```
typedef int semaphore;
    semaphore resource_1;
    semaphore resource_2;

    void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }

    void process_B(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }
```

(a)

```
    semaphore resource_1;
    semaphore resource_2;

    void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }

    void process_B(void) {
        down(&resource_2);
        down(&resource_1);
        use_both_resources( );
        up(&resource_1);
        up(&resource_2);
    }
```

(b)

Figure 3-2. (a) Deadlock-free code. (b) Code with a potential deadlock.

3.2 INTRODUCTION TO DEADLOCKS

Deadlock can be defined formally as follows:

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

Because all the processes are waiting, none of them will ever cause any of the events that could wake up any of the other members of the set, and all the processes continue to wait forever. For this model, we assume that processes have only a single thread and that there are no interrupts possible to wake up a blocked process. The no-interrupts condition is needed to prevent an otherwise deadlocked process from being awakened by, say, an alarm, and then causing events that release other processes in the set.

In most cases, the event that each process is waiting for is the release of some resource currently possessed by another member of the set. In other words, each member of the set of deadlocked processes is waiting for a resource that is owned by a deadlocked process. None of the processes can run, none of them can release any resources, and none of them can be awakened. The number of processes and the number and kind of resources possessed and requested are unimportant. This result holds for any kind of resource, including both hardware and software.

3.2.1 Conditions for Deadlock

Coffman et al. (1971) showed that four conditions must hold for there to be a deadlock:

1. Mutual exclusion condition. Each resource is either currently assigned to exactly one process or is available.
2. Hold and wait condition. Processes currently holding resources granted earlier can request new resources.
3. No preemption condition. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
4. Circular wait condition. There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

All four of these conditions must be present for a deadlock to occur. If one of them is absent, no deadlock is possible.

It is worth noting that each condition relates to a policy that a system can have or not have. Can a given resource be assigned to more than one process at once? Can a process hold a resource and ask for another? Can resources be preempted? Can circular waits exist? Later on we will see how deadlocks can be attacked by trying to negate some of these conditions.

3.2.2 Deadlock Modeling

Holt (1972) showed how these four conditions can be modeled using directed graphs. The graphs have two kinds of nodes: processes, shown as circles, and resources, shown as squares. An arc from a resource node (square) to a process node (circle) means that the resource has previously been requested by, granted to, and is currently held by that process. In Fig. 3-3(a), resource R is currently assigned to process A.

An arc from a process to a resource means that the process is currently blocked waiting for that resource. In Fig. 3-3(b), process B is waiting for resource S. In Fig. 3-3(c) we see a deadlock: process C is waiting for resource T, which is currently held by process D. Process D is not about to release resource T because it is waiting for resource U, held by C. Both processes will wait forever. A cycle in the graph means that there is a deadlock involving the processes and resources in the cycle (assuming that there is one resource of each kind). In this example, the cycle is C–T–D–U–C.

Now let us look at an example of how resource graphs can be used. Imagine that we have three processes, A, B, and C, and three resources, R, S, and T. The

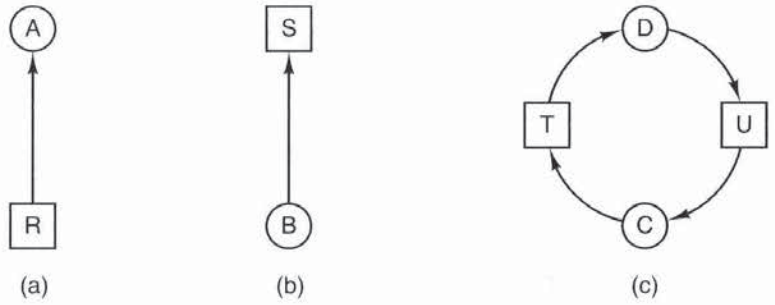


Figure 3-3. Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

requests and releases of the three processes are given in Fig. 3-4(a)-(c). The operating system is free to run any unblocked process at any instant, so it could decide to run *A* until *A* finished all its work, then run *B* to completion, and finally run *C*.

This ordering does not lead to any deadlocks (because there is no competition for resources) but it also has no parallelism at all. In addition to requesting and releasing resources, processes compute and do I/O. When the processes are run sequentially, there is no possibility that while one process is waiting for I/O, another can use the CPU. Thus running the processes strictly sequentially may not be optimal. On the other hand, if none of the processes do any I/O at all, shortest job first is better than round robin, so under some circumstances running all processes sequentially may be the best way.

Let us now suppose that the processes do both I/O and computing, so that round robin is a reasonable scheduling algorithm. The resource requests might occur in the order of Fig. 3-4(d). If these six requests are carried out in that order, the six resulting resource graphs are shown in Fig. 3-4(e)-(j). After request 4 has been made, *A* blocks waiting for *S*, as shown in Fig. 3-4(h). In the next two steps *B* and *C* also block, ultimately leading to a cycle and the deadlock of Fig. 3-4(j).

However, as we have already mentioned, the operating system is not required to run the processes in any special order. In particular, if granting a particular request might lead to deadlock, the operating system can simply suspend the process without granting the request (i.e., just not schedule the process) until it is safe. In Fig. 3-4, if the operating system knew about the impending deadlock, it could suspend *B* instead of granting it *S*. By running only *A* and *C*, we would get the requests and releases of Fig. 3-4(k) instead of Fig. 3-4(d). This sequence leads to the resource graphs of Fig. 3-4(l)-(q), which do not lead to deadlock.

After step (q), process *B* can be granted *S* because *A* is finished and *C* has everything it needs. Even if *B* should eventually block when requesting *T*, no deadlock can occur. *B* will just wait until *C* is finished.

Later in this chapter we will study a detailed algorithm for making allocation decisions that do not lead to deadlock. For the moment, the point to understand is

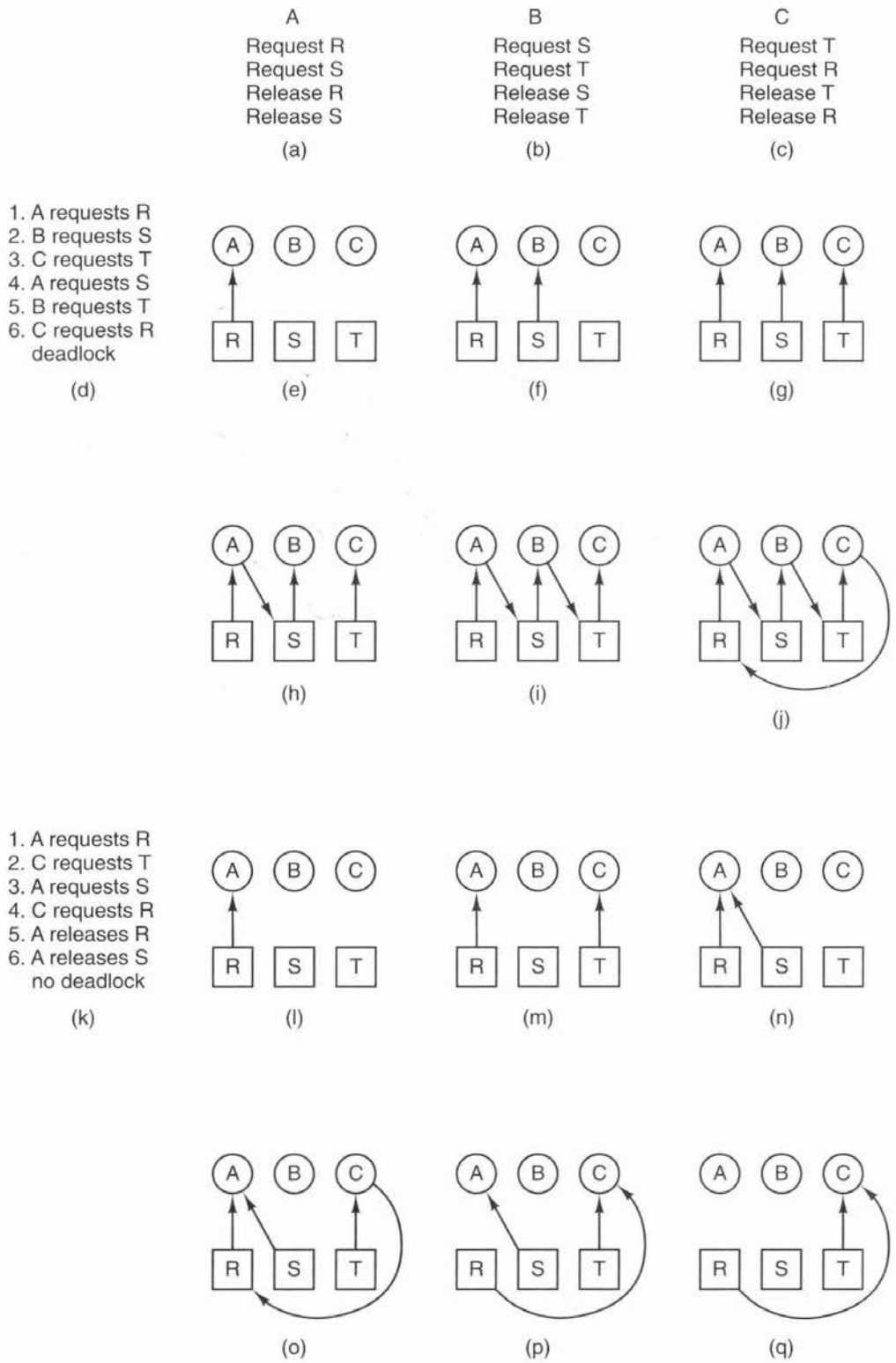


Figure 3-4. An example of how deadlock occurs and how it can be avoided.

that resource graphs are a tool that let us see if a given request/release sequence leads to deadlock. We just carry out the requests and releases step by step, and after every step check the graph to see if it contains any cycles. If so, we have a deadlock; if not, there is no deadlock. Although our treatment of resource graphs has been for the case of a single resource of each type, resource graphs can also be generalized to handle multiple resources of the same type (Holt, 1972).

In general, four strategies are used for dealing with deadlocks.

1. Just ignore the problem altogether. Maybe if you ignore it, it will ignore you.
2. Detection and recovery. Let deadlocks occur, detect them, and take action.
3. Dynamic avoidance by careful resource allocation.
4. Prevention, by structurally negating one of the four conditions necessary to cause a deadlock.

We will examine each of these methods in turn in the next four sections.

3.3 THE OSTRICH ALGORITHM

The simplest approach is the ostrich algorithm: stick your head in the sand and pretend there is no problem at all†. Different people react to this strategy in different ways. Mathematicians find it totally unacceptable and say that deadlocks must be prevented at all costs. Engineers ask how often the problem is expected, how often the system crashes for other reasons, and how serious a deadlock is. If deadlocks occur on the average once every five years, but system crashes due to hardware failures, compiler errors, and operating system bugs occur once a week, most engineers would not be willing to pay a large penalty in performance or convenience to eliminate deadlocks.

To make this contrast more specific, most operating systems potentially suffer from deadlocks that are not even detected, let alone automatically broken. The total number of processes in a system is determined by the number of entries in the process table. Thus process table slots are finite resources. If a fork fails because the table is full, a reasonable approach for the program doing the fork is to wait a random time and try again.

Now suppose that a UNIX system has 100 process slots. Ten programs are running, each of which needs to create 12 (sub)processes. After each process has

†Actually, this bit of folklore is nonsense. Ostriches can run at 60 km/hour and their kick is powerful enough to kill any lion with visions of a big chicken dinner.

created 9 processes, the 10 original processes and the 90 new processes have exhausted the table. Each of the 10 original processes now sits in an endless loop forking and failing—a deadlock. The probability of this happening is minuscule, but it *could* happen. Should we abandon processes and the fork call to eliminate the problem?

The maximum number of open files is similarly restricted by the size of the i-node table, so a similar problem occurs when it fills up. Swap space on the disk is another limited resource. In fact, almost every table in the operating system represents a finite resource. Should we abolish all of these because it might happen that a collection of n processes might each claim $1/n$ of the total, and then each try to claim another one?

Most operating systems, including UNIX and Windows, just ignore the problem on the assumption that most users would prefer an occasional deadlock to a rule restricting all users to one process, one open file, and one of everything. If deadlocks could be eliminated for free, there would not be much discussion. The problem is that the price is high, mostly in terms of putting inconvenient restrictions on processes, as we will see shortly. Thus we are faced with an unpleasant trade-off between convenience and correctness, and a great deal of discussion about which is more important, and to whom. Under these conditions, general solutions are hard to find.

3.4 DEADLOCK DETECTION AND RECOVERY

A second technique is detection and recovery. When this technique is used, the system does not attempt to prevent deadlocks from occurring. Instead, it lets them occur, tries to detect when this happens, and then takes some action to recover after the fact. In this section we will look at some of the ways deadlocks can be detected and some of the ways recovery from them can be handled.

3.4.1 Deadlock Detection with One Resource of Each Type

Let us begin with the simplest case: only one resource of each type exists. Such a system might have one scanner, one CD recorder, one plotter, and one tape drive, but no more than one of each class of resource. In other words, we are excluding systems with two printers for the moment. We will treat them later, using a different method.

For such a system, we can construct a resource graph of the sort illustrated in Fig. 3-3. If this graph contains one or more cycles, a deadlock exists. Any process that is part of a cycle is deadlocked. If no cycles exist, the system is not deadlocked.

As an example of a more complex system than the ones we have looked at so far, consider a system with seven processes, A though G, and six resources, R

through *W*. The state of which resources are currently owned and which ones are currently being requested is as follows:

1. Process *A* holds *R* and wants *S*.
2. Process *B* holds nothing but wants *T*.
3. Process *C* holds nothing but wants *S*.
4. Process *D* holds *U* and wants *S* and *T*.
5. Process *E* holds *T* and wants *V*.
6. Process *F* holds *W* and wants *S*.
7. Process *G* holds *V* and wants *U*.

The question is: "Is this system deadlocked, and if so, which processes are involved?"

To answer this question, we can construct the resource graph of Fig. 3-5(a). This graph contains one cycle, which can be seen by visual inspection. The cycle is shown in Fig. 3-5(b). From this cycle, we can see that processes *D*, *E*, and *G* are all deadlocked. Processes *A*, *C*, and *F* are not deadlocked because *S* can be allocated to any one of them, which then finishes and returns it. Then the other two can take it in turn and also complete.

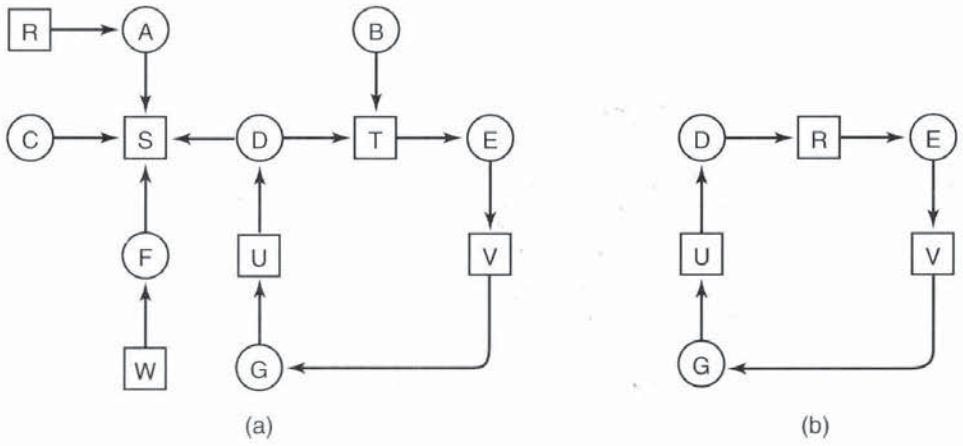


Figure 3-5. (a) A resource graph. (b) A cycle extracted from (a).

Although it is relatively simple to pick out the deadlocked processes by eye from a simple graph, for use in actual systems we need a formal algorithm for detecting deadlocks. Many algorithms for detecting cycles in directed graphs are known. Below we will give a simple one that inspects a graph and terminates either when it has found a cycle or when it has shown that none exist. It uses one data structure, *L*, a list of nodes. During the algorithm, arcs will be marked to indicate that they have already been inspected, to prevent repeated inspections.

The algorithm operates by carrying out the following steps as specified:

1. For each node, N in the graph, perform the following 5 steps with N as the starting node.
2. Initialize L to the empty list, and designate all the arcs as unmarked.
3. Add the current node to the end of L and check to see if the node now appears in L two times. If it does, the graph contains a cycle (listed in L) and the algorithm terminates.
4. From the given node, see if there are any unmarked outgoing arcs. If so, go to step 5; if not, go to step 6.
5. Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to step 3.
6. We have now reached a dead end. Remove it and go back to the previous node, that is, the one that was current just before this one, make that one the current node, and go to step 3. If this node is the initial node, the graph does not contain any cycles and the algorithm terminates.

What this algorithm does is take each node, in turn, as the root of what it hopes will be a tree, and does a depth-first search on it. If it ever comes back to a node it has already encountered, then it has found a cycle. If it exhausts all the arcs from any given node, it backtracks to the previous node. If it backtracks to the root and cannot go further, the subgraph reachable from the current node does not contain any cycles. If this property holds for all nodes, the entire graph is cycle free, so the system is not deadlocked.

To see how the algorithm works in practice, let us use it on the graph of Fig. 3-5(a). The order of processing the nodes is arbitrary, so let us just inspect them from left to right, top to bottom, first running the algorithm starting at R, then successively, A, B, C, S, D, T, E, F, and so forth. If we hit a cycle, the algorithm stops.

We start at R and initialize L to the empty list. Then we add R to the list and move to the only possibility, A, and add it to L, giving $L = [R, A]$. From A we go to S, giving $L = [R, A, S]$. S has no outgoing arcs, so it is a dead end, forcing us to backtrack to A. Since A has no unmarked outgoing arcs, we backtrack to R, completing our inspection of R.

Now we restart the algorithm starting at A, resetting L to the empty list. This search, too, quickly stops, so we start again at B. From B we continue to follow outgoing arcs until we get to D, at which time $L = [B, T, E, V, G, U, D]$. Now we must make a (random) choice. If we pick S we come to a dead end and backtrack to D. The second time we pick T and update L to be $[B, T, E, V, G, U, D, T]$, at which point we discover the cycle and stop the algorithm.

This algorithm is far from optimal. For a better one, see (Even, 1979). Nevertheless, it demonstrates that an algorithm for deadlock detection exists.

3.4.2 Deadlock Detection with Multiple Resource of Each Type

When multiple copies of some of the resources exist, a different approach is needed to detect deadlocks. We will now present a matrix-based algorithm for detecting deadlock among n processes, P_1 through P_n. Let the number of resource classes be m, with E_1 resources of class 1, E_2 resources of class 2, and generally, E_i resources of class i ($1 \le i \le m$). E is the **existing resource vector**. It gives the total number of instances of each resource in existence. For example, if class 1 is tape drives, then $E_1 = 2$ means the system has two tape drives.

At any instant, some of the resources are assigned and are not available. Let A be the **available resource vector**, with A_i giving the number of instances of resource i that are currently available (i.e., unassigned). If both of our two tape drives are assigned, A_1 will be 0.

Now we need two arrays, C, the **current allocation matrix**, and R, the **request matrix**. The i-th row of C tells how many instances of each resource class P_i currently holds. Thus C_{ij} is the number of instances of resource j that are held by process i. Similarly, R_{ij} is the number of instances of resource j that P_i wants. These four data structures are shown in Fig. 3-6.

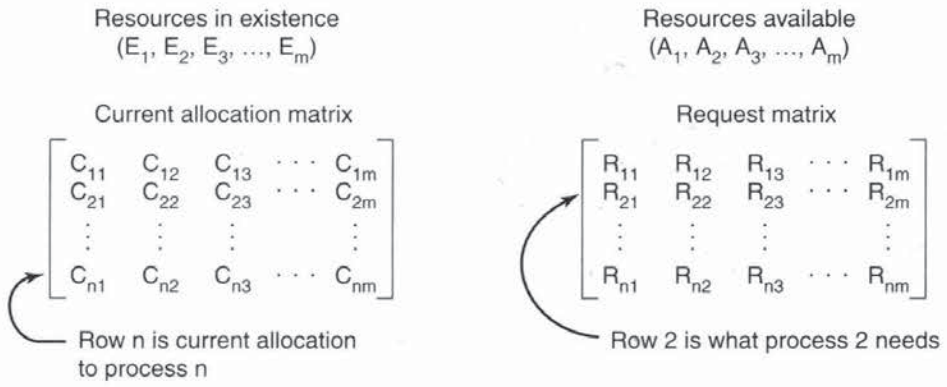


Figure 3-6. The four data structures needed by the deadlock detection algorithm.

An important invariant holds for these four data structures. In particular, every resource is either allocated or is available. This observation means that

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

In other words, if we add up all the instances of the resource j that have been

allocated and to this add all the instances that are available, the result is the number of instances of that resource class that exist.

The deadlock detection algorithm is based on comparing vectors. Let us define the relation $A \leq B$ on two vectors A and B to mean that each element of A is less than or equal to the corresponding element of B. Mathematically, $A \leq B$ holds if and only if $A_i \leq B_i$ for $1 \leq i \leq m$.

Each process is initially said to be unmarked. As the algorithm progresses, processes will be marked, indicating that they are able to complete and are thus not deadlocked. When the algorithm terminates, any unmarked processes are known to be deadlocked.

The deadlock detection algorithm can now be given, as follows.

1. Look for an unmarked process, P_i, for which the i-th row of R is less than or equal to A.
2. If such a process is found, add the i-th row of C to A, mark the process, and go back to step 1.
3. If no such process exists, the algorithm terminates.

When the algorithm finishes, all the unmarked processes, if any, are deadlocked.

What the algorithm is doing in step 1 is looking for a process that can be run to completion. Such a process is characterized as having resource demands that can be met by the currently available resources. The selected process is then run until it finishes, at which time it returns the resources it is holding to the pool of available resources. It is then marked as completed. If all the processes are ultimately able to run, none of them are deadlocked. If some of them can never run, they are deadlocked. Although the algorithm is nondeterministic (because it may run the processes in any feasible order), the result is always the same.

As an example of how the deadlock detection algorithm works, consider Fig. 3-7. Here we have three processes and four resource classes, which we have arbitrarily labeled tape drives, plotters, scanner, and CD-ROM drive. Process 1 has one scanner. Process 2 has two tape drives and a CD-ROM drive. Process 3 has a plotter and two scanners. Each process needs additional resources, as shown by the R matrix .

To run the deadlock detection algorithm, we look for a process whose resource request can be satisfied. The first one cannot be satisfied because there is no CD-ROM drive available. The second cannot be satisfied either, because there is no scanner free. Fortunately, the third one can be satisfied, so process 3 runs and eventually returns all its resources, giving

$$A = (2\ 2\ 2\ 0)$$

At this point process 2 can run and return its resources, giving

$$A = (4\ 2\ 2\ 1)$$

Now the remaining process can run. There is no deadlock in the system.

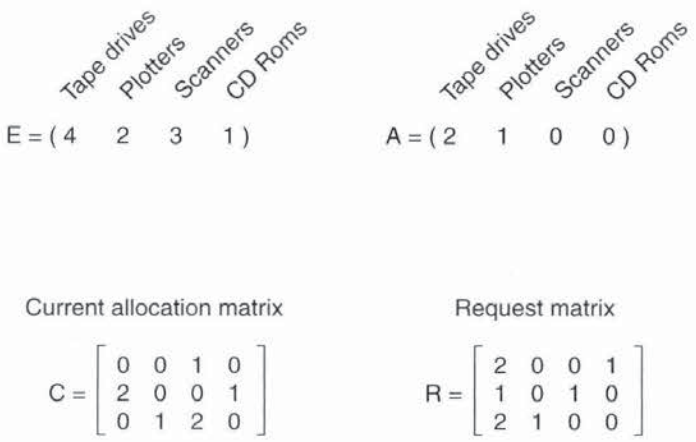


Figure 3-7. An example for the deadlock detection algorithm.

Now consider a minor variation of the situation of Fig. 3-7. Suppose that process 2 needs a CD-ROM drive as well as the two tape drives and the plotter. None of the requests can be satisfied, so the entire system is deadlocked.

Now that we know how to detect deadlocks, the question of when to look for them comes up. One possibility is to check every time a resource request is made. This is certain to detect them as early as possible, but it is potentially expensive in terms of CPU time. An alternative strategy is to check every k minutes, or perhaps only when the CPU utilization has dropped below some threshold. The reason for considering the CPU utilization is that if enough processes are deadlocked, there will be few runnable processes, and the CPU will often be idle.

3.4.3 Recovery from Deadlock

Suppose that our deadlock detection algorithm has succeeded and detected a deadlock. What next? Some way is needed to recover and get the system going again. In this section we will discuss various ways of recovering from deadlock. None of them are especially attractive, however.

Recovery through Preemption

In some cases it may be possible to temporarily take a resource away from its current owner and give it to another process. In many cases, manual intervention may be required, especially in batch processing operating systems running on mainframes.

For example, to take a laser printer away from its owner, the operator can collect all the sheets already printed and put them in a pile. Then the process can be suspended (marked as not runnable). At this point the printer can be assigned to

another process. When that process finishes, the pile of printed sheets can be put back in the printer's output tray and the original process restarted.

The ability to take a resource away from a process, have another process use it, and then give it back without the process noticing it is highly dependent on the nature of the resource. Recovering this way is frequently difficult or impossible. Choosing the process to suspend depends largely on which ones have resources that can easily be taken back.

Recovery through Rollback

If the system designers and machine operators know that deadlocks are likely, they can arrange to have processes **checkpointed** periodically. Checkpointing a process means that its state is written to a file so that it can be restarted later. The checkpoint contains not only the memory image, but also the resource state, that is, which resources are currently assigned to the process. To be most effective, new checkpoints should not overwrite old ones but should be written to new files, so as the process executes, a whole sequence of checkpoint files are accumulated.

When a deadlock is detected, it is easy to see which resources are needed. To do the recovery, a process that owns a needed resource is rolled back to a point in time before it acquired some other resource by starting one of its earlier checkpoints. All the work done since the checkpoint is lost (e.g., output printed since the checkpoint must be discarded, since it will be printed again). In effect, the process is reset to an earlier moment when it did not have the resource, which is now assigned to one of the deadlocked processes. If the restarted process tries to acquire the resource again, it will have to wait until it becomes available.

Recovery through Killing Processes

The crudest, but simplest way to break a deadlock is to kill one or more processes. One possibility is to kill a process in the cycle. With a little luck, the other processes will be able to continue. If this does not help, it can be repeated until the cycle is broken.

Alternatively, a process not in the cycle can be chosen as the victim in order to release its resources. In this approach, the process to be killed is carefully chosen because it is holding resources that some process in the cycle needs. For example, one process might hold a printer and want a plotter, with another process holding a plotter and wanting a printer. These two are deadlocked. A third process may hold another identical printer and another identical plotter and be happily running. Killing the third process will release these resources and break the deadlock involving the first two.

Where possible, it is best to kill a process that can be rerun from the beginning with no ill effects. For example, a compilation can always be rerun because

all it does is read a source file and produce an object file. If it is killed part way through, the first run has no influence on the second run.

On the other hand, a process that updates a database cannot always be run a second time safely. If the process adds 1 to some record in the database, running it once, killing it, and then running it again will add 2 to the record, which is incorrect.

3.5 DEADLOCK AVOIDANCE

In the discussion of deadlock detection, we tacitly assumed that when a process asks for resources, it asks for them all at once (the R matrix of Fig. 3-6). In most systems, however, resources are requested one at a time. The system must be able to decide whether granting a resource is safe or not and only make the allocation when it is safe. Thus the question arises: Is there an algorithm that can always avoid deadlock by making the right choice all the time? The answer is a qualified yes—we can avoid deadlocks, but only if certain information is available in advance. In this section we examine ways to avoid deadlock by careful resource allocation.

3.5.1 Resource Trajectories

The main algorithms for doing deadlock avoidance are based on the concept of safe states. Before describing the algorithms, we will make a slight digression to look at the concept of safety in a graphic and easy-to-understand way. Although the graphical approach does not translate directly into a usable algorithm, it gives a good intuitive feel for the nature of the problem.

In Fig. 3-8 we see a model for dealing with two processes and two resources, for example, a printer and a plotter. The horizontal axis represents the number of instructions executed by process A. The vertical axis represents the number of instructions executed by process B. At I_1 A requests a printer; at I_2 it needs a plotter. The printer and plotter are released at I_3 and I_4, respectively. Process B needs the plotter from I_5 to I_7 and the printer from I_6 to I_8.

Every point in the diagram represents a joint state of the two processes. Initially, the state is at p, with neither process having executed any instructions. If the scheduler chooses to run A first, we get to the point q, in which A has executed some number of instructions, but B has executed none. At point q the trajectory becomes vertical, indicating that the scheduler has chosen to run B. With a single processor, all paths must be horizontal or vertical, never diagonal. Furthermore, motion is always to the north or east, never to the south or west (processes cannot run backward).

When A crosses the I_1 line on the path from r to s, it requests and is granted the printer. When B reaches point t, it requests the plotter.

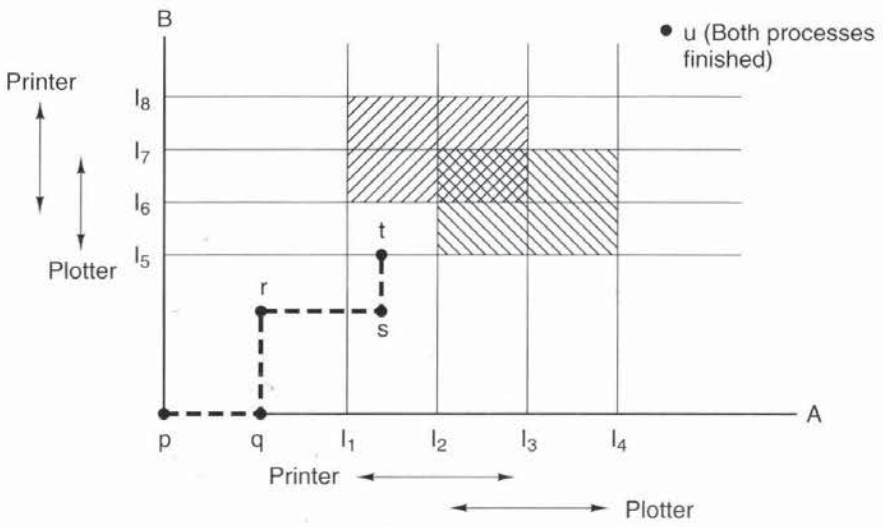


Figure 3-8. Two process resource trajectories.

The regions that are shaded are especially interesting. The region with lines slanting from southwest to northeast represents both processes having the printer. The mutual exclusion rule makes it impossible to enter this region. Similarly, the region shaded the other way represents both processes having the plotter, and is equally impossible.

If the system ever enters the box bounded by I_1 and I_2 on the sides and I_5 and I_6 top and bottom, it will eventually deadlock when it gets to the intersection of I_2 and I_6. At this point, A is requesting the plotter and B is requesting the printer, and both are already assigned. The entire box is unsafe and must not be entered. At point t the only safe thing to do is run process A until it gets to I_4. Beyond that, any trajectory to u will do.

The important thing to see here is at point t B is requesting a resource. The system must decide whether to grant it or not. If the grant is made, the system will enter an unsafe region and eventually deadlock. To avoid the deadlock, B should be suspended until A has requested and released the plotter.

3.5.2 Safe and Unsafe States

The deadlock avoidance algorithms that we will study use the information of Fig. 3-6. At any instant of time, there is a current state consisting of E, A, C, and R. A state is said to be **safe** if it is not deadlocked and there is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately. It is easiest to illustrate this concept by an example using one resource. In Fig. 3-9(a) we have a state in which A has 3 instances of the resource but may need as many as 9 eventually. B currently has 2 and may need 4 altogether, later. Similarly, C also has 2 but may

need an additional 5. A total of 10 instances of the resource exist, so with 7 resources already allocated, there are 3 still free.

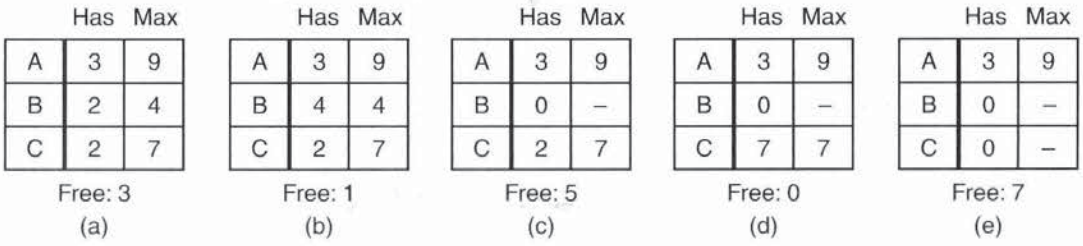


Figure 3-9. Demonstration that the state in (a) is safe.

The state of Fig. 3-9(a) is safe because there exists a sequence of allocations that allows all processes to complete. Namely, the scheduler could simply run *B* exclusively, until it asked for and got two more instances of the resource, leading to the state of Fig. 3-9(b). When *B* completes, we get the state of Fig. 3-9(c). Then the scheduler can run *C*, leading eventually to Fig. 3-9(d). When *C* completes, we get Fig. 3-9(e). Now *A* can get the six instances of the resource it needs and also complete. Thus the state of Fig. 3-9(a) is safe because the system, by careful scheduling, can avoid deadlock.

Now suppose we have the initial state shown in Fig. 3-10(a), but this time *A* requests and gets another resource, giving Fig. 3-10(b). Can we find a sequence that is guaranteed to work? Let us try. The scheduler could run *B* until it asked for all its resources, as shown in Fig. 3-10(c).

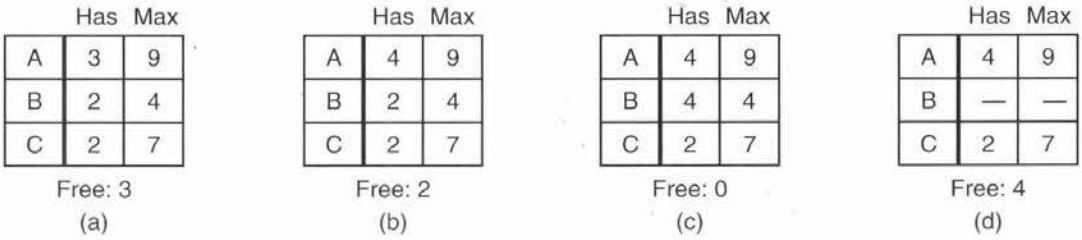


Figure 3-10. Demonstration that the state in (b) is not safe.

Eventually, *B* completes and we get the situation of Fig. 3-10(d). At this point we are stuck. We only have four instances of the resource free, and each of the active processes needs five. There is no sequence that guarantees completion. Thus the allocation decision that moved the system from Fig. 3-10(a) to Fig. 3-10(b) went from a safe state to an unsafe state. Running *A* or *C* next starting at Fig. 3-10(b) does not work either. In retrospect, *A*'s request should not have been granted.

It is worth noting that an unsafe state is not a deadlocked state. Starting at Fig. 3-10(b), the system can run for a while. In fact, one process can even complete. Furthermore, it is possible that *A* might release a resource before asking for

any more, allowing C to complete and avoiding deadlock altogether. Thus the difference between a safe state and an unsafe state is that from a safe state the system can *guarantee* that all processes will finish; from an unsafe state, no such guarantee can be given.

3.5.3 The Banker's Algorithm for a Single Resource

A scheduling algorithm that can avoid deadlocks is due to Dijkstra (1965) and is known as the **banker's algorithm** and is an extension of the deadlock detection algorithm given in Sec. 3.4.1. It is modeled on the way a small-town banker might deal with a group of customers to whom he has granted lines of credit. What the algorithm does is check to see if granting the request leads to an unsafe state. If it does, the request is denied. If granting the request leads to a safe state, it is carried out. In Fig. 3-11(a) we see four customers, A, B, C, and D, each of whom has been granted a certain number of credit units (e.g., 1 unit is 1K dollars). The banker knows that not all customers will need their maximum credit immediately, so he has reserved only 10 units rather than 22 to service them. (In this analogy, customers are processes, units are, say, tape drives, and the banker is the operating system.)

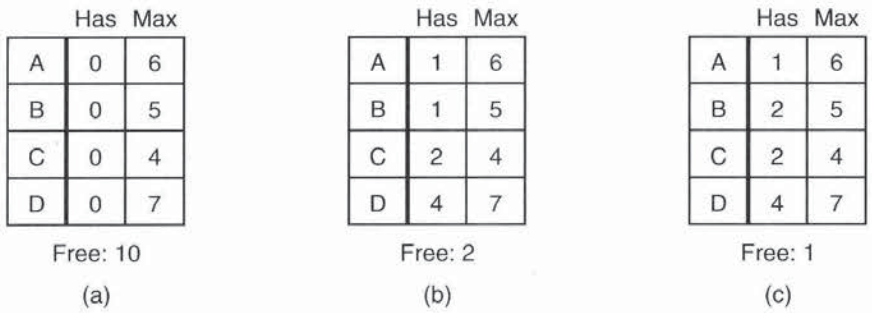


Figure 3-11. Three resource allocation states: (a) Safe. (b) Safe. (c) Unsafe.

The customers go about their respective businesses, making loan requests from time to time (i.e., asking for resources). At a certain moment, the situation is as shown in Fig. 3-11(b). This state is safe because with two units left, the banker can delay any requests except C's, thus letting C finish and release all four of his resources. With four units in hand, the banker can let either D or B have the necessary units, and so on.

Consider what would happen if a request from B for one more unit were granted in Fig. 3-11(b). We would have situation Fig. 3-11(c), which is unsafe. If all the customers suddenly asked for their maximum loans, the banker could not satisfy any of them, and we would have a deadlock. An unsafe state does not *have* to lead to deadlock, since a customer might not need the entire credit line available, but the banker cannot count on this behavior.

The banker's algorithm considers each request as it occurs, and see if granting it leads to a safe state. If it does, the request is granted; otherwise, it is postponed

until later. To see if a state is safe, the banker checks to see if he has enough resources to satisfy some customer. If so, those loans are assumed to be repaid, and the customer now closest to the limit is checked, and so on. If all loans can eventually be repaid, the state is safe and the initial request can be granted.

3.5.4 The Banker's Algorithm for Multiple Resources

The banker's algorithm can be generalized to handle multiple resources. Figure 3-12 shows how it works.

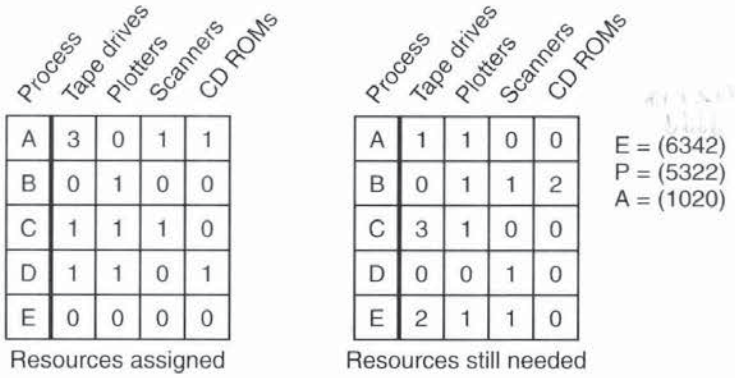


Figure 3-12. The banker's algorithm with multiple resources.

In Fig. 3-12 we see two matrices. The one on the left shows how many of each resource are currently assigned to each of the five processes. The matrix on the right shows how many resources each process still needs in order to complete. These matrices are just C and R from Fig. 3-6. As in the single resource case, processes must state their total resource needs before executing, so that the system can compute the right-hand matrix at each instant.

The three vectors at the right of the figure show the existing resources, E, the possessed resources, P, and the available resources, A, respectively. From E we see that the system has six tape drives, three plotters, four printers, and two CD-ROM drives. Of these, five tape drives, three plotters, two printers, and two CD-ROM drives are currently assigned. This fact can be seen by adding up the four resource columns in the left-hand matrix. The available resource vector is simply the difference between what the system has and what is currently in use.

The algorithm for checking to see if a state is safe can now be stated.

1. Look for a row, R, whose unmet resource needs are all smaller than or equal to A. If no such row exists, the system will eventually deadlock since no process can run to completion.
2. Assume the process of the row chosen requests all the resources it needs (which is guaranteed to be possible) and finishes. Mark that process as terminated and add all its resources to the A vector.

3. Repeat steps 1 and 2 until either all processes are marked terminated, in which case the initial state was safe, or until a deadlock occurs, in which case it was not.

If several processes are eligible to be chosen in step 1, it does not matter which one is selected: the pool of available resources either gets larger, or at worst, stays the same.

Now let us get back to the example of Fig. 3-12. The current state is safe. Suppose that process *B* now requests a printer. This request can be granted because the resulting state is still safe (process *D* can finish, and then processes *A* or *E*, followed by the rest).

Now imagine that after giving *B* one of the two remaining printers, *E* wants the last printer. Granting that request would reduce the vector of available resources to (1 0 0 0), which leads to deadlock. Clearly *E*'s request must be deferred for a while.

The banker's algorithm was first published by Dijkstra in 1965. Since that time, nearly every book on operating systems has described it in detail. Innumerable papers have been written about various aspects of it. Unfortunately, few authors have had the audacity to point out that although in theory the algorithm is wonderful, in practice it is essentially useless because processes rarely know in advance what their maximum resource needs will be. In addition, the number of processes is not fixed, but dynamically varying as new users log in and out. Furthermore, resources that were thought to be available can suddenly vanish (tape drives can break). Thus in practice, few, if any, existing systems use the banker's algorithm for avoiding deadlocks.

3.6 DEADLOCK PREVENTION

Having seen that deadlock avoidance is essentially impossible, because it requires information about future requests, which is not known, how do real systems avoid deadlock? The answer is to go back to the four conditions stated by Coffman et al. (1971) to see if they can provide a clue. If we can ensure that at least one of these conditions is never satisfied, then deadlocks will be structurally impossible (Havender, 1968).

3.6.1 Attacking the Mutual Exclusion Condition

First let us attack the mutual exclusion condition. If no resource were ever assigned exclusively to a single process, we would never have deadlocks. However, it is equally clear that allowing two processes to write on the printer at the

same time will lead to chaos. By spooling printer output, several processes can generate output at the same time. In this model, the only process that actually requests the physical printer is the printer daemon. Since the daemon never requests any other resources, we can eliminate deadlock for the printer.

Unfortunately, not all devices can be spooled (the process table does not lend itself well to being spooled). Furthermore, competition for disk space for spooling can itself lead to deadlock. What would happen if two processes each filled up half of the available spooling space with output and neither was finished producing output? If the daemon was programmed to begin printing even before all the output was spooled, the printer might lie idle if an output process decided to wait several hours after the first burst of output. For this reason, daemons are normally programmed to print only after the complete output file is available. In this case we have two processes that have each finished part, but not all, of their output, and cannot continue. Neither process will ever finish, so we have a deadlock on the disk.

Nevertheless, there is a germ of an idea here that is frequently applicable. Avoid assigning a resource when that is not absolutely necessary, and try to make sure that as few processes as possible may actually claim the resource.

3.6.2 Attacking the Hold and Wait Condition

The second of the conditions stated by Coffman et al. looks slightly more promising. If we can prevent processes that hold resources from waiting for more resources, we can eliminate deadlocks. One way to achieve this goal is to require all processes to request all their resources before starting execution. If everything is available, the process will be allocated whatever it needs and can run to completion. If one or more resources are busy, nothing will be allocated and the process would just wait.

An immediate problem with this approach is that many processes do not know how many resources they will need until they have started running. In fact, if they knew, the banker's algorithm could be used. Another problem is that resources will not be used optimally with this approach. Take, as an example, a process that reads data from an input tape, analyzes it for an hour, and then writes an output tape as well as plotting the results. If all resources must be requested in advance, the process will tie up the output tape drive and the plotter for an hour.

Nevertheless, some mainframe batch systems require the user to list all the resources on the first line of each job. The system then acquires all resources immediately and keeps them until the job finishes. While this method puts a burden on the programmer and wastes resources, it does prevent deadlocks.

A slightly different way to break the hold-and-wait condition is to require a process requesting a resource to first temporarily release all the resources it currently holds. Then it tries to get everything it needs all at once.

3.6.3 Attacking the No Preemption Condition

Attacking the third condition (no preemption) is even less promising than attacking the second one. If a process has been assigned the printer and is in the middle of printing its output, forcibly taking away the printer because a needed plotter is not available is tricky at best and impossible at worst.

3.6.4 Attacking the Circular Wait Condition

Only one condition is left. The circular wait can be eliminated in several ways. One way is simply to have a rule saying that a process is entitled only to a single resource at any moment. If it needs a second one, it must release the first one. For a process that needs to copy a huge file from a tape to a printer, this restriction is unacceptable.

Another way to avoid the circular wait is to provide a global numbering of all the resources, as shown in Fig. 3-13(a). Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order. A process may request first a printer and then a tape drive, but it may not request first a plotter and then a printer.



Figure 3-13. (a) Numerically ordered resources. (b) A resource graph.

With this rule, the resource allocation graph can never have cycles. Let us see why this is true for the case of two processes, in Fig. 3-13(b). We can get a deadlock only if A requests resource j and B requests resource i. Assuming i and j are distinct resources, they will have different numbers. If $i > j$, then A is not allowed to request j because that is lower than what it already has. If $i < j$, then B is not allowed to request i because that is lower than what it already has. Either way, deadlock is impossible.

With multiple processes, the same logic holds. At every instant, one of the assigned resources will be highest. The process holding that resource will never ask for a resource already assigned. It will either finish, or at worst, request even higher numbered resources, all of which are available. Eventually, it will finish and free its resources. At this point, some other process will hold the highest resource and can also finish. In short, there exists a scenario in which all processes finish, so no deadlock is present.

A minor variation of this algorithm is to drop the requirement that resources be acquired in strictly increasing sequence and merely insist that no process request a resource lower than what it is already holding. If a process initially requests 9 and 10, and then releases both of them, it is effectively starting all over, so there is no reason to prohibit it from now requesting resource 1.

Although numerically ordering the resources eliminates the problem of deadlocks, it may be impossible to find an ordering that satisfies everyone. When the resources include process table slots, disk spooler space, locked database records, and other abstract resources, the number of potential resources and different uses may be so large that no ordering could possibly work.

The various approaches to deadlock prevention are summarized in Fig. 3-14.

Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

Figure 3-14. Summary of approaches to deadlock prevention.

3.7 OTHER ISSUES

In this section we will discuss a few miscellaneous issues related to deadlocks. These include two-phase locking, nonresource deadlocks, and starvation.

3.7.1 Two-Phase Locking

Although both avoidance and prevention are not terribly promising in the general case, for specific applications, many excellent special-purpose algorithms are known. As an example, in many database systems, an operation that occurs frequently is requesting locks on several records and then updating all the locked records. When multiple processes are running at the same time, there is a real danger of deadlock.

The approach often used is called **two-phase locking**. In the first phase, the process tries to lock all the records it needs, one at a time. If it succeeds, it begins the second phase, performing its updates and releasing the locks. No real work is done in the first phase.

If during the first phase, some record is needed that is already locked, the process just releases all its locks and starts the first phase all over. In a certain

sense, this approach is similar to requesting all the resources needed in advance, or at least before anything irreversible is done. In some versions of two-phase locking, there is no release and restart if a lock is encountered during the first phase. In these versions, deadlock can occur.

However, this strategy is not applicable in general. In real-time systems and process control systems, for example, it is not acceptable to just terminate a process partway through because a resource is not available and start all over again. Neither is it acceptable to start over if the process has read or written messages to the network, updated files, or anything else that cannot be safely repeated. The algorithm works only in those situations where the programmer has very carefully arranged things so that the program can be stopped at any point during the first phase and restarted. Many applications cannot be structured this way.

3.7.2 Nonresource Deadlocks

All of our work so far has concentrated on resource deadlocks. One process wants something that another process has and must wait until the first one gives it up. Deadlocks can also occur in other situations, however, including those not involving resources at all.

For example, it can happen that two processes deadlock each waiting for the other one to do something. This often happens with semaphores. In Chap. 2 we saw examples in which a process had to do a down on two semaphores, typically *mutex* and another one. If these are done in the wrong order, deadlock can result.

3.7.3 Starvation

A problem closely related to deadlock is **starvation**. In a dynamic system, requests for resources happen all the time. Some policy is needed to make a decision about who gets which resource when. This policy, although seemingly reasonable, may lead to some processes never getting service even though they are not deadlocked.

As an example, consider allocation of the printer. Imagine that the system uses some kind of algorithm to ensure that allocating the printer does not lead to deadlock. Now suppose that several processes all want it at once. Which one should get it?

One possible allocation algorithm is to give it to the process with the smallest file to print (assuming this information is available). This approach maximizes the number of happy customers and seems fair. Now consider what happens in a busy system when one process has a huge file to print. Every time the printer is free, the system will look around and choose the process with the shortest file. If there is a constant stream of processes with short files, the process with the huge

file will never be allocated the printer. It will simply starve to death (be postponed indefinitely, even though it is not blocked).

Starvation can be avoided by using a first-come, first-serve, resource allocation policy. With this approach, the process waiting the longest gets served next. In due course of time, any given process will eventually become the oldest and thus get the needed resource.

3.8 RESEARCH ON DEADLOCKS

If ever there was a subject that was investigated mercilessly during the early days of operating systems, it was deadlocks. The reason for this is that deadlock detection is a nice little graph theory problem that one mathematically-inclined graduate student can get his jaws around and chew on for 3 or 4 years. All kinds of algorithms were devised, each one more exotic and less practical than the previous one. Essentially, all this research has died out, with only a very occasional new paper appearing (e.g., Karacali et al., 2000). When an operating system wants to do deadlock detection or prevention, which few of them do, they use one of the methods discussed in this chapter,

There is still a little research on distributed deadlock detection, however. We will not treat that here because (1) it is outside the scope of this book, and (2) none of it is even remotely practical in real systems. Its main function seems to be keeping otherwise unemployed graph theorists off the streets.

3.9 SUMMARY

Deadlock is a potential problem in any operating system. It occurs when a group of processes each have been granted exclusive access to some resources, and each one wants yet another resource that belongs to another process in the group. All of them are blocked and none will ever run again.

Deadlock can be avoided by keeping track of which states are safe and which are unsafe. A safe state is one in which there exists a sequence of events that guarantee that all processes can finish. An unsafe state has no such guarantee. The banker's algorithm avoids deadlock by not granting a request if that request will put the system in an unsafe state.

Deadlock can be structurally prevented by building the system in such a way that it can never occur by design. For example, by allowing a process to hold only one resource at any instant the circular wait condition required for deadlock is broken. Deadlock can also be prevented by numbering all the resources, and making processes request them in strictly increasing order. Starvation can be avoided by a first-come, first-served allocation policy.

PROBLEMS

1. Give an example of a deadlock taken from politics.
2. Students working at individual PCs in a computer laboratory send their files to be printed by a server which spools the files on its hard disk. Under what conditions may a deadlock occur if the disk space for the print spool is limited? How may the deadlock be avoided?
3. In the preceding question which resources are preemptable and which are nonpreemptable?
4. In Fig. 3-1 the resources are returned in the reverse order of their acquisition. Would giving them back in the other order be just as good?
5. Fig. 3-3 shows the concept of a resource graph. Do illegal graphs exist, that is graphs that structurally violate the model we have used of resource usage? If so, give an example of one.
6. The discussion of the ostrich algorithm mentions the possibility of process table slots or other system tables filling up. Can you suggest a way to enable a system administrator to recover from such a situation?
7. Consider Fig. 3-4. Suppose that in step (o) C requested S instead of requesting R. Would this lead to deadlock? Suppose that it requested both S and R?
8. At a crossroads with STOP signs on all four approaches, the rule is that each driver yields the right of way to the driver on his right. This rule is not adequate when four vehicles arrive simultaneously. Fortunately, humans are sometimes capable of acting more intelligently than computers and the problem is usually resolved when one driver signals the driver to his left to go ahead. Can you draw an analogy between this behavior and any of the ways of recovering from deadlock described in Sec. 3.4.3? Why is a problem with such a simple solution in the human world so difficult to apply to a computer system?
9. Suppose that in Fig. 3-6 $C_{ij} + R_{ij} > E_j$ for some i. What implications does this have for all the processes finishing without deadlock?
10. All the trajectories in Fig. 3-8 are horizontal or vertical. Can you envision any circumstances in which diagonal trajectories were also possible?
11. Can the resource trajectory scheme of Fig. 3-8 also be used to illustrate the problem of deadlocks with three processes and three resources? If so, how can this be done? If not, why not?
12. In theory, resource trajectory graphs could be used to avoid deadlocks. By clever scheduling, the operating system could avoid unsafe regions. Suggest a practical problem with actually doing this.
13. Take a careful look at Fig. 3-11(b). If D asks for one more unit, does this lead to a safe state or an unsafe one? What if the request came from C instead of D?
14. Can a system be in a state that is neither deadlocked nor safe? If so, give an example. If not, prove that all states are either deadlocked or safe.

15. A system has two processes and three identical resources. Each process needs a maximum of two resources. Is deadlock possible? Explain your answer.

16. Consider the previous problem again, but now with p processes each needing a maximum of m resources and a total of r resources available. What condition must hold to make the system deadlock free?

17. Suppose that process A in Fig. 3-12 requests the last tape drive. Does this action lead to a deadlock?

18. A computer has six tape drives, with n processes competing for them. Each process may need two drives. For which values of n is the system deadlock free?

19. The banker's algorithm is being run in a system with m resource classes and n processes. In the limit of large m and n, the number of operations that must be performed to check a state for safety is proportional to $m^a n^b$. What are the values of a and b?

20. A system has four processes and five allocatable resources. The current allocation and maximum needs are as follows:

	Allocated	*Maximum*	*Available*
Process A	1 0 2 1 1	1 1 2 1 3	0 0 x 1 1
Process B	2 0 1 1 0	2 2 2 1 0	
Process C	1 1 0 1 0	2 1 3 1 0	
Process D	1 1 1 1 0	1 1 2 2 1	

What is the smallest value of x for which this is a safe state?

21. A distributed system using mailboxes has two IPC primitives, send and receive. The latter primitive specifies a process to receive from and blocks if no message from that process is available, even though messages may be waiting from other processes. There are no shared resources, but processes need to communicate frequently about other matters. Is deadlock possible? Discuss.

22. Two processes, A and B, each need three records, 1, 2, and 3, in a database. If A asks for them in the order 1, 2, 3, and B asks for them in the same order, deadlock is not possible. However, if B asks for them in the order 3, 2, 1, then deadlock is possible. With three resources, there are 3! or 6 possible combinations each process can request the resources. What fraction of all the combinations are guaranteed to be deadlock free?

23. Now reconsider the above problem, but using two-phase locking. Will that eliminate the potential for deadlock? Does it have any other undesirable characteristics, however? If so, which ones?

24. In an electronic funds transfer system, there are hundreds of identical processes that work as follows. Each process reads an input line specifying an amount of money, the account to be credited, and the account to be debited. Then it locks both accounts and transfers the money, releasing the locks when done. With many processes running in parallel, there is a very real danger that having locked account x it will be unable to lock y because y has been locked by a process now waiting for x. Devise a scheme that avoids deadlocks. Do not release an account record until you have completed the

transactions. (In other words, solutions that lock one account and then release it immediately if the other is locked are not allowed.)

25. One way to prevent deadlocks is to eliminate the hold-and-wait condition. In the text it was proposed that before asking for a new resource, a process must first release whatever resources it already holds (assuming that is possible). However, doing so introduces the danger that it may get the new resource but lose some of the existing ones to competing processes. Propose an improvement to this scheme.

26. A computer science student assigned to work on deadlocks thinks of the following brilliant way to eliminate deadlocks. When a process requests a resource, it specifies a time limit. If the process blocks because the resource is not available, a timer is started. If the time limit is exceeded, the process is released and allowed to run again. If you were the professor, what grade would you give this proposal and why.

27. Cinderella and the Prince are getting divorced. To divide their property, they have agreed on the following algorithm. Every morning, each one may send a letter to the other's lawyer requesting one item of property. Since it takes a day for letters to be delivered, they have agreed that if both discover that they have requested the same item on the same day, the next day they will send a letter canceling the request. Among their property is their dog, Woofer, Woofer's doghouse, their canary, Tweeter, and Tweeter's cage. The animals love their houses, so it has been agreed that any division of property separating an animal from its house is invalid, requiring the whole division to start over from scratch. Both Cinderella and the Prince desperately want Woofer. So they can go on (separate) vacations, each spouse has programmed a personal computer to handle the negotiation. When they come back from vacation, the computers are still negotiating. Why? Is deadlock possible? Is starvation possible? Discuss.

28. A student majoring in anthropology and minoring in computer science has embarked on a research project to see if African baboons can be taught about deadlocks. He locates a deep canyon and fastens a rope across it, so the baboons can cross hand-over-hand. Several baboons can cross at the same time, provided that they are all going in the same direction. If eastward moving and westward moving baboons ever get onto the rope at the same time, a deadlock will result (the baboons will get stuck in the middle) because it is impossible for one baboon to climb over another one while suspended over the canyon. If a baboon wants to cross the canyon, he must check to see that no other baboon is currently crossing in the opposite direction. Write a program using semaphores that avoids deadlock. Do not worry about a series of eastward moving baboons holding up the westward moving baboons indefinitely.

29. Repeat the previous problem, but now avoid starvation. When a baboon that wants to cross to the east arrives at the rope and finds baboons crossing to the west, he waits until the rope is empty, but no more westward moving baboons are allowed to start until at least one baboon has crossed the other way.

30. Program a simulation of the banker's algorithm. Your program should cycle through each of the bank clients asking for a request and evaluating whether it is safe or unsafe. Output a log of requests and decisions to a file.

4

MEMORY MANAGEMENT

Memory is an important resource that must be carefully managed. While the average home computer nowadays has a thousand times as much memory as the IBM 7094, the largest computer in the world in the early 1960s, programs are getting bigger faster than memories. To paraphrase Parkinson's law, "Programs expand to fill the memory available to hold them." In this chapter we will study how operating systems manage memory.

Ideally, what every programmer would like is an infinitely large, infinitely fast memory that is also nonvolatile, that is, does not lose its contents when the electric power fails. While we are at it, why not also ask for it to be inexpensive, too? Unfortunately technology does not provide such memories. Consequently, most computers have a **memory hierarchy**, with a small amount of very fast, expensive, volatile cache memory, tens of megabytes of medium-speed, medium-price, volatile main memory (RAM), and tens or hundreds of gigabytes of slow, cheap, nonvolatile disk storage. It is the job of the operating system to coordinate how these memories are used.

The part of the operating system that manages the memory hierarchy is called the **memory manager**. Its job is to keep track of which parts of memory are in use and which parts are not in use, to allocate memory to processes when they need it and deallocate it when they are done, and to manage swapping between main memory and disk when main memory is too small to hold all the processes.

In this chapter we will investigate a number of different memory management schemes, ranging from very simple to highly sophisticated. We will start at the

beginning and look first at the simplest possible memory management system and then gradually progress to more and more elaborate ones.

As we pointed out in Chap. 1, history tends to repeat itself in the computer world. While the simplest memory management schemes are no longer used on desktop computers, they are still used in some palmtop, embedded, and smart card systems. For this reason, they are still worth studying.

4.1 BASIC MEMORY MANAGEMENT

Memory management systems can be divided into two classes: those that move processes back and forth between main memory and disk during execution (swapping and paging), and those that do not. The latter are simpler, so we will study them first. Later in the chapter we will examine swapping and paging. Throughout this chapter the reader should keep in mind that swapping and paging are largely artifacts caused by the lack of sufficient main memory to hold all the programs at once. If main memory ever gets so large that there is truly enough of it, the arguments in favor of one kind of memory management scheme or another may become obsolete.

On the other hand, as mentioned above, software seems to be growing even faster than memory, so efficient memory management may always be needed. In the 1980s, there were many universities that ran a timesharing system with dozens of (more-or-less satisfied) users on a 4 MB VAX. Now Microsoft recommends having at least 64 MB for a single-user Windows 2000 system. The trend toward multimedia puts even more demands on memory, so good memory management is probably going to be needed for the next decade at least.

4.1.1 Monoprogramming without Swapping or Paging

The simplest possible memory management scheme is to run just one program at a time, sharing the memory between that program and the operating system. Three variations on this theme are shown in Fig. 4-1. The operating system may be at the bottom of memory in RAM (Random Access Memory), as shown in Fig. 4-1(a), or it may be in ROM (Read-Only Memory) at the top of memory, as shown in Fig. 4-1(b), or the device drivers may be at the top of memory in a ROM and the rest of the system in RAM down below, as shown in Fig. 4-1(c). The first model was formerly used on mainframes and minicomputers but is rarely used any more. The second model is used on some palmtop computers and embedded systems. The third model was used by early personal computers (e.g., running MS-DOS), where the portion of the system in the ROM is called the **BIOS** (Basic Input Output System).

When the system is organized in this way, only one process at a time can be running. As soon as the user types a command, the operating system copies the

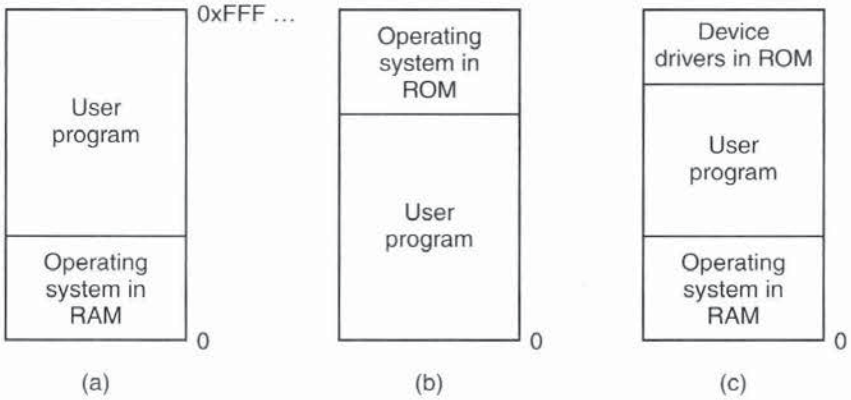


Figure 4-1. Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

requested program from disk to memory and executes it. When the process finishes, the operating system displays a prompt character and waits for a new command. When it receives the command, it loads a new program into memory, overwriting the first one.

4.1.2 Multiprogramming with Fixed Partitions

Except on simple embedded systems, monoprogramming is hardly used any more. Most modern systems allow multiple processes to run at the same time. Having multiple processes running at once means that when one process is blocked waiting for I/O to finish, another one can use the CPU. Thus multiprogramming increases the CPU utilization. Network servers always have the ability to run multiple processes (for different clients) at the same time, but most client (i.e., desktop) machines also have this ability nowadays.

The easiest way to achieve multiprogramming is simply to divide memory up into n (possibly unequal) partitions. This partitioning can, for example, be done manually when the system is started up.

When a job arrives, it can be put into the input queue for the smallest partition large enough to hold it. Since the partitions are fixed in this scheme, any space in a partition not used by a job is lost. In Fig. 4-2(a) we see how this system of fixed partitions and separate input queues looks.

The disadvantage of sorting the incoming jobs into separate queues becomes apparent when the queue for a large partition is empty but the queue for a small partition is full, as is the case for partitions 1 and 3 in Fig. 4-2(a). Here small jobs have to wait to get into memory, even though plenty of memory is free. An alternative organization is to maintain a single queue as in Fig. 4-2(b). Whenever a partition becomes free, the job closest to the front of the queue that fits in it could be loaded into the empty partition and run. Since it is undesirable to waste a large

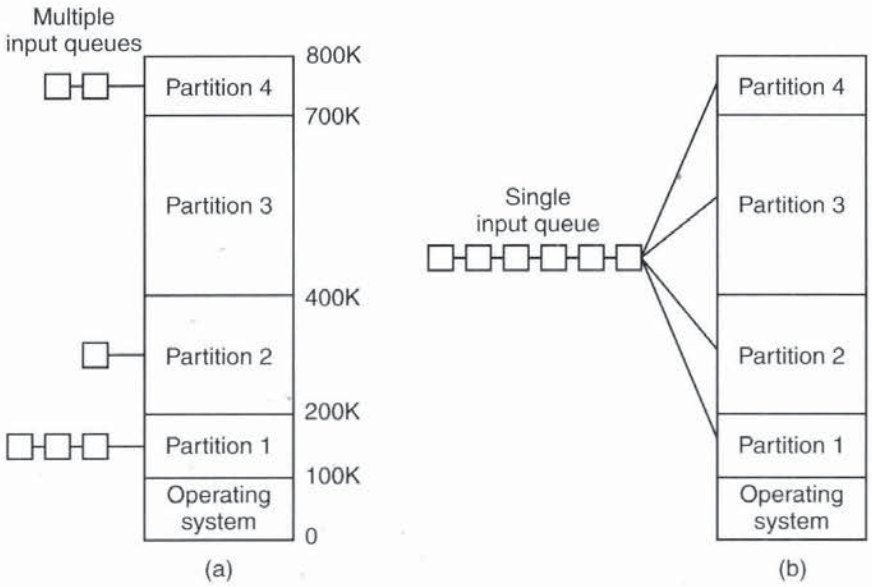


Figure 4-2. (a) Fixed memory partitions with separate input queues for each partition. (b) Fixed memory partitions with a single input queue.

partition on a small job, a different strategy is to search the whole input queue whenever a partition becomes free and pick the largest job that fits. Note that the latter algorithm discriminates against small jobs as being unworthy of having a whole partition, whereas usually it is desirable to give the smallest jobs (often interactive jobs) the best service, not the worst.

One way out is to have at least one small partition around. Such a partition will allow small jobs to run without having to allocate a large partition for them.

Another approach is to have a rule stating that a job that is eligible to run may not be skipped over more than k times. Each time it is skipped over, it gets one point. When it has acquired k points, it may not be skipped again.

This system, with fixed partitions set up by the operator in the morning and not changed thereafter, was used by OS/360 on large IBM mainframes for many years. It was called **MFT** (Multiprogramming with a Fixed number of Tasks or OS/MFT). It is simple to understand and equally simple to implement: incoming jobs are queued until a suitable partition is available, at which time the job is loaded into that partition and run until it terminates. Nowadays, few, if any, operating systems, support this model.

4.1.3 Modeling Multiprogramming

When multiprogramming is used, the CPU utilization can be improved. Crudely put, if the average process computes only 20 percent of the time it is sitting in memory, with five processes in memory at once, the CPU should be busy

all the time. This model is unrealistically optimistic, however, since it assumes that all five processes will never be waiting for I/O at the same time.

A better model is to look at CPU usage from a probabilistic viewpoint. Suppose that a process spends a fraction p of its time waiting for I/O to complete. With n processes in memory at once, the probability that all n processes are waiting for I/O (in which case the CPU will be idle) is p^n. The CPU utilization is then given by the formula

$$\text{CPU utilization} = 1 - p^n$$

Figure 4-3 shows the CPU utilization as a function of n, which is called the **degree of multiprogramming**.

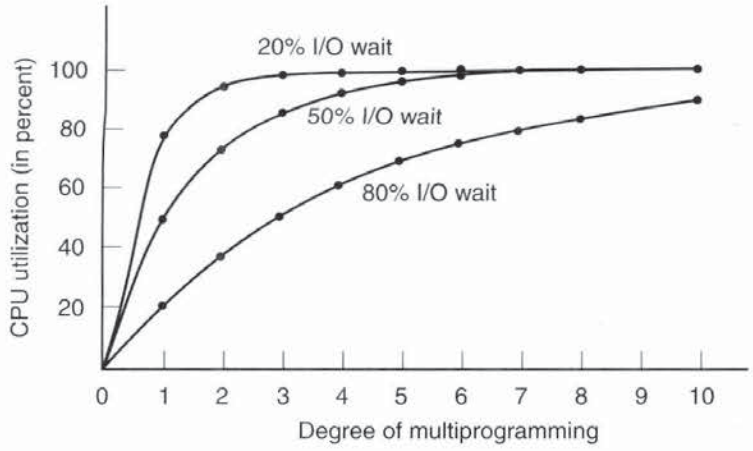


Figure 4-3. CPU utilization as a function of the number of processes in memory.

From the figure it is clear that if processes spend 80 percent of their time waiting for I/O, at least 10 processes must be in memory at once to get the CPU waste below 10 percent. When you realize that an interactive process waiting for a user to type something at a terminal is in I/O wait state, it should be clear that I/O wait times of 80 percent and more are not unusual. But even in batch systems, processes doing a lot of disk I/O will often have this percentage or more.

For the sake of complete accuracy, it should be pointed out that the probabilistic model just described is only an approximation. It implicitly assumes that all n processes are independent, meaning that it is quite acceptable for a system with five processes in memory to have three running and two waiting. But with a single CPU, we cannot have three processes running at once, so a process becoming ready while the CPU is busy will have to wait. Thus the processes are not independent. A more accurate model can be constructed using queueing theory, but the point we are making—multiprogramming lets processes use the CPU when it would be otherwise idle—is, of course, still valid, even if the true curves of Fig. 4-3 are slightly different.

Even though the model of Fig. 4-3 is simple-minded, it can nevertheless be used to make specific, although approximate, predictions about CPU performance.

Suppose, for example, that a computer has 32 MB of memory, with the operating system taking up 16 MB and each user program taking up 4 MB. These sizes allow four user programs to be in memory at once. With an 80 percent average I/O wait, we have a CPU utilization (ignoring operating system overhead) of $1 - 0.8^4$ or about 60 percent. Adding another 16 MB of memory allows the system to go from four-way multiprogramming to eight-way multiprogramming, thus raising the CPU utilization to 83 percent. In other words, the additional 16 MB will raise the throughput by 38 percent.

Adding yet another 16 MB would only increase CPU utilization from 83 percent to 93 percent, thus raising the throughput by only another 12 percent. Using this model the computer's owner might decide that the first addition is a good investment but that the second is not.

4.1.4 Analysis of Multiprogramming System Performance

The model discussed above can also be used to analyze batch systems. Consider, for example, a computer center whose jobs average 80 percent I/O wait time. On a particular morning, four jobs are submitted as shown in Fig. 4-4(a). The first job, arriving at 10:00 A.M., requires 4 minutes of CPU time. With 80 percent I/O wait, the job uses only 12 seconds of CPU time for each minute it is sitting in memory, even if no other jobs are competing with it for the CPU. The other 48 seconds are spent waiting for I/O to complete. Thus the job will have to sit in memory for at least 20 minutes in order to get 4 minutes of CPU work done, even in the absence of competition for the CPU.

From 10:00 A.M. to 10:10 A.M., job 1 is all by itself in memory and gets 2 minutes of work done. When job 2 arrives at 10:10 A.M., the CPU utilization increases from 0.20 to 0.36, due to the higher degree of multiprogramming (see Fig. 4-3). However, with round-robin scheduling, each job gets half of the CPU, so each job gets 0.18 minutes of CPU work done for each minute it is in memory. Notice that the addition of a second job costs the first job only 10 percent of its performance. It goes from getting 0.20 CPU minutes per minute of real time to getting 0.18 CPU minutes per minute of real time.

At 10:15 A.M. the third job arrives. At this point job 1 has received 2.9 minutes of CPU and job 2 has had 0.9 minutes of CPU. With three-way multiprogramming, each job gets 0.16 minutes of CPU time per minute of real time, as shown in Fig. 4-4(b). From 10:15 A.M. to 10:20 A.M. each of the three jobs gets 0.8 minutes of CPU time. At 10:20 A.M. a fourth job arrives. Fig. 4-4(c) shows the complete sequence of events.

4.1.5 Relocation and Protection

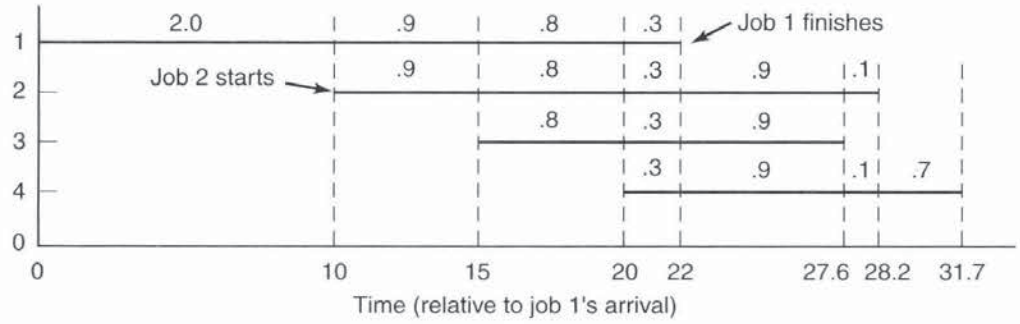
Multiprogramming introduces two essential problems that must be solved—relocation and protection. Look at Fig. 4-2. From the figure it is clear that different jobs will be run at different addresses. When a program is linked (i.e., the

Job	Arrival time	CPU minutes needed
1	10:00	4
2	10:10	3
3	10:15	2
4	10:20	2

(a)

	# Processes 1	2	3	4
CPU idle	.80	.64	.51	.41
CPU busy	.20	.36	.49	.59
CPU/process	.20	.18	.16	.15

(b)



(c)

Figure 4-4. (a) Arrival and work requirements of four jobs. (b) CPU utilization for 1 to 4 jobs with 80 percent I/O wait. (c) Sequence of events as jobs arrive and finish. The numbers above the horizontal lines show how much CPU time, in minutes, each job gets in each interval.

main program, user-written procedures, and library procedures are combined into a single address space), the linker must know at what address the program will begin in memory.

For example, suppose that the first instruction is a call to a procedure at absolute address 100 within the binary file produced by the linker. If this program is loaded in partition 1 (at address 100K), that instruction will jump to absolute address 100, which is inside the operating system. What is needed is a call to 100K + 100. If the program is loaded into partition 2, it must be carried out as a call to 200K + 100, and so on. This problem is known as the **relocation** problem.

One possible solution is to actually modify the instructions as the program is loaded into memory. Programs loaded into partition 1 have 100K added to each address, programs loaded into partition 2 have 200K added to addresses, and so forth. To perform relocation during loading like this, the linker must include in the binary program a list or bitmap telling which program words are addresses to be relocated and which are opcodes, constants, or other items that must not be relocated. OS/MFT worked this way.

Relocation during loading does not solve the protection problem. A malicious program can always construct a new instruction and jump to it. Because programs in this system use absolute memory addresses rather than addresses relative to a

register, there is no way to stop a program from building an instruction that reads or writes any word in memory. In multiuser systems, it is highly undesirable to let processes read and write memory belonging to other users.

The solution that IBM chose for protecting the 360 was to divide memory into blocks of 2-KB bytes and assign a 4-bit protection code to each block. The PSW (Program Status Word) contained a 4-bit key. The 360 hardware trapped any attempt by a running process to access memory whose protection code differed from the PSW key. Since only the operating system could change the protection codes and key, user processes were prevented from interfering with one another and with the operating system itself.

An alternative solution to both the relocation and protection problems is to equip the machine with two special hardware registers, called the **base** and **limit** registers. When a process is scheduled, the base register is loaded with the address of the start of its partition, and the limit register is loaded with the length of the partition. Every memory address generated automatically has the base register contents added to it before being sent to memory. Thus if the base register contains the value 100K, a CALL 100 instruction is effectively turned into a CALL 100K + 100 instruction, without the instruction itself being modified. Addresses are also checked against the limit register to make sure that they do not attempt to address memory outside the current partition. The hardware protects the base and limit registers to prevent user programs from modifying them.

A disadvantage of this scheme is the need to perform an addition and a comparison on every memory reference. Comparisons can be done fast, but additions are slow due to carry propagation time unless special addition circuits are used.

The CDC 6600—the world's first supercomputer—used this scheme. The Intel 8088 CPU used for the original IBM PC used a weaker version of this scheme—base registers, but no limit registers. Few computers use it any more though.

4.2 SWAPPING

With a batch system, organizing memory into fixed partitions is simple and effective. Each job is loaded into a partition when it gets to the head of the queue. It stays in memory until it has finished. As long as enough jobs can be kept in memory to keep the CPU busy all the time, there is no reason to use anything more complicated.

With timesharing systems or graphically oriented personal computers, the situation is different. Sometimes there is not enough main memory to hold all the currently active processes, so excess processes must be kept on disk and brought in to run dynamically.

Two general approaches to memory management can be used, depending (in part) on the available hardware. The simplest strategy, called **swapping**, consists

of bringing in each process in its entirety, running it for a while, then putting it back on the disk. The other strategy, called **virtual memory**, allows programs to run even when they are only partially in main memory. Below we will study swapping; in Sec. 4.3 we will examine virtual memory.

The operation of a swapping system is illustrated in Fig. 4-5. Initially only process *A* is in memory. Then processes *B* and *C* are created or swapped in from disk. In Fig. 4-5(d) *A* is swapped out to disk. Then *D* comes in and *B* goes out. Finally *A* comes in again. Since *A* is now at a different location, addresses contained in it must be relocated, either by software when it is swapped in or (more likely) by hardware during program execution.

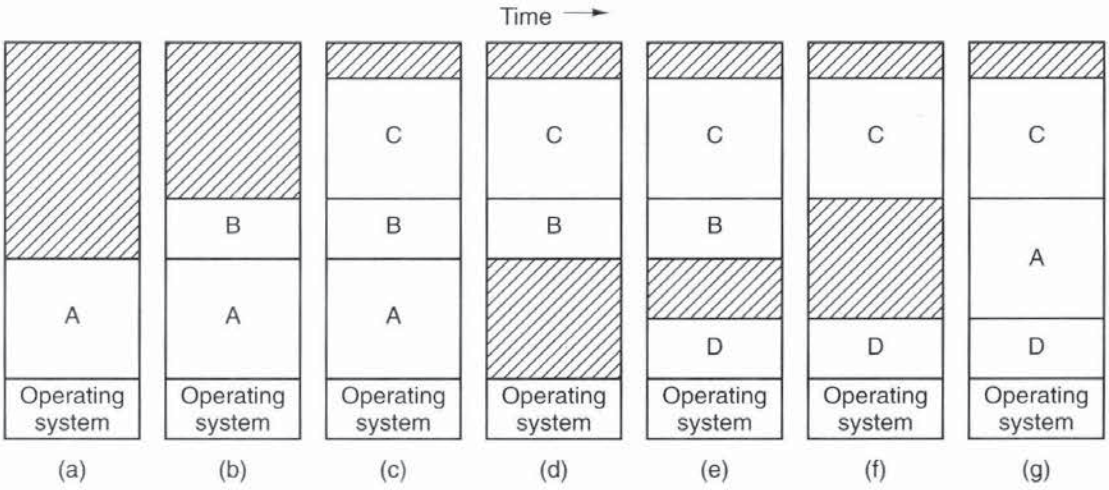


Figure 4-5. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

The main difference between the fixed partitions of Fig. 4-2 and the variable partitions of Fig. 4-5 is that the number, location, and size of the partitions vary dynamically in the latter as processes come and go, whereas they are fixed in the former. The flexibility of not being tied to a fixed number of partitions that may be too large or too small improves memory utilization, but it also complicates allocating and deallocating memory, as well as keeping track of it.

When swapping creates multiple holes in memory, it is possible to combine them all into one big one by moving all the processes downward as far as possible. This technique is known as **memory compaction**. It is usually not done because it requires a lot of CPU time. For example, on a 256-MB machine that can copy 4 bytes in 40 nsec, it takes about 2.7 sec to compact all of memory.

A point that is worth making concerns how much memory should be allocated for a process when it is created or swapped in. If processes are created with a fixed size that never changes, then the allocation is simple: the operating system allocates exactly what is needed, no more and no less.

If, however, processes' data segments can grow, for example, by dynamically allocating memory from a heap, as in many programming languages, a problem occurs whenever a process tries to grow. If a hole is adjacent to the process, it can be allocated and the process allowed to grow into the hole. On the other hand, if the process is adjacent to another process, the growing process will either have to be moved to a hole in memory large enough for it, or one or more processes will have to be swapped out to create a large enough hole. If a process cannot grow in memory and the swap area on the disk is full, the process will have to wait or be killed.

If it is expected that most processes will grow as they run, it is probably a good idea to allocate a little extra memory whenever a process is swapped in or moved, to reduce the overhead associated with moving or swapping processes that no longer fit in their allocated memory. However, when swapping processes to disk, only the memory actually in use should be swapped; it is wasteful to swap the extra memory as well. In Fig. 4-6(a) we see a memory configuration in which space for growth has been allocated to two processes.

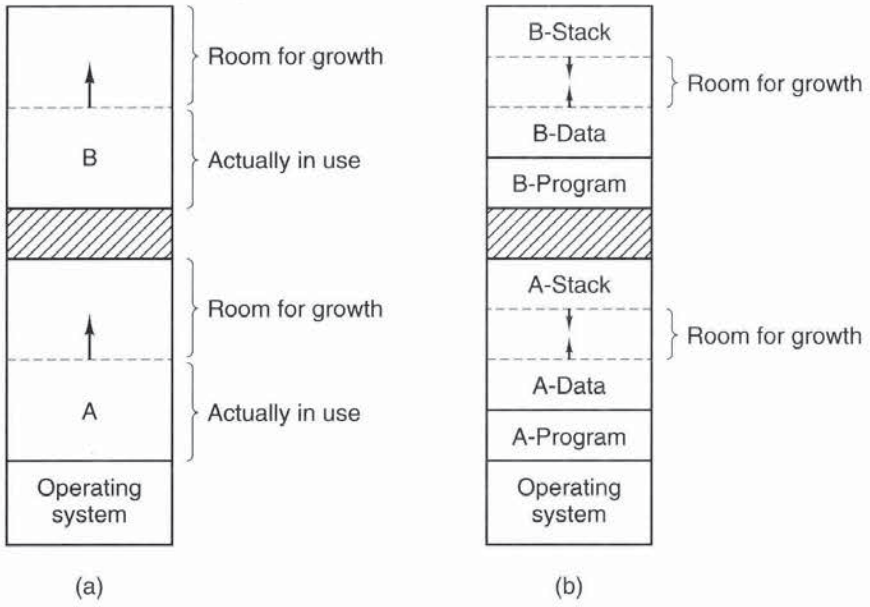


Figure 4-6. (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment.

If processes can have two growing segments, for example, the data segment being used as a heap for variables that are dynamically allocated and released and a stack segment for the normal local variables and return addresses, an alternative arrangement suggests itself, namely that of Fig. 4-6(b). In this figure we see that each process illustrated has a stack at the top of its allocated memory that is growing downward, and a data segment just beyond the program text that is growing

upward. The memory between them can be used for either segment. If it runs out, either the process will have to be moved to a hole with enough space, swapped out of memory until a large enough hole can be created, or killed.

4.2.1 Memory Management with Bitmaps

When memory is assigned dynamically, the operating system must manage it. In general terms, there are two ways to keep track of memory usage: bitmaps and free lists. In this section and the next one we will look at these two methods in turn.

With a bitmap, memory is divided up into allocation units, perhaps as small as a few words and perhaps as large as several kilobytes. Corresponding to each allocation unit is a bit in the bitmap, which is 0 if the unit is free and 1 if it is occupied (or vice versa). Figure 4-7 shows part of memory and the corresponding bitmap.

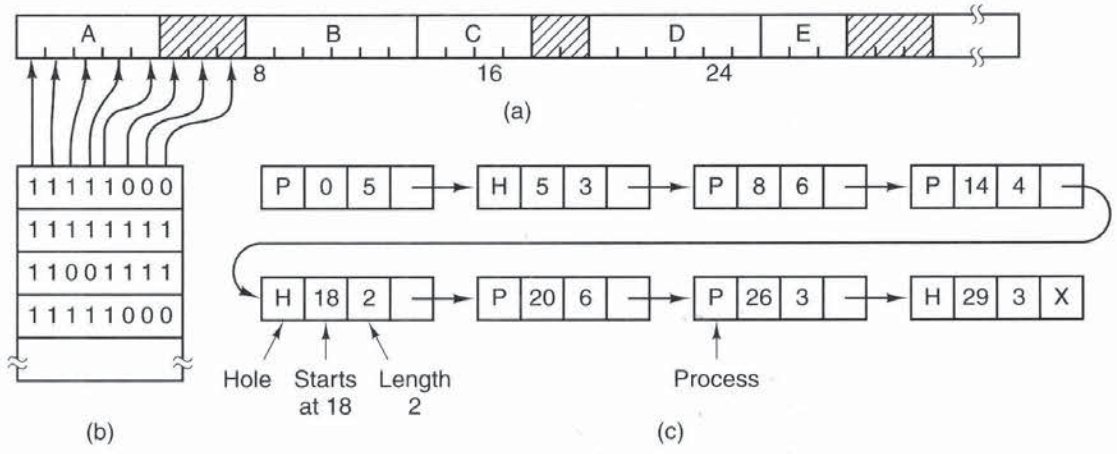


Figure 4-7. (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

The size of the allocation unit is an important design issue. The smaller the allocation unit, the larger the bitmap. However, even with an allocation unit as small as 4 bytes, 32 bits of memory will require only 1 bit of the map. A memory of $32n$ bits will use n map bits, so the bitmap will take up only 1/33 of memory. If the allocation unit is chosen large, the bitmap will be smaller, but appreciable memory may be wasted in the last unit of the process if the process size is not an exact multiple of the allocation unit.

A bitmap provides a simple way to keep track of memory words in a fixed amount of memory because the size of the bitmap depends only on the size of memory and the size of the allocation unit. The main problem with it is that when

it has been decided to bring a k unit process into memory, the memory manager must search the bitmap to find a run of k consecutive 0 bits in the map. Searching a bitmap for a run of a given length is a slow operation (because the run may straddle word boundaries in the map); this is an argument against bitmaps.

4.2.2 Memory Management with Linked Lists

Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment is either a process or a hole between two processes. The memory of Fig. 4-7(a) is represented in Fig. 4-7(c) as a linked list of segments. Each entry in the list specifies a hole (H) or process (P), the address at which it starts, the length, and a pointer to the next entry.

In this example, the segment list is kept sorted by address. Sorting this way has the advantage that when a process terminates or is swapped out, updating the list is straightforward. A terminating process normally has two neighbors (except when it is at the very top or bottom of memory). These may be either processes or holes, leading to the four combinations of Fig. 4-8. In Fig. 4-8(a) updating the list requires replacing a P by an H. In Fig. 4-8(b) and Fig. 4-8(c), two entries are coalesced into one, and the list becomes one entry shorter. In Fig. 4-8(d), three entries are merged and two items are removed from the list. Since the process table slot for the terminating process will normally point to the list entry for the process itself, it may be more convenient to have the list as a double-linked list, rather than the single-linked list of Fig. 4-7(c). This structure makes it easier to find the previous entry and to see if a merge is possible.

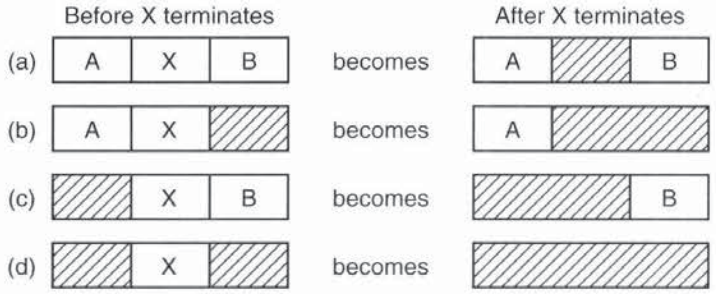


Figure 4-8. Four neighbor combinations for the terminating process, X.

When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a newly created process (or an existing process being swapped in from disk). We assume that the memory manager knows how much memory to allocate. The simplest algorithm is **first fit**. The memory manager scans along the list of segments until it finds a hole that is big enough. The hole is then broken up into two pieces, one for the process and one for the unused memory, except in the statistically unlikely case of an exact fit. First fit is a fast algorithm because it searches as little as possible.

A minor variation of first fit is **next fit**. It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole. The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of always at the beginning, as first fit does. Simulations by Bays (1977) show that next fit gives slightly worse performance than first fit.

Another well-known algorithm is **best fit**. Best fit searches the entire list and takes the smallest hole that is adequate. Rather than breaking up a big hole that might be needed later, best fit tries to find a hole that is close to the actual size needed.

As an example of first fit and best fit, consider Fig. 4-7 again. If a block of size 2 is needed, first fit will allocate the hole at 5, but best fit will allocate the hole at 18.

Best fit is slower than first fit because it must search the entire list every time it is called. Somewhat surprisingly, it also results in more wasted memory than first fit or next fit because it tends to fill up memory with tiny, useless holes. First fit generates larger holes on the average.

To get around the problem of breaking up nearly exact matches into a process and a tiny hole, one could think about **worst fit**, that is, always take the largest available hole, so that the hole broken off will be big enough to be useful. Simulation has shown that worst fit is not a very good idea either.

All four algorithms can be speeded up by maintaining separate lists for processes and holes. In this way, all of them devote their full energy to inspecting holes, not processes. The inevitable price that is paid for this speedup on allocation is the additional complexity and slowdown when deallocating memory, since a freed segment has to be removed from the process list and inserted into the hole list.

If distinct lists are maintained for processes and holes, the hole list may be kept sorted on size, to make best fit faster. When best fit searches a list of holes from smallest to largest, as soon as it finds a hole that fits, it knows that the hole is the smallest one that will do the job, hence the best fit. No further searching is needed, as it is with the single list scheme. With a hole list sorted by size, first fit and best fit are equally fast, and next fit is pointless.

When the holes are kept on separate lists from the processes, a small optimization is possible. Instead of having a separate set of data structures for maintaining the hole list, as is done in Fig. 4-7(c), the holes themselves can be used. The first word of each hole could be the hole size, and the second word a pointer to the following entry. The nodes of the list of Fig. 4-7(c), which require three words and one bit (P/H), are no longer needed.

Yet another allocation algorithm is **quick fit**, which maintains separate lists for some of the more common sizes requested. For example, it might have a table with n entries, in which the first entry is a pointer to the head of a list of 4-KB holes, the second entry is a pointer to a list of 8-KB holes, the third entry a pointer to 12-KB holes, and so on. Holes of say, 21 KB, could either be put on the 20-KB

list or on a special list of odd-sized holes. With quick fit, finding a hole of the required size is extremely fast, but it has the same disadvantage as all schemes that sort by hole size, namely, when a process terminates or is swapped out, finding its neighbors to see if a merge is possible is expensive. If merging is not done, memory will quickly fragment into a large number of small holes into which no processes fit.

4.3 VIRTUAL MEMORY

Many years ago people were first confronted with programs that were too big to fit in the available memory. The solution usually adopted was to split the program into pieces, called **overlays**. Overlay 0 would start running first. When it was done, it would call another overlay. Some overlay systems were highly complex, allowing multiple overlays in memory at once. The overlays were kept on the disk and swapped in and out of memory by the operating system, dynamically, as needed.

Although the actual work of swapping overlays in and out was done by the system, the work of splitting the program into pieces had to be done by the programmer. Splitting up large programs into small, modular pieces was time consuming and boring. It did not take long before someone thought of a way to turn the whole job over to the computer.

The method that was devised (Fotheringham, 1961) has come to be known as **virtual memory**. The basic idea behind virtual memory is that the combined size of the program, data, and stack may exceed the amount of physical memory available for it. The operating system keeps those parts of the program currently in use in main memory, and the rest on the disk. For example, a 16-MB program can run on a 4-MB machine by carefully choosing which 4 MB to keep in memory at each instant, with pieces of the program being swapped between disk and memory as needed.

Virtual memory can also work in a multiprogramming system, with bits and pieces of many programs in memory at once. While a program is waiting for part of itself to be brought in, it is waiting for I/O and cannot run, so the CPU can be given to another process, the same way as in any other multiprogramming system.

4.3.1 Paging

Most virtual memory systems use a technique called **paging**, which we will now describe. On any computer, there exists a set of memory addresses that programs can produce. When a program uses an instruction like

```
MOV REG,1000
```

it does this to copy the contents of memory address 1000 to REG (or vice versa,

depending on the computer). Addresses can be generated using indexing, base registers, segment registers, and other ways.

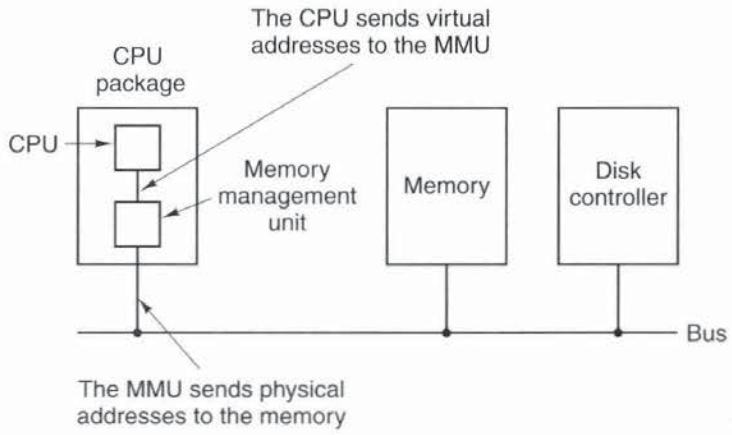


Figure 4-9. The position and function of the MMU. Here the MMU is shown as being a part of the CPU chip because it commonly is nowadays. However, logically it could be a separate chip and was in years gone by.

These program-generated addresses are called **virtual addresses** and form the **virtual address space**. On computers without virtual memory, the virtual address is put directly onto the memory bus and causes the physical memory word with the same address to be read or written. When virtual memory is used, the virtual addresses do not go directly to the memory bus. Instead, they go to an **MMU** (**Memory Management Unit**) that maps the virtual addresses onto the physical memory addresses as illustrated in Fig. 4-9.

A very simple example of how this mapping works is shown in Fig. 4-10. In this example, we have a computer that can generate 16-bit addresses, from 0 up to 64K. These are the virtual addresses. This computer, however, has only 32 KB of physical memory, so although 64-KB programs can be written, they cannot be loaded into memory in their entirety and run. A complete copy of a program's core image, up to 64 KB, must be present on the disk, however, so that pieces can be brought in as needed.

The virtual address space is divided up into units called **pages**. The corresponding units in the physical memory are called **page frames**. The pages and page frames are always the same size. In this example they are 4 KB, but page sizes from 512 bytes to 64 KB have been used in real systems. With 64 KB of virtual address space and 32 KB of physical memory, we get 16 virtual pages and 8 page frames. Transfers between RAM and disk are always in units of a page.

When the program tries to access address 0, for example, using the instruction

```
MOV REG,0
```

virtual address 0 is sent to the MMU. The MMU sees that this virtual address falls

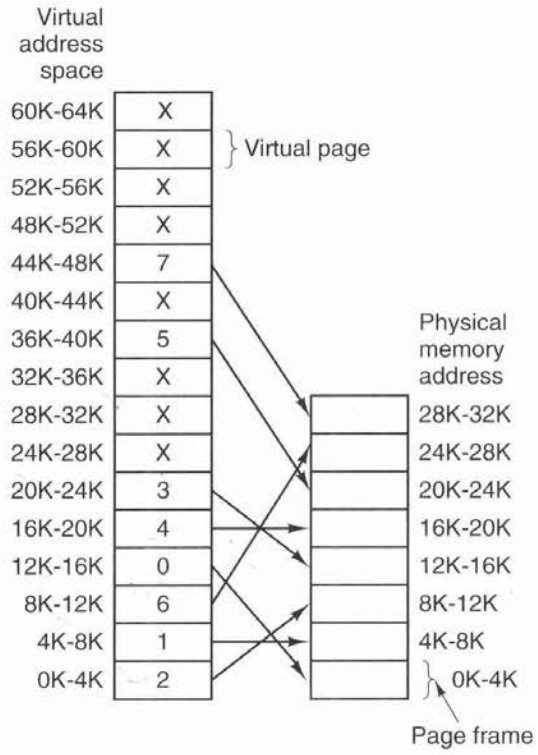


Figure 4-10. The relation between virtual addresses and physical memory addresses is given by the page table.

in page 0 (0 to 4095), which according to its mapping is page frame 2 (8192 to 12287). It thus transforms the address to 8192 and outputs address 8192 onto the bus. The memory knows nothing at all about the MMU and just sees a request for reading or writing address 8192, which it honors. Thus, the MMU has effectively mapped all virtual addresses between 0 and 4095 onto physical addresses 8192 to 12287.

Similarly, an instruction

```
MOV REG,8192
```

is effectively transformed into

```
MOV REG,24576
```

because virtual address 8192 is in virtual page 2 and this page is mapped onto physical page frame 6 (physical addresses 24576 to 28671). As a third example, virtual address 20500 is 20 bytes from the start of virtual page 5 (virtual addresses 20480 to 24575) and maps onto physical address 12288 + 20 = 12308.

By itself, this ability to map the 16 virtual pages onto any of the eight page frames by setting the MMU's map appropriately does not solve the problem that the virtual address space is larger than the physical memory. Since we have only

eight physical page frames, only eight of the virtual pages in Fig. 4-10 are mapped onto physical memory. The others, shown as a cross in the figure, are not mapped. In the actual hardware, a **Present/absent bit** keeps track of which pages are physically present in memory.

What happens if the program tries to use an unmapped page, for example, by using the instruction

```
MOV REG,32780
```

which is byte 12 within virtual page 8 (starting at 32768)? The MMU notices that the page is unmapped (indicated by a cross in the figure) and causes the CPU to trap to the operating system. This trap is called a **page fault**. The operating system picks a little-used page frame and writes its contents back to the disk. It then fetches the page just referenced into the page frame just freed, changes the map, and restarts the trapped instruction.

For example, if the operating system decided to evict page frame 1, it would load virtual page 8 at physical address 4K and make two changes to the MMU map. First, it would mark virtual page 1's entry as unmapped, to trap any future accesses to virtual addresses between 4K and 8K. Then it would replace the cross in virtual page 8's entry with a 1, so that when the trapped instruction is re-executed, it will map virtual address 32780 onto physical address 4108.

Now let us look inside the MMU to see how it works and why we have chosen to use a page size that is a power of 2. In Fig. 4-11 we see an example of a virtual address, 8196 (0010000000000100 in binary), being mapped using the MMU map of Fig. 4-10. The incoming 16-bit virtual address is split into a 4-bit page number and a 12-bit offset. With 4 bits for the page number, we can have 16 pages, and with 12 bits for the offset, we can address all 4096 bytes within a page.

The page number is used as an index into the **page table**, yielding the number of the page frame corresponding to that virtual page. If the *Present/absent* bit is 0, a trap to the operating system is caused. If the bit is 1, the page frame number found in the page table is copied to the high-order 3 bits of the output register, along with the 12-bit offset, which is copied unmodified from the incoming virtual address. Together they form a 15-bit physical address. The output register is then put onto the memory bus as the physical memory address.

4.3.2 Page Tables

In the simplest case, the mapping of virtual addresses onto physical addresses is as we have just described it. The virtual address is split into a virtual page number (high-order bits) and an offset (low-order bits). For example, with a 16-bit address and a 4-KB page size, the upper 4 bits could specify one of the 16 virtual pages and the lower 12 bits would then specify the byte offset (0 to 4095) within the selected page. However a split with 3 or 5 or some other number of bits for the page is also possible. Different splits imply different page sizes.

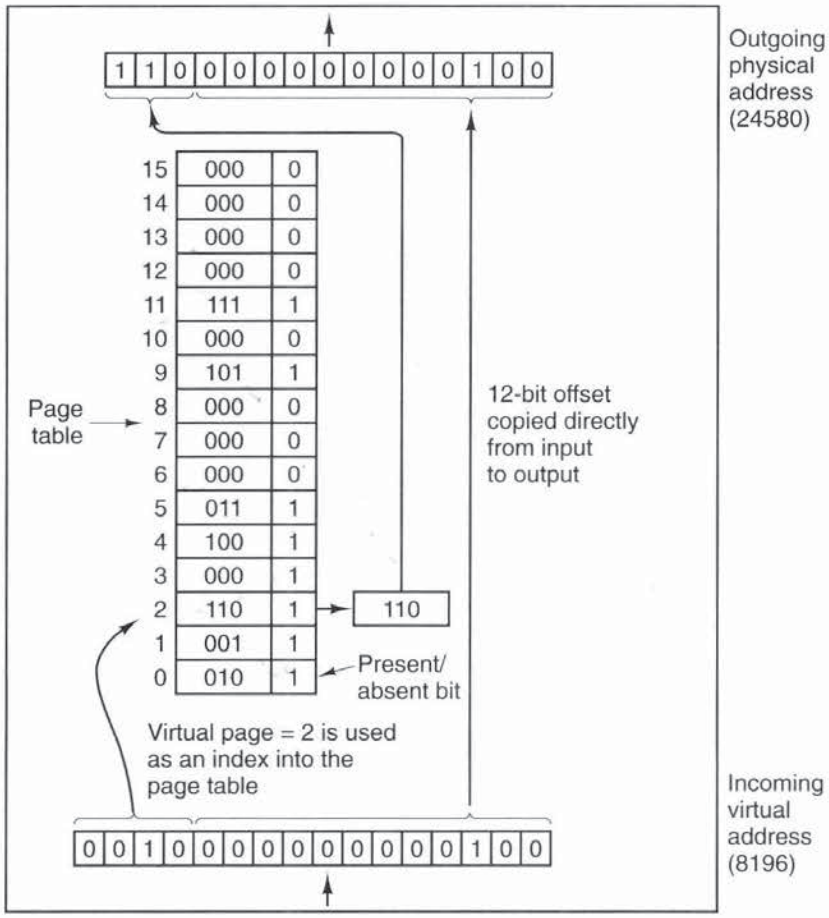


Figure 4-11. The internal operation of the MMU with 16 4-KB pages.

The virtual page number is used as an index into the page table to find the entry for that virtual page. From the page table entry, the page frame number (if any) is found. The page frame number is attached to the high-order end of the offset, replacing the virtual page number, to form a physical address that can be sent to the memory.

The purpose of the page table is to map virtual pages onto page frames. Mathematically speaking, the page table is a function, with the virtual page number as argument and the physical frame number as result. Using the result of this function, the virtual page field in a virtual address can be replaced by a page frame field, thus forming a physical memory address.

Despite this simple description, two major issues must be faced:

1. The page table can be extremely large.
2. The mapping must be fast.

The first point follows from the fact that modern computers use virtual addresses of at least 32 bits. With, say, a 4-KB page size, a 32-bit address space has 1 million pages, and a 64-bit address space has more than you want to contemplate. With 1 million pages in the virtual address space, the page table must have 1 million entries. And remember that each process needs its own page table (because it has its own virtual address space).

The second point is a consequence of the fact that the virtual-to-physical mapping must be done on every memory reference. A typical instruction has an instruction word, and often a memory operand as well. Consequently, it is necessary to make 1, 2, or sometimes more page table references per instruction. If an instruction takes, say, 4 nsec, the page table lookup must be done in under 1 nsec to avoid becoming a major bottleneck.

The need for large, fast page mapping is a significant constraint on the way computers are built. Although the problem is most serious with top-of-the-line machines, it is also an issue at the low end as well, where cost and the price/performance ratio are critical. In this section and the following ones, we will look at page table design in detail and show a number of hardware solutions that have been used in actual computers.

The simplest design (at least conceptually) is to have a single page table consisting of an array of fast hardware registers, with one entry for each virtual page, indexed by virtual page number, as shown in Fig. 4-11. When a process is started up, the operating system loads the registers with the process' page table, taken from a copy kept in main memory. During process execution, no more memory references are needed for the page table. The advantages of this method are that it is straightforward and requires no memory references during mapping. A disadvantage is that it is potentially expensive (if the page table is large). Having to load the full page table at every context switch hurts performance.

At the other extreme, the page table can be entirely in main memory. All the hardware needs then is a single register that points to the start of the page table. This design allows the memory map to be changed at a context switch by reloading one register. Of course, it has the disadvantage of requiring one or more memory references to read page table entries during the execution of each instruction. For this reason, this approach is rarely used in its most pure form, but below we will study some variations that have much better performance.

Multilevel Page Tables

To get around the problem of having to store huge page tables in memory all the time, many computers use a multilevel page table. A simple example is shown in Fig. 4-12. In Fig. 4-12(a) we have a 32-bit virtual address that is partitioned into a 10-bit *PT1* field, a 10-bit *PT2* field, and a 12-bit *Offset* field. Since offsets are 12 bits, pages are 4 KB, and there are a total of 2^{20} of them.

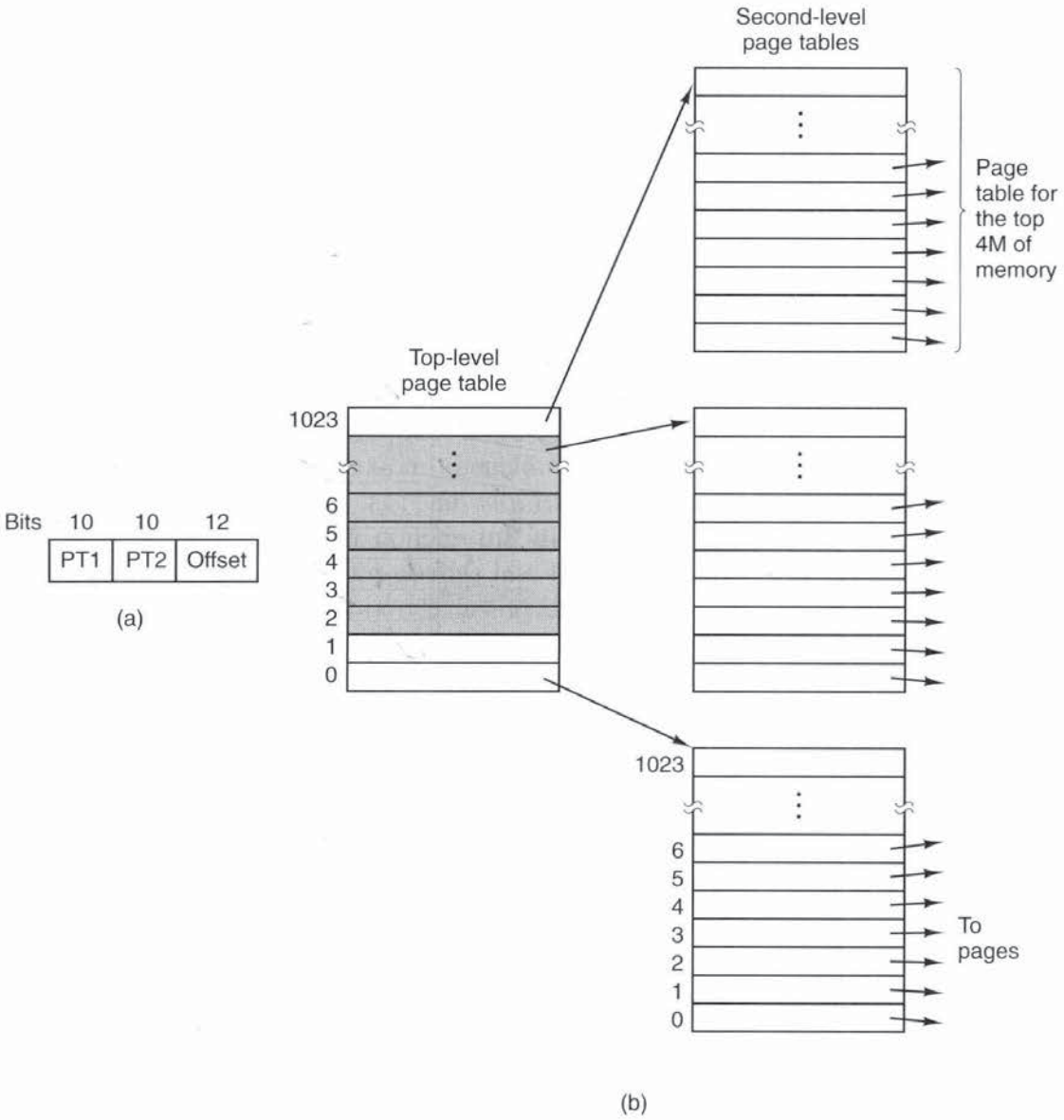


Figure 4-12. (a) A 32-bit address with two page table fields. (b) Two-level page tables.

The secret to the multilevel page table method is to avoid keeping all the page tables in memory all the time. In particular, those that are not needed should not be kept around. Suppose, for example, that a process needs 12 megabytes, the bottom 4 megabytes of memory for program text, the next 4 megabytes for data, and the top 4 megabytes for the stack. In between the top of the data and the bottom of the stack is a gigantic hole that is not used.

In Fig. 4-12(b) we see how the two-level page table works in this example. On the left we have the top-level page table, with 1024 entries, corresponding to

the 10-bit *PT1* field. When a virtual address is presented to the MMU, it first extracts the *PT1* field and uses this value as an index into the top-level page table. Each of these 1024 entries represents 4M because the entire 4-gigabyte (i.e., 32-bit) virtual address space has been chopped into chunks of 1024 bytes.

The entry located by indexing into the top-level page table yields the address or the page frame number of a second-level page table. Entry 0 of the top-level page table points to the page table for the program text, entry 1 points to the page table for the data, and entry 1023 points to the page table for the stack. The other (shaded) entries are not used. The *PT2* field is now used as an index into the selected second-level page table to find the page frame number for the page itself.

As an example, consider the 32-bit virtual address 0x00403004 (4,206,596 decimal), which is 12,292 bytes into the data. This virtual address corresponds to $PT1 = 1$, $PT2 = 2$, and $Offset = 4$. The MMU first uses *PT1* to index into the top-level page table and obtain entry 1, which corresponds to addresses 4M to 8M. It then uses *PT2* to index into the second-level page table just found and extract entry 3, which corresponds to addresses 12288 to 16383 within its 4M chunk (i.e., absolute addresses 4,206,592 to 4,210,687). This entry contains the page frame number of the page containing virtual address 0x00403004. If that page is not in memory, the *Present/absent* bit in the page table entry will be zero, causing a page fault. If the page is in memory, the page frame number taken from the second-level page table is combined with the offset (4) to construct a physical address. This address is put on the bus and sent to memory.

The interesting thing to note about Fig. 4-12 is that although the address space contains over a million pages, only four page tables are actually needed: the top-level table, and the second-level tables for 0 to 4M, 4M to 8M, and the top 4M. The *Present/absent* bits in 1021 entries of the top-level page table are set to 0, forcing a page fault if they are ever accessed. Should this occur, the operating system will notice that the process is trying to reference memory that it is not supposed to and will take appropriate action, such as sending it a signal or killing it. In this example we have chosen round numbers for the various sizes and have picked *PT1* equal to *PT2* but in actual practice other values are also possible, of course.

The two-level page table system of Fig. 4-12 can be expanded to three, four, or more levels. Additional levels give more flexibility, but it is doubtful that the additional complexity is worth it beyond three levels.

Structure of a Page Table Entry

Let us now turn from the structure of the page tables in the large, to the details of a single page table entry. The exact layout of an entry is highly machine dependent, but the kind of information present is roughly the same from machine to machine. In Fig. 4-13 we give a sample page table entry. The size varies from computer to computer, but 32 bits is a common size. The most important field is

the *Page frame number*. After all, the goal of the page mapping is to locate this value. Next to it we have the *Present/absent* bit. If this bit is 1, the entry is valid and can be used. If it is 0, the virtual page to which the entry belongs is not currently in memory. Accessing a page table entry with this bit set to 0 causes a page fault.

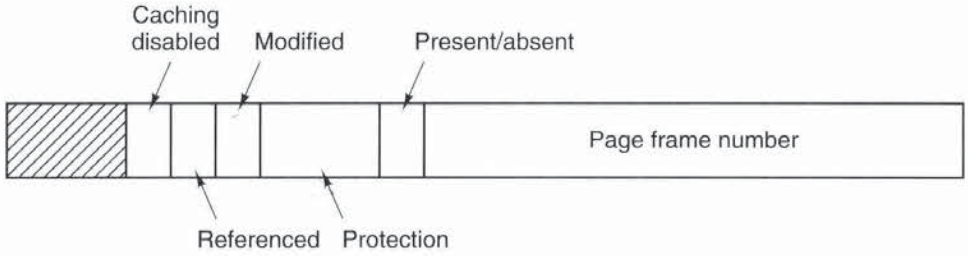


Figure 4-13. A typical page table entry.

The *Protection* bits tell what kinds of access are permitted. In the simplest form, this field contains 1 bit, with 0 for read/write and 1 for read only. A more sophisticated arrangement is having 3 bits, one bit each for enabling reading, writing, and executing the page.

The *Modified* and *Referenced* bits keep track of page usage. When a page is written to, the hardware automatically sets the *Modified* bit. This bit is of value when the operating system decides to reclaim a page frame. If the page in it has been modified (i.e., is "dirty"), it must be written back to the disk. If it has not been modified (i.e., is "clean"), it can just be abandoned, since the disk copy is still valid. The bit is sometimes called the **dirty bit**, since it reflects the page's state.

The *Referenced* bit is set whenever a page is referenced, either for reading or writing. Its value is to help the operating system choose a page to evict when a page fault occurs. Pages that are not being used are better candidates than pages that are, and this bit plays an important role in several of the page replacement algorithms that we will study later in this chapter.

Finally, the last bit allows caching to be disabled for the page. This feature is important for pages that map onto device registers rather than memory. If the operating system is sitting in a tight loop waiting for some I/O device to respond to a command it was just given, it is essential that the hardware keep fetching the word from the device, and not use an old cached copy. With this bit, caching can be turned off. Machines that have a separate I/O space and do not use memory mapped I/O do not need this bit.

Note that the disk address used to hold the page when it is not in memory is not part of the page table. The reason is simple. The page table holds only that information the hardware needs to translate a virtual address to a physical address. Information the operating system needs to handle page faults is kept in software tables inside the operating system. The hardware does not need it.

4.3.3 TLBs—Translation Lookaside Buffers

In most paging schemes, the page tables are kept in memory, due to their large size. Potentially, this design has an enormous impact on performance. Consider, for example, an instruction that copies one register to another. In the absence of paging, this instruction makes only one memory reference, to fetch the instruction. With paging, additional memory references will be needed to access the page table. Since execution speed is generally limited by the rate the CPU can get instructions and data out of the memory, having to make two page table references per memory reference reduces performance by 2/3. Under these conditions, no one would use it.

Computer designers have known about this problem for years and have come up with a solution. Their solution is based on the observation that most programs tend to make a large number of references to a small number of pages, and not the other way around. Thus only a small fraction of the page table entries are heavily read; the rest are barely used at all.

The solution that has been devised is to equip computers with a small hardware device for mapping virtual addresses to physical addresses without going through the page table. The device, called a **TLB** (**Translation Lookaside Buffer**) or sometimes an **associative memory**, is illustrated in Fig. 4-14. It is usually inside the MMU and consists of a small number of entries, eight in this example, but rarely more than 64. Each entry contains information about one page, including the virtual page number, a bit that is set when the page is modified, the protection code (read/write/execute permissions), and the physical page frame in which the page is located. These fields have a one-to-one correspondence with the fields in the page table. Another bit indicates whether the entry is valid (i.e., in use) or not.

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figure 4-14. A TLB to speed up paging.

An example that might generate the TLB of Fig. 4-14 is a process in a loop that spans virtual pages 19, 20, and 21, so these TLB entries have protection codes

for reading and executing. The main data currently being used (say, an array being processed) are on pages 129 and 130. Page 140 contains the indices used in the array calculations. Finally, the stack is on pages 860 and 861.

Let us now see how the TLB functions. When a virtual address is presented to the MMU for translation, the hardware first checks to see if its virtual page number is present in the TLB by comparing it to all the entries simultaneously (i.e., in parallel). If a valid match is found and the access does not violate the protection bits, the page frame is taken directly from the TLB, without going to the page table. If the virtual page number is present in the TLB but the instruction is trying to write on a read-only page, a protection fault is generated, the same way as it would be from the page table itself.

The interesting case is what happens when the virtual page number is not in the TLB. The MMU detects the miss and does an ordinary page table lookup. It then evicts one of the entries from the TLB and replaces it with the page table entry just looked up. Thus if that page is used again soon, the second time it will result in a hit rather than a miss. When an entry is purged from the TLB, the modified bit is copied back into the page table entry in memory. The other values are already there. When the TLB is loaded from the page table, all the fields are taken from memory.

Software TLB Management

Up until now, we have assumed that every machine with paged virtual memory has page tables recognized by the hardware, plus a TLB. In this design, TLB management and handling TLB faults are done entirely by the MMU hardware. Traps to the operating system occur only when a page is not in memory.

In the past, this assumption was true. However, many modern RISC machines, including the SPARC, MIPS, Alpha, and HP PA, do nearly all of this page management in software. On these machines, the TLB entries are explicitly loaded by the operating system. When a TLB miss occurs, instead of the MMU just going to the page tables to find and fetch the needed page reference, it just generates a TLB fault and tosses the problem into the lap of the operating system. The system must find the page, remove an entry from the TLB, enter the new one, and restart the instruction that faulted. And, of course, all of this must be done in a handful of instructions because TLB misses occur much more frequently than page faults.

Surprisingly enough, if the TLB is reasonably large (say, 64 entries) to reduce the miss rate, software management of the TLB turns out to be acceptably efficient. The main gain here is a much simpler MMU, which frees up a considerable amount of area on the CPU chip for caches and other features that can improve performance. Software TLB management is discussed by Uhlig et al. (1994).

Various strategies have been developed to improve performance on machines that do TLB management in software. One approach attacks both reducing TLB

misses and reducing the cost of a TLB miss when it does occur (Bala et al., 1994). To reduce TLB misses, sometimes the operating system can use its intuition to figure out which pages are likely to be used next and to preload entries for them in the TLB. For example, when a client process sends a message to a server process on the same machine, it is very likely that the server will have to run soon. Knowing this, while processing the trap to do the send, the system can also check to see where the server's code, data, and stack pages are and map them in before they can cause TLB faults.

The normal way to process a TLB miss, whether in hardware or in software, is to go to the page table and perform the indexing operations to locate the page referenced. The problem with doing this search in software is that the pages holding the page table may not be in the TLB, which will cause additional TLB faults during the processing. These faults can be reduced by maintaining a large (e.g., 4-KB) software cache of TLB entries in a fixed location whose page is always kept in the TLB. By first checking the software cache, the operating system can substantially reduce TLB misses.

4.3.4 Inverted Page Tables

Traditional page tables of the type described so far require one entry per virtual page, since they are indexed by virtual page number. If the address space consists of 2^{32} bytes, with 4096 bytes per page, then over 1 million page table entries are needed. As a bare minimum, the page table will have to be at least 4 megabytes. On larger systems, this size is probably doable.

However, as 64-bit computers become more common, the situation changes drastically. If the address space is now 2^{64} bytes, with 4-KB pages, we need a page table with 2^{52} entries. If each entry is 8 bytes, the table is over 30 million gigabytes. Tying up 30 million gigabytes just for the page table is not doable, not now and not for years to come, if ever. Consequently, a different solution is needed for 64-bit paged virtual address spaces.

One such solution is the **inverted page table**. In this design, there is one entry per page frame in real memory, rather than one entry per page of virtual address space. For example, with 64-bit virtual addresses, a 4-KB page, and 256 MB of RAM, an inverted page table only requires 65,536 entries. The entry keeps track of which (process, virtual page) is located in the page frame.

Although inverted page tables save vast amounts of space, at least when the virtual address space is much larger than the physical memory, they have a serious downside: virtual-to-physical translation becomes much harder. When process n references virtual page p, the hardware can no longer find the physical page by using p as an index into the page table. Instead, it must search the entire inverted page table for an entry (n, p). Furthermore, this search must be done on every memory reference, not just on page faults. Searching a 64K table on every memory reference is not the way to make your machine blindingly fast.

The way out of this dilemma is to use the TLB. If the TLB can hold all of the heavily used pages, translation can happen just as fast as with regular page tables. On a TLB miss, however, the inverted page table has to be searched in software. One feasible way to accomplish this search is to have a hash table hashed on the virtual address. All the virtual pages currently in memory that have the same hash value are chained together, as shown in Fig. 4-15. If the hash table has as many slots as the machine has physical pages, the average chain will be only one entry long, greatly speeding up the mapping. Once the page frame number has been found, the new (virtual, physical) pair is entered into the TLB.

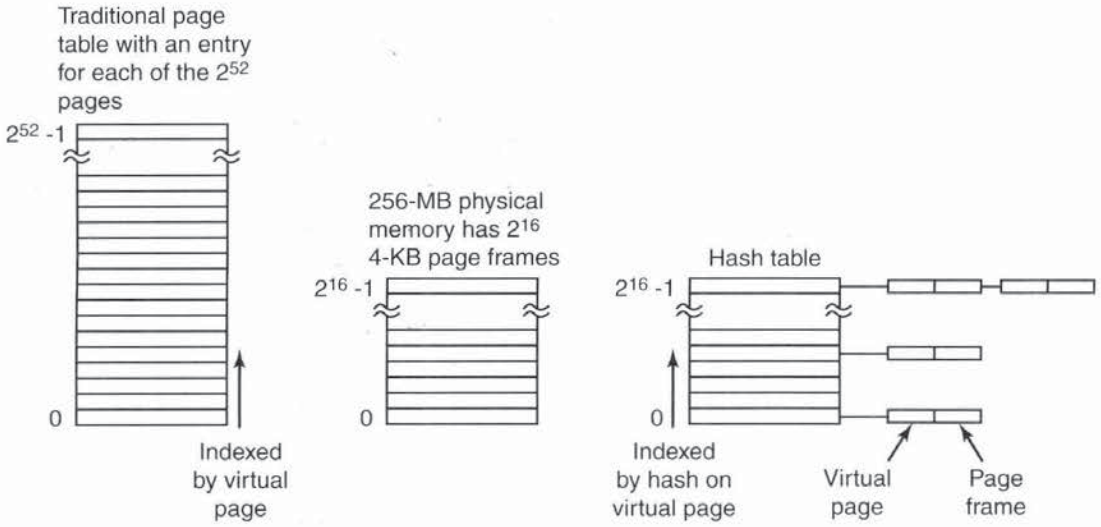


Figure 4-15. Comparison of a traditional page table with an inverted page table.

Inverted page tables are currently used on some IBM and Hewlett-Packard workstations and will become more common as 64-bit machines become widespread. Other approaches to handling large virtual memories can be found in (Huck and Hays, 1993; Talluri and Hill, 1994; and Talluri et al., 1995).

4.4 PAGE REPLACEMENT ALGORITHMS

When a page fault occurs, the operating system has to choose a page to remove from memory to make room for the page that has to be brought in. If the page to be removed has been modified while in memory, it must be rewritten to the disk to bring the disk copy up to date. If, however, the page has not been changed (e.g., it contains program text), the disk copy is already up to date, so no rewrite is needed. The page to be read in just overwrites the page being evicted.

While it would be possible to pick a random page to evict at each page fault, system performance is much better if a page that is not heavily used is chosen. If

a heavily used page is removed, it will probably have to be brought back in quickly, resulting in extra overhead. Much work has been done on the subject of page replacement algorithms, both theoretical and experimental. Below we will describe some of the most important algorithms.

It is worth noting that the problem of "page replacement" occurs in other areas of computer design as well. For example, most computers have one or more memory caches consisting of recently used 32-byte or 64-byte memory blocks. When the cache is full, some block has to be chosen for removal. This problem is precisely the same as page replacement except on a shorter time scale (it has to be done in a few nanoseconds, not milliseconds as with page replacement). The reason for the shorter time scale is that cache block misses are satisfied from main memory, which has no seek time and no rotational latency.

A second example is in a Web server. The server can keep a certain number of heavily used Web pages in its memory cache. However, when the memory cache is full and a new page is referenced, a decision has to be made which Web page to evict. The considerations are similar to pages of virtual memory, except for the fact that the Web pages are never modified in the cache, so there is always a fresh copy on disk. In a virtual memory system, pages in main memory may be either clean or dirty.

4.4.1 The Optimal Page Replacement Algorithm

The best possible page replacement algorithm is easy to describe but impossible to implement. It goes like this. At the moment that a page fault occurs, some set of pages is in memory. One of these pages will be referenced on the very next instruction (the page containing that instruction). Other pages may not be referenced until 10, 100, or perhaps 1000 instructions later. Each page can be labeled with the number of instructions that will be executed before that page is first referenced.

The optimal page algorithm simply says that the page with the highest label should be removed. If one page will not be used for 8 million instructions and another page will not be used for 6 million instructions, removing the former pushes the page fault that will fetch it back as far into the future as possible. Computers, like people, try to put off unpleasant events for as long as they can.

The only problem with this algorithm is that it is unrealizable. At the time of the page fault, the operating system has no way of knowing when each of the pages will be referenced next. (We saw a similar situation earlier with the shortest job first scheduling algorithm—how can the system tell which job is shortest?) Still, by running a program on a simulator and keeping track of all page references, it is possible to implement optimal page replacement on the *second* run by using the page reference information collected during the *first* run.

In this way it is possible to compare the performance of realizable algorithms with the best possible one. If an operating system achieves a performance of, say,

only 1 percent worse than the optimal algorithm, effort spent in looking for a better algorithm will yield at most a 1 percent improvement.

To avoid any possible confusion, it should be made clear that this log of page references refers only to the one program just measured and then with only one specific input. The page replacement algorithm derived from it is thus specific to that one program and input data. Although this method is useful for evaluating page replacement algorithms, it is of no use in practical systems. Below we will study algorithms that *are* useful on real systems.

4.4.2 The Not Recently Used Page Replacement Algorithm

In order to allow the operating system to collect useful statistics about which pages are being used and which ones are not, most computers with virtual memory have two status bits associated with each page. *R* is set whenever the page is referenced (read or written). *M* is set when the page is written to (i.e., modified). The bits are contained in each page table entry, as shown in Fig. 4-13. It is important to realize that these bits must be updated on every memory reference, so it is essential that they be set by the hardware. Once a bit has been set to 1, it stays 1 until the operating system resets it to 0 in software.

If the hardware does not have these bits, they can be simulated as follows. When a process is started up, all of its page table entries are marked as not in memory. As soon as any page is referenced, a page fault will occur. The operating system then sets the *R* bit (in its internal tables), changes the page table entry to point to the correct page, with mode READ ONLY, and restarts the instruction. If the page is subsequently written on, another page fault will occur, allowing the operating system to set the *M* bit and change the page's mode to READ/WRITE.

The *R* and *M* bits can be used to build a simple paging algorithm as follows. When a process is started up, both page bits for all its pages are set to 0 by the operating system. Periodically (e.g., on each clock interrupt), the *R* bit is cleared, to distinguish pages that have not been referenced recently from those that have been.

When a page fault occurs, the operating system inspects all the pages and divides them into four categories based on the current values of their *R* and *M* bits:

Class 0: not referenced, not modified.
Class 1: not referenced, modified.
Class 2: referenced, not modified.
Class 3: referenced, modified.

Although class 1 pages seem, at first glance, impossible, they occur when a class 3 page has its *R* bit cleared by a clock interrupt. Clock interrupts do not clear the *M* bit because this information is needed to know whether the page has to be rewritten to disk or not. Clearing *R* but not *M* leads to a class 1 page.

The **NRU** (**Not Recently Used**) algorithm removes a page at random from the lowest numbered nonempty class. Implicit in this algorithm is that it is better to remove a modified page that has not been referenced in at least one clock tick (typically 20 msec) than a clean page that is in heavy use. The main attraction of NRU is that it is easy to understand, moderately efficient to implement, and gives a performance that, while certainly not optimal, may be adequate.

4.4.3 The First-In, First-Out (FIFO) Page Replacement Algorithm

Another low-overhead paging algorithm is the **FIFO** (**First-In, First-Out**) algorithm. To illustrate how this works, consider a supermarket that has enough shelves to display exactly k different products. One day, some company introduces a new convenience food—instant, freeze-dried, organic yogurt that can be reconstituted in a microwave oven. It is an immediate success, so our finite supermarket has to get rid of one old product in order to stock it.

One possibility is to find the product that the supermarket has been stocking the longest (i.e., something it began selling 120 years ago) and get rid of it on the grounds that no one is interested any more. In effect, the supermarket maintains a linked list of all the products it currently sells in the order they were introduced. The new one goes on the back of the list; the one at the front of the list is dropped.

As a page replacement algorithm, the same idea is applicable. The operating system maintains a list of all pages currently in memory, with the page at the head of the list the oldest one and the page at the tail the most recent arrival. On a page fault, the page at the head is removed and the new page added to the tail of the list. When applied to stores, FIFO might remove mustache wax, but it might also remove flour, salt, or butter. When applied to computers the same problem arises. For this reason, FIFO in its pure form is rarely used.

4.4.4 The Second Chance Page Replacement Algorithm

A simple modification to FIFO that avoids the problem of throwing out a heavily used page is to inspect the R bit of the oldest page. If it is 0, the page is both old and unused, so it is replaced immediately. If the R bit is 1, the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated as though it had just arrived in memory. Then the search continues.

The operation of this algorithm, called **second chance**, is shown in Fig. 4-16. In Fig. 4-16(a) we see pages A through H kept on a linked list and sorted by the time they arrived in memory.

Suppose that a page fault occurs at time 20. The oldest page is A, which arrived at time 0, when the process started. If A has the R bit cleared, it is evicted from memory, either by being written to the disk (if it is dirty), or just abandoned (if it is clean). On the other hand, if the R bit is set, A is put onto the end of the

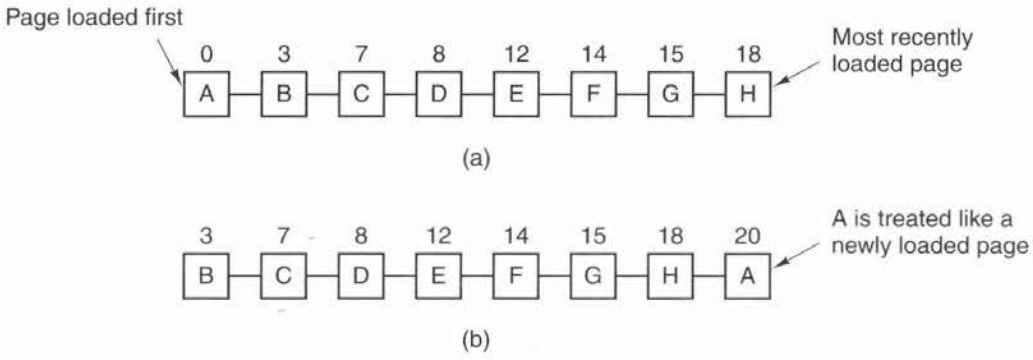


Figure 4-16. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and *A* has its *R* bit set. The numbers above the pages are their loading times.

list and its "load time" is reset to the current time (20). The *R* bit is also cleared. The search for a suitable page continues with *B*.

What second chance is doing is looking for an old page that has not been referenced in the previous clock interval. If all the pages have been referenced, second chance degenerates into pure FIFO. Specifically, imagine that all the pages in Fig. 4-16(a) have their *R* bits set. One by one, the operating system moves the pages to the end of the list, clearing the *R* bit each time it appends a page to the end of the list. Eventually, it comes back to page *A*, which now has its *R* bit cleared. At this point *A* is evicted. Thus the algorithm always terminates.

4.4.5 The Clock Page Replacement Algorithm

Although second chance is a reasonable algorithm, it is unnecessarily inefficient because it is constantly moving pages around on its list. A better approach is to keep all the page frames on a circular list in the form of a clock, as shown in Fig. 4-17. A hand points to the oldest page.

When a page fault occurs, the page being pointed to by the hand is inspected. If its *R* bit is 0, the page is evicted, the new page is inserted into the clock in its place, and the hand is advanced one position. If *R* is 1, it is cleared and the hand is advanced to the next page. This process is repeated until a page is found with $R = 0$. Not surprisingly, this algorithm is called **clock**. It differs from second chance only in the implementation.

4.4.6 The Least Recently Used (LRU) Page Replacement Algorithm

A good approximation to the optimal algorithm is based on the observation that pages that have been heavily used in the last few instructions will probably be heavily used again in the next few. Conversely, pages that have not been used for ages will probably remain unused for a long time. This idea suggests a realizable

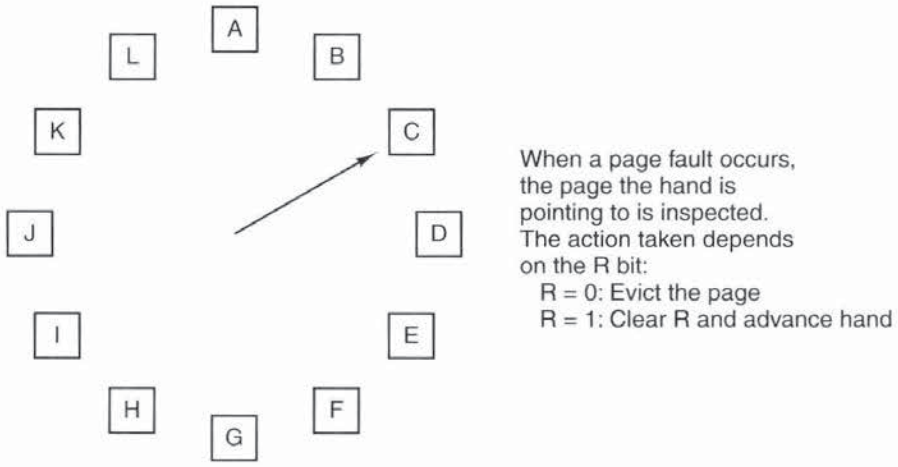


Figure 4-17. The clock page replacement algorithm.

algorithm: when a page fault occurs, throw out the page that has been unused for the longest time. This strategy is called **LRU (Least Recently Used)** paging.

Although LRU is theoretically realizable, it is not cheap. To fully implement LRU, it is necessary to maintain a linked list of all pages in memory, with the most recently used page at the front and the least recently used page at the rear. The difficulty is that the list must be updated on every memory reference. Finding a page in the list, deleting it, and then moving it to the front is a very time consuming operation, even in hardware (assuming that such hardware could be built).

However, there are other ways to implement LRU with special hardware. Let us consider the simplest way first. This method requires equipping the hardware with a 64-bit counter, C, that is automatically incremented after each instruction. Furthermore, each page table entry must also have a field large enough to contain the counter. After each memory reference, the current value of C is stored in the page table entry for the page just referenced. When a page fault occurs, the operating system examines all the counters in the page table to find the lowest one. That page is the least recently used.

Now let us look at a second hardware LRU algorithm. For a machine with n page frames, the LRU hardware can maintain a matrix of $n \times n$ bits, initially all zero. Whenever page frame k is referenced, the hardware first sets all the bits of row k to 1, then sets all the bits of column k to 0. At any instant, the row whose binary value is lowest is the least recently used, the row whose value is next lowest is next least recently used, and so forth. The workings of this algorithm are given in Fig. 4-18 for four page frames and page references in the order

0 1 2 3 2 1 0 3 2 3

After page 0 is referenced, we have the situation of Fig. 4-18(a). After page 1 is reference, we have the situation of Fig. 4-18(b), and so forth.

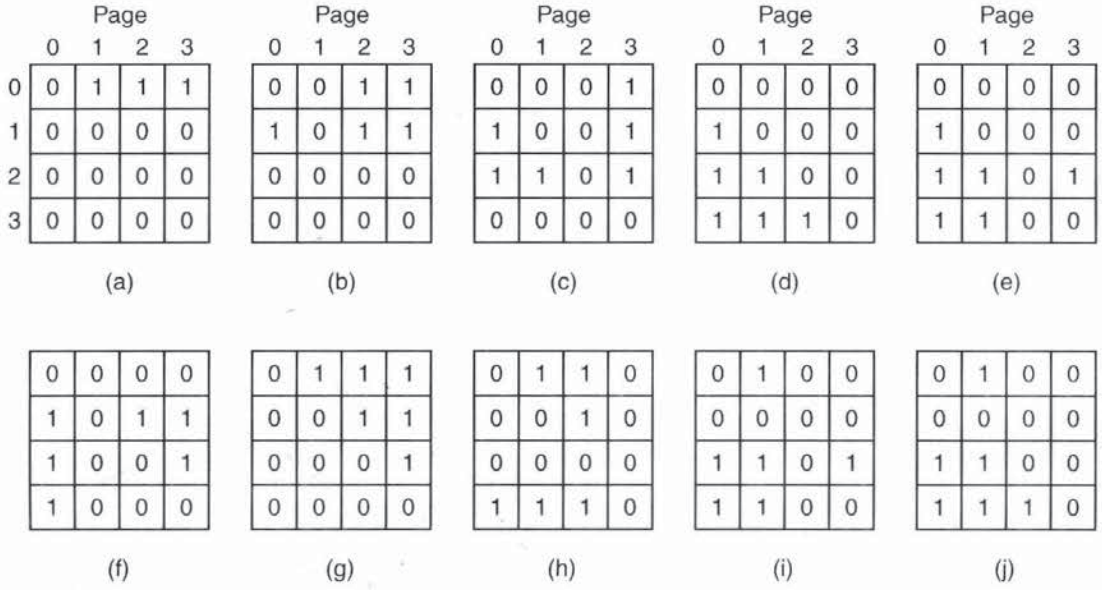


Figure 4-18. LRU using a matrix when pages are referenced in the order 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

4.4.7 Simulating LRU in Software

Although both of the previous LRU algorithms are realizable in principle, few, if any, machines have this hardware, so they are of little use to the operating system designer who is making a system for a machine that does not have this hardware. Instead, a solution that can be implemented in software is needed. One possibility is called the **NFU** (**Not Frequently Used**) algorithm. It requires a software counter associated with each page, initially zero. At each clock interrupt, the operating system scans all the pages in memory. For each page, the *R* bit, which is 0 or 1, is added to the counter. In effect, the counters are an attempt to keep track of how often each page has been referenced. When a page fault occurs, the page with the lowest counter is chosen for replacement.

The main problem with NFU is that it never forgets anything. For example, in a multipass compiler, pages that were heavily used during pass 1 may still have a high count well into later passes. In fact, if pass 1 happens to have the longest execution time of all the passes, the pages containing the code for subsequent passes may always have lower counts than the pass 1 pages. Consequently, the operating system will remove useful pages instead of pages no longer in use.

Fortunately, a small modification to NFU makes it able to simulate LRU quite well. The modification has two parts. First, the counters are each shifted right 1 bit before the *R* bit is added in. Second, the *R* bit is added to the leftmost, rather than the rightmost bit.

Figure 4-19 illustrates how the modified algorithm, known as **aging**, works. Suppose that after the first clock tick the *R* bits for pages 0 to 5 have the values 1,

0, 1, 0, 1, and 1, respectively (page 0 is 1, page 1 is 0, page 2 is 1, etc.). In other words, between tick 0 and tick 1, pages 0, 2, 4, and 5 were referenced, setting their *R* bits to 1, while the other ones remain 0. After the six corresponding counters have been shifted and the *R* bit inserted at the left, they have the values shown in Fig. 4-19(a). The four remaining columns show the six counters after the next four clock ticks.

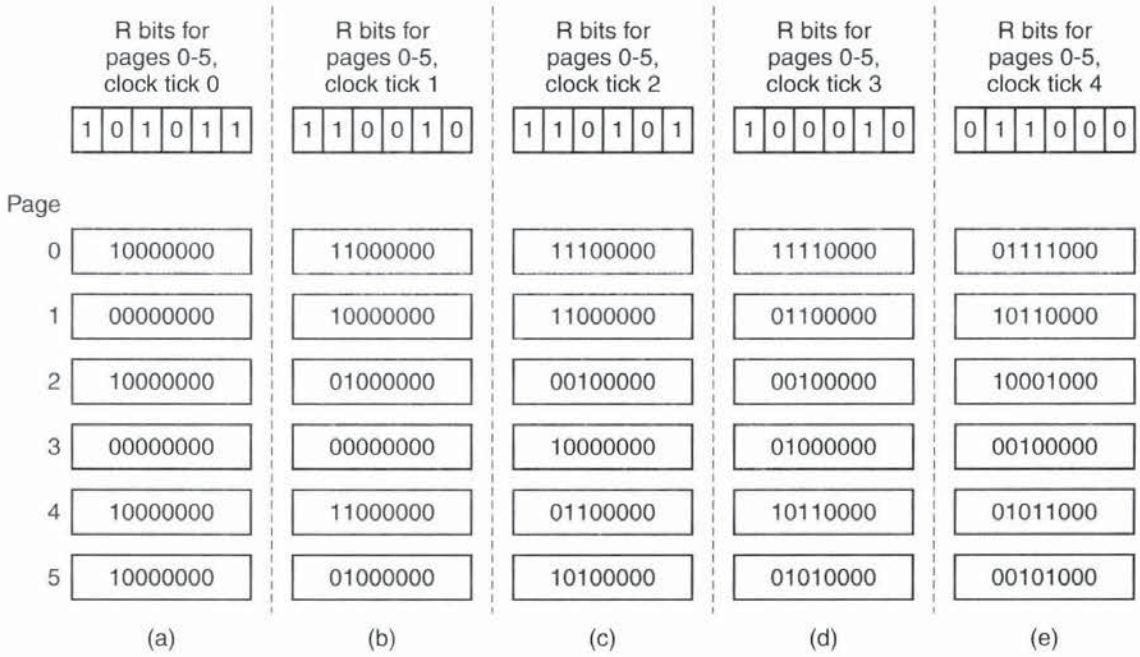


Figure 4-19. The aging algorithm simulates LRU in software. Shown are six pages for five clock ticks. The five clock ticks are represented by (a) to (e).

When a page fault occurs, the page whose counter is the lowest is removed. It is clear that a page that has not been referenced for, say, four clock ticks will have four leading zeros in its counter and thus will have a lower value than a counter that has not been referenced for three clock ticks.

This algorithm differs from LRU in two ways. Consider pages 3 and 5 in Fig. 4-19(e). Neither has been referenced for two clock ticks; both were referenced in the tick prior to that. According to LRU, if a page must be replaced, we should choose one of these two. The trouble is, we do not know which of these two was referenced last in the interval between tick 1 and tick 2. By recording only one bit per time interval, we have lost the ability to distinguish references early in the clock interval from those occurring later. All we can do is remove page 3, because page 5 was also referenced two ticks earlier and page 3 was not.

The second difference between LRU and aging is that in aging the counters have a finite number of bits, 8 bits in this example. Suppose that two pages each have a counter value of 0. All we can do is pick one of them at random. In

reality, it may well be that one of the pages was last referenced 9 ticks ago and the other was last referenced 1000 ticks ago. We have no way of seeing that. In practice, however, 8 bits is generally enough if a clock tick is around 20 msec. If a page has not been referenced in 160 msec, it probably is not that important.

4.4.8 The Working Set Page Replacement Algorithm

In the purest form of paging, processes are started up with none of their pages in memory. As soon as the CPU tries to fetch the first instruction, it gets a page fault, causing the operating system to bring in the page containing the first instruction. Other page faults for global variables and the stack usually follow quickly. After a while, the process has most of the pages it needs and settles down to run with relatively few page faults. This strategy is called **demand paging** because pages are loaded only on demand, not in advance.

Of course, it is easy enough to write a test program that systematically reads all the pages in a large address space, causing so many page faults that there is not enough memory to hold them all. Fortunately, most processes do not work this way. They exhibit a **locality of reference**, meaning that during any phase of execution, the process references only a relatively small fraction of its pages. Each pass of a multipass compiler, for example, references only a fraction of all the pages, and a different fraction at that.

The set of pages that a process is currently using is called its **working set** (Denning, 1968a; Denning, 1980). If the entire working set is in memory, the process will run without causing many faults until it moves into another execution phase (e.g., the next pass of the compiler). If the available memory is too small to hold the entire working set, the process will cause many page faults and run slowly since executing an instruction takes a few nanoseconds and reading in a page from the disk typically takes 10 milliseconds. At a rate of one or two instructions per 10 milliseconds, it will take ages to finish. A program causing page faults every few instructions is said to be **thrashing** (Denning, 1968b).

In a multiprogramming system, processes are frequently moved to disk (i.e., all their pages are removed from memory) to let other processes have a turn at the CPU. The question arises of what to do when a process is brought back in again. Technically, nothing need be done. The process will just cause page faults until its working set has been loaded. The problem is that having 20, 100, or even 1000 page faults every time a process is loaded is slow, and it also wastes considerable CPU time, since it takes the operating system a few milliseconds of CPU time to process a page fault.

Therefore, many paging systems try to keep track of each process' working set and make sure that it is in memory before letting the process run. This approach is called the **working set model** (Denning, 1970). It is designed to greatly reduce the page fault rate. Loading the pages *before* letting processes run is also called **prepaging.** Note that the working set changes over time.

It has been long known that most programs do not reference their address space uniformly, but that the references tend to cluster on a small number of pages. A memory reference may fetch an instruction, it may fetch data, or it may store data. At any instant of time, t, there exists a set consisting of all the pages used by the k most recent memory references. This set, $w(k, t)$, is the working set. Because the $k = 1$ most recent references must have used all the pages used by the $k > 1$ most recent references, and possibly others, $w(k, t)$ is a monotonically nondecreasing function of k. The limit of $w(k, t)$ as k becomes large is finite because a program cannot reference more pages than its address space contains, and few programs will use every single page. Figure 4-20 depicts the size of the working set as a function of k.

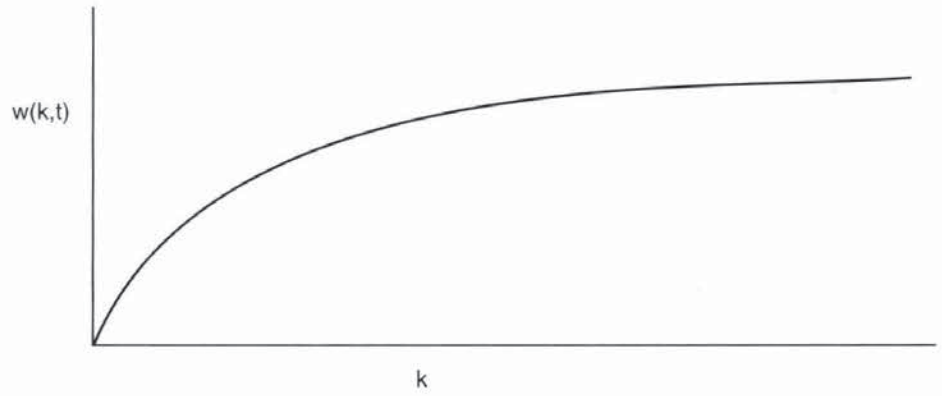


Figure 4-20. The working set is the set of pages used by the k most recent memory references. The function $w(k, t)$ is the size of the working set at time t.

The fact that most programs randomly access a small number of pages, but that this set changes slowly in time explains the initial rapid rise of the curve and then the slow rise for large k. For example, a program that is executing a loop occupying two pages using data on four pages, may reference all six pages every 1000 instructions, but the most recent reference to some other page may be a million instructions earlier, during the initialization phase. Due to this asymptotic behavior, the contents of the working set is not sensitive to the value of k chosen. To put it differently, there exists a wide range of k values for which the working set is unchanged. Because the working set varies slowly with time, it is possible to make a reasonable guess as to which pages will be needed when the program is restarted on the basis of its working set when it was last stopped. Prepaging consists of loading these pages before the process is allowed to run again.

To implement the working set model, it is necessary for the operating system to keep track of which pages are in the working set. Having this information also immediately leads to a possible page replacement algorithm: when a page fault occurs, find a page not in the working set and evict it. To implement such an algorithm, we need a precise way of determining which pages are in the working set and which are not at any given moment in time.

As we mentioned above, the working set is the set of pages used in the k most recent memory references (some authors use the k most recent page references, but the choice is arbitrary). To implement any working set algorithm, some value of k must be chosen in advance. Once some value has been selected, after every memory reference, the set of pages used by the previous k memory references is uniquely determined.

Of course, having an operational definition of the working set does not mean that there is an efficient way to monitor it in real time, during program execution. One could imagine a shift register of length k, with every memory reference shifting the register left one position and inserting the most recently referenced page number on the right. The set of all k page numbers in the shift register would be the working set. In theory, at a page fault, the contents of the shift register could be read out and sorted. Duplicate pages could then be removed. The result would be the working set. However, maintaining the shift register and processing it at a page fault would both be prohibitively expensive, so this technique is never used.

Instead, various approximations are used. One commonly used approximation is to drop the idea of counting back k memory references and use execution time instead. For example, instead of defining the working set as those pages used during the previous 10 million memory references, we can define it as the set of pages used during the past 100 msec of execution time. In practice, such a definition is just as good and much easier to use. Note that for each process, only its own execution time counts. Thus if a process starts running at time T and has had 40 msec of CPU time at real time $T + 100$ msec, for working set purposes, its time is 40 msec. The amount of CPU time a process has actually used has since it started is often called its **current virtual time**. With this approximation, the working set of a process is the set of pages it has referenced during the past τ seconds of virtual time.

Now let us look at a page replacement algorithm based on the working set. The basic idea is to find a page that is not in the working set and evict it. In Fig. 4-21 we see a portion of a page table for some machine. Because only pages that are in memory are considered as candidates for eviction, pages that are absent from memory are ignored by this algorithm. Each entry contains (at least) two items of information: the approximate time the page was last used and the R (Referenced) bit. The empty white rectangle symbolizes the other fields not needed for this algorithm, such as the page frame number, the protection bits, and the M (Modified) bit.

The algorithm works as follows. The hardware is assumed to set the R and M bits, as we have discussed before. Similarly, a periodic clock interrupt is assumed to cause software to run that clears the *Referenced* bit on every clock tick. On every page fault, the page table is scanned to look for a suitable page to evict.

As each entry is processed, the R bit is examined. If it is 1, the current virtual time is written into the *Time of last use* field in the page table, indicating that the page was in use at the time the fault occurred. Since the page has been referenced

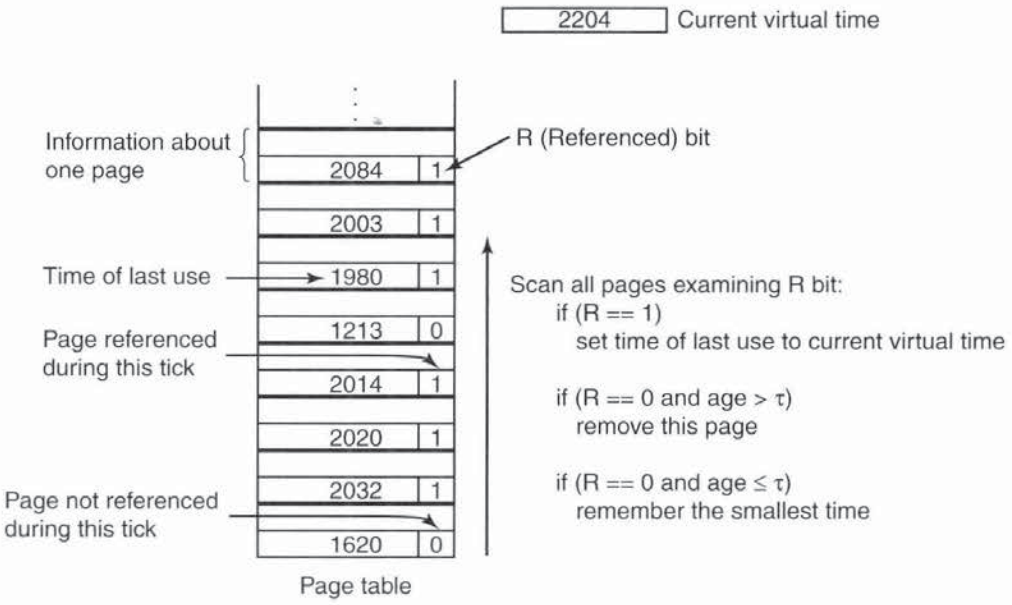


Figure 4-21. The working set algorithm.

during the current clock tick, it is clearly in the working set and is not a candidate for removal (τ is assumed to span multiple clock ticks).

If R is 0, the page has not been referenced during the current clock tick and may be a candidate for removal. To see whether or not it should be removed, its age, that is, the current virtual time minus its *Time of last use* is computed and compared to τ. If the age is greater than τ, the page is no longer in the working set. It is reclaimed and the new page loaded here. The scan continues updating the remaining entries, however.

However, if R is 0 but the age is less than or equal to τ, the page is still in the working set. The page is temporarily spared, but the page with the greatest age (smallest value of *Time of last use*) is noted. If the entire table is scanned without finding a candidate to evict, that means that all pages are in the working set. In that case, if one or more pages with $R = 0$ were found, the one with the greatest age is evicted. In the worst case, all pages have been referenced during the current clock tick (and thus all have $R = 1$), so one is chosen at random for removal, preferably a clean page, if one exists.

4.4.9 The WSClock Page Replacement Algorithm

The basic working set algorithm is cumbersome since the entire page table has to be scanned at each page fault until a suitable candidate is located. An improved algorithm, that is based on the clock algorithm but also uses the working set information is called **WSClock** (Carr and Hennessey, 1981). Due to its simplicity of implementation and good performance, it is widely used in practice.

The data structure needed is a circular list of page frames, as in the clock algorithm, and as shown in Fig. 4-22(a). Initially, this list is empty. When the first page is loaded, it is added to the list. As more pages are added, they go into the list to form a ring. Each entry contains the *Time of last use* field from the basic working set algorithm, as well as the *R* bit (shown) and the *M* bit (not shown).

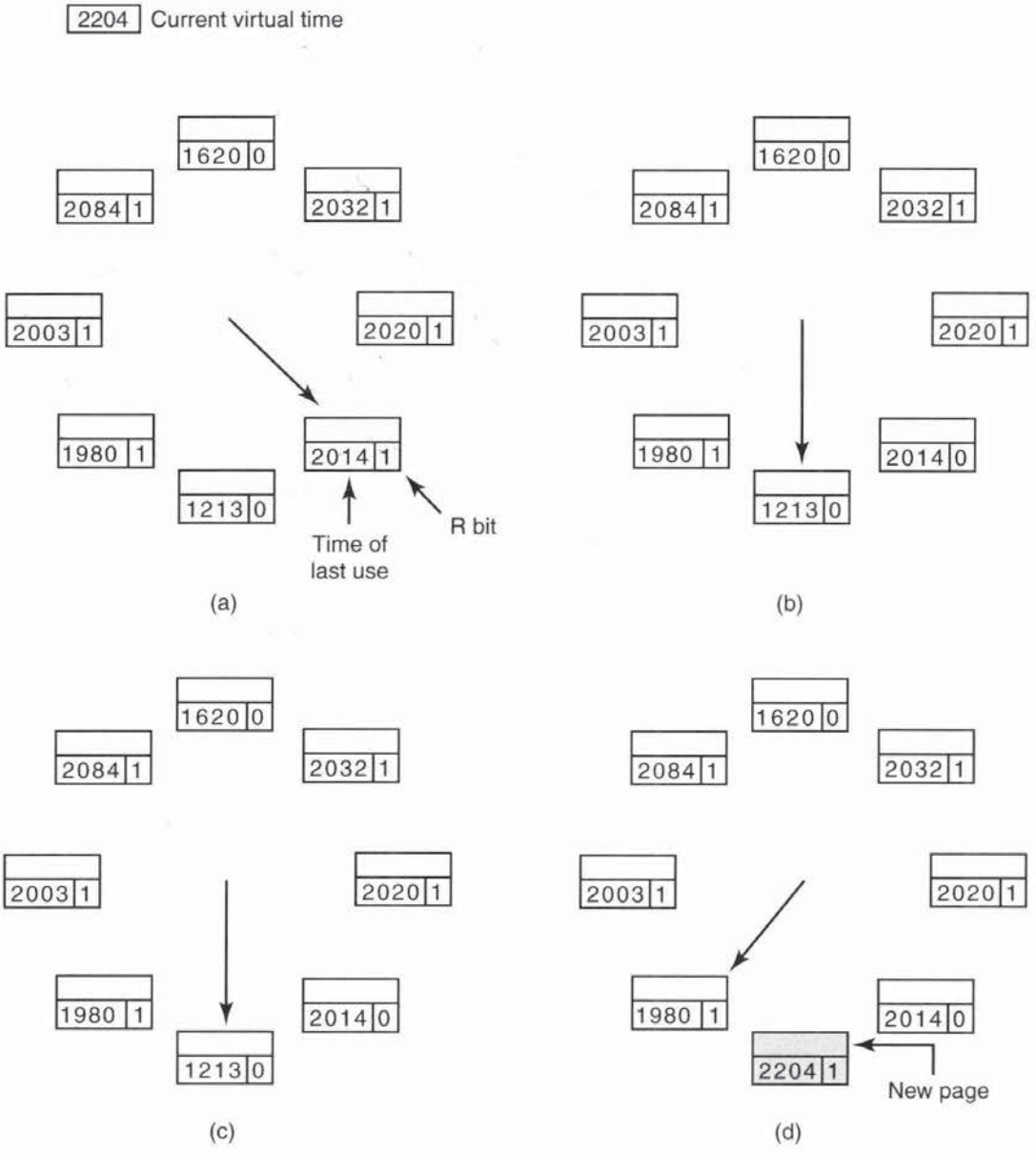


Figure 4-22. Operation of the WSClock algorithm. (a) and (b) give an example of what happens when $R = 1$. (c) and (d) give an example of $R = 0$.

As with the clock algorithm, at each page fault the page pointed to by the hand is examined first. If the *R* bit is set to 1, the page has been used during the

current tick so it is not an ideal candidate to remove. The R bit is then set to 0, the hand advanced to the next page, and the algorithm repeated for that page. The state after this sequence of events is shown in Fig. 4-22(b).

Now consider what happens if the page pointed to has $R = 0$, as shown in Fig. 4-22(c). If the age is greater than τ and the page is clean, it is not in the working set and a valid copy exists on the disk. The page frame is simply claimed and the new page put there, as shown in Fig. 4-22(d). On the other hand, if the page is dirty, it cannot be claimed immediately since no valid copy is present on disk. To avoid a process switch, the write to disk is scheduled, but the hand is advanced and the algorithm continues with the next page. After all, there might be an old, clean page further down the line that can be used immediately.

In principle, all pages might be scheduled for disk I/O on one cycle around the clock. To reduce disk traffic, a limit might be set, allowing a maximum of n pages to be written back. Once this limit has been reached, no new writes are scheduled.

What happens if the hand comes all the way around to its starting point? There are two cases to distinguish:

1. At least one write has been scheduled.
2. No writes have been scheduled.

In the former case, the hand just keeps moving, looking for a clean page. Since one or more writes have been scheduled, eventually some write will complete and its page will be marked as clean. The first clean page encountered is evicted. This page is not necessarily the first write scheduled because the disk driver may reorder writes in order to optimize disk performance.

In the latter case, all pages are in the working set, otherwise at least one write would have been scheduled. Lacking additional information, the simplest thing to do is claim any clean page and use it. The location of a clean page could be kept track of during the sweep. If no clean pages exist, then the current page is chosen and written back to disk.

4.4.10 Summary of Page Replacement Algorithms

We have now looked at a variety of page replacement algorithms. In this section we will briefly summarize them. The list of algorithms discussed is given in Fig. 4-23.

The optimal algorithm replaces the page referenced last among the current pages. Unfortunately, there is no way to determine which page will be last, so in practice this algorithm cannot be used. It is useful as a benchmark against which other algorithms can be measured, however.

The NRU algorithm divides pages into four classes depending on the state of the R and M bits. A random page from the lowest numbered class is chosen. This algorithm is easy to implement, but it is very crude. Better ones exist.

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Figure 4-23. Page replacement algorithms discussed in the text.

FIFO keeps track of the order pages were loaded into memory by keeping them in a linked list. Removing the oldest page then becomes trivial, but that page might still be in use, so FIFO is a bad choice.

Second chance is a modification to FIFO that checks if a page is in use before removing it. If it is, the page is spared. This modification greatly improves the performance. Clock is simply a different implementation of second chance. It has the same performance properties, but takes a little less time to execute the algorithm.

LRU is an excellent algorithm, but it cannot be implemented without special hardware. If this hardware is not available, it cannot be used. NFU is a crude attempt to approximate LRU. It is not very good. However, aging is a much better approximation to LRU and can be implemented efficiently. It is a good choice.

The last two algorithms use the working set. The working set algorithm is reasonable performance, but it is somewhat expensive to implement. WSClock is a variant that not only gives good performance but is also efficient to implement.

All in all, the two best algorithms are aging and WSClock. They are based on LRU and the working set, respectively. Both give good paging performance and can be implemented efficiently. A few other algorithms exist, but these two are probably the most important in practice.

4.5 MODELING PAGE REPLACEMENT ALGORITHMS

Over the years, some work has been done on modeling page replacement algorithms from a theoretical perspective. In this section we will discuss some of these ideas, just to see how the modeling process works.

4.5.1 Belady's Anomaly

Intuitively, it might seem that the more page frames the memory has, the fewer page faults a program will get. Surprisingly enough, this is not always the case. Belady et al. (1969) discovered a counterexample, in which FIFO caused more page faults with four page frames than with three. This strange situation has become known as **Belady's anomaly**. It is illustrated in Fig. 4-24 for a program with five virtual pages, numbered from 0 to 4. The pages are referenced in the order

0 1 2 3 0 1 4 0 1 2 3 4

In Fig. 4-24(a) we see how with three page frames a total of nine page faults are caused. In Fig. 4-24(b) we get ten page faults with four page frames.

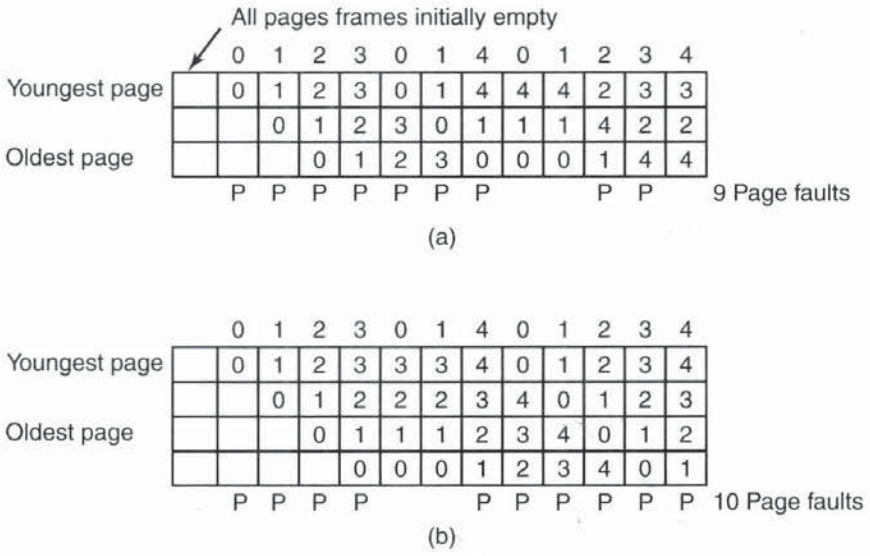


Figure 4-24. Belady's anomaly. (a) FIFO with three page frames. (b) FIFO with four page frames. The *P*'s show which page references cause page faults.

4.5.2 Stack Algorithms

Many researchers in computer science were dumbfounded by Belady's anomaly and began investigating it. This work led to the development of a whole theory of paging algorithms and their properties. While most of this work is beyond the scope of this book, we will give a brief introduction below. For more details see (Maekawa et al., 1987).

All of this work begins with the observation that every process generates a sequence of memory references as it runs. Each memory reference corresponds to

a specific virtual page. Thus conceptually, a process' memory access can be characterized by an (ordered) list of page numbers. This list is called the **reference string**, and plays a central role in the theory. For simplicity, in the rest of this section we will consider only the case of a machine with one process, so each machine has a single, deterministic reference string (with multiple processes, we would have to take into account the interleaving of their reference strings due to the multiprogramming).

A paging system can be characterized by three items:

1. The reference string of the executing process.
2. The page replacement algorithm.
3. The number of page frames available in memory, m.

Conceptually, we can imagine an abstract interpreter that works as follows. It maintains an internal array, M, that keeps track of the state of memory. It has as many elements as the process has virtual pages, which we will call n. The array M is divided into two parts. The top part, with m entries, contains all the pages that are currently in memory. The bottom part, with $n - m$ pages, contains all the pages that have been referenced once but have been paged out and are not currently in memory. Initially, M is the empty set, since no pages have been referenced and no pages are in memory.

As execution begins, the process begins emitting the pages in the reference string, one at a time. As each one comes out, the interpreter checks to see if the page is in memory (i.e., in the top part of M). If it is not, a page fault occurs. If there is an empty slot in memory (i.e., the top part of M contains fewer than m entries), the page is loaded and entered in the top part of M. This situation arises only at the start of execution. If memory is full (i.e., the top part of M contains m entries), the page replacement algorithm is invoked to remove a page from memory. In the model, what happens is that one page is moved from the top part of M to the bottom part, and the needed page entered into the top part. In addition, the top part and the bottom part may be separately rearranged.

To make the operation of the interpreter clearer, let us look at a concrete example using LRU page replacement. The virtual address space has eight pages and the physical memory has four page frames. At the top of Fig. 4-25 we have a reference string consisting of the 24 pages:

0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 4 1

Under the reference string, we have 25 columns of 8 items each. The first column, which is empty, reflects the state of M before execution begins. Each successive column shows M after one page has been emitted by the reference and processed by the paging algorithm. The heavy outline denotes the top of M, that is, the first four slots, which correspond to page frames in memory. Pages inside the heavy box are in memory, and pages below it have been paged out to disk.

Reference string	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
		0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	1	3	4
			0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3
				0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7
					0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	4	4	5	5
						0	2	2	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6
							0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Page faults	P	P	P	P	P	P	P		P					P			P						P	
Distance string	∞	∞	∞	∞	∞	∞	∞	4	∞	4	2	3	1	5	1	2	6	1	1	4	2	3	5	3

Figure 4-25. The state of the memory array, M, after each item in the reference string is processed. The distance string will be discussed in the next section.

The first page in the reference string is 0, so it is entered in the top of memory, as shown in the second column. The second page is 2, so it is entered at the top of the third column. This action causes 0 to move down. In this example, a newly loaded page is always entered at the top, and everything else moved down, as needed.

Each of the first seven pages in the reference string causes a page fault. The first four can be handled without removing a page, but starting with the reference to page 5, loading a new page requires removing an old page.

The second reference to page 3 does not cause a page fault, because 3 is already in memory. Nevertheless, the interpreter removes it from where it was and puts it on the top, as shown. The process continues for a while, until page 5 is referenced. This page is moved from bottom part of M to the top part (i.e., it is loaded into memory from disk). Whenever a page is referenced that is not within the heavy box, a page fault occurs, as indicated by the P's below the matrix.

Let us now briefly summarize some of the properties of this model. First, when a page is referenced, it is always moved to the top entry in M. Second, if the page referenced was already in M, all pages above it move down one position. A transition from within the box to outside of it corresponds to a page being evicted from memory. Third, pages that were below the referenced page are not moved. In this manner, the contents of M exactly represent the contents of the LRU algorithm.

Although this example uses LRU, the model works equally well with other algorithms. In particular, there is one class of algorithms that is especially interesting: algorithms that have the property

$$M(m, r) \subseteq M(m + 1, r)$$

where m varies over the page frames and r is an index into the reference string. What this says is that the set of pages included in the top part of M for a memory

with m page frames after r memory references are also included in M for a memory with $m + 1$ page frames. In other words, if we increase memory size by one page frame and re-execute the process, at every point during the execution, all the pages that were present in the first run are also present in the second run, along with one additional page.

From examination of Fig. 4-25 and a little thought about how it works, it should be clear that LRU has this property. Some other algorithms (e.g., optimal page replacement) also have it, but FIFO does not. Algorithms that have this property are called **stack algorithms**. These algorithms do not suffer from Belady's anomaly and are thus much loved by virtual memory theorists.

4.5.3 The Distance String

For stack algorithms, it is often convenient to represent the reference string in a more abstract way than the actual page numbers. A page reference will be henceforth denoted by the distance from the top of the stack where the referenced page was located. For example, the reference to page 1 in the last column of Fig. 4-25 is a reference to a page at a distance 3 from the top of the stack (because page 1 was in third place *before* the reference). Pages that have not yet been referenced and thus are not yet on the stack (i.e., not yet in M) are said to be at a distance ∞. The distance string for Fig. 4-25 is given at the bottom of the figure.

Note that the distance string depends not only on the reference string, but also on the paging algorithm. With the same original reference string, a different paging algorithm would make different choices about which pages to evict. As a result, a different sequence of stacks arises.

The statistical properties of the distance string have a big impact on the performance of the algorithm. In Fig. 4-26(a) we see the probability density function for the entries in a (ficticious) distance string, d. Most of the entries in the string are between 1 and k. With a memory of k page frames, few page faults occur.

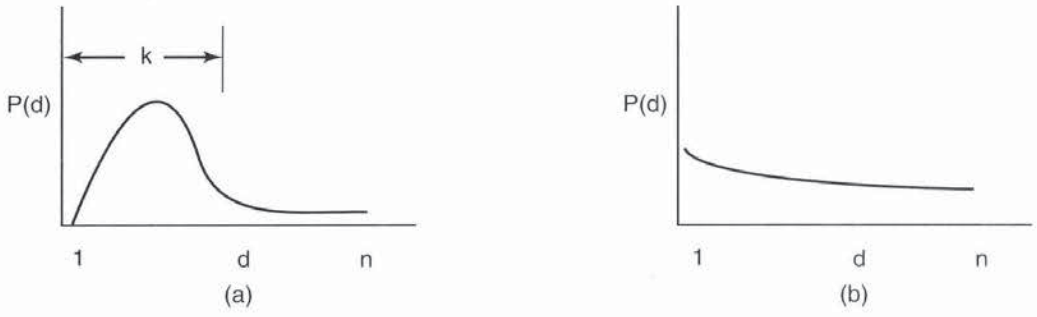


Figure 4-26. Probability density functions for two hypothetical distance strings.

In contrast, in Fig. 4-26(b), the references are so spread out that the only way to avoid a large number of page faults is to give the program as many page frames as it has virtual pages. Having a program like this is just bad luck.

4.5.4 Predicting Page Fault Rates

One of the nice properties of the distance string is that it can be used to predict the number of page faults that will occur with memories of different sizes. We will demonstrate how this computation can be made based on the example of Fig. 4-25. The goal is to make one pass over the distance string and, from the information collected, to be able to predict how many page faults the process would have in memories with 1, 2, 3, ..., n page frames, where n is the number of virtual pages in the process' address space.

The algorithm starts by scanning the distance string, page by page. It keeps track of the number of times 1 occurs, the number of times 2 occurs, and so on. Let C_i be the number of occurrences of i. For the distance string of Fig. 4-25, the C vector is illustrated in Fig. 4-27(a). In this example, it happens four times that the page referenced is already on top of the stack. Three times the reference is to the next-to-the-top page, and so forth. Let C_∞ be the number of times ∞ occurs in the distance string.

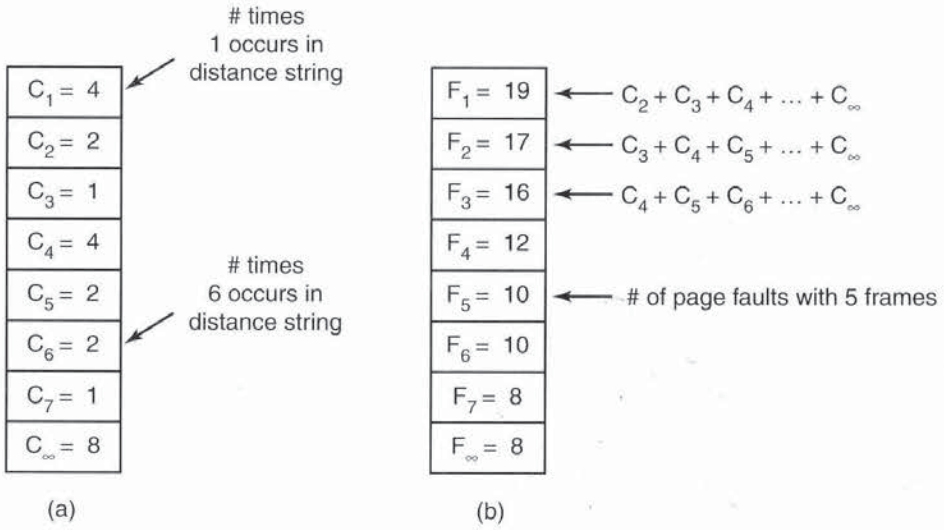


Figure 4-27. Computation of the page fault rate from the distance string. (a) The C vector. (b) F vector.

Now compute the F vector according to the formula

$$F_m = \sum_{k=m+1}^{n} C_k + C_\infty$$

The value of F_m is the number of page faults that will occur with the given distance string and m page frames. For the distance string of Fig. 4-25, Fig. 4-27(b) gives the F vector. For example, F_1 is 20, meaning that with a memory holding only 1 page frame, out of the 24 references in the string, all get page faults except the four that are the same as the previous page reference.

To see why this formula works, go back to the heavy box in Fig. 4-25. Let m be the number of page frames in the top part of M. A page fault occurs any time an element of the distance string is $m + 1$ or more. The summation in the formula above adds up the number of times such elements occur. This model can be used to make other predictions as well (Maekawa et al., 1987).

4.6 DESIGN ISSUES FOR PAGING SYSTEMS

In the previous sections we have explained how paging works and have given a few of the basic page replacement algorithms and shown how to model them. But knowing the bare mechanics is not enough. To design a system, you have to know a lot more to make it work well. It is like the difference between knowing how to move the rook, knight, bishop, and other pieces in chess, and being a good player. In the following sections, we will look at other issues that operating system designers must consider carefully in order to get good performance from a paging system.

4.6.1 Local versus Global Allocation Policies

In the preceding sections we have discussed several algorithms for choosing a page to replace when a fault occurs. A major issue associated with this choice (which we have carefully swept under the rug until now) is how memory should be allocated among the competing runnable processes.

Take a look at Fig. 4-28(a). In this figure, three processes, A, B, and C, make up the set of runnable processes. Suppose A gets a page fault. Should the page replacement algorithm try to find the least recently used page considering only the six pages currently allocated to A, or should it consider all the pages in memory? If it looks only at A's pages, the page with the lowest age value is $A5$, so we get the situation of Fig. 4-28(b).

On the other hand, if the page with the lowest age value is removed without regard to whose page it is, page $B3$ will be chosen and we will get the situation of Fig. 4-28(c). The algorithm of Fig. 4-28(b) is said to be a **local** page replacement algorithm, whereas that of Fig. 4-28(c) is said to be a **global** algorithm. Local algorithms effectively correspond to allocating every process a fixed fraction of the memory. Global algorithms dynamically allocate page frames among the runnable processes. Thus the number of page frames assigned to each process varies in time.

In general, global algorithms work better, especially when the working set size can vary over the lifetime of a process. If a local algorithm is used and the working set grows, thrashing will result, even if there are plenty of free page frames. If the working set shrinks, local algorithms waste memory. If a global algorithm is used, the system must continually decide how many page frames to assign to each process. One way is to monitor the working set size as indicated by

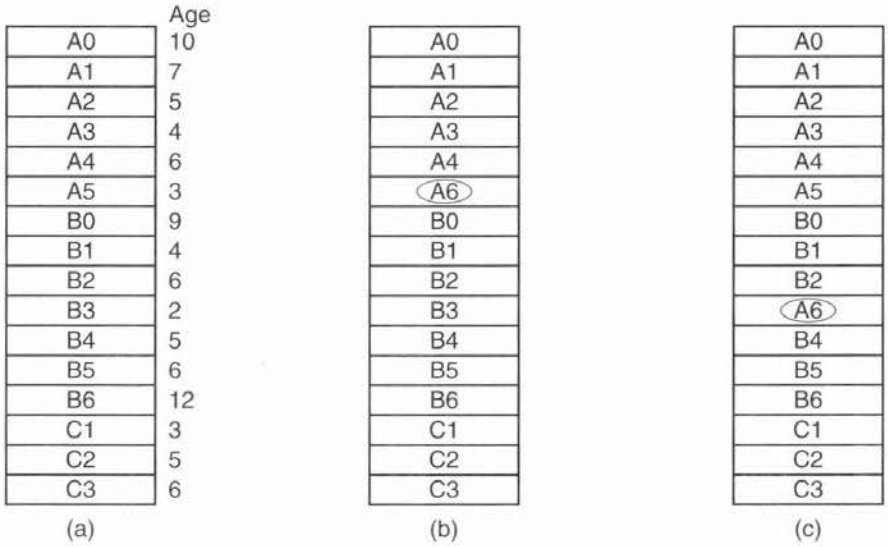


Figure 4-28. Local versus global page replacement. (a) Original configuration. (b) Local page replacement. (c) Global page replacement.

the aging bits, but this approach does not necessarily prevent thrashing. The working set may change size in microseconds, whereas the aging bits are a crude measure spread over a number of clock ticks.

Another approach is to have an algorithm for allocating page frames to processes. One way is to periodically determine the number of running processes and allocate each process an equal share. Thus with 12,416 available (i.e., non-operating system) page frames and 10 processes, each process gets 1241 frames. The remaining 6 go into a pool to be used when page faults occur.

Although this method seems fair, it makes little sense to give equal shares of the memory to a 10-KB process and a 300-KB process. Instead, pages can be allocated in proportion to each process' total size, with a 300-KB process getting 30 times the allotment of a 10-KB process. It is probably wise to give each process some minimum number, so it can run, no matter how small it is. On some machines, for example, a single two-operand instruction may need as many as six pages because the instruction itself, the source operand, and the destination operand may all straddle page boundaries. With an allocation of only five pages, programs containing such instructions cannot execute at all.

If a global algorithm is used, it may be possible to start each process up with some number of pages proportional to the process' size, but the allocation has to be updated dynamically as the processes run. One way to manage the allocation is to use the **PFF** (**Page Fault Frequency**) algorithm. It tells when to increase or decrease a process' page allocation but says nothing about which page to replace on a fault. It just controls the size of the allocation set.

For a large class of page replacement algorithms, including LRU, it is known that the fault rate decreases as more pages are assigned, as we discussed above. This is the assumption behind PFF. This property is illustrated in Fig. 4-29.

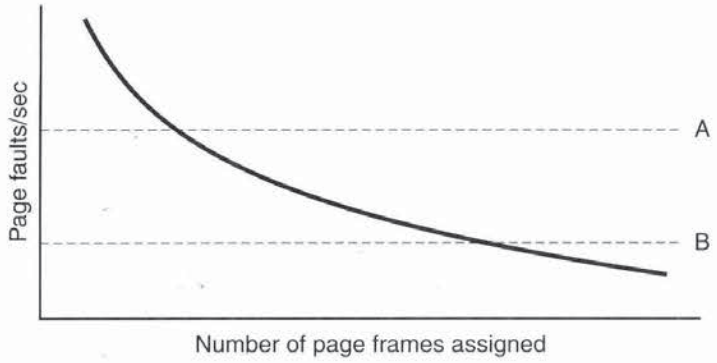


Figure 4-29. Page fault rate as a function of the number of page frames assigned.

Measuring the page fault rate is straightforward: just count the number of faults per second, possibly taking a running mean over past seconds as well. One easy way to do this is to add the present second's value to the current running mean and divide by two. The dashed line marked *A* corresponds to a page fault rate that is unacceptably high, so the faulting process is given more page frames to reduce the fault rate. The dashed line marked *B* corresponds to a page fault rate so low that it can be concluded that the process has too much memory. In this case page frames may be taken away from it. Thus, PFF tries to keep the paging rate for each process within acceptable bounds.

It is important to note that some page replacement algorithms can work with either a local replacement policy or a global one. For example, FIFO can replace the oldest page in all of memory (global algorithm) or the oldest page owned by the current process (local algorithm). Similarly, LRU or some approximation to it can replace the least recently page in all of memory (global algorithm) or the least recently used page owned by the current process (local algorithm). The choice of local versus global is independent of the algorithm in some cases.

On the other hand, for other page replacement algorithms, only a local strategy makes sense. In particular, the working set and WSClock algorithms refer to some specific process and must be applied in that context. There really is no working set for the machine as a whole and trying to use the union of all the working sets would lose the locality property and not work well.

4.6.2 Load Control

Even with the best page replacement algorithm and optimal global allocation of page frames to processes, it can happen that the system thrashes. In fact, whenever the combined working sets of all processes exceed the capacity of memory,

thrashing can be expected. One symptom of this situation is that the PFF algorithm indicates that some processes need more memory but no processes need less memory. In this case there is no way to give more memory to those processes needing it without hurting some other processes. The only real solution is to temporarily get rid of some processes.

The way to reduce the number of processes competing for memory is to swap some of them to the disk and free up all the pages they are holding. For example, one process can be swapped to disk and its page frames divided up among other processes that are thrashing. If the thrashing stops, the system can run for a while this way. If it does not stop, another process has to be swapped out, and so on, until the thrashing stops. Thus even with paging, swapping is still needed, only now swapping is used to reduce potential demand for memory, rather than to reclaim blocks of it for immediate use.

Swapping processes out to relieve the load on memory is reminiscent of two-level scheduling, in which some processes are put on disk and a short-term scheduler is used to schedule the remaining processes. Clearly, the two ideas can be combined, with just enough processes swapped out to make the page-fault rate acceptable. Periodically, some processes are brought in from disk and other ones are swapped out there.

However, another factor to consider is the degree of multiprogramming. As we saw in Fig. 4-4, when the number of processes in main memory is too low, the CPU may be idle for substantial periods of time. This consideration argues for considering not only process size and paging rate when deciding which process to swap out, but also its characteristics, such as whether it is CPU bound or I/O bound, and what characteristics the remaining processes have as well.

4.6.3 Page Size

The page size is often a parameter that can be chosen by the operating system. Even if the hardware has been designed with, for example, 512-byte pages, the operating system can easily regard pages 0 and 1, 2 and 3, 4 and 5, and so on, as 1-KB pages by always allocating two consecutive 512-byte page frames for them.

Determining the best page size requires balancing several competing factors. As a result, there is no overall optimum. To start with, there are two factors that argue for a small page size. A randomly chosen text, data, or stack segment will not fill an integral number of pages. On the average, half of the final page will be empty. The extra space in that page is wasted. This wastage is called **internal fragmentation**. With n segments in memory and a page size of p bytes, $np/2$ bytes will be wasted on internal fragmentation. This reasoning argues for a small page size.

Another argument for a small page size becomes apparent if we think about a program consisting of eight sequential phases of 4 KB each. With a 32-KB page

size, the program must be allocated 32 KB all the time. With a 16-KB page size, it needs only 16 KB. With a page size of 4 KB or smaller, it requires only 4 KB at any instant. In general, a large page size will cause more unused program to be in memory than a small page size.

On the other hand, small pages mean that programs will need many pages, hence a large page table. A 32-KB program needs only four 8-KB pages, but 64 512-byte pages. Transfers to and from the disk are generally a page at a time, with most of the time being for the seek and rotational delay, so that transferring a small page takes almost as much time as transferring a large page. It might take 64×10 msec to load 64 512-byte pages, but only 4×12 msec to load four 8-KB pages.

On some machines, the page table must be loaded into hardware registers every time the CPU switches from one process to another. On these machines having a small page size means that the time required to load the page registers gets longer as the page size gets smaller. Furthermore, the space occupied by the page table increases as the page size decreases.

This last point can be analyzed mathematically. Let the average process size be s bytes and the page size be p bytes. Furthermore, assume that each page entry requires e bytes. The approximate number of pages needed per process is then s/p, occupying se/p bytes of page table space. The wasted memory in the last page of the process due to internal fragmentation is $p/2$. Thus, the total overhead due to the page table and the internal fragmentation loss is given by the sum of these two terms:

$$\text{overhead} = se/p + p/2$$

The first term (page table size) is large when the page size is small. The second term (internal fragmentation) is large when the page size is large. The optimum must lie somewhere in between. By taking the first derivative with respect to p and equating it to zero, we get the equation

$$-se/p^2 + 1/2 = 0$$

From this equation we can derive a formula that gives the optimum page size (considering only memory wasted in fragmentation and page table size). The result is:

$$p = \sqrt{2se}$$

For s = 1MB and e = 8 bytes per page table entry, the optimum page size is 4 KB. Commercially available computers have used page sizes ranging from 512 bytes to 64 KB. A typical value used to 1 KB, but nowadays 4 KB or 8 KB are more common. As memories get larger, the page size tends to get larger as well (but not linearly). Quadrupling the RAM size rarely even doubles the page size.

4.6.4 Separate Instruction and Data Spaces

Most computers have a single address space that holds both programs and data, as shown in Fig. 4-30(a). If this address space is large enough, everything works fine. However, it is often too small, forcing programmers to stand on their heads to fit everything into the address space.

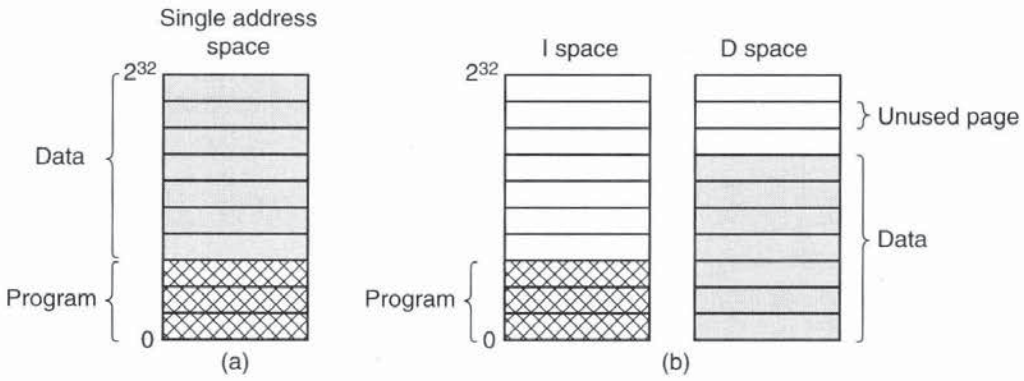


Figure 4-30. (a) One address space. (b) Separate I and D spaces.

One solution, pioneered on the (16-bit) PDP-11, is to have separate address spaces for instructions (program text) and data. These are called **I-space** and **D-space**, respectively. Each address space runs from 0 to some maximum, typically $2^{16} - 1$ or $2^{32} - 1$. In Fig. 4-30(b) we see both of them. The linker must know when separate I- and D-spaces are being used because when they are, the data are relocated to virtual address 0 instead of starting after the program.

In a computer with this design, both address spaces can be paged, independently from one another. Each one has its own page table, with its own mapping of virtual pages to physical page frames. When the hardware wants to fetch an instruction, it knows that it must use I-space and the I-space page table. Similarly, references to data must go through the D-space page table. Other than this distinction, having separate I- and D-spaces does not introduce any special complications and it does double the available address space.

4.6.5 Shared Pages

Another design issue is sharing. In a large multiprogramming system, it is common for several users to be running the same program at the same time. It is clearly more efficient to share the pages, to avoid having two copies of the same page in memory at the same time. One problem is that not all pages are sharable. In particular, pages that are read-only, such as program text, can be shared, but data pages cannot.

If separate I- and D-spaces are supported, it is relatively straightforward to share programs by having two or more processes use the same page table for their

I-space but different page tables for their D-spaces. Typically in an implementation that supports sharing in this way, page tables are data structures independent of the process table. Each process then has two pointers in its process table: one to the I-space page table and one to the D-space page table, as shown in Fig. 4-31. When the scheduler chooses a process to run, it uses these pointers to locate the appropriate page tables and sets up the MMU using them. Even without separate I- and D-spaces, processes can share programs (or sometimes, libraries), but the mechanism is more complicated.

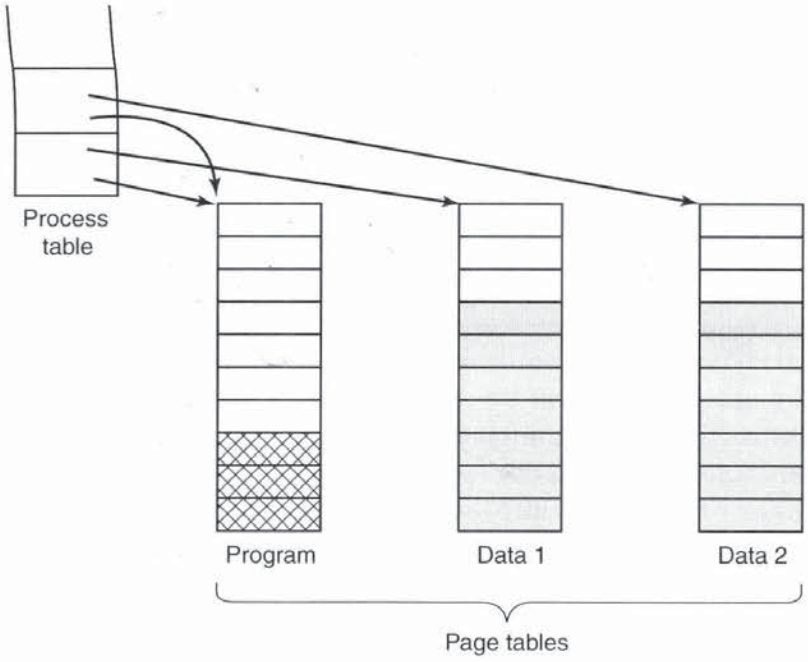


Figure 4-31. Two processes sharing the same program sharing its page table.

When two or more processes share some code, a problem occurs with the shared pages. Suppose that processes *A* and *B* are both running the editor and sharing its pages. If the scheduler decides to remove *A* from memory, evicting all its pages and filling the empty page frames with some other program will cause *B* to generate a large number of page faults to bring them back in again.

Similarly, when *A* terminates, it is essential to be able to discover that the pages are still in use so that their disk space will not be freed by accident. Searching all the page tables to see if a page is shared is usually too expensive, so special data structures are needed to keep track of shared pages, especially if the unit of sharing is the individual page (or run of pages), rather than an entire page table.

Sharing data is trickier than sharing code, but it is not impossible. In particular, in UNIX, after a fork system call, the parent and child are required to share both program text and data. In a paged system, what is often done is to give each of these processes its own page table and have both of them point to the same set

of pages. Thus no copying of pages is done at fork time. However, all the data pages are mapped into both processes as READ ONLY.

As long as both processes just read their data, without modifying it, this situation can continue. As soon as either process updates a memory word, the violation of the read-only protection causes a trap to the operating system. A copy is then made of the page so that each process now has its own private copy. Both copies are now set to READ-WRITE so subsequent writes to either copy proceed without trapping. This strategy means that those pages that are never written (including all the program pages) need not be copied. Only the data pages that are actually written need be copied. This approach, called **copy on write**, improves performance by reducing copying.

4.6.6 Cleaning Policy

Paging works best when there are plenty of free page frames that can be claimed as page faults occur. If every page frame is full, and furthermore modified, before a new page can be brought in, an old page must first be written to disk. To insure a plentiful supply of free page frames, many paging systems have a background process, called the **paging daemon**, that sleeps most of the time but is awakened periodically to inspect the state of memory. If too few page frames are free, the paging daemon begins selecting pages to evict using the chosen page replacement algorithm. If these pages have been modified since being loaded, they are written to disk.

In any event, the previous contents of the page are remembered. In the event one of the evicted pages is needed again before its frame has been overwritten, it can be reclaimed by removing it from the pool of free page frames. Keeping a supply of page frames around yields better performance than using all of memory and then trying to find a frame at the moment it is needed. At the very least, the paging daemon ensures that all the free frames are clean, so they need not be written to disk in a big hurry when they are required.

One way to implement this cleaning policy is with a two-handed clock. The front hand is controlled by the paging daemon. When it points to a dirty page, that page it written back to disk and the front hand advanced. When it points to a clean page, it is just advanced. The back hand is used for page replacement, as in the standard clock algorithm. Only now, the probability of the back hand hitting a clean page is increased due to the work of the paging daemon.

4.6.7 Virtual Memory Interface

Up until now, our whole discussion has assumed that virtual memory is transparent to processes and programmers, that is, all they see is a large virtual address space on a computer with a small(er) physical memory. With many systems, that is true, but in some advanced systems, programmers have some control over the

memory map and can use it in nontraditional ways to enhance program behavior. In this section, we will briefly look at a few of these.

One reason for giving programmers control over their memory map is to allow two or more processes to share the same memory. If programmers can name regions of their memory, it may be possible for one process to give another process the name of a memory region so that process can also map it in. With two (or more) processes sharing the same pages, high bandwidth sharing becomes possible—one process writes into the shared memory and another one reads from it.

Sharing of pages can also be used to implement a high-performance message-passing system. Normally, when messages are passed, the data are copied from one address space to another, at considerable cost. If processes can control their page map, a message can be passed by having the sending process unmap the page(s) containing the message, and the receiving process mapping them in. Here only the page names have to be copied, instead of all the data.

Yet another advanced memory management technique is **distributed shared memory** (Feeley et al., 1995; Li, 1986; Li and Hudak, 1989; and Zekauskas et al., 1994). The idea here is to allow multiple processes over a network to share a set of pages, possibly, but not necessarily, as a single shared linear address space. When a process references a page that is not currently mapped in, it gets a page fault. The page fault handler, which may be in the kernel or in user space, then locates the machine holding the page and sends it a message asking it to unmap the page and send it over the network. When the page arrives, it is mapped in and the faulting instruction is restarted. We will examine distributed shared memory in more detail in Chap. 8.

4.7 IMPLEMENTATION ISSUES

Implementers of virtual memory systems have to make choices among the major theoretical algorithms such as second chance versus aging, local versus global page allocation, and demand paging versus prepaging. But they also have to be aware of a number of practical implementation issues as well. In this section we will take a look at a few of the more common problems and some solutions.

4.7.1 Operating System Involvement with Paging

There are four times when the operating system has work to do relating to paging: process creation time, process execution time, page fault time, and process termination time. We will now briefly examine each of these to see what has to be done.

When a new process is created in a paging system, the operating system has to determine how large the program and data will be (initially) and create a page

table for it. Space has to be allocated in memory for the page table and it has to be initialized. The page table need not be resident when the process is swapped out but has to be in memory when the process is running. In addition, space has to be allocated in the swap area on disk so that when a page is swapped out, it has somewhere to go. The swap area also has to be initialized with program text and data so that when the new process starts getting page faults, the pages can be brought in from disk. Some systems page the program text directly from the executable file, thus saving disk space and initialization time. Finally, information about the page table and swap area on disk must be recorded in the process table.

When a process is scheduled for execution, the MMU has to be reset for the new process and the TLB flushed, to get rid of traces of the previously executing process. The new process' page table has to be made current, usually by copying it or a pointer to it to some hardware register(s). Optionally, some or all of the process' pages can be brought into memory to reduce the number of page faults initially.

When a page fault occurs, the operating system has to read out hardware registers to determine which virtual address caused the fault. From this information, it must compute which page is needed and locate that page on disk. It must then find an available page frame to put the new page, evicting some old page if need be. Then it must read the needed page into the page frame. Finally, it must back up the program counter to have it point to the faulting instruction and let that instruction execute again.

When a process exits, the operating system must release its page table, its pages, and the disk space that the pages occupy when they are on disk. If some of the pages are shared with other processes, the pages in memory and on disk can only be released when the last process using them has terminated.

4.7.2 Page Fault Handling

We are finally in a position to describe what happens on a page fault in some detail. The sequence of events is as follows:

1. The hardware traps to the kernel, saving the program counter on the stack. On most machines, some information about the state of the current instruction is saved in special CPU registers.
2. An assembly code routine is started to save the general registers and other volatile information, to keep the operating system from destroying it. This routine calls the operating system as a procedure.
3. The operating system discovers that a page fault has occurred, and tries to discover which virtual page is needed. Often one of the hardware registers contains this information. If not, the operating system must retrieve the program counter, fetch the instruction, and parse it in software to figure out what it was doing when the fault hit.

4. Once the virtual address that caused the fault is known, the system checks to see if this address is valid and the protection consistent with the access. If not, the process is sent a signal or killed. If the address is valid and no protection fault has occurred, the system checks to see if a page frame is free. If no frames are free, the page replacement algorithm is run to select a victim.
5. If the page frame selected is dirty, the page is scheduled for transfer to the disk, and a context switch takes place, suspending the faulting process and letting another one run until the disk transfer has completed. In any event, the frame is marked as busy to prevent it from being used for another purpose.
6. As soon as the page frame is clean (either immediately or after it is written to disk), the operating system looks up the disk address where the needed page is, and schedules a disk operation to bring it in. While the page is being loaded, the faulting process is still suspended and another user process is run, if one is available.
7. When the disk interrupt indicates that the page has arrived, the page tables are updated to reflect its position, and the frame is marked as being in normal state.
8. The faulting instruction is backed up to the state it had when it began and the program counter is reset to point to that instruction.
9. The faulting process is scheduled, and the operating system returns to the assembly language routine that called it.
10. This routine reloads the registers and other state information and returns to user space to continue execution, as if no fault had occurred.

4.7.3 Instruction Backup

When a program references a page that is not in memory, the instruction causing the fault is stopped part way through and a trap to the operating system occurs. After the operating system has fetched the page needed, it must restart the instruction causing the trap. This is easier said than done.

To see the nature of this problem at its worst, consider a CPU that has instructions with two addresses, such as the Motorola 680x0, widely used in embedded systems. The instruction

```
MOV.L #6(A1),2(A0)
```

is 6 bytes, for example (see Fig. 4-32). In order to restart the instruction, the operating system must determine where the first byte of the instruction is. The value

of the program counter at the time of the trap depends on which operand faulted and how the CPU's microcode has been implemented.

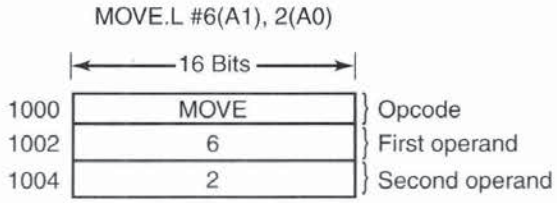


Figure 4-32. An instruction causing a page fault.

In Fig. 4-32, we have an instruction starting at address 1000 that makes three memory references: the instruction word itself, and two offsets for the operands. Depending on which of these three memory references caused the page fault, the program counter might be 1000, 1002, or 1004 at the time of the fault. It is frequently impossible for the operating system to determine unambiguously where the instruction began. If the program counter is 1002 at the time of the fault, the operating system has no way of telling whether the word in 1002 is a memory address associated with an instruction at 1000 (e.g., the location of an operand), or an instruction opcode.

Bad as this problem may be, it could have been worse. Some 680x0 addressing modes use autoincrementing, which means that a side effect of executing the instruction is to increment one or more registers. Instructions that use autoincrement mode can also fault. Depending on the details of the microcode, the increment may be done before the memory reference, in which case the operating system must decrement the register in software before restarting the instruction. Or, the autoincrement may be done after the memory reference, in which case it will not have been done at the time of the trap and must not be undone by the operating system. Autodecrement mode also exists and causes a similar problem. The precise details of whether autoincrements and autodecrements have or have not been done before the corresponding memory references may differ from instruction to instruction and from CPU model to CPU model.

Fortunately, on some machines the CPU designers provide a solution, usually in the form of a hidden internal register into which the program counter is copied just before each instruction is executed. These machines may also have a second register telling which registers have already been autoincremented or autodecremented, and by how much. Given this information, the operating system can unambiguously undo all the effects of the faulting instruction so it can be started all over again. If this information is not available, the operating system has to jump through hoops to figure out what happened and how to repair it. It is as though the hardware designers were unable to solve the problem, so they threw up their hands and told the operating system writers to deal with it. Nice guys.

4.7.4 Locking Pages in Memory

Although we have not discussed I/O much in this chapter, the fact that a computer has virtual memory does not mean that I/O is absent. Virtual memory and I/O interact in subtle ways. Consider a process that has just issued a system call to read from some file or device into a buffer within its address space. While waiting for the I/O to complete, the process is suspended and another process is allowed to run. This other process gets a page fault.

If the paging algorithm is global, there is a small, but nonzero, chance that the page containing the I/O buffer will be chosen to be removed from memory. If an I/O device is currently in the process of doing a DMA transfer to that page, removing it will cause part of the data to be written in the buffer where they belong, and part of the data to be written over the newly loaded page. One solution to this problem is to lock pages engaged in I/O in memory so that they will not be removed. Locking a page is often called **pinning** it in memory. Another solution is to do all I/O to kernel buffers and then copy the data to user pages later.

4.7.5 Backing Store

In our discussion of page replacement algorithms, we saw how a page is selected for removal. We have not said much about where on the disk it is put when it is paged out. Let us now describe some of the issues related to disk management.

The simplest algorithm for allocating page space on the disk is to have a special swap area on the disk. When the system is booted, this area is empty, and is represented in memory as a single entry giving its origin and size. When the first process is started, a chunk of the swap area the size of the first process is reserved and the remaining area reduced by that amount. As new processes are started, they are assigned chunks of the swap area equal in size to their core images. As they finish, their disk space is freed. The swap area is managed as a list of free chunks.

Associated with each process is the disk address of its swap area, kept in the process table. Calculating the address to write a page to becomes simple: just add the offset of the page within the virtual address space to the start of the swap area. However, before a process can start, the swap area must be initialized. One way is to copy the entire process image to the swap area, so that it can be brought *in* as needed. The other is to load the entire process in memory and let it be paged *out* as needed.

However, this simple model has a problem: processes can increase in size after starting. Although the program text is usually fixed, the data area can sometimes grow, and the stack can always grow. Consequently, it may be better to reserve separate swap areas for the text, data, and stack and allow each of these areas to consist of more than one chunk on the disk.

The other extreme is to allocate nothing in advance and allocate disk space for each page when it is swapped out and deallocate it when it is swapped back in. In this way, processes in memory do not tie up any swap space. The disadvantage is that a disk address is needed in memory to keep track of each page on disk. In other words, there must a table per process telling for each page on disk where it is. The two alternatives are shown in Fig. 4-33.

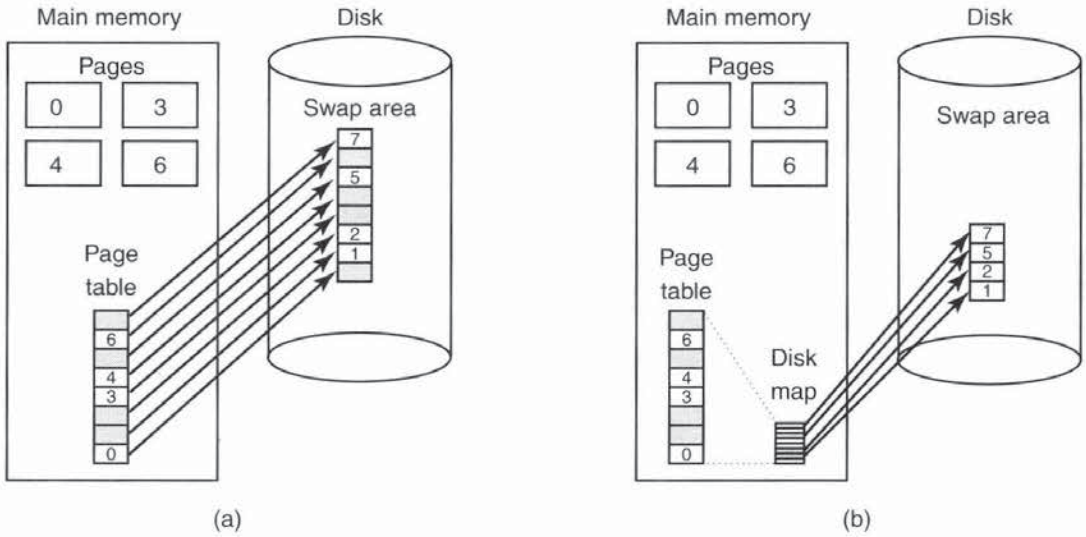


Figure 4-33. (a) Paging to a static swap area. (b) Backing up pages dynamically.

In Fig. 4-33(a), a page table with eight pages is illustrated. Pages 0, 3, 4, and 6 are in main memory. Pages 1, 2, 5, and 7 are on disk. The swap area on disk is as large as the process virtual address space (eight pages) with each page having a fixed location to which it is written when it is evicted from main memory. Calculation of this address requires knowing only where the process' paging area begins, since pages are stored in it contiguously in order of their virtual page number. Pages that are in memory always have a shadow copy on disk, but this copy may be out of date if the page has been modified since being loaded.

In Fig. 4-33(b), pages do not have fixed addresses on disk. When a page is swapped out, an empty disk page is chosen on the fly and the disk map (which has room for one disk address per virtual page) is updated accordingly. Pages that are in memory have no copy on disk. Their entries in the disk map contain an invalid disk address or a bit marking them as not in use.

4.7.6 Separation of Policy and Mechanism

An important tool for managing the complexity of any system is to separate the policy from the mechanism. This principle can be applied to memory management by having most of the memory manager run as a user-level process.

Such a separation was first done in Mach (Young et al., 1987) and in MINIX (Tanenbaum, 1987). The discussion below is loosely based on Mach.

A simple example of how policy and mechanism can be separated is shown in Fig. 4-34. Here the memory management system is divided into three parts:

1. A low-level MMU handler.
2. A page fault handler that is part of the kernel.
3. An external pager running in user space.

All the details of how the MMU works are encapsulated in the MMU handler, which is machine-dependent code and has to be rewritten for each new platform the operating system is ported to. The page-fault handler is machine independent code and contains most of the mechanism for paging. The policy is largely determined by the external pager, which runs as a user process.

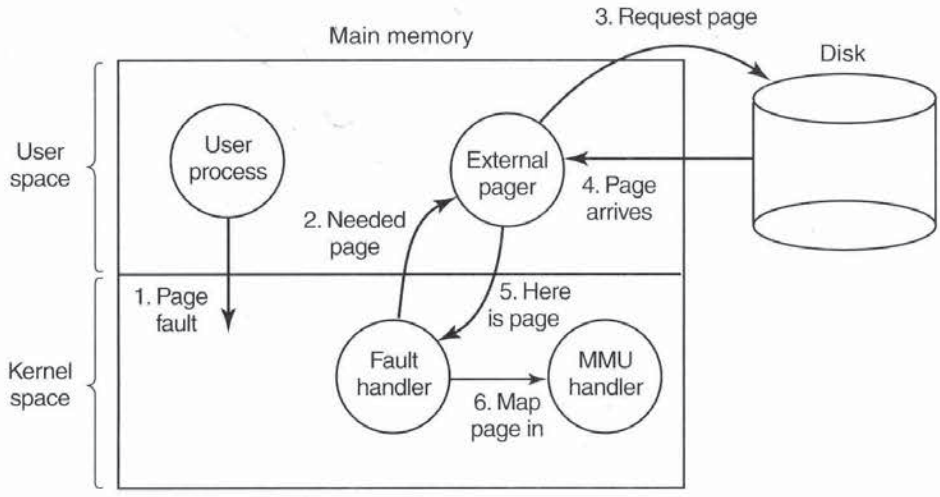


Figure 4-34. Page fault handling with an external pager.

When a process starts up, the external pager is notified in order to set up the process page map and allocate backing store on the disk if need be. As the process runs, it may map new objects into its address space, so the external pager is again notified.

Once the process starts running, it may get a page fault. The fault handler figures out which virtual page is needed and sends a message to the external pager, telling what the problem is. The external pager then reads the needed page in from the disk and copies it to a portion of its own address space. Then it tells the fault handler where the page is. The fault handler then unmaps the page from the external pager's address space and asks the MMU handler to put it into the user's address space at the right place. Then the user process can be restarted.

This implementation leaves open where the page replacement algorithm is put. It would be cleanest to have it in the external pager, but there are some problems with this approach. Principal among these is that the external pager does not have access to the *R* and *M* bits of all the pages. These bits play a role in many of the paging algorithms. Thus either some mechanism is needed to pass this information up to the external pager, or the page replacement algorithm must go in the kernel. In the latter case, the fault handler tells the external pager which page it has selected for eviction and provides the data, either by mapping it into the external pager's address space or including it in a message. Either way, the external pager writes the data to disk.

The main advantage of this implementation is more modular code and greater flexibility. The main disadvantage is the extra overhead of crossing the user-kernel boundary several times and the overhead of the various messages being sent between the pieces of the system. At the moment, the subject is highly controversial, but as computers get faster and faster, and the software gets more and more complex, in the long run sacrificing some performance for more reliable software will probably be acceptable to most implementers.

4.8 SEGMENTATION

The virtual memory discussed so far is one-dimensional because the virtual addresses go from 0 to some maximum address, one address after another. For many problems, having two or more separate virtual address spaces may be much better than having only one. For example, a compiler has many tables that are built up as compilation proceeds, possibly including

1. The source text being saved for the printed listing (on batch systems).
2. The symbol table, containing the names and attributes of variables.
3. The table containing all the integer and floating-point constants used.
4. The parse tree, containing the syntactic analysis of the program.
5. The stack used for procedure calls within the compiler.

Each of the first four tables grows continuously as compilation proceeds. The last one grows and shrinks in unpredictable ways during compilation. In a one-dimensional memory, these five tables would have to be allocated contiguous chunks of virtual address space, as in Fig. 4-35.

Consider what happens if a program has an exceptionally large number of variables but a normal amount of everything else. The chunk of address space allocated for the symbol table may fill up, but there may be lots of room in the other tables. The compiler could, of course, simply issue a message saying that the compilation cannot continue due to too many variables, but doing so does not seem very sporting when unused space is left in the other tables.

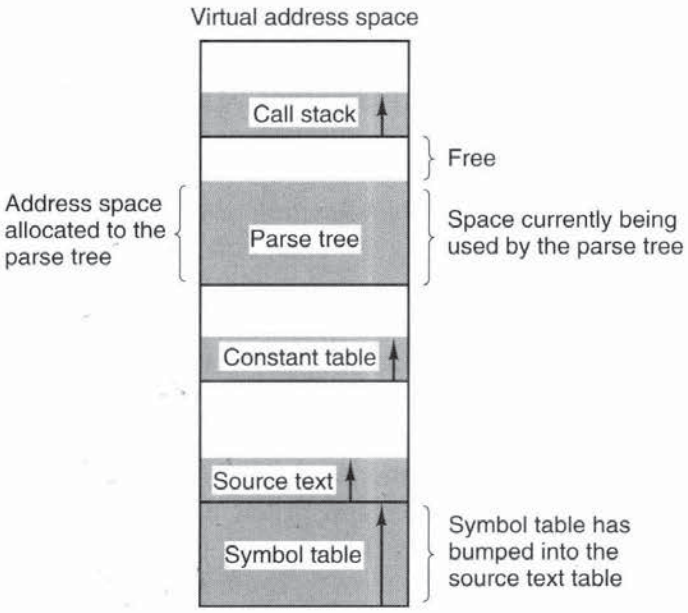


Figure 4-35. In a one-dimensional address space with growing tables, one table may bump into another.

Another possibility is to play Robin Hood, taking space from the tables with an excess of room and giving it to the tables with little room. This shuffling can be done, but it is analogous to managing one's own overlays—a nuisance at best and a great deal of tedious, unrewarding work at worst.

What is really needed is a way of freeing the programmer from having to manage the expanding and contracting tables, in the same way that virtual memory eliminates the worry of organizing the program into overlays.

A straightforward and extremely general solution is to provide the machine with many completely independent address spaces, called **segments**. Each segment consists of a linear sequence of addresses, from 0 to some maximum. The length of each segment may be anything from 0 to the maximum allowed. Different segments may, and usually do, have different lengths. Moreover, segment lengths may change during execution. The length of a stack segment may be increased whenever something is pushed onto the stack and decreased whenever something is popped off the stack.

Because each segment constitutes a separate address space, different segments can grow or shrink independently, without affecting each other. If a stack in a certain segment needs more address space to grow, it can have it, because there is nothing else in its address space to bump into. Of course, a segment can fill up but segments are usually very large, so this occurrence is rare. To specify an address in this segmented or two-dimensional memory, the program must supply a two-part address, a segment number, and an address within the segment.

Figure 4-36 illustrates a segmented memory being used for the compiler tables discussed earlier. Five independent segments are shown here.

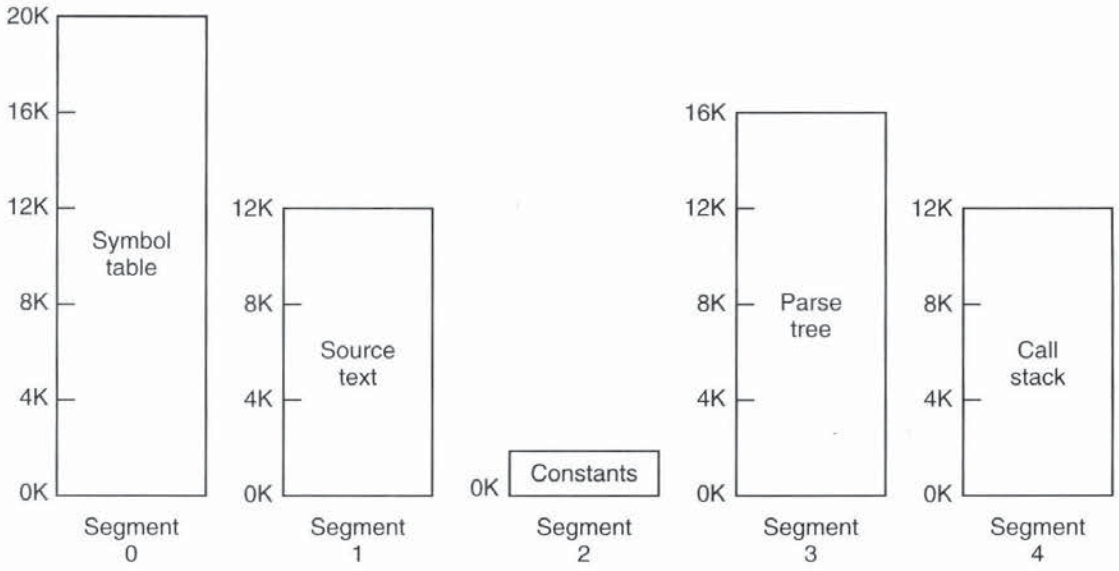


Figure 4-36. A segmented memory allows each table to grow or shrink independently of the other tables.

We emphasize that a segment is a logical entity, which the programmer is aware of and uses as a logical entity. A segment might contain a procedure, or an array, or a stack, or a collection of scalar variables, but usually it does not contain a mixture of different types.

A segmented memory has other advantages besides simplifying the handling of data structures that are growing or shrinking. If each procedure occupies a separate segment, with address 0 as its starting address, the linking up of procedures compiled separately is greatly simplified. After all the procedures that constitute a program have been compiled and linked up, a procedure call to the procedure in segment n will use the two-part address $(n, 0)$ to address word 0 (the entry point).

If the procedure in segment n is subsequently modified and recompiled, no other procedures need be changed (because no starting addresses have been modified), even if the new version is larger than the old one. With a one-dimensional memory, the procedures are packed tightly next to each other, with no address space between them. Consequently, changing one procedure's size can affect the starting address of other, unrelated procedures. This, in turn, requires modifying all procedures that call any of the moved procedures, in order to incorporate their new starting addresses. If a program contains hundreds of procedures, this process can be costly.

Segmentation also facilitates sharing procedures or data between several processes. A common example is the **shared library**. Modern workstations that

run advanced window systems often have extremely large graphical libraries compiled into nearly every program. In a segmented system, the graphical library can be put in a segment and shared by multiple processes, eliminating the need for having it in every process' address space. While it is also possible to have shared libraries in pure paging systems, it is much more complicated. In effect, these systems do it by simulating segmentation.

Because each segment forms a logical entity of which the programmer is aware, such as a procedure, or an array, or a stack, different segments can have different kinds of protection. A procedure segment can be specified as execute only, prohibiting attempts to read from it or store into it. A floating-point array can be specified as read/write but not execute, and attempts to jump to it will be caught. Such protection is helpful in catching programming errors.

You should try to understand why protection makes sense in a segmented memory but not in a one-dimensional paged memory. In a segmented memory the user is aware of what is in each segment. Normally, a segment would not contain a procedure and a stack, for example, but one or the other. Since each segment contains only one type of object, the segment can have the protection appropriate for that particular type. Paging and segmentation are compared in Fig. 4-37.

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Figure 4-37. Comparison of paging and segmentation.

The contents of a page are, in a sense, accidental. The programmer is unaware of the fact that paging is even occurring. Although putting a few bits in each entry of the page table to specify the access allowed would be possible, to utilize this feature the programmer would have to keep track of where in his address space the page boundaries were. That is precisely the sort of administration that paging was invented to eliminate. Because the user of a segmented memory has the illusion that all segments are in main memory all the time—that is, he can address them as though they were—he can protect each segment separately, without having to be concerned with the administration of overlaying them.

4.8.1 Implementation of Pure Segmentation

The implementation of segmentation differs from paging in an essential way: pages are fixed size and segments are not. Figure 4-38(a) shows an example of physical memory initially containing five segments. Now consider what happens if segment 1 is evicted and segment 7, which is smaller, is put in its place. We arrive at the memory configuration of Fig. 4-38(b). Between segment 7 and segment 2 is an unused area—that is, a hole. Then segment 4 is replaced by segment 5, as in Fig. 4-38(c), and segment 3 is replaced by segment 6, as in Fig. 4-38(d). After the system has been running for a while, memory will be divided up into a number of chunks, some containing segments and some containing holes. This phenomenon, called **checkerboarding** or **external fragmentation**, wastes memory in the holes. It can be dealt with by compaction, as shown in Fig. 4-38(e).

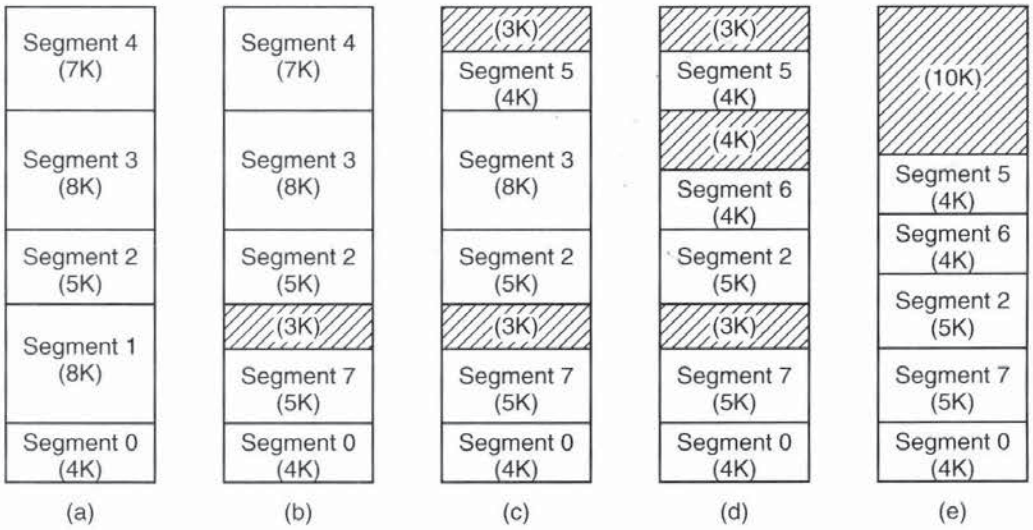


Figure 4-38. (a)-(d) Development of checkerboarding. (e) Removal of the checkerboarding by compaction.

4.8.2 Segmentation with Paging: MULTICS

If the segments are large, it may be inconvenient, or even impossible, to keep them in main memory in their entirety. This leads to the idea of paging them, so that only those pages that are actually needed have to be around. Several significant systems have supported paged segments. In this section we will describe the first one: MULTICS. In the next one we will discuss a more recent one: the Intel Pentium.

MULTICS ran on the Honeywell 6000 machines and their descendants and provided each program with a virtual memory of up to 2^{18} segments (more than 250,000), each of which could be up to 65,536 (36-bit) words long. To implement this, the MULTICS designers chose to treat each segment as a virtual memory and to page it, combining the advantages of paging (uniform page size and not having to keep the whole segment in memory if only part of it is being used) with the advantages of segmentation (ease of programming, modularity, protection, and sharing).

Each MULTICS program has a segment table, with one descriptor per segment. Since there are potentially more than a quarter of a million entries in the table, the segment table is itself a segment and is paged. A segment descriptor contains an indication of whether the segment is in main memory or not. If any part of the segment is in memory, the segment is considered to be in memory, and its page table will be in memory. If the segment is in memory, its descriptor contains an 18-bit pointer to its page table [see Fig. 4-39(a)]. Because physical addresses are 24 bits and pages are aligned on 64-byte boundaries (implying that the low-order 6 bits of page addresses are 000000), only 18 bits are needed in the descriptor to store a page table address. The descriptor also contains the segment size, the protection bits, and a few other items. Figure 4-39(b) illustrates a MULTICS segment descriptor. The address of the segment in secondary memory is not in the segment descriptor but in another table used by the segment fault handler.

Each segment is an ordinary virtual address space and is paged in the same way as the nonsegmented paged memory described earlier in this chapter. The normal page size is 1024 words (although a few small segments used by MULTICS itself are not paged or are paged in units of 64 words to save physical memory).

An address in MULTICS consists of two parts: the segment and the address within the segment. The address within the segment is further divided into a page number and a word within the page, as shown in Fig. 4-40. When a memory reference occurs, the following algorithm is carried out.

1. The segment number is used to find the segment descriptor.
2. A check is made to see if the segment's page table is in memory. If the page table is in memory, it is located. If it is not, a segment fault occurs. If there is a protection violation, a fault (trap) occurs.

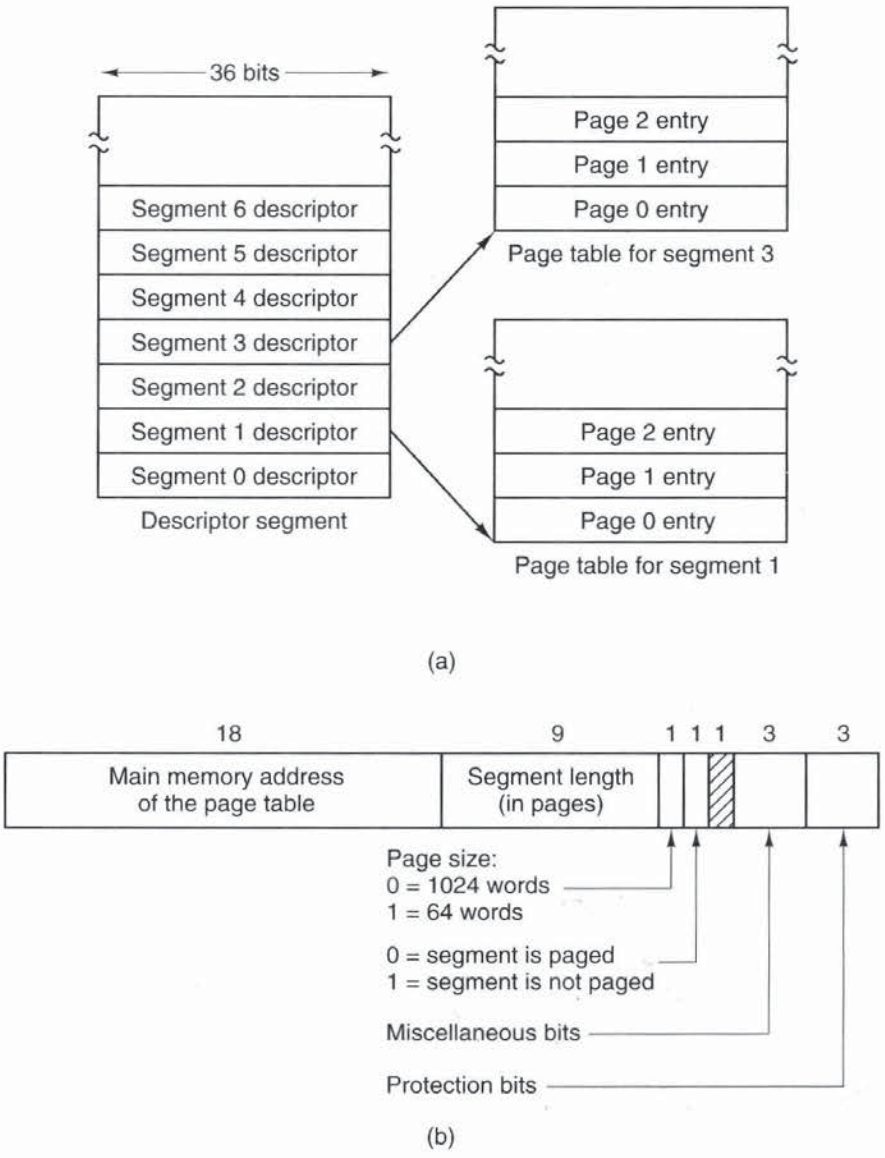


Figure 4-39. The MULTICS virtual memory. (a) The descriptor segment points to the page tables. (b) A segment descriptor. The numbers are the field lengths.

3. The page table entry for the requested virtual page is examined. If the page is not in memory, a page fault occurs. If it is in memory, the main memory address of the start of the page is extracted from the page table entry.
4. The offset is added to the page origin to give the main memory address where the word is located.
5. The read or store finally takes place.

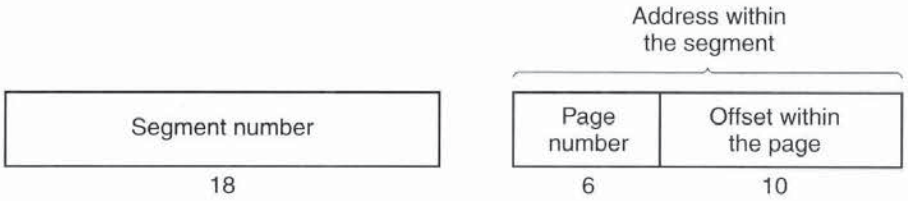


Figure 4-40. A 34-bit MULTICS virtual address.

This process is illustrated in Fig. 4-41. For simplicity, the fact that the descriptor segment is itself paged has been omitted. What really happens is that a register (the descriptor base register), is used to locate the descriptor segment's page table, which, in turn, points to the pages of the descriptor segment. Once the descriptor for the needed segment has been found, the addressing proceeds as shown in Fig. 4-41.

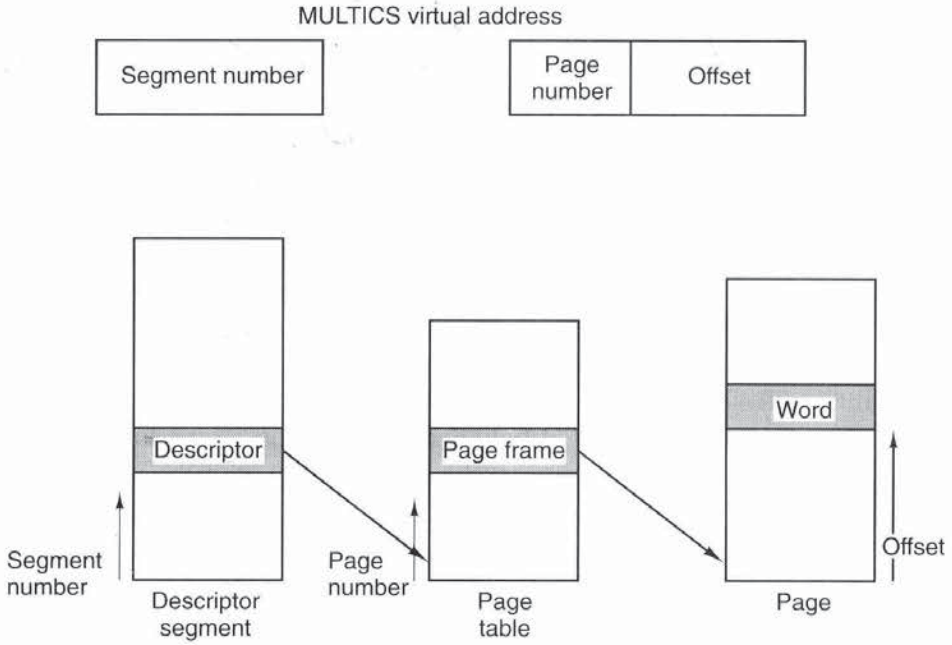


Figure 4-41. Conversion of a two-part MULTICS address into a main memory address.

As you have no doubt guessed by now, if the preceding algorithm were actually carried out by the operating system on every instruction, programs would not run very fast. In reality, the MULTICS hardware contains a 16-word high-speed TLB that can search all its entries in parallel for a given key. It is illustrated in Fig. 4-42. When an address is presented to the computer, the addressing hardware first checks to see if the virtual address is in the TLB. If so, it gets the page frame number directly from the TLB and forms the actual address of the referenced word without having to look in the descriptor segment or page table.

Comparison field: Segment number	Comparison field: Virtual page	Page frame	Protection	Age	Is this entry used?
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

Figure 4-42. A simplified version of the MULTICS TLB. The existence of two page sizes makes the actual TLB more complicated.

The addresses of the 16 most recently referenced pages are kept in the TLB. Programs whose working set is smaller than the TLB size will come to equilibrium with the addresses of the entire working set in the TLB and therefore will run efficiently. If the page is not in the TLB, the descriptor and page tables are actually referenced to find the page frame address, and the TLB is updated to include this page, the least recently used page being thrown out. The age field keeps track of which entry is the least recently used. The reason that a TLB is used is for comparing the segment and page number of all the entries in parallel.

4.8.3 Segmentation with Paging: The Intel Pentium

In many ways, the virtual memory on the Pentium resembles MULTICS, including the presence of both segmentation and paging. Whereas MULTICS has 256K independent segments, each up to 64K 36-bit words, the Pentium has 16K independent segments, each holding up to 1 billion 32-bit words. Although there are fewer segments, the larger segment size is far more important, as few programs need more than 1000 segments, but many programs need large segments.

The heart of the Pentium virtual memory consists of two tables, the **LDT** (**Local Descriptor Table**) and the **GDT** (**Global Descriptor Table**). Each program has its own LDT, but there is a single GDT, shared by all the programs on the computer. The LDT describes segments local to each program, including its code, data, stack, and so on, whereas the GDT describes system segments, including the operating system itself.

To access a segment, a Pentium program first loads a selector for that segment into one of the machine's six segment registers. During execution, the CS register holds the selector for the code segment and the DS register holds the selector for

the data segment. The other segment registers are less important. Each selector is a 16-bit number, as shown in Fig. 4-43.

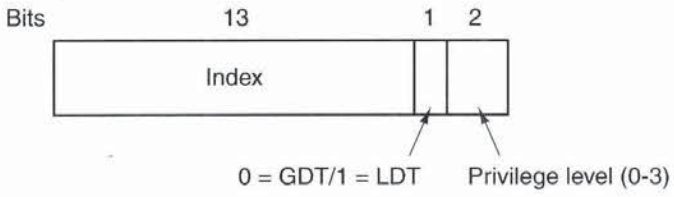


Figure 4-43. A Pentium selector.

One of the selector bits tells whether the segment is local or global (i.e., whether it is in the LDT or GDT). Thirteen other bits specify the LDT or GDT entry number, so these tables are each restricted to holding 8K segment descriptors. The other 2 bits relate to protection, and will be described later. Descriptor 0 is forbidden. It may be safely loaded into a segment register to indicate that the segment register is not currently available. It causes a trap if used.

At the time a selector is loaded into a segment register, the corresponding descriptor is fetched from the LDT or GDT and stored in microprogram registers, so it can be accessed quickly. A descriptor consists of 8 bytes, including the segment's base address, size, and other information, as depicted in Fig. 4-44.

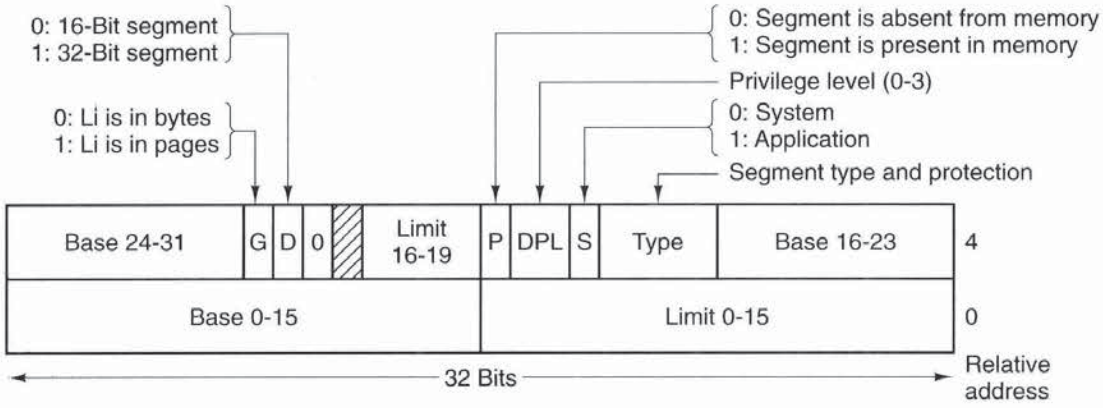


Figure 4-44. Pentium code segment descriptor. Data segments differ slightly.

The format of the selector has been cleverly chosen to make locating the descriptor easy. First either the LDT or GDT is selected, based on selector bit 2. Then the selector is copied to an internal scratch register, and the 3 low-order bits set to 0. Finally, the address of either the LDT or GDT table is added to it, to give a direct pointer to the descriptor. For example, selector 72 refers to entry 9 in the GDT, which is located at address GDT + 72.

Let us trace the steps by which a (selector, offset) pair is converted to a physical address. As soon as the microprogram knows which segment register is being

used, it can find the complete descriptor corresponding to that selector in its internal registers. If the segment does not exist (selector 0), or is currently paged out, a trap occurs.

It then checks to see if the offset is beyond the end of the segment, in which case a trap also occurs. Logically, there should simply be a 32-bit field in the descriptor giving the size of the segment, but there are only 20 bits available, so a different scheme is used. If the *Gbit* (Granularity) field is 0, the *Limit* field is the exact segment size, up to 1 MB. If it is 1, the *Limit* field gives the segment size in pages instead of bytes. The Pentium page size is fixed at 4 KB, so 20 bits are enough for segments up to 2^{32} bytes.

Assuming that the segment is in memory and the offset is in range, the Pentium then adds the 32-bit *Base* field in the descriptor to the offset to form what is called a **linear address**, as shown in Fig. 4-45. The *Base* field is broken up into three pieces and spread all over the descriptor for compatibility with the 286, in which the *Base* is only 24 bits. In effect, the *Base* field allows each segment to start at an arbitrary place within the 32-bit linear address space.

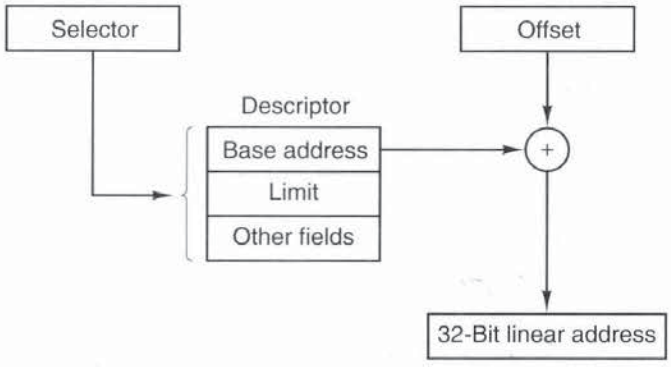


Figure 4-45. Conversion of a (selector, offset) pair to a linear address.

If paging is disabled (by a bit in a global control register), the linear address is interpreted as the physical address and sent to the memory for the read or write. Thus with paging disabled, we have a pure segmentation scheme, with each segment's base address given in its descriptor. Segments are permitted to overlap, incidentally, probably because it would be too much trouble and take too much time to verify that they were all disjoint.

On the other hand, if paging is enabled, the linear address is interpreted as a virtual address and mapped onto the physical address using page tables, pretty much as in our earlier examples. The only real complication is that with a 32-bit virtual address and a 4-KB page, a segment might contain 1 million pages, so a two-level mapping is used to reduce the page table size for small segments.

Each running program has a **page directory** consisting of 1024 32-bit entries. It is located at an address pointed to by a global register. Each entry in this

directory points to a page table also containing 1024 32-bit entries. The page table entries point to page frames. The scheme is shown in Fig. 4-46.

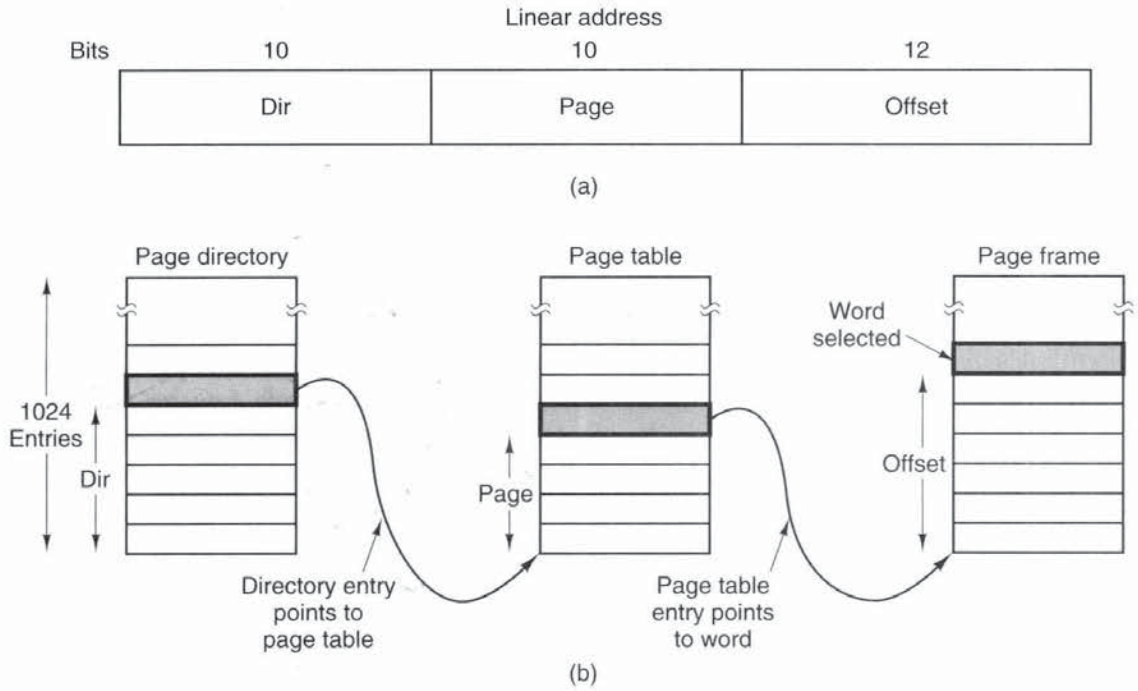


Figure 4-46. Mapping of a linear address onto a physical address.

In Fig. 4-46(a) we see a linear address divided into three fields, *Dir*, *Page*, and *Offset*. The *Dir* field is used to index into the page directory to locate a pointer to the proper page table. Then the *Page* field is used as an index into the page table to find the physical address of the page frame. Finally, *Offset* is added to the address of the page frame to get the physical address of the byte or word needed.

The page table entries are 32 bits each, 20 of which contain a page frame number. The remaining bits contain access and dirty bits, set by the hardware for the benefit of the operating system, protection bits, and other utility bits.

Each page table has entries for 1024 4-KB page frames, so a single page table handles 4 megabytes of memory. A segment shorter than 4M will have a page directory with a single entry, a pointer to its one and only page table. In this way, the overhead for short segments is only two pages, instead of the million pages that would be needed in a one-level page table.

To avoid making repeated references to memory, the Pentium, like MULTICS, has a small TLB that directly maps the most recently used *Dir–Page* combinations onto the physical address of the page frame. Only when the current combination is not present in the TLB is the mechanism of Fig. 4-46 actually carried out and the TLB updated. As long as TLB misses are rare, performance is good.

It is also worth noting that if some application does not need segmentation but is content with a single, paged, 32-bit address space, that model is possible. All the segment registers can be set up with the same selector, whose descriptor has *Base* = 0 and *Limit* set to the maximum. The instruction offset will then be the linear address, with only a single address space used—in effect, normal paging. In fact, all current operating systems for the Pentium work this way. OS/2 was the only one that used the full power of the Intel MMU architecture.

All in all, one has to give credit to the Pentium designers. Given the conflicting goals of implementing pure paging, pure segmentation, and paged segments, while at the same time being compatible with the 286, and doing all of this efficiently, the resulting design is surprisingly simple and clean.

Although we have covered the complete architecture of the Pentium virtual memory, albeit briefly, it is worth saying a few words about protection, since this subject is intimately related to the virtual memory. Just as the virtual memory scheme is closely modeled on MULTICS, so is the protection system. The Pentium supports four protection levels with level 0 being the most privileged and level 3 the least. These are shown in Fig. 4-47. At each instant, a running program is at a certain level, indicated by a 2-bit field in its PSW. Each segment in the system also has a level.

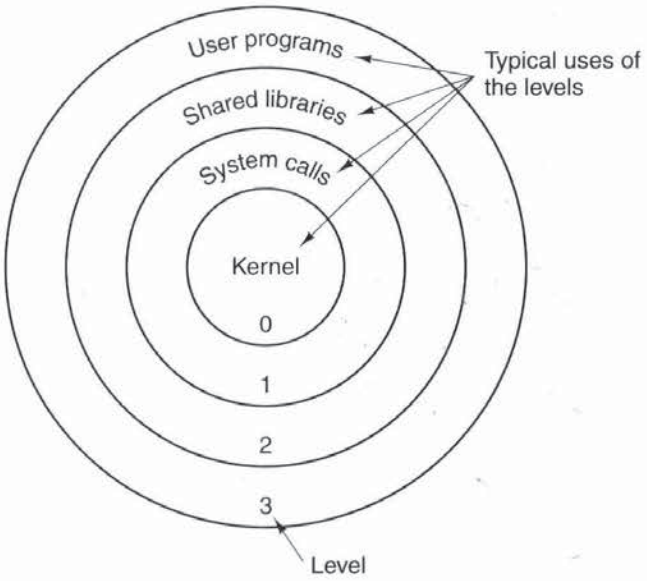


Figure 4-47. Protection on the Pentium.

As long as a program restricts itself to using segments at its own level, everything works fine. Attempts to access data at a higher level are permitted. Attempts to access data at a lower level are illegal and cause traps. Attempts to call procedures at a different level (higher or lower) are allowed, but in a carefully controlled way. To make an interlevel call, the CALL instruction must contain a selector instead of an address. This selector designates a descriptor called a **call**

gate, which gives the address of the procedure to be called. Thus it is not possible to jump into the middle of an arbitrary code segment at a different level. Only official entry points may be used. The concepts of protection levels and call gates were pioneered in MULTICS, where they were viewed as **protection rings**.

A typical use for this mechanism is suggested in Fig. 4-47. At level 0, we find the kernel of the operating system, which handles I/O, memory management, and other critical matters. At level 1, the system call handler is present. User programs may call procedures here to have system calls carried out, but only a specific and protected list of procedures may be called. Level 2 contains library procedures, possibly shared among many running programs. User programs may call these procedures and read their data, but they may not modify them. Finally, user programs run at level 3, which has the least protection.

Traps and interrupts use a mechanism similar to the call gates. They, too, reference descriptors, rather than absolute addresses, and these descriptors point to specific procedures to be executed. The *Type* field in Fig. 4-44 distinguishes between code segments, data segments, and the various kinds of gates.

4.9 RESEARCH ON MEMORY MANAGEMENT

Memory management, especially paging algorithms, was once a fruitful area for research, but most of that seems to have largely died off, at least for general-purpose systems. Most real systems tend to use some variation on clock, because it is easy to implement and relatively effective. One recent exception, however, is a redesign of the 4.4 BSD virtual memory system (Cranor and Parulkar, 1999).

There is still research going on concerning paging in special-purpose and newer kinds of systems though. Some of this work relates to allowing user processes to field their own page faults and handle their own memory management, possibly in an application-specific way (Engler et al., 1995). One area where applications may need to do their own paging in special ways is multimedia, so some research has looked at this (Hand, 1999). Another area that has some special requirements is hand-held personal communicators (Abutaleb and Li, 1997; and Wan and Lin, 1997). One final area is systems with 64-bit address spaces shared by many processes (Talluri et al., 1995).

4.10 SUMMARY

In this chapter we have examined memory management. We saw that the simplest systems do not swap or page at all. Once a program is loaded into memory, it remains there until it finishes. Some operating systems allow only one process at a time in memory, while others support multiprogramming.

The next step up is swapping. When swapping is used, the system can handle more processes than it has room for in memory. Processes for which there is no room are swapped out to the disk. Free space in memory and on disk can be kept track of with a bitmap or a hole list.

Modern computers often have some form of virtual memory. In the simplest form, each process' address space is divided up into uniform sized blocks called pages, which can be placed into any available page frame in memory. There are many page replacement algorithms; two of the better algorithms are aging and WSClock.

Paging systems can be modeled by abstracting the page reference string from the program and using the same reference string with different algorithms. These models can be used to make some predictions about paging behavior.

To make paging systems work well, choosing an algorithm is not enough; attention to issues such as determining the working set, memory allocation policy, and page size are required.

Segmentation helps in handling data structures that change size during execution and simplifies linking and sharing. It also facilitates providing different protection for different segments. Sometimes segmentation and paging are combined to provide a two-dimensional virtual memory. The MULTICS system and the Intel Pentium support segmentation and paging.

PROBLEMS

1. A computer system has enough room to hold four programs in its main memory. These programs are idle waiting for I/O half the time. What fraction of the CPU time is wasted?

2. In Fig. 4-21 we saw an example of how multiple jobs can run in parallel and finish faster than if they had run sequentially. Suppose that two jobs, each of which needs 10 minutes of CPU time, start simultaneously. How long will the last one take to complete if they run sequentially? How long if they run in parallel? Assume 50% I/O wait.

3. A swapping system eliminates holes by compaction. Assuming a random distribution of many holes and many data segments and a time to read or write a 32-bit memory word of 10 nsec, about how long does it take to compact 128 MB? For simplicity, assume that word 0 is part of a hole and that the highest word in memory contains valid data.

4. In this problem you are to compare the storage needed to keep track of free memory using a bitmap versus using a linked list. The 128-MB memory is allocated in units of

n bytes. For the linked list, assume that memory consists of an alternating sequence of segments and holes, each 64 KB. Also assume that each node in the linked list needs a 32-bit memory address, a 16-bit length, and a 16-bit next-node field. How many bytes of storage is required for each method? Which one is better?

5. Consider a swapping system in which memory consists of the following hole sizes in memory order: 10 KB, 4 KB, 20 KB, 18 KB, 7 KB, 9 KB, 12 KB, and 15 KB. Which hole is taken for successive segment requests of

(a) 12 KB
(b) 10 KB
(c) 9 KB

for first fit? Now repeat the question for best fit, worst fit, and next fit.

6. What is the difference between a physical address and a virtual address?

7. For each of the following decimal virtual addresses, compute the virtual page number and offset for a 4-KB page and for an 8 KB page: 20000, 32768, 60000.

8. Using the page table of Fig. 4-10, give the physical address corresponding to each of the following virtual addresses:

(a) 20
(b) 4100
(c) 8300

9. The Intel 8086 processor does not support virtual memory. Nevertheless, some companies previously sold systems that contained an unmodified 8086 CPU and do paging. Make an educated guess as to how they did it. *Hint*: Think about the logical location of the MMU.

10. The amount of disk space that must be available for page storage is related to the maximum number of processes, n, the number of bytes in the virtual address space, v, and the number of bytes of RAM, r. Give an expression for the worst case disk space requirements. How realistic is this amount?

11. If an instruction takes 10 nsec and a page fault takes an additional n nsec, give a formula for the effective instruction time if page faults occur every k instructions.

12. A machine has a 32-bit address space and an 8-KB page. The page table is entirely in hardware, with one 32-bit word per entry. When a process starts, the page table is copied to the hardware from memory, at one word every 100 nsec. If each process runs for 100 msec (including the time to load the page table), what fraction of the CPU time is devoted to loading the page tables?

13. A computer with a 32-bit address uses a two-level page table. Virtual addresses are split into a 9-bit top-level page table field, an 11-bit second-level page table field, and an offset. How large are the pages and how many are there in the address space?

14. Suppose that a 32-bit virtual address is broken up into four fields, a, b, c, and d. The first three are used for a three-level page table system. The fourth field, d, is the offset. Does the number of pages depend on the sizes of all four fields? If not, which ones matter and which ones do not?

15. A computer has 32-bit virtual addresses and 4-KB pages. The program and data together fit in the lowest page (0–4095) The stack fits in the highest page. How many entries are needed in the page table if traditional (one-level) paging is used? How many page table entries are needed for two-level paging, with 10 bits in each part?

16. Below is an execution trace of a program fragment for a computer with 512-byte pages. The program is located at address 1020, and its stack pointer is at 8192 (the stack grows toward 0). Give the page reference string generated by this program. Each instruction occupies 4 bytes (1 word) including immediate constants. Both instruction and data references count in the reference string.

Load word 6144 into register 0
Push register 0 onto the stack
Call a procedure at 5120, stacking the return address
Subtract the immediate constant 16 from the stack pointer
Compare the actual parameter to the immediate constant 4
Jump if equal to 5152

17. A computer whose processes have 1024 pages in their address spaces keeps its page tables in memory. The overhead required for reading a word from the page table is 5 nsec. To reduce this overhead, the computer has a TLB, which holds 32 (virtual page, physical page frame) pairs, and can do a look up in 1 nsec. What hit rate is needed to reduce the mean overhead to 2 nsec?

18. The TLB on the VAX does not contain an *R* bit. Why?

19. How can the associative memory device needed for a TLB be implemented in hardware, and what are the implications of such a design for expandability?

20. A machine has 48-bit virtual addresses and 32-bit physical addresses. Pages are 8 KB. How many entries are needed for the page table?

21. A computer with an 8-KB page, a 256-KB main memory, and a 64-GB virtual address space uses an inverted page table to implement its virtual memory. How big should the hash table be to ensure a mean hash chain length of less than 1? Assume that the hash table size is a power of two.

22. A student in a compiler design course proposes to the professor a project of writing a compiler that will produce a list of page references that can be used to implement the optimal page replacement algorithm. Is this possible? Why or why not? Is there anything that could be done to improve paging efficiency at run time?

23. If FIFO page replacement is used with four page frames and eight pages, how many page faults will occur with the reference string 0172327103 if the four frames are initially empty? Now repeat this problem for LRU.

24. Consider the page sequence of Fig. 4-16(b). Suppose that the *R* bits for the pages *B* through *A* are 11011011, respectively. Which page will second chance remove?

25. A small computer has four page frames. At the first clock tick, the *R* bits are 0111 (page 0 is 0, the rest are 1). At subsequent clock ticks, the values are 1011, 1010, 1101, 0010, 1010, 1100, and 0001. If the aging algorithm is used with an 8-bit counter, give the values of the four counters after the last tick.

26. Suppose that $\tau = 400$ in Fig. 4-21. Which page will be removed?

27. In the WSClock algorithm of Fig. 4-22(c), the hand points to a page with $R = 0$. If $\tau = 400$, will this page be removed? What about if $\tau = 1000$?

28. How long does it take to load a 64-KB program from a disk whose average seek time is 10 msec, whose rotation time is 10 msec, and whose tracks hold 32 KB

(a) for a 2-KB page size?
(b) for a 4-KB page size?

The pages are spread randomly around the disk and the number of cylinders is so large that the chance of two pages being on the same cylinder is negligible.

29. A computer has four page frames. The time of loading, time of last access, and the *R* and *M* bits for each page are as shown below (the times are in clock ticks):

Page	Loaded	Last ref.	R	M
0	126	280	1	0
1	230	265	0	01
2	140	270	0	0
3	110	285	1	1

(a) Which page will NRU replace?
(b) Which page will FIFO replace?
(c) Which page will LRU replace?
(d) Which page will second chance replace?

30. One of the first timesharing machines, the PDP-1, had a memory of 4K 18-bit words. It held one process at a time in memory. When the scheduler decided to run another process, the process in memory was written to a paging drum, with 4K 18-bit words around the circumference of the drum. The drum could start writing (or reading) at any word, rather than only at word 0. Why do you suppose this drum was chosen?

31. A computer provides each process with 65,536 bytes of address space divided into pages of 4096 bytes. A particular program has a text size of 32,768 bytes, a data size of 16,386 bytes, and a stack size of 15,870 bytes. Will this program fit in the address space? If the page size were 512 bytes, would it fit? Remember that a page may not contain parts of two different segments.

32. Can a page be in two working sets at the same time? Explain.

33. If a page is shared between two processes, is it possible that the page is read-only for one process and read-write for the other? Why or why not?

34. It has been observed that the number of instructions executed between page faults is directly proportional to the number of page frames allocated to a program. If the available memory is doubled, the mean interval between page faults is also doubled. Suppose that a normal instruction takes 1 microsec, but if a page fault occurs, it takes 2001 μsec (i.e., 2 msec to handle the fault). If a program takes 60 sec to run, during which time it gets 15,000 page faults, how long would it take to run if twice as much memory were available?

35. A group of operating system designers for the Frugal Computer Company are thinking about ways of reducing the amount of backing store needed in their new operating system. The head guru has just suggested not bothering to save the program text in the swap area at all, but just page it in directly from the binary file whenever it is needed. Under what conditions, if any, does this idea work for the program text? Under what conditions, if any, does it work for the data?

36. A machine language instruction to load a 32-bit word into a register contains the 32-bit address of the word to be loaded. What is the maximum number of page faults this instruction can cause?

37. Explain the difference between internal fragmentation and external fragmentation. Which one occurs in paging systems? Which one occurs in systems using pure segmentation?

38. When segmentation and paging are both being used, as in MULTICS, first the segment descriptor must be looked up, then the page descriptor. Does the TLB also work this way, with two levels of lookup?

39. Plot a histogram and calculate the mean and median of the sizes of executable binary files on a computer to which you have access. On a Windows system, look at all .exe and .dll files; on a UNIX system look at all executable files in */bin*, */usr/bin*, and */local/bin* that are not scripts (or use the *file* utility to find all executables). Determine the optimal page size for this computer just considering the code (not data). Consider internal fragmentation and page table size, making some reasonable assumption about the size of a page table entry. Assume that all programs are equally likely to be run and thus should be weighted equally.

40. Small programs for MS-DOS can be compiled as *.COM* files. These files are always loaded at address 0x100 in a single memory segment that is used for code, data, and stack. Instructions that transfer control of execution, such as JMP and CALL, or that access static data from fixed addresses have the addresses compiled into the object code. Write a program that can relocate such a program file to run starting at an arbitrary address. Your program must scan through code looking for object codes for instructions that refer to fixed memory addresses, then modify those addresses that point to memory locations within the range to be relocated. You can find the object codes in an assembly language programming text. Note that doing this perfectly without additional information is, in general, an impossible task, because some data words may have values that mimic instruction object codes.

41. Write a program that simulates a paging system. At the start of the program, the user should be asked to choose a page replacement algorithm, choosing from FIFO, LRU, and at least one other. On each cycle, read the number of the referenced page from a file. Generate a listing similar to Fig. 4-25, except rotated 90 degrees so that each new page reference increases the length of the output file by one line.

42. Write a program that models the distance string algorithm described in the text. The input to the program is a list of page references (contained in a file), plus the number of page frames of physical memory available. If possible, use trace data from real programs instead of randomly-generated page references. The program should maintain the stack of pages, analogous to Fig. 4-25. At each page fault, a procedure should

be called to choose a replacement page. When the run is completed, the program should plot the distance string, analogous to Fig. 4-26. Make multiple runs for different values of the memory size and see what conclusions you can draw.

5

INPUT/OUTPUT

One of the main functions of an operating system is to control all the computer's I/O (Input/Output) devices. It must issue commands to the devices, catch interrupts, and handle errors. It should also provide an interface between the devices and the rest of the system that is simple and easy to use. To the extent possible, the interface should be the same for all devices (device independence). The I/O code represents a significant fraction of the total operating system. How the operating system manages I/O is the subject of this chapter.

This chapter is organized as follows. First we will look at some of the principles of I/O hardware, and then we will look at I/O software in general. I/O software can be structured in layers, with each layer having a well-defined task to perform. We will look at these layers to see what they do and how they fit together.

Following that introduction, we will look at several I/O devices in detail: disks, clocks, keyboards, and displays. For each device we will look at its hardware and software. Finally, we will consider power management.

5.1 PRINCIPLES OF I/O HARDWARE

Different people look at I/O hardware in different ways. Electrical engineers look at it in terms of chips, wires, power supplies, motors, and all the other physical components that make up the hardware. Programmers look at the interface

presented to the software—the commands the hardware accepts, the functions it carries out, and the errors that can be reported back. In this book we are concerned with programming I/O devices, not designing, building, or maintaining them, so our interest will be restricted to how the hardware is programmed, not how it works inside. Nevertheless, the programming of many I/O devices is often intimately connected with their internal operation. In the next three sections we will provide a little general background on I/O hardware as it relates to programming. It may be regarded as a review and expansion of the introductory material in Sec. 1.4.

5.1.1 I/O Devices

I/O devices can be roughly divided into two categories: **block devices** and **character devices**. A block device is one that stores information in fixed-size blocks, each one with its own address. Common block sizes range from 512 bytes to 32,768 bytes. The essential property of a block device is that it is possible to read or write each block independently of all the other ones. Disks are the most common block devices.

If you look closely, the boundary between devices that are block addressable and those that are not is not well defined. Everyone agrees that a disk is a block addressable device because no matter where the arm currently is, it is always possible to seek to another cylinder and then wait for the required block to rotate under the head. Now consider a tape drive used for making disk backups. Tapes contain a sequence of blocks. If the tape drive is given a command to read block *N*, it can always rewind the tape and go forward until it comes to block *N*. This operation is analogous to a disk doing a seek, except that it takes much longer. Also, it may or may not be possible to rewrite one block in the middle of a tape. Even if it were possible to use tapes as random access block devices, that is stretching the point somewhat: they are normally not used that way.

The other type of I/O device is the character device. A character device delivers or accepts a stream of characters, without regard to any block structure. It is not addressable and does not have any seek operation. Printers, network interfaces, mice (for pointing), rats (for psychology lab experiments), and most other devices that are not disk-like can be seen as character devices.

This classification scheme is not perfect. Some devices just do not fit in. Clocks, for example, are not block addressable. Nor do they generate or accept character streams. All they do is cause interrupts at well-defined intervals. Memory-mapped screens do not fit the model well either. Still, the model of block and character devices is general enough that it can be used as a basis for making some of the operating system software dealing with I/O device independent. The file system, for example, deals just with abstract block devices and leaves the device-dependent part to lower-level software.

I/O devices cover a huge range in speeds, which puts considerable pressure on the software to perform well over many orders of magnitude in data rates. Fig. 5-1 shows the data rates of some common devices. Most of these devices tend to get faster as time goes on.

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Telephone channel	8 KB/sec
Dual ISDN lines	16 KB/sec
Laser printer	100 KB/sec
Scanner	400 KB/sec
Classic Ethernet	1.25 MB/sec
USB (Universal Serial Bus)	1.5 MB/sec
Digital camcorder	4 MB/sec
IDE disk	5 MB/sec
40x CD-ROM	6 MB/sec
Fast Ethernet	12.5 MB/sec
ISA bus	16.7 MB/sec
EIDE (ATA-2) disk	16.7 MB/sec
FireWire (IEEE 1394)	50 MB/sec
XGA Monitor	60 MB/sec
SONET OC-12 network	78 MB/sec
SCSI Ultra 2 disk	80 MB/sec
Gigabit Ethernet	125 MB/sec
Ultrium tape	320 MB/sec
PCI bus	528 MB/sec
Sun Gigaplane XB backplane	20 GB/sec

Figure 5-1. Some typical device, network, and bus data rates.

5.1.2 Device Controllers

I/O units typically consist of a mechanical component and an electronic component. It is often possible to separate the two portions to provide a more modular and general design. The electronic component is called the **device controller** or **adapter**. On personal computers, it often takes the form of a printed circuit card

that can be inserted into an expansion slot. The mechanical component is the device itself. This arrangement is shown in Fig. 1-5.

The controller card usually has a connector on it, into which a cable leading to the device itself can be plugged. Many controllers can handle two, four, or even eight identical devices. If the interface between the controller and device is a standard interface, either an official ANSI, IEEE, or ISO standard or a de facto one, then companies can make controllers or devices that fit that interface. Many companies, for example, make disk drives that match the IDE or SCSI interface.

The interface between the controller and the device is often a very low-level interface. A disk, for example, might be formatted with 256 sectors of 512 bytes per track. What actually comes off the drive, however, is a serial bit stream, starting with a **preamble**, then the 4096 bits in a sector, and finally a checksum, also called an **Error-Correcting Code** (**ECC**). The preamble is written when the disk is formatted and contains the cylinder and sector number, the sector size, and similar data, as well as synchronization information.

The controller's job is to convert the serial bit stream into a block of bytes and perform any error correction necessary. The block of bytes is typically first assembled, bit by bit, in a buffer inside the controller. After its checksum has been verified and the block declared to be error free, it can then be copied to main memory.

The controller for a monitor also works as a bit serial device at an equally low level. It reads bytes containing the characters to be displayed from memory and generates the signals used to modulate the CRT beam to cause it to write on the screen. The controller also generates the signals for making the CRT beam do a horizontal retrace after it has finished a scan line, as well as the signals for making it do a vertical retrace after the entire screen has been scanned. If it were not for the CRT controller, the operating system programmer would have to explicitly program the analog scanning of the tube. With the controller, the operating system initializes the controller with a few parameters, such as the number of characters or pixels per line and number of lines per screen, and lets the controller take care of actually driving the beam.

5.1.3 Memory-Mapped I/O

Each controller has a few registers that are used for communicating with the CPU. By writing into these registers, the operating system can command the device to deliver data, accept data, switch itself on or off, or otherwise perform some action. By reading from these registers, the operating system can learn what the device's state is, whether it is prepared to accept a new command, and so on.

In addition to the control registers, many devices have a data buffer that the operating system can read and write. For example, a common way for computers to display pixels on the screen is to have a video RAM, which is basically just a data buffer, available for programs or the operating system to write into.

The issue thus arises of how the CPU communicates with the control registers and the device data buffers. Two alternatives exist. In the first approach, each control register is assigned an **I/O port** number, an 8- or 16-bit integer. Using a special I/O instruction such as

```
IN REG,PORT,
```

the CPU can read in control register PORT and store the result in CPU register REG. Similarly, using

```
OUT PORT,REG
```

the CPU can write the contents of REG to a control register. Most early computers, including nearly all mainframes, such as the IBM 360 and all of its successors, worked this way.

In this scheme, the address spaces for memory and I/O are different, as shown in Fig. 5-2(a). The instructions

```
IN R0,4
```

and

```
MOV R0,4
```

are completely different in this design. The former reads the contents of I/O port 4 and puts it in R0 whereas the latter reads the contents of memory word 4 and puts it in R0. The 4s in these examples thus refer to different and unrelated address spaces.

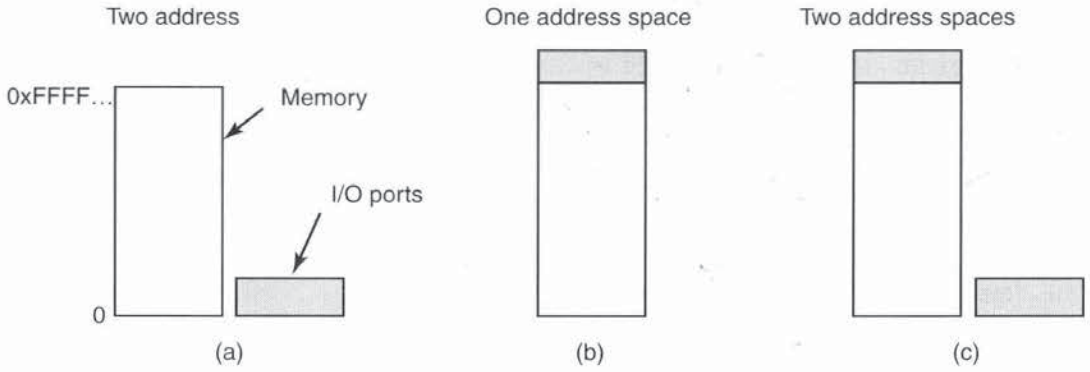


Figure 5-2. (a) Separate I/O and memory space. (b) Memory-mapped I/O. (c) Hybrid.

The second approach, introduced with the PDP-11, is to map all the control registers into the memory space, as shown in Fig. 5-2(b). Each control register is assigned a unique memory address to which no memory is assigned. This system is called **memory-mapped I/O**. Usually, the assigned addresses are at the top of the address space. A hybrid scheme, with memory-mapped I/O data buffers and

separate I/O ports for the control registers is shown in Fig. 5-2(c). The Pentium uses this architecture, with addresses 640K to 1M being reserved for device data buffers in IBM PC compatibles, in addition to I/O ports 0 through 64K.

How do these schemes work? In all cases, when the CPU wants to read a word, either from memory or from an I/O port, it puts the address it needs on the bus' address lines and then asserts a READ signal on a bus' control line. A second signal line is used to tell whether I/O space or memory space is needed. If it is memory space, the memory responds to the request. If it is I/O space, the I/O device responds to the request. If there is only memory space [as in Fig. 5-2(b)], every memory module and every I/O device compares the address lines to the range of addresses that it services. It the address falls in its range, it responds to the request. Since no address is ever assigned to both memory and an I/O device, there is no ambiguity and no conflict.

The two schemes for addressing the controllers have different strengths and weaknesses. Let us start with the advantages of memory-mapped I/O. First, if special I/O instructions are needed to read and write the device control registers, access to them requires the use of assembly code since there is no way to execute an IN or OUT instruction in C or C++. Calling such a procedure adds overhead to controlling I/O. In contrast, with memory-mapped I/O, device control registers are just variables in memory and can be addressed in C the same way as any other variables. Thus with memory-mapped I/O, a I/O device driver can be written entirely in C. Without memory-mapped I/O, some assembly code is needed.

Second, with memory-mapped I/O, no special protection mechanism is needed to keep user processes from performing I/O. All the operating system has to do is refrain from putting that portion of the address space containing the control registers in any user's virtual address space. Better yet, if each device has its control registers on a different page of the address space, the operating system can give a user control over specific devices but not others by simply including the desired pages in its page table. Such a scheme can allow different device drivers to be placed in different address spaces, not only reducing kernel size but also keeping one driver from interfering with others.

Third, with memory-mapped I/O, every instruction that can reference memory can also reference control registers. For example, if there is an instruction, TEST, that tests a memory word for 0, it can also be used to test a control register for 0, which might be the signal that the device is idle and can accept a new command. The assembly language code might look like this:

```
LOOP:  TEST PORT_4          // check if port 4 is 0
       BEQ READY            // if it is 0, go to ready
       BRANCH LOOP          // otherwise, continue testing
READY:
```

If memory-mapped I/O is not present, the control register must first be read into the CPU, then tested, requiring two instructions instead of one. In the case of the

loop given above, a fourth instruction has to be added, slightly slowing down the responsiveness of detecting an idle device.

In computer design, practically everything involves trade-offs, and that is the case here too. Memory-mapped I/O also has its disadvantages. First, most computers nowadays have some form of caching of memory words. Caching a device control register would be disastrous. Consider the assembly code loop given above in the presence of caching. The first reference to PORT_4 would cause it to be cached. Subsequent references would just take the value from the cache and not even ask the device. Then when the device finally became ready, the software would have no way of finding it out. Instead, the loop would go on forever.

To prevent this situation with memory-mapped I/O, the hardware has to be equipped with the ability to selectively disable caching, for example, on a per page basis. This feature adds extra complexity to both the hardware and the operating system, which has to manage the selective caching.

Second, if there is only one address space, then all memory modules and all I/O devices must examine all memory references to see which ones to respond to. If the computer has a single bus, as in Fig. 5-3(a), having everyone look at every address is straightforward.

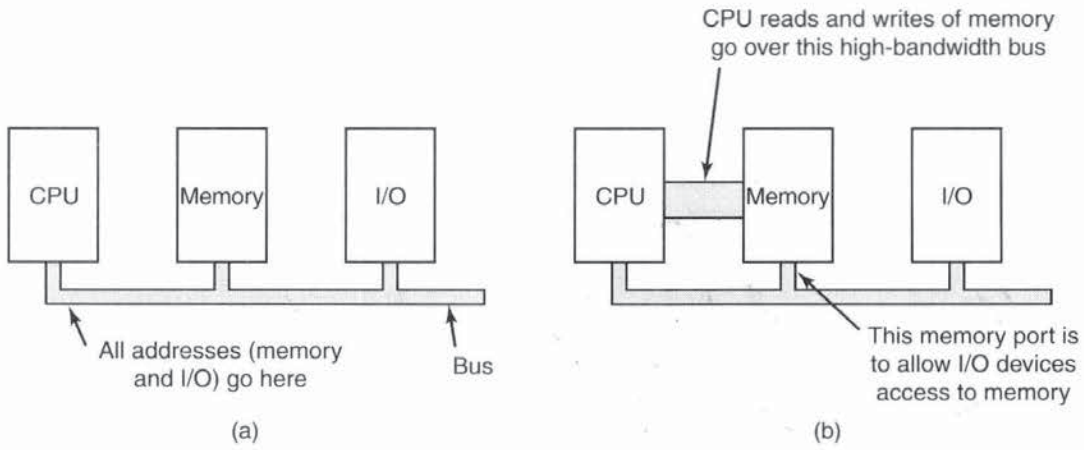


Figure 5-3. (a) A single-bus architecture. (b) A dual-bus memory architecture.

However, the trend in modern personal computers is to have a dedicated high-speed memory bus as shown in Fig. 5-3(b), a property also found in mainframes, incidentally. This bus is tailored for optimize memory performance, with no compromises for the sake of slow I/O devices. Pentium systems even have three external buses (memory, PCI, ISA), as shown in Fig. 1-11.

The trouble with having a separate memory bus on memory-mapped machines is that the I/O devices have no way of seeing memory addresses as they go by on the memory bus, so they have no way of responding. Again, special measures have to be taken to make memory-mapped I/O work on a system with multiple buses. One possibility is to first send all memory references to the memory. If

the memory fails to respond, then the CPU tries the other buses. This design can be made to work but requires additional hardware complexity.

A second possible design is to put a snooping device on the memory bus to pass all addresses presented to potentially interested I/O devices. The problem here is that I/O devices may not be able to process requests at the speed the memory can.

A third possible design, which is the one used on the Pentium configuration of Fig. 1-11, is to filter addresses in the PCI bridge chip. This chip contains range registers that are preloaded at boot time. For example, 640K to 1M could be marked as a nonmemory range. Addresses that fall within one of the ranges marked as nonmemory are forwarded onto the PCI bus instead of to memory. The disadvantage of this scheme is the need for figuring out at boot time which memory addresses are not really memory addresses. Thus each scheme has arguments for and against it, so compromises and trade-offs are inevitable.

5.1.4 Direct Memory Access (DMA)

No matter whether a CPU does or does not have memory-mapped I/O, it needs to address the device controllers to exchange data with them. The CPU can request data from an I/O controller one byte at a time but doing so wastes the CPU's time, so a different scheme, called **DMA** (**Direct Memory Access**) is often used. The operating system can only use DMA if the hardware has a DMA controller, which most systems do. Sometimes this controller is integrated into disk controllers and other controllers, but such a design requires a separate DMA controller for each device. More commonly, a single DMA controller is available (e.g., on the parentboard) for regulating transfers to multiple devices, often concurrently.

No matter where it is physically located, the DMA controller has access to the system bus independent of the CPU, as shown in Fig. 5-4. It contains several registers that can be written and read by the CPU. These include a memory address register, a byte count register, and one or more control registers. The control registers specify the I/O port to use, the direction of the transfer (reading from the I/O device or writing to the I/O device), the transfer unit (byte at a time or word at a time), and the number of bytes to transfer in one burst.

To explain how DMA works, let us first look at how disk reads occur when DMA is not used. First the controller reads the block (one or more sectors) from the drive serially, bit by bit, until the entire block is in the controller's internal buffer. Next, it computes the checksum to verify that no read errors have occurred. Then the controller causes an interrupt. When the operating system starts running, it can read the disk block from the controller's buffer a byte or a word at a time by executing a loop, with each iteration reading one byte or word from a controller device register and storing it in main memory.

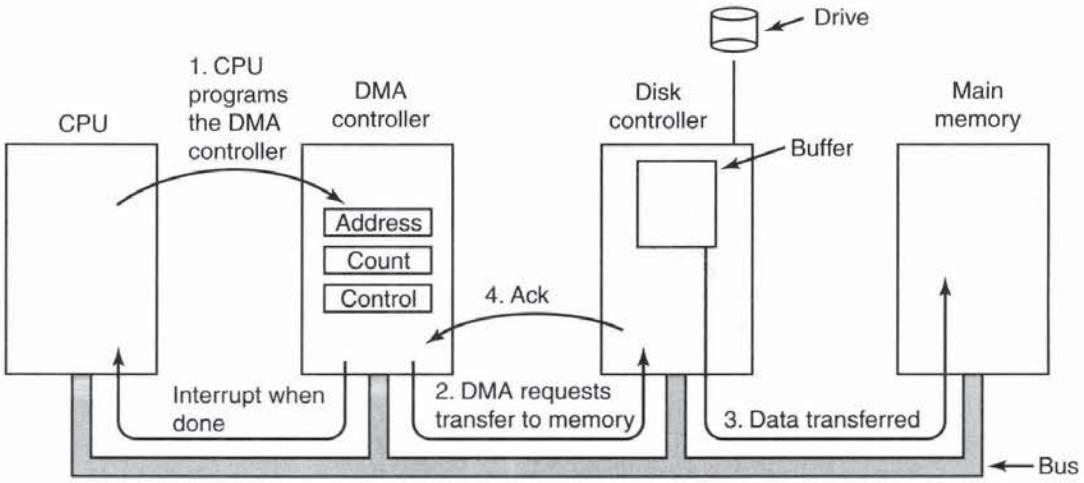


Figure 5-4. Operation of a DMA transfer.

When DMA is used, the procedure is different. First the CPU programs the DMA controller by setting its registers so it knows what to transfer where (step 1 in Fig. 5-4). It also issues a command to the disk controller telling it to read data from the disk into its internal buffer and verify the checksum. When valid data are in the disk controller's buffer, DMA can begin.

The DMA controller initiates the transfer by issuing a read request over the bus to the disk controller (step 2). This read request looks like any other read request, and the disk controller does not know or care whether it came from the CPU or from a DMA controller. Typically, the memory address to write to is on the bus' address lines so when the disk controller fetches the next word from its internal buffer, it knows where to write it. The write to memory is another standard bus cycle (step 3). When the write is complete, the disk controller sends an acknowledgement signal to the disk controller, also over the bus (step 4). The DMA controller then increments the memory address to use and decrements the byte count. If the byte count is still greater than 0, steps 2 through 4 are repeated until the count reaches 0. At that time, the DMA controller interrupts the CPU to let it know that the transfer is now complete. When the operating system starts up, it does not have to copy the disk block to memory; it is already there.

DMA controllers vary considerably in their sophistication. The simplest ones handle one transfer at a time, as described above. More complex ones can be programmed to handle multiple transfers at once. Such controllers have multiple sets of registers internally, one for each channel. The CPU starts by loading each set of registers with the relevant parameters for its transfer. Each transfer must use a different device controller. After each word is transferred (steps 2 through 4) in Fig. 5-4, the DMA controller decides which device to service next. It may be set up to use a round-robin algorithm, or it may have a priority scheme design to favor some devices over others. Multiple requests to different device controllers

may be pending at the same time, provided that there is an unambiguous way to tell the acknowledgements apart. Often a different acknowledgement line on the bus is used for each DMA channel for this reason.

Many buses can operate in two modes: word-at-a-time mode and block mode. Some DMA controllers can also operate in either mode. In the former mode, the operation is as described above: the DMA controller requests for the transfer of one word and gets it. If the CPU also wants the bus, it has to wait. The mechanism is called **cycle stealing** because the device controller sneaks in and steals an occasional bus cycle from the CPU once in a while, delaying it slightly. In block mode, the DMA controller tells the device to acquire the bus, issue a series of transfers, then release the bus. This form of operation is called **burst mode**. It is more efficient than cycle stealing because acquiring the bus takes time and multiple words can be transferred for the price of one bus acquisition. The down side to burst mode is that it can block the CPU and other devices for a substantial period of time if a long burst is being transferred.

In the model we have been discussing, sometimes called **fly-by mode**, the DMA controller tells the device controller to transfer the data directly to main memory. An alternative mode that some DMA controllers use is to have the device controller send the word to the DMA controller, which then issues a second bus request to write the word to wherever it is supposed to go. This scheme requires an extra bus cycle per word transferred, but is more flexible in that it can also perform device-to-device copies and even memory-to-memory copies (by first issuing a read to memory and then issuing a write to memory at a different address).

Most DMA controllers use physical memory addresses for their transfers. Using physical addresses requires the operating system to convert the virtual address of the intended memory buffer into a physical address and write this physical address into the DMA controller's address register. An alternative scheme used in a few DMA controllers is to write virtual addresses into the DMA controller instead. Then the DMA controller must use the MMU to have the virtual-to-physical translation done. Only in the case that the MMU is part of the memory (possible, but rare) rather than part of the CPU, can virtual addresses be put on the bus.

We mentioned earlier that the disk first reads data into its internal buffer before DMA can start. You may be wondering why the controller does not just store the bytes in main memory as soon as it gets them from the disk. In other words, why does it need an internal buffer? There are two reasons. First, by doing internal buffering, the disk controller can verify the checksum before starting a transfer. If the checksum is incorrect, an error is signaled and no transfer is done.

The second reason is that once a disk transfer has started, the bits keep arriving from the disk at a constant rate, whether the controller is ready for them or not. If the controller tried to write data directly to memory, it would have to go

over the system bus for each word transferred. If the bus were busy due to some other device using it (e.g., in burst mode), the controller would have to wait. If the next disk word arrived before the previous one had been stored, the controller would have to store it somewhere. If the bus were very busy, the controller might end up storing quite a few words and having a lot of administration to do as well. When the block is buffered internally, the bus is not needed until the DMA begins, so the design of the controller is much simpler because the DMA transfer to memory is not time critical. (Some older controllers did, in fact, go directly to memory with only a small amount of internal buffering, but when the bus was very busy, a transfer might have had to be terminated with an overrun error.)

Not all computers use DMA. The argument against it is that the main CPU is often far faster than the DMA controller and can do the job much faster (when the limiting factor is not the speed of the I/O device). If there is no other work for it to do, having the (fast) CPU wait for the (slow) DMA controller to finish is pointless. Also, getting rid of the DMA controller and having the CPU do all the work in software saves money, important on low-end (embedded) computers.

5.1.5 Interrupts Revisited

We briefly introduced interrupts in Sec. 1.4.3, but there is more to be said. In a typical personal computer system, the interrupt structure is as shown in Fig. 5-5. At the hardware level, interrupts work as follows. When an I/O device has finished the work given to it, it causes an interrupt (assuming that interrupts have been enabled by the operating system). It does this by asserting a signal on a bus line that it has been assigned. This signal is detected by the interrupt controller chip on the parentboard, which then decides what to do.

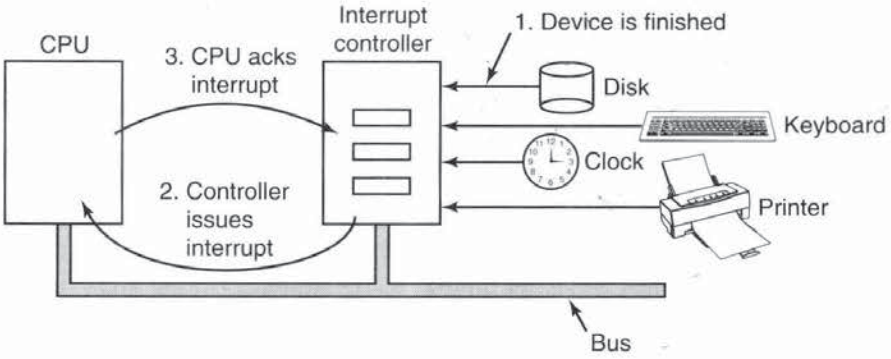


Figure 5-5. How an interrupt happens. The connections between the devices and the interrupt controller actually use interrupt lines on the bus rather than dedicated wires.

If no other interrupts are pending, the interrupt controller processes the interrupt immediately. If another one is in progress, or another device has made a simultaneous request on a higher-priority interrupt request line on the bus, the

device is just ignored for the moment. In this case it continues to assert an interrupt signal on the bus until it is serviced by the CPU.

To handle the interrupt, the controller puts a number on the address lines specifying which device wants attention and asserts a signal that interrupts the CPU.

The interrupt signal causes the CPU to stop what it is doing and start doing something else. The number on the address lines is used as an index into a table called the **interrupt vector** to fetch a new program counter. This program counter points to the start of the corresponding interrupt service procedure. Typically traps and interrupts use the same mechanism from this point on, and frequently share the same interrupt vector. The location of the interrupt vector can be hardwired into the machine or it can be anywhere in memory, with a CPU register (loaded by the operating system) pointing to its origin.

Shortly after it starts running, the interrupt service procedure acknowledges the interrupt by writing a certain value to one of the interrupt controller's I/O ports. This acknowledgement tells the controller that it is free to issue another interrupt. By having the CPU delay this acknowledgement until it is ready to handle the next interrupt, race conditions involving multiple almost simultaneous interrupts can be avoided. As an aside, some (older) computers do not have a centralized interrupt controller chip, so each device controller requests its own interrupts.

The hardware always saves certain information before starting the service procedure. Which information is saved and where it is saved varies greatly from CPU to CPU. As a bare minimum, the program counter must be saved, so the interrupted process can be restarted. At the other extreme, all the visible registers and a large number of internal registers may be saved as well.

One issue is where to save this information. One option is to put it in internal registers that the operating system can read out as needed. A problem with this approach is that then the interrupt controller cannot be acknowledged until all potentially relevant information has been read out, lest a second interrupt overwrite the internal registers saving the state. This strategy leads to long dead times when interrupts are disabled and possibly lost interrupts and lost data.

Consequently, most CPUs save the information on the stack. However, this approach, too, has problems. To start with: whose stack? If the current stack is used, it may well be a user process stack. The stack pointer may not even be legal, which would cause a fatal error when the hardware tried to write some words at it. Also, it might point to the end of a page. After several memory writes, the page boundary might be exceeded and a page fault generated. Having a page fault occur during the hardware interrupt processing creates a bigger problem: where to save the state to handle the page fault?

If the kernel stack is used, there is a much better chance of the stack pointer being legal and pointing to a pinned page. However, switching into kernel mode may require changing MMU contexts and will probably invalidate most or all of

the cache and TLB. Reloading all of these, statically or dynamically will increase the time to process an interrupt and thus waste CPU time.

Another problem is caused by the fact that most modern CPUs are heavily pipelined and often superscalar (internally parallel). In older systems, after each instruction was finished executing, the microprogram or hardware checked to see if there was an interrupt pending. If so, the program counter and PSW were pushed onto the stack and the interrupt sequence begun. After the interrupt handler ran, the reverse process took place, with the old PSW and program counter popped from the stack and the previous process continued.

This model makes the implicit assumption that if an interrupt occurs just after some instruction, all the instructions up to and including that instruction have been executed completely, and no instructions after it have executed at all. On older machines, this assumption was always valid. On modern ones it may not be.

For starters, consider the pipeline model of Fig. 1-6(a). What happens if an interrupt occurs while the pipeline is full (the usual case)? Many instructions are in various stages of execution. When the interrupt occurs, the value of the program counter may not reflect the correct boundary between executed instructions and nonexecuted instructions. More likely, it reflects the address of the next instruction to be fetched and pushed into the pipeline rather than the address of the instruction that just was processed by the execution unit.

As a consequence, there may be a well-defined boundary between instructions that have actually executed and those that have not, but the hardware may not know what it is. Consequently, when the operating system must return from an interrupt, it cannot just start filling the pipeline from the address contained in the program counter. It must figure out what the last executed instruction was, often a complex task that may require analyzing the state of the machine.

Although this situation is bad, interrupts on a superscalar machine, such as that of Fig. 1-6(b) are far worse. Because instructions may execute out of order, there may be no well-defined boundary between the executed and nonexecuted instructions. It may well be that instructions 1, 2, 3, 5, and 8 have executed, but instructions 4, 6, 7, 9, 10, and beyond have not. Furthermore, the program counter may now be pointing to instruction 9, 10, or 11.

An interrupt that leaves the machine in a well-defined state is called a **precise interrupt** (Walker and Cragon, 1995). Such an interrupt has four properties:

1. The PC (Program Counter) is saved in a known place.
2. All instructions before the one pointed to by the PC have fully executed.
3. No instruction beyond the one pointed to by the PC has been executed.
4. The execution state of the instruction pointed to by the PC is known.

Note that there is no prohibition on instructions beyond the one pointed to by the PC from starting. It is just that any changes they make to registers or memory

must be undone before the interrupt happens. It is permitted that the instruction pointed to has been executed. It is also permitted that it has not been executed. However, it must be clear which case applies. Often, if the interrupt is an I/O interrupt, the instruction will not yet have started. However, if the interrupt is really a trap or page fault, then the PC generally points to the instruction that caused the fault so it can be restarted later.

An interrupt that does not meet these requirements is called an **imprecise interrupt** and makes life extremely unpleasant for the operating system writer, who now has to figure out what has happened and what still has to happen. Machines with imprecise interrupts usually vomit a large amount of internal state onto the stack to give the operating system the possibility of figuring out what was going on. Saving a large amount of information to memory on every interrupt makes interrupts slow and recovery even worse. This leads to the ironic situation of having very fast superscalar CPUs sometimes being unsuitable for real-time work due to slow interrupts.

Some computers are designed so that some kinds of interrupts and traps are precise and others are not. For example, having I/O interrupts be precise but traps due to fatal programming errors be imprecise is not so bad since no attempt need be made to restart the running process. Some machines have a bit that can be set to force all interrupts to be precise. The downside of setting this bit is that it forces the CPU to carefully log everything it is doing and maintain shadow copies of registers so it can generate a precise interrupt at any instant. All this overhead has a major impact on performance.

Some superscalar machines, such as the Pentium Pro and all of its successors, have precise interrupts to allow old 386, 486, and Pentium I programs to work correctly (superscalar was introduced in the Pentium Pro; the Pentium I just had two pipelines). The price paid for precise interrupts is extremely complex interrupt logic within the CPU to make sure that when the interrupt controller signals that it wants to cause an interrupt, all instructions up to some point are allowed to finish and none beyond that point are allowed to have any noticeable effect on the machine state. Here the price is paid not in time, but in chip area and in complexity of the design. If precise interrupts were not required for backward compatibility purposes, this chip area would be available for larger on-chip caches, making the CPU faster. On the other hand, imprecise interrupts make the operating system far more complicated and slower, so it is hard to tell which approach is really better.

5.2 PRINCIPLES OF I/O SOFTWARE

Let us now turn away from the I/O hardware and look at the I/O software. First we will look at the goals of the I/O software and then at the different ways I/O can be done from the point of view of the operating system.

5.2.1 Goals of the I/O Software

A key concept in the design of I/O software is known as **device independence**. What it means is that it should be possible to write programs that can access any I/O device without having to specify the device in advance. For example, a program that reads a file as input should be able to read a file on a floppy disk, on a hard disk, or on a CD-ROM, without having to modify the program for each different device. Similarly, one should be able to type a command such as

```
sort <input >output
```

and have it work with input coming from a floppy disk, an IDE disk, a SCSI disk, or the keyboard, and the output going to any kind of disk or the screen. It is up to the operating system to take care of the problems caused by the fact that these devices really are different and require very different command sequences to read or write.

Closely related to device independence is the goal of **uniform naming**. The name of a file or a device should simply be a string or an integer and not depend on the device in any way. In UNIX, all disks can be integrated in the file system hierarchy in arbitrary ways so the user need not be aware of which name corresponds to which device. For example, a floppy disk can be **mounted** on top of the directory */usr/ast/backup* so that copying a file to */usr/ast/backup/monday* copies the file to the floppy disk. In this way, all files and devices are addressed the same way: by a path name.

Another important issue for I/O software is **error handling**. In general, errors should be handled as close to the hardware as possible. If the controller discovers a read error, it should try to correct the error itself if it can. If it cannot, then the device driver should handle it, perhaps by just trying to read the block again. Many errors are transient, such as read errors caused by specks of dust on the read head, and will go away if the operation is repeated. Only if the lower layers are not able to deal with the problem should the upper layers be told about it. In many cases, error recovery can be done transparently at a low level without the upper levels even knowing about the error.

Still another key issue is **synchronous** (blocking) versus **asynchronous** (interrupt-driven) transfers. Most physical I/O is asynchronous—the CPU starts the transfer and goes off to do something else until the interrupt arrives. User programs are much easier to write if the I/O operations are blocking—after a read system call the program is automatically suspended until the data are available in the buffer. It is up to the operating system to make operations that are actually interrupt-driven look blocking to the user programs.

Another issue for the I/O software is **buffering**. Often data that come off a device cannot be stored directly in its final destination. For example, when a packet comes in off the network, the operating system does not know where to put it until it has stored the packet somewhere and examined it. Also, some devices

have severe real-time constraints (for example, digital audio devices), so the data must be put into an output buffer in advance to decouple the rate at which the buffer is filled from the rate at which it is emptied, in order to avoid buffer underruns. Buffering involves considerable copying and often has a major impact on I/O performance.

The final concept that we will mention here is sharable versus dedicated devices. Some I/O devices, such as disks, can be used by many users at the same time. No problems are caused by multiple users having open files on the same disk at the same time. Other devices, such as tape drives, have to be dedicated to a single user until that user is finished. Then another user can have the tape drive. Having two or more users writing blocks intermixed at random to the same tape will definitely not work. Introducing dedicated (unshared) devices also introduces a variety of problems, such as deadlocks. Again, the operating system must be able to handle both shared and dedicated devices in a way that avoids problems.

5.2.2 Programmed I/O

There are three fundamentally different ways that I/O can be performed. In this section we will look at the first one (programmed I/O). In the next two sections we will examine the others (interrupt-driven I/O and I/O using DMA). The simplest form of I/O is to have the CPU do all the work. This method is called **programmed I/O**.

It is simplest to illustrate programmed I/O by means of an example. Consider a user process that wants to print the eight-character string "ABCDEFGH" on the printer. It first assembles the string in a buffer in user space, as shown in Fig. 5-6(a).

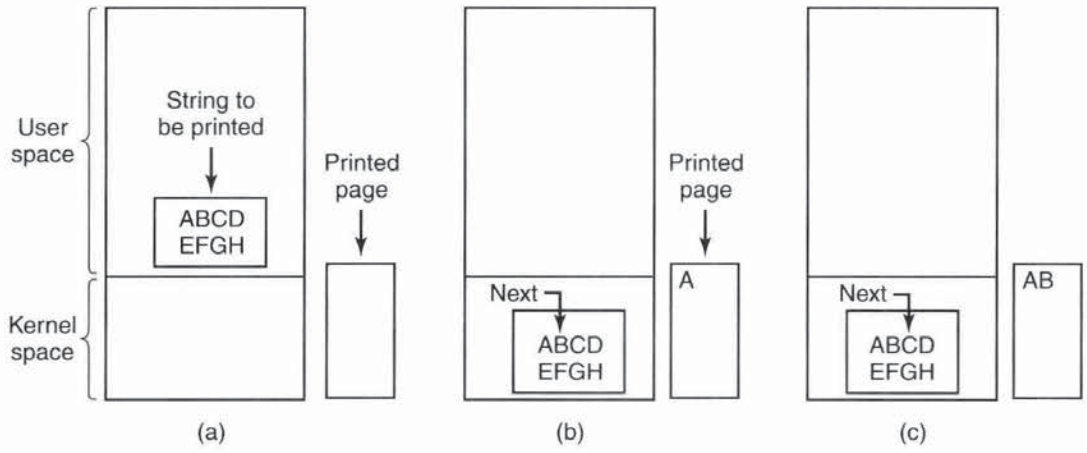


Figure 5-6. Steps in printing a string.

The user process then acquires the printer for writing by making a system call to open it. If the printer is currently in use by another process, this call will fail

and return an error code or will block until the printer is available, depending on the operating system and the parameters of the call. Once it has the printer, the user process makes a system call telling the operating system to print the string on the printer.

The operating system then (usually) copies the buffer with the string to an array, say, *p*, in kernel space, where it is more easily accessed (because the kernel may have to change the memory map to get at user space). It then checks to see if the printer is currently available. If not, it waits until it is available. As soon as the printer is available, the operating system copies the first character to the printer's data register, in this example using memory-mapped I/O. This action activates the printer. The character may not appear yet because some printers buffer a line or a page before printing anything. In Fig. 5-6(b), however, we see that the first character has been printed and that the system has marked the "B" as the next character to be printed.

As soon as it has copied the first character to the printer, the operating system checks to see if the printer is ready to accept another one. Generally, the printer has a second register, which gives its status. The act of writing to the data register causes the status to become not ready. When the printer controller has processed the current character, it indicates its availability by setting some bit in its status register or putting some value in it.

At this point the operating system waits for the printer to become ready again. When that happens, it prints the next character, as shown in Fig. 5-6(c). This loop continues until the entire string has been printed. Then control returns to the user process.

The actions followed by the operating system are summarized in Fig. 5-7. First the data are copied to the kernel. Then the operating system enters a tight loop outputting the characters one at a time. The essential aspect of programmed I/O, clearly illustrated in this figure, is that after outputting a character, the CPU continuously polls the device to see if it is ready to accept another one. This behavior is often called **polling** or **busy waiting**.

```
copy_from_user(buffer, p, count);                /* p is the kernel bufer */
for (i = 0; i < count; i++) {                    /* loop on every character */
     while (*printer_status_reg != READY) ;      /* loop until ready */
     *printer_data_register = p[i];              /* output one character */
}
return_to_user( );
```

Figure 5-7. Writing a string to the printer using programmed I/O.

Programmed I/O is simple but has the disadvantage of tying up the CPU full time until all the I/O is done. If the time to "print" a character is very short (because all the printer is doing is copying the new character to an internal buffer), then

busy waiting is fine. Also, in an embedded system, where the CPU has nothing else to do, busy waiting is reasonable. However, in more complex systems, where the CPU has other work to do, busy waiting is inefficient. A better I/O method is needed.

5.2.3 Interrupt-Driven I/O

Now let us consider the case of printing on a printer that does not buffer characters but prints each one as it arrives. If the printer can print, say 100 characters/sec, each character takes 10 msec to print. This means that after every character is written to the printer's data register, the CPU will sit in an idle loop for 10 msec waiting to be allowed to output the next character. This is more than enough time to do a context switch and run some other process for the 10 msec that would otherwise be wasted.

The way to allow the CPU to do something else while waiting for the printer to become ready is to use interrupts. When the system call to print the string is made, the buffer is copied to kernel space, as we showed earlier, and the first character is copied to the printer as soon as it is willing to accept a character. At that point the CPU calls the scheduler and some other process is run. The process that asked for the string to be printed is blocked until the entire string has printed. The work done on the system call is shown in Fig. 5-8(a).

```
copy_from_user(buffer, p, count);
enable_interrupts( );
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler( );
```

(a)

```
if (count == 0) {
    unblock_user( );
} else {
    *printer_data_register = p[i];
    count = count – 1;
    i = i + 1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

(b)

Figure 5-8. Writing a string to the printer using interrupt-driven I/O. (a) Code executed when the print system call is made. (b) Interrupt service procedure.

When the printer has printed the character and is prepared to accept the next one, it generates an interrupt. This interrupt stops the current process and saves its state. Then the printer interrupt service procedure is run. A crude version of this code is shown in Fig. 5-8(b). If there are no more characters to print, the interrupt handler takes some action to unblock the user. Otherwise, it outputs the next character, acknowledges the interrupt, and returns to the process that was running just before the interrupt, which continues from where it left off.

5.2.4 I/O Using DMA

An obvious disadvantage of interrupt-driven I/O is that an interrupt occurs on every character. Interrupts take time, so this scheme wastes a certain amount of CPU time. A solution is to use DMA. Here the idea is to let the DMA controller feed the characters to the printer one at time, without the CPU being bothered. In essence, DMA is programmed I/O, only with the DMA controller doing all the work, instead of the main CPU. An outline of the code is given in Fig. 5-9.

```
copy_from_user(buffer, p, count);
set_up_DMA_controller( );
scheduler( );
```

(a)

```
acknowledge_interrupt( );
unblock_user( );
return_from_interrupt( );
```

(b)

Figure 5-9. Printing a string using DMA. (a) Code executed when the print system call is made. (b) Interrupt service procedure.

The big win with DMA is reducing the number of interrupts from one per character to one per buffer printed. If there are many characters and interrupts are slow, this can be a major improvement. On the other hand, the DMA controller is usually much slower than the main CPU. If the DMA controller is not capable of driving the device at full speed, or the CPU usually has nothing to do anyway while waiting for the DMA interrupt, then interrupt-driven I/O or even programmed I/O may be better.

5.3 I/O SOFTWARE LAYERS

I/O software is typically organized in four layers, as shown in Fig. 5-10. Each layer has a well-defined function to perform and a well-defined interface to the adjacent layers. The functionality and interfaces differ from system to system, so the discussion that follows, which examines all the layers starting at the bottom, is not specific to one machine.

5.3.1 Interrupt Handlers

While programmed I/O is occasionally useful, for most I/O, interrupts are an unpleasant fact of life and cannot be avoided. They should be hidden away, deep in the bowels of the operating system, so that as little of the operating system as possible knows about them. The best way to hide them is to have the driver starting an I/O operation block until the I/O has completed and the interrupt occurs. The driver can block itself by doing a down on a semaphore, a wait on a condition variable, a receive on a message, or something similar, for example.

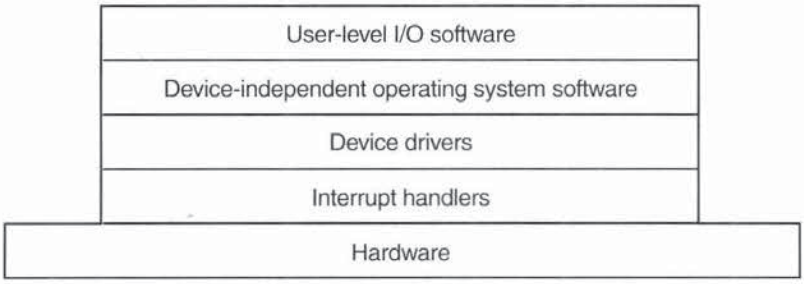


Figure 5-10. Layers of the I/O software system.

When the interrupt happens, the interrupt procedure does whatever it has to in order to handle the interrupt. Then it can unblock the driver that started it. In some cases it will just complete up on a semaphore. In others it will do a signal on a condition variable in a monitor. In still others, it will send a message to the blocked driver. In all cases the net effect of the interrupt will be that a driver that was previously blocked will now be able to run. This model works best if drivers are structured as kernel processes, with their own states, stacks, and program counters.

Of course, reality is not quite so simple. Processing an interrupt is not just a matter of taking the interrupt, doing an up on some semaphore, and then executing an IRET instruction to return from the interrupt to the previous process. There is a great deal more work involved for the operating system. We will now give an outline of this work as a series of steps that must be performed in software after the hardware interrupt has completed. It should be noted that the details are very system dependent, so some of the steps listed below may not be needed on a particular machine and steps not listed may be required. Also, the steps that do occur may be in a different order on some machines.

1. Save any registers (including the PSW) that have not already been saved by the interrupt hardware.
2. Set up a context for the interrupt service procedure. Doing this may involve setting up the TLB, MMU and a page table.
3. Set up a stack for the interrupt service procedure.
4. Acknowledge the interrupt controller. If there is no centralized interrupt controller, reenable interrupts.
5. Copy the registers from where they were saved (possibly some stack) to the process table.
6. Run the interrupt service procedure. It will extract information from the interrupting device controller's registers.

7. Choose which process to run next. If the interrupt has caused some high-priority process that was blocked to become ready, it may be chosen to run now.
8. Set up the MMU context for the process to run next. Some TLB set up may also be needed.
9. Load the new process' registers, including its PSW.
10. Start running the new process.

As can be seen, interrupt processing is far from trivial. It also takes a considerable number of CPU instructions, especially on machines in which virtual memory is present and page tables have to be set up or the state of the MMU stored (e.g., the *R* and *M* bits). On some machines the TLB and CPU cache may also have to be managed when switching between user and kernel modes, which takes additional machine cycles.

5.3.2 Device Drivers

Earlier in this chapter we looked at what device controllers do. We saw that each controller has some device registers used to give it commands or some device registers used to read out its status or both. The number of device registers and the nature of the commands vary radically from device to device. For example, a mouse driver has to accept information from the mouse telling how far it has moved and which buttons are currently depressed. In contrast, a disk driver has to know about sectors, tracks, cylinders, heads, arm motion, motor drives, head settling times, and all the other mechanics of making the disk work properly. Obviously, these drivers will be very different.

As a consequence, each I/O device attached to a computer needs some device-specific code for controlling it. This code, called the **device driver**, is generally written by the device's manufacturer and delivered along with the device. Since each operating system needs its own drivers, device manufacturers commonly supply drivers for several popular operating systems.

Each device driver normally handles one device type, or at most, one class of closely related devices. For example, a SCSI disk driver can usually handle multiple SCSI disks of different sizes and different speeds, and perhaps a SCSI CD-ROM as well. On the other hand, a mouse and joystick are so different that different drivers are usually required. However, there is no technical restriction on having one device driver control multiple unrelated devices. It is just not a good idea.

In order to access the device's hardware, meaning the controller's registers, the device driver normally has to be part of the operating system kernel, at least with current architectures. Actually, it would be possible to construct drivers that

ran in user space, with system calls for reading and writing the device registers. In fact, this design would be a good idea, since it would isolate the kernel from the drivers and the drivers from each other. Doing this would eliminate a major source of system crashes—buggy drivers that interfere with the kernel in one way or another. However, since current operating systems expect drivers to run in the kernel, that is the model we will consider here.

Since the designers of every operating system know that pieces of code (drivers) written by outsiders will be installed in it, it needs to have an architecture that allows such installation. This means having a well-defined model of what a driver does and how it interacts with the rest of the operating system. Device drivers are normally positioned below the rest of the operating system, as illustrated in Fig. 5-11.

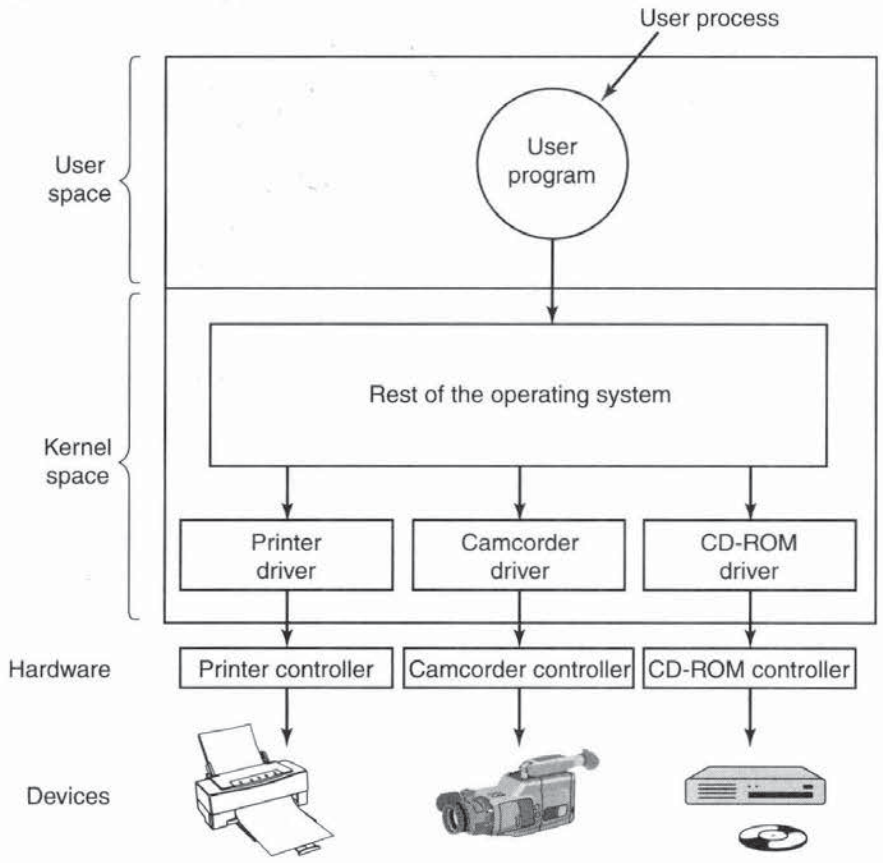


Figure 5-11. Logical positioning of device drivers. In reality all communication between drivers and device controllers goes over the bus.

Operating systems usually classify drivers into one of a small number of categories. The most common categories are the **block devices**, such as disks, which

contain multiple data blocks that can be addressed independently, and the **character devices**, such as keyboards and printers, which generate or accept a stream of characters.

Most operating systems define a standard interface that all block drivers must support and a second standard interface that all character drivers must support. These interfaces consist of a number of procedures that the rest of the operating system can call to get the driver to do work for it. Typical procedures are those to read a block (block device) or write a character string (character device).

In some systems, the operating system is a single binary program that contains all of the drivers that it will need compiled into it. This scheme was the norm for years with UNIX systems because they were run by computer centers and I/O devices rarely changed. If a new device was added, the system administrator simply recompiled the kernel with the new driver to build a new binary.

With the advent of personal computers, with their myriad of I/O devices, this model no longer worked. Few users are capable of recompiling or relinking the kernel, even if they have the source code or object modules, which is not always the case. Instead, operating systems, starting with MS-DOS, went over to a model in which drivers were dynamically loaded into the system during execution. Different systems handle loading drivers in different ways.

A device driver has several functions. The most obvious one is to accept abstract read and write requests from the device-independent software above it and see that they are carried out. But there are also a few other functions they must perform. For example, the driver must initialize the device, if needed. It may also need to manage its power requirements and log events.

Many device drivers have a similar general structure. A typical driver starts out by checking the input parameters to see if they are valid. If not, an error is returned. If they are valid, a translation from abstract to concrete terms may be needed. For a disk driver, this may mean converting a linear block number into the head, track, sector, and cylinder numbers for the disk's geometry.

Next the driver may check if the device is currently in use. If it is, the request will be queued for later processing. If the device is idle, the hardware status will be examined to see if the request can be handled now. It may be necessary to switch the device on or start a motor before transfers can be begun. Once the device is on and ready to go, the actual control can begin.

Controlling the device means issuing a sequence of commands to it. The driver is the place where the command sequence is determined, depending on what has to be done. After the driver knows which commands it is going to issue, it starts writing them into the controller's device registers. After writing each command to the controller, it may be necessary to check to see if the controller accepted the command and is prepared to accept the next one. This sequence continues until all the commands have been issued. Some controllers can be given a linked list of commands (in memory) and told to read and process them all by itself without further help from the operating system.

After the commands have been issued, one of two situations will apply. In many cases the device driver must wait until the controller does some work for it, so it blocks itself until the interrupt comes in to unblock it. In other cases, however, the operation finishes without delay, so the driver need not block. As an example of the latter situation, scrolling the screen in character mode requires just writing a few bytes into the controller's registers. No mechanical motion is needed, so the entire operation can be completed in nanoseconds.

In the former case, the blocked driver will be awakened by the interrupt. In the latter case, it will never go to sleep. Either way, after the operation has been completed, the driver must check for errors. If everything is all right, the driver may have data to pass to the device-independent software (e.g., a block just read). Finally, it returns some status information for error reporting back to its caller. If any other requests are queued, one of them can now be selected and started. If nothing is queued, the driver blocks waiting for the next request.

This simple model is only a rough approximation to reality. Many factors make the code much more complicated. For one thing, an I/O device may complete while a driver is running, interrupting the driver. The interrupt may cause a device driver to run. In fact, it may cause the current driver to run. For example, while the network driver is processing an incoming packet, another packet may arrive. Consequently, drivers have to be **reentrant**, meaning that a running driver has to expect that it will be called a second time before the first call has completed.

In a hot pluggable system, devices can be added or removed while the computer is running. As a result, while a driver is busy reading from some device, the system may inform it that the user has suddenly removed that device from the system. Not only must the current I/O transfer be aborted without damaging any kernel data structures, but any pending requests for the now-vanished device must also be gracefully removed from the system and their callers given the bad news. Furthermore, the unexpected addition of new devices may cause the kernel to juggle resources (e.g., interrupt request lines), taking old ones away from the driver and giving it new ones in their place.

Drivers are not allowed to make system calls, but they often need to interact with the rest of the kernel. Usually, calls to certain kernel procedures are permitted. For example, there are usually calls to allocate and deallocate hardwired pages of memory for use as buffers. Other useful calls are needed to manage the MMU, timers, the DMA controller, the interrupt controller, and so on.

5.3.3 Device-Independent I/O Software

Although some of the I/O software is device specific, other parts of it are device independent. The exact boundary between the drivers and the device-independent software is system (and device) dependent, because some functions that could be done in a device-independent way may actually be done in the

drivers, for efficiency or other reasons. The functions shown in Fig. 5-12 are typically done in the device-independent software.

Uniform interfacing for device drivers
Buffering
Error reporting
Allocating and releasing dedicated devices
Providing a device-independent block size

Figure 5-12. Functions of the device-independent I/O software.

The basic function of the device-independent software is to perform the I/O functions that are common to all devices and to provide a uniform interface to the user-level software. Below we will look at the above issues in more detail.

Uniform Interfacing for Device Drivers

A major issue in an operating system is how to make all I/O devices and drivers look more-or-less the same. If disks, printers, keyboards, etc., are all interfaced in different ways, every time a new device comes along, the operating system must be modified for the new device. Having to hack on the operating system for each new device is not a good idea.

One aspect of this issue is the interface between the device drivers and the rest of the operating system. In Fig. 5-13(a) we illustrate a situation in which each device driver has a different interface to the operating system. What this means is that the driver functions available for the system to call differ from driver to driver. It might also mean that the kernel functions that the driver needs also differ from driver to driver. Taken together, it means that interfacing each new driver requires a lot of new programming effort.

In contrast, in Fig. 5-13(b), we show a different design in which all drivers have the same interface. Now it becomes much easier to plug in a new driver, providing it conforms to the driver interface. It also means that driver writers know what is expected of them (e.g., what functions they must provide and what kernel functions they may call). In practice, not all devices are absolutely identical, but usually there are only a small number of device types and even these are generally almost the same. For example, even block and character devices have many functions in common.

Another aspect of having a uniform interface is how I/O devices are named. The device-independent software takes care of mapping symbolic device names onto the proper driver. For example, in UNIX a device name, such as */dev/disk0*, uniquely specifies the i-node for a special file, and this i-node contains the **major device number**, which is used to locate the appropriate driver. The i-node also

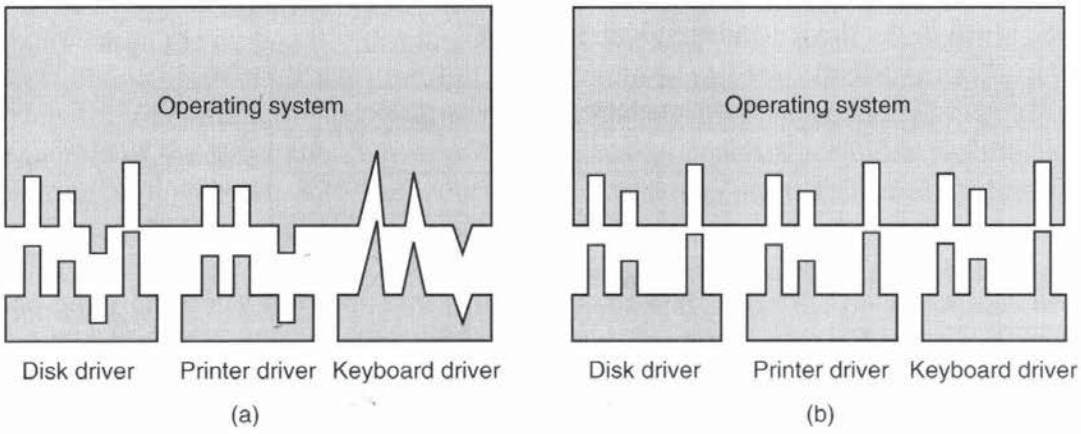


Figure 5-13. (a) Without a standard driver interface. (b) With a standard driver interface.

contains the **minor device number**, which is passed as a parameter to the driver in order to specify the unit to be read or written. All devices have major and minor numbers, and all drivers are accessed by using the major device number to select the driver.

Closely related to naming is protection. How does the system prevent users from accessing devices that they are not entitled to access? In both UNIX and Windows 2000 devices appears in the file system as named objects, which means that the usual protection rules for files also apply to I/O devices. The system administrator can then set the proper permissions for each device.

Buffering

Buffering is also an issue, both for block and character devices for a variety of reasons. To see one of them, consider a process that wants to read data from a modem. One possible strategy for dealing with the incoming characters is to have the user process do a read system call and block waiting for one character. Each arriving character causes an interrupt. The interrupt service procedure hands the character to the user process and unblocks it. After putting the character somewhere, the process reads another character and blocks again. This model is indicated in Fig. 5-14(a).

The trouble with this way of doing business is that the user process has to be started up for every incoming character. Allowing a process to run many times for short runs is inefficient, so this design is not a good one.

An improvement is shown in Fig. 5-14(b). Here the user process provides an n-character buffer in user space and does a read of n characters. The interrupt service procedure puts incoming characters in this buffer until it fills up. Then it

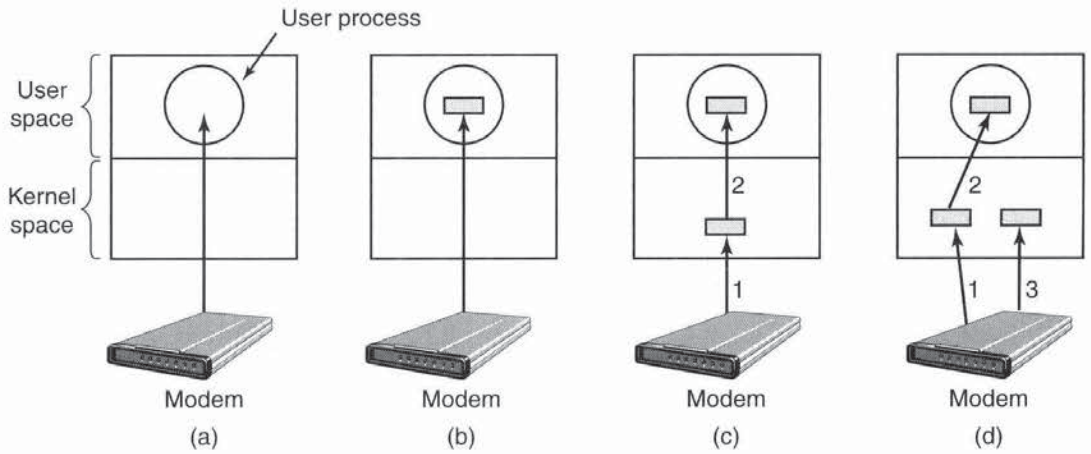


Figure 5-14. (a) Unbuffered input. (b) Buffering in user space. (c) Buffering in the kernel followed by copying to user space. (d) Double buffering in the kernel.

wakes up the user process. This scheme is far more efficient than the previous one, but it, too, has a drawback: what happens if the buffer is paged out when a character arrives? The buffer could be locked in memory, but if many processes start locking pages in memory, the pool of available pages will shrink and performance will degrade.

Yet another approach is to create a buffer inside the kernel and have the interrupt handler put the characters there, as shown in Fig. 5-14(c). When this buffer is full, the page with the user buffer is brought in, if needed, and the buffer copied there in one operation. This scheme is far more efficient.

However, even this scheme suffers from a problem: What happens to characters that arrive while the page with the user buffer is being brought in from the disk? Since the buffer is full, there is no place to put them. A way out is to have a second kernel buffer. After the first buffer fills up, but before it has been emptied, the second one is used, as shown in Fig. 5-14(d). When the second buffer fills up, it is available to be copied to the user (assuming the user has asked for it). While the second buffer is being copied to user space, the first one can be used for new characters. In this way, the two buffers take turns: while one is being copied to user space, the other is accumulating new input. A buffering scheme like this is called **double buffering**.

Buffering is also important on output. Consider, for example, how output is done to the modem without buffering using the model of Fig. 5-14(b). The user process executes a write system call to output n characters. The system has two choices at this point. It can block the user until all the characters have been written, but this could take a very long time over a slow telephone line. It could also release the user immediately and do the I/O while the user computes some more, but this leads to an even worse problem: how does the user process know that the output has been completed and it can reuse the buffer? The system could generate

a signal or software interrupt, but that style of programming is difficult and prone to race conditions. A much better solution is for the kernel to copy the data to a kernel buffer, analogous in Fig. 5-14(c) (but the other way), and unblock the caller immediately. Now it does not matter when the actual I/O has been completed. The user is free to reuse the buffer the instant it is unblocked.

Buffering is a widely-used technique, but it has a downside as well. If data get buffered too many times, performance suffers. Consider, for example, the network of Fig. 5-15. Here a user does a system call to write to the network. The kernel copies the packet to a kernel buffer to allow the user to proceed immediately (step 1).

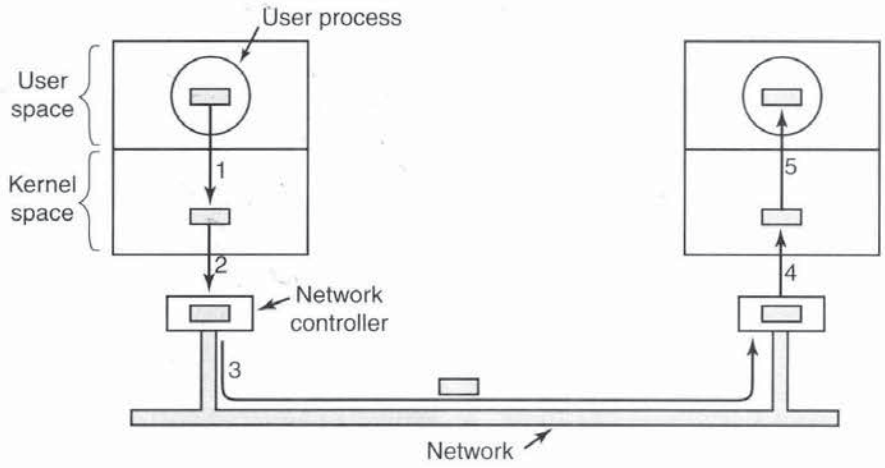


Figure 5-15. Networking may involve many copies of a packet.

When the driver is called, it copies the packet to the controller for output (step 2). The reason it does not output to the wire directly from kernel memory is that once a packet transmission has been started, it must continue at a uniform speed. The driver cannot guarantee that it can get to memory at a uniform speed because DMA channels and other I/O devices may be stealing many cycles. Failing to get a word on time would ruin the packet. By buffering the packet inside the controller, this problem is avoided.

After the packet has been copied to the controller's internal buffer, it is copied out onto the network (step 3). Bits arrive at the receiver shortly after being sent, so just after the last bit has been sent, that bit arrives at the receiver, where the packet has been buffered in the controller. Next the packet is copied to the receiver's kernel buffer (step 4). Finally, it is copied to the receiving process' buffer (step 5). Usually, the receiver then sends back an acknowledgement. When the sender gets the acknowledgement, it is free to send the next packet. However, it should be clear that all this copying is going to slow down the transmission rate considerably because all the steps must happen sequentially.

Error Reporting

Errors are far more common in the context of I/O than in other contexts. When they occur, the operating system must handle them as best it can. Many errors are device-specific and must be handled by the appropriate driver, but the framework for error handling is device independent.

One class of I/O errors are programming errors. These occur when a process asks for something impossible, such as writing to an input device (keyboard, mouse, scanner, etc.) or reading from an output device (printer, plotter, etc.). Other errors include providing an invalid buffer address or other parameter, and specifying an invalid device (e.g., disk 3 when the system has only two disks). The action to take on these errors is straightforward: just report back an error code to the caller.

Another class of errors is the class of actual I/O errors, for example, trying to write a disk block that has been damaged or trying to read from a camcorder that has been switched off. In these circumstances, it is up to the driver to determine what to do. If the driver does not know what to do, it may pass the problem back up to device-independent software.

What this software does depends on the environment and the nature of the error. If it is a simple read error and there is an interactive user available, it may display a dialog box asking the user what to do. The options may include retrying a certain number of times, ignoring the error, or killing the calling process. If there is no user available, probably the only real option is to have the system call fail with an error code.

However, some errors cannot be handled this way. For example, a critical data structure, such as the root directory or free block list, may have been destroyed. In this case, the system may have to display an error message and terminate.

Allocating and Releasing Dedicated Devices

Some devices, such as CD-ROM recorders, can be used only by a single process at any given moment. It is up to the operating system to examine requests for device usage and accept or reject them, depending on whether the requested device is available or not. A simple way to handle these requests is to require processes to perform opens on the special files for devices directly. If the device is unavailable, the open fails. Closing such a dedicated device then releases it.

An alternative approach is to have special mechanisms for requesting and releasing dedicated devices. An attempt to acquire a device that is not available blocks the caller instead of failing. Blocked processes are put on a queue. Sooner or later, the requested device becomes available and the first process on the queue is allowed to acquire it and continue execution.

Device-Independent Block Size

Different disks may have different sector sizes. It is up to the device-independent software to hide this fact and provide a uniform block size to higher layers, for example, by treating several sectors as a single logical block. In this way, the higher layers only deal with abstract devices that all use the same logical block size, independent of the physical sector size. Similarly, some character devices deliver their data one byte at a time (e.g., modems), while others deliver theirs in larger units (e.g., network interfaces). These differences may also be hidden.

5.3.4 User-Space I/O Software

Although most of the I/O software is within the operating system, a small portion of it consists of libraries linked together with user programs, and even whole programs running outside the kernel. System calls, including the I/O system calls, are normally made by library procedures. When a C program contains the call

```
count = write(fd, buffer, nbytes);
```

the library procedure *write* will be linked with the program and contained in the binary program present in memory at run time. The collection of all these library procedures is clearly part of the I/O system.

While these procedures do little more than put their parameters in the appropriate place for the system call, there are other I/O procedures that actually do real work. In particular, formatting of input and output is done by library procedures. One example from C is *printf*, which takes a format string and possibly some variables as input, builds an ASCII string, and then calls write to output the string. As an example of *printf*, consider the statement

```
printf("The square of %3d is  %6d\n", i, i*i);
```

It formats a string consisting of the 14-character string "The square of " followed by the value i as a 3-character string, then the 4-character string " is ", then i^2 as six characters, and finally a line feed.

An example of a similar procedure for input is *scanf* which reads input and stores it into variables described in a format string using the same syntax as *printf*. The standard I/O library contains a number of procedures that involve I/O and all run as part of user programs.

Not all user-level I/O software consists of library procedures. Another important category is the spooling system. **Spooling** is a way of dealing with dedicated I/O devices in a multiprogramming system. Consider a typical spooled device: a printer. Although it would be technically easy to let any user process open the character special file for the printer, suppose a process opened it and then did nothing for hours. No other process could print anything.

Instead what is done is to create a special process, called a **daemon**, and a special directory, called a **spooling directory**. To print a file, a process first generates the entire file to be printed and puts it in the spooling directory. It is up to the daemon, which is the only process having permission to use the printer's special file, to print the files in the directory. By protecting the special file against direct use by users, the problem of having someone keeping it open unnecessarily long is eliminated.

Spooling is not only used for printers. It is also used in other situations. For example, file transfer over a network often uses a network daemon. To send a file somewhere, a user puts it in a network spooling directory. Later on, the network daemon takes it out and transmits it. One particular use of spooled file transmission is the USENET News system. This network consists of millions of machines around the world communicating using the Internet. Thousands of news groups exist on many topics. To post a news message, the user invokes a news program, which accepts the message to be posted and then deposits it in a spooling directory for transmission to other machines later. The entire news system runs outside the operating system.

Figure 5-16 summarizes the I/O system, showing all the layers and the principal functions of each layer. Starting at the bottom, the layers are the hardware, interrupt handlers, device drivers, device-independent software, and finally the user processes.

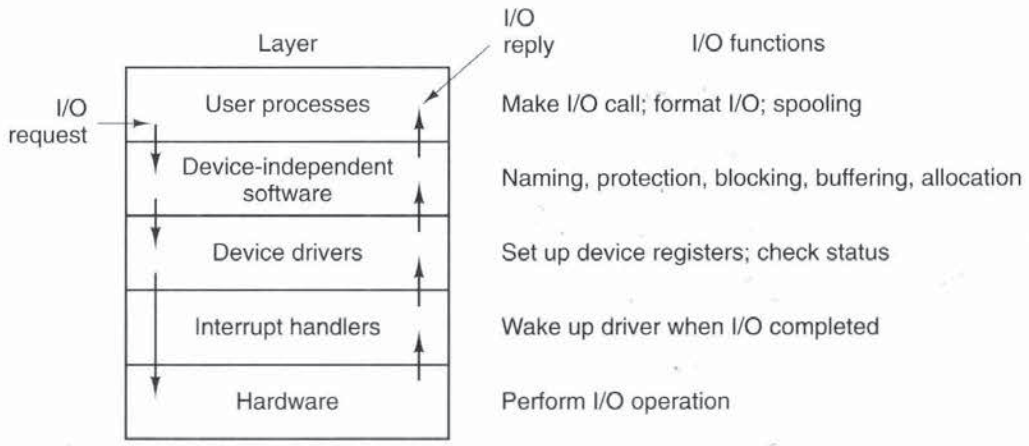


Figure 5-16. Layers of the I/O system and the main functions of each layer.

The arrows in Fig. 5-16 show the flow of control. When a user program tries to read a block from a file, for example, the operating system is invoked to carry out the call. The device-independent software looks for it in the buffer cache, for example. If the needed block is not there, it calls the device driver to issue the request to the hardware to go get it from the disk. The process is then blocked until the disk operation has been completed.

When the disk is finished, the hardware generates an interrupt. The interrupt handler is run to discover what has happened, that is, which device wants attention right now. It then extracts the status from the device and wakes up the sleeping process to finish off the I/O request and let the user process continue.

5.4 DISKS

Now we will begin studying some real I/O devices. We will begin with disks. After that we will examine clocks, keyboards, and displays.

5.4.1 Disk Hardware

Disks come in a variety of types. The most common ones are the magnetic disks (hard disks and floppy disks). They are characterized by the fact that reads and writes are equally fast, which makes them ideal as secondary memory (paging, file systems, etc.). Arrays of these disks are sometimes used to provide highly-reliable storage. For distribution of programs, data, and movies, various kinds of optical disks (CD-ROMs, CD-Recordables, and DVDs) are also important. In the following sections we will first describe the hardware and then the software for these devices.

Magnetic Disks

Magnetic disks are organized into cylinders, each one containing as many tracks as there are heads stacked vertically. The tracks are divided into sectors, with the number of sectors around the circumference typically being 8 to 32 on floppy disks, and up to several hundred on hard disks. The number of heads varies from 1 to about 16.

Some magnetic disks have little electronics and just deliver a simple serial bit stream. On these disks, the controller does most of the work. On other disks, in particular, **IDE** (**Integrated Drive Electronics**) disks, the drive itself contains a microcontroller that does some work and allows the real controller to issue a set of higher-level commands.

A device feature that has important implications for the disk driver is the possibility of a controller doing seeks on two or more drives at the same time. These are known as **overlapped seeks**. While the controller and software are waiting for a seek to complete on one drive, the controller can initiate a seek on another drive. Many controllers can also read or write on one drive while seeking on one or more other drives, but a floppy disk controller cannot read or write on two drives at the same time. (Reading or writing requires the controller to move bits on a microsecond time scale, so one transfer uses up most of its computing power.) The situation is different for hard disks with integrated controllers, and in

a system with more than one of these hard drives they can operate simultaneously, at least to the extent of transferring between the disk and the controller's buffer memory. Only one transfer between the controller and the main memory is possible at once, however. The ability to perform two or more operations at the same time can reduce the average access time considerably.

Figure 5-17 compares parameters of the standard storage medium for the original IBM PC with parameters of a modern hard disk to show how much disks have changed in the past two decades. It is interesting to note that not all parameters have improved as much. Average seek time is seven times better, transfer rate is 1300 times better, while capacity is up by a factor of 50,000. This pattern has to do with relatively gradual improvements in the moving parts, but much higher bit densities on the recording surfaces.

Parameter	IBM 360-KB floppy disk	WD 18300 hard disk
Number of cylinders	40	10601
Tracks per cylinder	2	12
Sectors per track	9	281 (avg)
Sectors per disk	720	35742000
Bytes per sector	512	512
Disk capacity	360 KB	18.3 GB
Seek time (adjacent cylinders)	6 msec	0.8 msec
Seek time (average case)	77 msec	6.9 msec
Rotation time	200 msec	8.33 msec
Motor stop/start time	250 msec	20 sec
Time to transfer 1 sector	22 msec	17 μsec

Figure 5-17. Disk parameters for the original IBM PC 360-KB floppy disk and a Western Digital WD 18300 hard disk.

One thing to be aware of in looking at the specifications of modern hard disks is that the geometry specified, and used by the driver software, may be different than the physical format. On older disks, the number of sectors per track was the same for all cylinders. Modern disks are divided into zones with more sectors on the outer zones than the inner ones. Fig. 5-18(a) illustrates a tiny disk with two zones. The outer zone has 32 sectors per track; the inner one has 16 sectors per track. A real disk, such as the WD 18300, often has 16 zones, with the number of sectors increasing by about 4% per zone as one goes out from the innermost zone to the outermost zone.

To hide the details of how many sectors each track has, most modern disks have a virtual geometry that is presented to the operating system. The software is instructed to act as though there are x cylinders, y heads, and z sectors per track. The controller then remaps a request for (x, y, z) onto the real cylinder, head, and

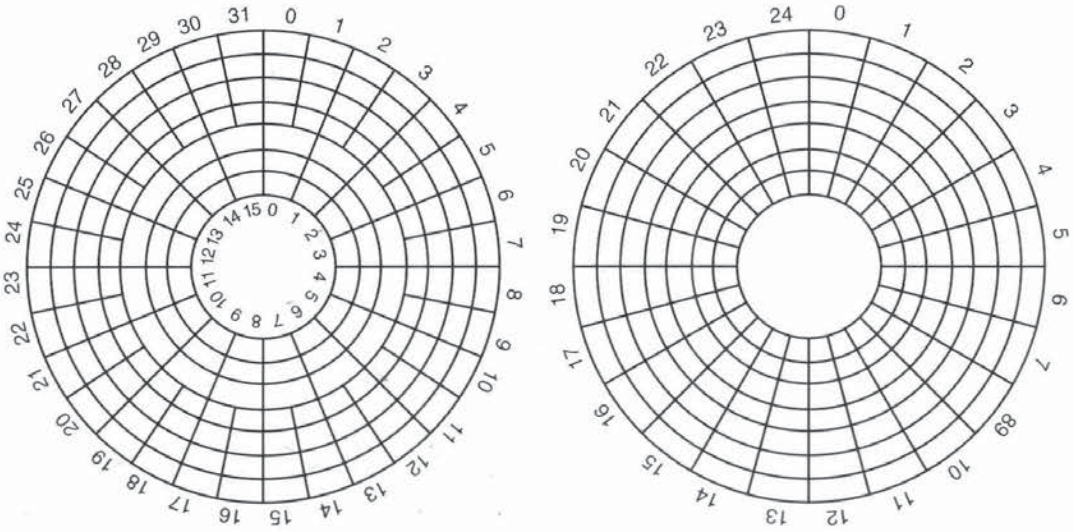


Figure 5-18. (a) Physical geometry of a disk with two zones. (b) A possible virtual geometry for this disk.

sector. A possible virtual geometry for the physical disk of Fig. 5-18(a) is shown in Fig. 5-18(b). In both cases the disk has 192 sectors, only the published arrangement is different than the real one.

For Pentium-based computers, the maximum values for these three parameters are often (65535, 16, and 63), due to the need to be backward compatible with the limitations of the original IBM PC. On this machine, 16-, 4-, and 6-bit fields were used to specify these numbers, with cylinders and sectors numbered starting at 1 and heads numbered starting at 0. With these parameters and 512 bytes per sector, the largest possible disk is 31.5 GB. To get around this limit, many disks now support a system called **logical block addressing**, in which disk sectors are just numbered consecutively starting at 0, without regard to the disk geometry.

RAID

CPU performance has been increasing exponentially over the past decade, roughly doubling every 18 months. Not so with disk performance. In the 1970s, average seek times on minicomputer disks were 50 to 100 msec. Now seek times are slightly under 10 msec. In most technical industries (say, automobiles or aviation), a factor of 5 to 10 performance improvement in two decades would be major news, but in the computer industry it is an embarrassment. Thus the gap between CPU performance and disk performance has become much larger over time.

As we have seen, parallel processing is being used more and more to speed up CPU performance. It has occurred to various people over the years that parallel

I/O might be a good idea too. In their 1988 paper, Patterson et al. suggested six specific disk organizations that could be used to improve disk performance, reliability, or both (Patterson et al., 1988). These ideas were quickly adopted by industry and have led to a new class of I/O device called a **RAID**. Patterson et al. defined RAID as **Redundant Array of Inexpensive Disks**, but industry redefined the I to be "Independent" rather than "Inexpensive" (maybe so they could use expensive disks?). Since a villain was also needed (as in RISC versus CISC, also due to Patterson), the bad guy here was the **SLED** (**Single Large Expensive Disk**).

The basic idea behind a RAID is to install a box full of disks next to the computer, typically a large server, replace the disk controller card with a RAID controller, copy the data over to the RAID, and then continue normal operation. In other words, a RAID should look like a SLED to the operating system but have better performance and better reliability. Since SCSI disks have good performance, low price, and the ability to have up to 7 drives on a single controller (15 for wide SCSI), it is natural that most RAIDs consist of a RAID SCSI controller plus a box of SCSI disks that appear to the operating system as a single large disk. In this way, no software changes are required to use the RAID, a big selling point for many system administrators.

In addition to appearing like a single disk to the software, all RAIDs have the property that the data are distributed over the drives, to allow parallel operation. Several different schemes for doing this were defined by Patterson et al., and they are now known as RAID level 0 through RAID level 5. In addition, there are a few other minor levels that we will not discuss. The term "level" is something of a misnomer since there is no hierarchy involved; there are simply six different organizations possible.

RAID level 0 is illustrated in Fig. 5-19(a). It consists of viewing the virtual single disk simulated by the RAID as being divided up into strips of k sectors each, with sectors 0 to $k - 1$ being strip 0, sectors k to $2k - 1$ as strip 1, and so on. For $k = 1$, each strip is a sector; for $k = 2$ a strip is two sectors, etc. The RAID level 0 organization writes consecutive strips over the drives in round-robin fashion, as depicted in Fig. 5-19(a) for a RAID with four disk drives. Distributing data over multiple drives like this is called **striping**. For example, if the software issues a command to read a data block consisting of four consecutive strips starting at a strip boundary, the RAID controller will break this command up into four separate commands, one for each of the four disks, and have them operate in parallel. Thus we have parallel I/O without the software knowing about it.

RAID level 0 works best with large requests, the bigger the better. If a request is larger than the number of drives times the strip size, some drives will get multiple requests, so that when they finish the first request they start the second one. It is up to the controller to split the request up and feed the proper commands to the proper disks in the right sequence and then assemble the results in memory correctly. Performance is excellent and the implementation is straightforward.

RAID level 0 works worst with operating systems that habitually ask for data one sector at a time. The results will be correct, but there is no parallelism and hence no performance gain. Another disadvantage of this organization is that the reliability is potentially worse than having a SLED. If a RAID consists of four disks, each with a mean time to failure of 20,000 hours, about once every 5000 hours a drive will fail and all the data will be completely lost. A SLED with a mean time to failure of 20,000 hours would be four times more reliable. Because no redundancy is present in this design, it is not really a true RAID.

The next option, RAID level 1, shown in Fig. 5-19(b), is a true RAID. It duplicates all the disks, so there are four primary disks and four backup disks. On a write, every strip is written twice. On a read, either copy can be used, distributing the load over more drives. Consequently, write performance is no better than for a single drive, but read performance can be up to twice as good. Fault tolerance is excellent: if a drive crashes, the copy is simply used instead. Recovery consists of simply installing a new drive and copying the entire backup drive to it.

Unlike levels 0 and 1, which work with strips of sectors, RAID level 2 works on a word basis, possibly even a byte basis. Imagine splitting each byte of the single virtual disk into a pair of 4-bit nibbles, then adding a Hamming code to each one to form a 7-bit word, of which bits 1, 2, and 4 were parity bits. Further imagine that the seven drives of Fig. 5-19(c) were synchronized in terms of arm position and rotational position. Then it would be possible to write the 7-bit Hamming coded word over the seven drives, one bit per drive.

The Thinking Machines' CM-2 computer used this scheme, taking 32-bit data words and adding 6 parity bits to form a 38-bit Hamming word, plus an extra bit for word parity, and spread each word over 39 disk drives. The total throughput was immense, because in one sector time it could write 32 sectors worth of data. Also, losing one drive did not cause problems, because loss of a drive amounted to losing 1 bit in each 39-bit word read, something the Hamming code could handle on the fly.

On the down side, this scheme requires all the drives to be rotationally synchronized, and it only makes sense with a substantial number of drives (even with 32 data drives and 6 parity drives, the overhead is 19 percent). It also asks a lot of the controller, since it must do a Hamming checksum every bit time.

RAID level 3 is a simplified version of RAID level 2. It is illustrated in Fig. 5-19(d). Here a single parity bit is computed for each data word and written to a parity drive. As in RAID level 2, the drives must be exactly synchronized, since individual data words are spread over multiple drives.

At first thought, it might appear that a single parity bit gives only error detection, not error correction. For the case of random undetected errors, this observation is true. However, for the case of a drive crashing, it provides full 1-bit error correction since the position of the bad bit is known. If a drive crashes, the controller just pretends that all its bits are 0s. If a word has a parity error, the bit from the dead drive must have been a 1, so it is corrected. Although both RAID levels

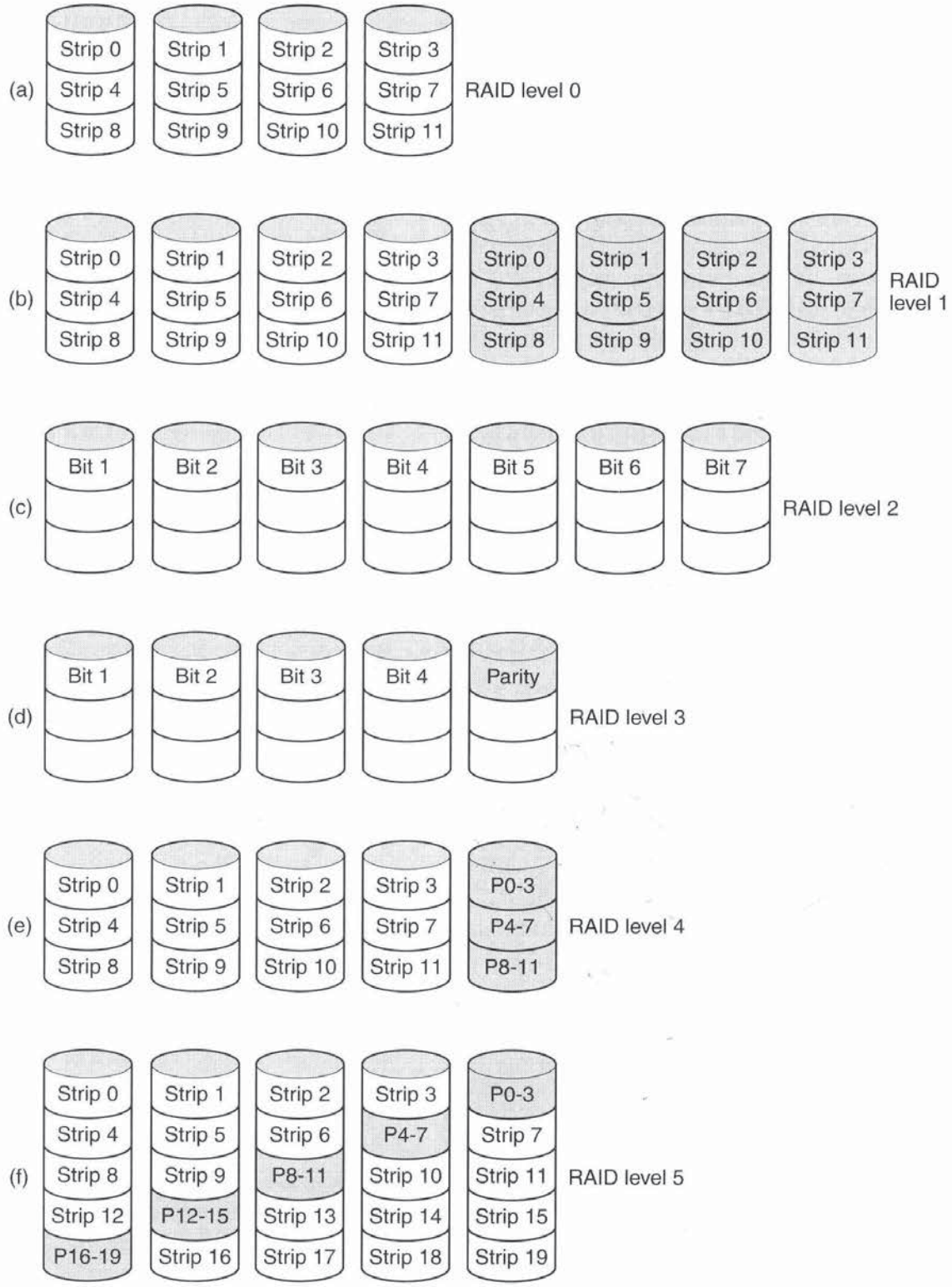


Figure 5-19. RAID levels 0 through 5. Backup and parity drives are shown shaded.

2 and 3 offer very high data rates, the number of separate I/O requests per second they can handle is no better than for a single drive.

RAID levels 4 and 5 work with strips again, not individual words with parity, and do not require synchronized drives. RAID level 4 [see Fig. 5-19(e)] is like RAID level 0, with a strip-for-strip parity written onto an extra drive. For example, if each strip is k bytes long, all the strips are EXCLUSIVE ORed together, resulting in a parity strip k bytes long. If a drive crashes, the lost bytes can be recomputed from the parity drive.

This design protects against the loss of a drive but performs poorly for small updates. If one sector is changed, it is necessary to read all the drives in order to recalculate the parity, which must then be rewritten. Alternatively, it can read the old user data and the old parity data and recompute the new parity from them. Even with this optimization, a small update requires two reads and two writes.

As a consequence of the heavy load on the parity drive, it may become a bottleneck. This bottleneck is eliminated in RAID level 5 by distributing the parity bits uniformly over all the drives, round robin fashion, as shown in Fig. 5-19(f). However, in the event of a drive crash, reconstructing the contents of the failed drive is a complex process.

CD-ROMs

In recent years, optical (as opposed to magnetic) disks have become available. They have much higher recording densities than conventional magnetic disks. Optical disks were originally developed for recording television programs, but they can be put to more esthetic use as computer storage devices. Due to their potentially enormous capacity, optical disks have been the subject of a great deal of research and have gone through an incredibly rapid evolution.

First-generation optical disks were invented by the Dutch electronics conglomerate Philips for holding movies. They were 30 cm across and marketed under the name LaserVision, but they did not catch on, except in Japan.

In 1980, Philips, together with Sony, developed the CD (Compact Disc), which rapidly replaced the 33 1/3-rpm vinyl record for music (except among connoisseurs, who still prefer vinyl). The precise technical details for the CD were published in an official International Standard (IS 10149), popularly called the **Red Book**, due to the color of its cover. (International Standards are issued by the International Organization for Standardization, which is the international counterpart of national standards groups like ANSI, DIN, etc. Each one has an IS number.) The point of publishing the disk and drive specifications as an International Standard is to allow CDs from different music publishers and players from different electronics manufacturers to work together. All CDs are 120 mm across and 1.2 mm thick, with a 15-mm hole in the middle. The audio CD was the first successful mass market digital storage medium. They are supposed to last 100 years. Please check back in 2080 for an update on how well the first batch did.

A CD is prepared by using a high-power infrared laser to burn 0.8-micron diameter holes in a coated glass master disk. From this master, a mold is made, with bumps where the laser holes were. Into this mold, molten polycarbonate resin is injected to form a CD with the same pattern of holes as the glass master. Then a very thin layer of reflective aluminum is deposited on the polycarbonate, topped by a protective lacquer and finally a label. The depressions in the polycarbonate substrate are called **pits**; the unburned areas between the pits are called **lands**.

When played back, a low-power laser diode shines infrared light with a wavelength of 0.78 micron on the pits and lands as they stream by. The laser is on the polycarbonate side, so the pits stick out toward the laser as bumps in the otherwise flat surface. Because the pits have a height of one-quarter the wavelength of the laser light, light reflecting off a pit is half a wavelength out of phase with light reflecting off the surrounding surface. As a result, the two parts interfere destructively and return less light to the player's photodetector than light bouncing off a land. This is how the player tells a pit from a land. Although it might seem simpler to use a pit to record a 0 and a land to record a 1, it is more reliable to use a pit/land or land/pit transition for a 1 and its absence as a 0, so this scheme is used.

The pits and lands are written in a single continuous spiral starting near the hole and working out a distance of 32 mm toward the edge. The spiral makes 22,188 revolutions around the disk (about 600 per mm). If unwound, it would be 5.6 km long. The spiral is illustrated in Fig. 5-20.

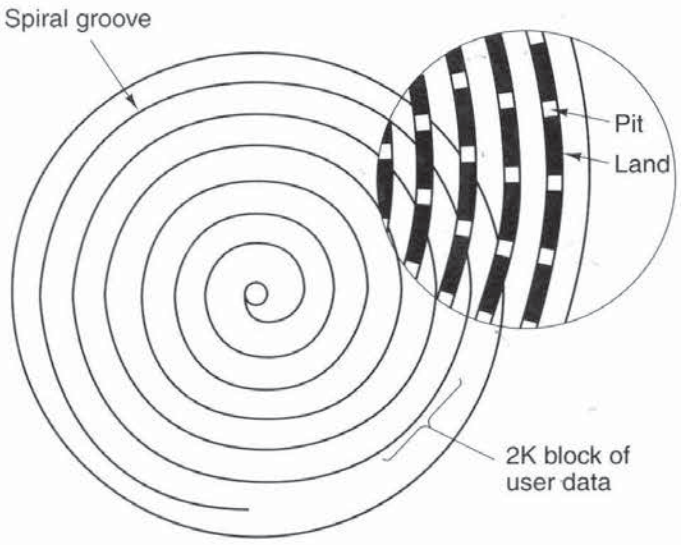


Figure 5-20. Recording structure of a compact disc or CD-ROM.

To make the music play at a uniform rate, it is necessary for the pits and lands to stream by at a constant linear velocity. Consequently the rotation rate of the

CD must be continuously reduced as the reading head moves from the inside of the CD to the outside. At the inside, the rotation rate is 530 rpm to achieve the desired streaming rate of 120 cm/sec; at the outside it has to drop to 200 rpm to give the same linear velocity at the head. A constant linear velocity drive is quite different than a magnetic disk drive, which operates at a constant angular velocity, independent of where the head is currently positioned. Also, 530 rpm is a far cry from the 3600 to 7200 rpm that most magnetic disks whirl at.

In 1984, Philips and Sony realized the potential for using CDs to store computer data, so they published the **Yellow Book** defining a precise standard for what are now called **CD-ROM**s (**Compact Disc - Read Only Memory**). To piggyback on the by-then already substantial audio CD market, CD-ROMs were to be the same physical size as audio CDs, mechanically and optically compatible with them, and produced using the same polycarbonate injection molding machines. The consequences of this decision were not only that slow variable-speed motors were required, but also that the manufacturing cost of a CD-ROM would be well under one dollar in moderate volume.

What the Yellow Book defined was the formatting of the computer data. It also improved the error-correcting abilities of the system, an essential step because although music lovers do not mind losing a bit here and there, computer lovers tend to be Very Picky about that. The basic format of a CD-ROM consists of encoding every byte in a 14-bit symbol. As we saw above, 14 bits is enough to Hamming encode an 8-bit byte with 2 bits left over. In fact, a more powerful encoding system is used. The 14-to-8 mapping for reading is done in hardware by table lookup.

At the next level up, a group of 42 consecutive symbols forms a 588-bit **frame**. Each frame holds 192 data bits (24 bytes). The remaining 396 bits are used for error correction and control. So far, this scheme is identical for audio CDs and CD-ROMs.

What the Yellow Book adds is the grouping of 98 frames into a **CD-ROM sector**, as shown in Fig. 5-21. Every CD-ROM sector begins with a 16-byte preamble, the first 12 of which are 00FFFFFFFFFFFFFFFFFF00 (hexadecimal), to allow the player to recognize the start of a CD-ROM sector. The next 3 bytes contain the sector number, needed because seeking on a CD-ROM with its single data spiral is much more difficult than on a magnetic disk with its uniform concentric tracks. To seek, the software in the drive calculates approximately where to go, moves the head there, and then starts hunting around for a preamble to see how good its guess was. The last byte of the preamble is the mode.

The Yellow Book defines two modes. Mode 1 uses the layout of Fig. 5-21, with a 16-byte preamble, 2048 data bytes, and a 288-byte error-correcting code (a crossinterleaved Reed-Solomon code). Mode 2 combines the data and ECC fields into a 2336-byte data field for those applications that do not need (or cannot afford the time to perform) error correction, such as audio and video. Note that to provide excellent reliability, three separate error-correcting schemes are used:

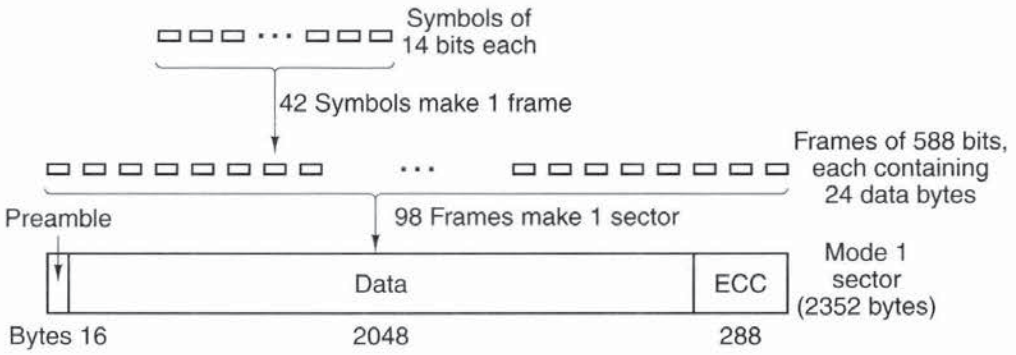


Figure 5-21. Logical data layout on a CD-ROM.

within a symbol, within a frame, and within a CD-ROM sector. Single-bit errors are corrected at the lowest level, short burst errors are corrected at the frame level, and any residual errors are caught at the sector level. The price paid for this reliability is that it takes 98 frames of 588 bits (7203 bytes) to carry a single 2048-byte payload, an efficiency of only 28 percent.

Single-speed CD-ROM drives operate at 75 sectors/sec, which gives a data rate of 153,600 bytes/sec in mode 1 and 175,200 bytes/sec in mode 2. Double-speed drives are twice as fast, and so on up to the highest speed. Thus a 40x drive can deliver data at a rate of $40 \times 153{,}600$ bytes/sec, assuming that the drive interface, bus, and operating system can all handle this data rate. A standard audio CD has room for 74 minutes of music, which, if used for mode 1 data, gives a capacity of 681,984,000 bytes. This figure is usually reported as 650 MB because 1 MB is 2^{20} bytes (1,048,576 bytes), not 1,000,000 bytes.

Note that even a 32x CD-ROM drive (4,915,200 bytes/sec) is no match for a fast SCSI-2 magnetic disk drive at 10 MB/sec, even though many CD-ROM drives use the SCSI interface (IDE CD-ROM drives also exist). When you realize that the seek time is usually several hundred milliseconds, it should be clear that CD-ROM drives are not in the same performance category as magnetic disk drives, despite their large capacity.

In 1986, Philips struck again with the **Green Book**, adding graphics and the ability to interleave audio, video and data in the same sector, a feature essential for multimedia CD-ROMs.

The last piece of the CD-ROM puzzle is the file system. To make it possible to use the same CD-ROM on different computers, agreement was needed on CD-ROM file systems. To get this agreement, representatives of many computer companies met at Lake Tahoe in the High Sierras on the California-Nevada boundary and devised a file system that they called **High Sierra**. It later evolved into an International Standard (IS 9660). It has three levels. Level 1 uses file names of up to 8 characters optionally followed by an extension of up to 3 characters (the

MS-DOS file naming convention). File names may contain only upper case letters, digits, and the underscore. Directories may be nested up to eight deep, but directory names may not contain extensions. Level 1 requires all files to be contiguous, which is not a problem on a medium written only once. Any CD-ROM conformant to IS 9660 level 1 can be read using MS-DOS, an Apple computer, a UNIX computer, or just about any other computer. CD-ROM publishers regard this property as being a big plus.

IS 9660 level 2 allows names up to 32 characters, and level 3 allows noncontiguous files. The Rock Ridge extensions (whimsically named after the town in the Gene Wilder film *Blazing Saddles*) allow very long names (for UNIX), UIDs, GIDs, and symbolic links, but CD-ROMs not conforming to level 1 will not be readable on all computers.

CD-ROMs have become extremely popular for publishing games, movies, encylopedias, atlases, and reference works of all kinds. Most commercial software now comes on CD-ROMs. Their combination of large capacity and low manufacturing cost makes them well suited to innumerable applications.

CD-Recordables

Initially, the equipment needed to produce a master CD-ROM (or audio CD, for that matter) was extremely expensive. But as usual in the computer industry, nothing stays expensive for long. By the mid 1990s, CD recorders no bigger than a CD player were a common peripheral available in most computer stores. These devices were still different from magnetic disks because once written, CD-ROMs could not be erased. Nevertheless, they quickly found a niche as a backup medium for large hard disks and also allowed individuals or startup companies to manufacture their own small-run CD-ROMs or make masters for delivery to high-volume commercial CD duplication plants. These drives are known as **CD-Rs** (**CD-Recordables**).

Physically, CD-Rs start with 120-mm polycarbonate blanks that are like CD-ROMs, except that they contain a 0.6-mm wide groove to guide the laser for writing. The groove has a sinusoidal excursion of 0.3 mm at a frequency of exactly 22.05 kHz to provide continuous feedback so the rotation speed can be accurately monitored and adjusted if need be. CD-Rs look like regular CD-ROMs, except that they are gold colored on top instead of silver colored. The gold color comes from the use of real gold instead of aluminum for the reflective layer. Unlike silver CDs, which have physical depressions on them, on CD-Rs the differing reflectivity of pits and lands has to be simulated. This is done by adding a layer of dye between the polycarbonate and the reflective gold layer, as shown in Fig. 5-22. Two kinds of dye are used: cyanine, which is green, and pthalocyanine, which is a yellowish orange. Chemists can argue endlessly about which one is better. These dyes are similar to those used in photography, which explains why Eastman Kodak and Fuji are major manufacturers of blank CD-Rs.

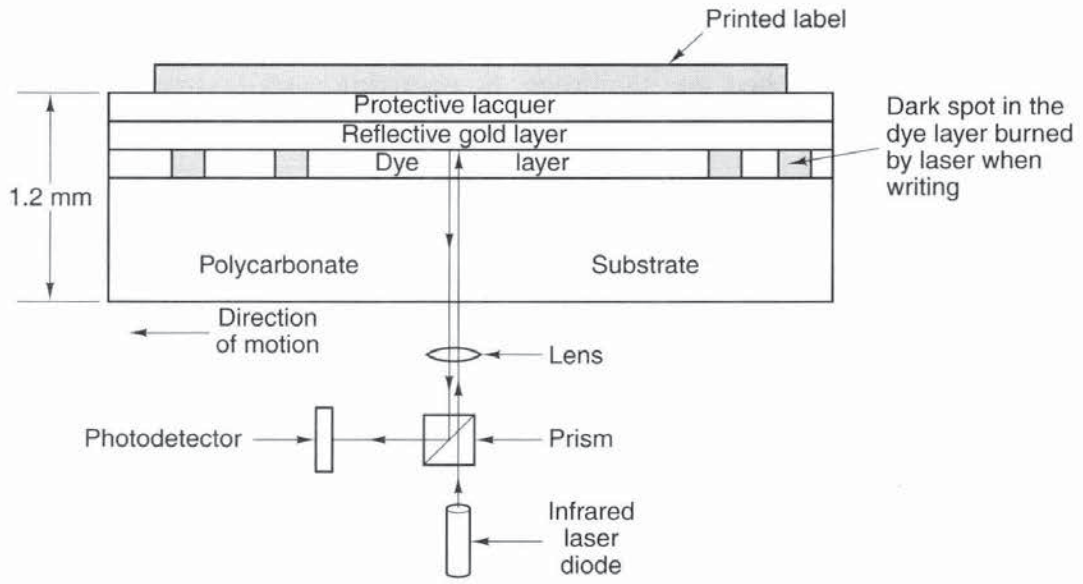


Figure 5-22. Cross section of a CD-R disk and laser (not to scale). A silver CD-ROM has a similar structure, except without the dye layer and with a pitted aluminum layer instead of a gold layer.

In its initial state, the dye layer is transparent and lets the laser light pass through and reflect off the gold layer. To write, the CD-R laser is turned up to high power (8–16 mW). When the beam hits a spot of dye, it heats up, breaking a chemical bond. This change to the molecular structure creates a dark spot. When read back (at 0.5 mW), the photodetector sees a difference between the dark spots where the dye has been hit and transparent areas where it is intact. This difference is interpreted as the difference between pits and lands, even when read back on a regular CD-ROM reader or even on an audio CD player.

No new kind of CD could hold up its head with pride without a colored book, so CD-R has the **Orange Book**, published in 1989. This document defines CD-R and also a new format, **CD-ROM XA**, which allows CD-Rs to be written incrementally, a few sectors today, a few tomorrow, and a few next month. A group of consecutive sectors written at once is called a **CD-ROM track**.

One of the first uses of CD-R was for the Kodak PhotoCD. In this system the customer brings a roll of exposed film and his old PhotoCD to the photo processor and gets back the same PhotoCD with the new pictures added after the old ones. The new batch, which is created by scanning in the negatives, is written onto the PhotoCD as a separate CD-ROM track. Incremental writing was needed because when this product was introduced, the CD-R blanks were too expensive to provide a new one for every film roll.

However, incremental writing creates a new problem. Prior to the Orange Book, all CD-ROMs had a single **VTOC** (**Volume Table of Contents**) at the

start. That scheme does not work with incremental (i.e., multitrack) writes. The Orange Book's solution is to give each CD-ROM track its own VTOC. The files listed in the VTOC can include some or all of the files from previous tracks. After the CD-R is inserted into the drive, the operating system searches through all the CD-ROM tracks to locate the most recent VTOC, which gives the current status of the disk. By including some, but not all, of the files from previous tracks in the current VTOC, it is possible to give the illusion that files have been deleted. Tracks can be grouped into **sessions**, leading to **multisession** CD-ROMs. Standard audio CD players cannot handle multisession CDs since they expect a single VTOC at the start.

Each track has to be written in a single continuous operation without stopping. As a consequence, the hard disk from which the data are coming has to be fast enough to deliver it on time. If the files to be copied are spread all over the hard disk, the seek times may cause the data stream to the CD-R to dry up and cause a buffer underrun. A buffer underrun results in producing a nice shiny (but somewhat expensive) coaster for your drinks, or a 120 mm gold-colored frisbee. CD-R software usually offers the option of collecting all the input files into a single contiguous 650-MB CD-ROM image prior to burning the CD-R, but this process typically doubles the effective writing time, requires 650 MB of free disk space, and still does not protect against hard disks that panic and decide to do a thermal recalibration when they get too hot.

CD-R makes it possible for individuals and companies to easily copy CD-ROMs (and audio CDs), generally in violation of the publisher's copyright. Several schemes have been devised to make such piracy harder and to make it difficult to read a CD-ROM using anything other than the publisher's software. One of them involves recording all the file lengths on the CD-ROM as multigigabyte, thwarting any attempts to copy the files to hard disk using standard copying software. The true lengths are embedded in the publisher's software or hidden (possibly encrypted) on the CD-ROM in an unexpected place. Another scheme uses intentionally incorrect ECCs in selected sectors, in the expectation that CD copying software will "fix" the errors. The application software checks the ECCs itself, refusing to work if they are correct. Using nonstandard gaps between the tracks and other physical "defects" are also possibilities.

CD-Rewritables

Although people are used to other write-once media such as paper and photographic film, there is a demand for a rewritable CD-ROM. One technology now available is **CD-RW** (**CD-ReWritable**), which uses the same size media as CD-R. However, instead of cyanine or pthalocyanine dye, CR-RW uses an alloy of silver, indium, antimony, and tellurium for the recording layer. This alloy has two stable states: crystalline and amorphous, with different reflectivities.

CD-RW drives use lasers with three different powers. At high power, the laser melts the alloy, converting it from the high-reflectivity crystalline state to the low-reflectivity amorphous state to represent a pit. At medium power, the alloy melts and reforms in its natural crystalline state to become a land again. At low power, the state of the material is sensed (for reading), but no phase transition occurs.

The reason CD-RW has not replaced CD-R is that the CD-RW blanks are much more expensive than the CR-R blanks. Also, for applications consisting of backing up hard disks, the fact that once written, a CD-R cannot be accidentally erased is a big plus.

DVD

The basic CD/CD-ROM format has been around since 1980. The technology has improved since then, so higher-capacity optical disks are now economically feasible and there is great demand for them. Hollywood would dearly love to replace analog video tapes by digital disks, since disks have a higher quality, are cheaper to manufacture, last longer, take up less shelf space in video stores, and do not have to be rewound. The consumer electronics companies are looking for a new blockbuster product, and many computer companies want to add multimedia features to their software.

This combination of technology and demand by three immensely rich and powerful industries has led to **DVD**, originally an acronym for **Digital Video Disk**, but now officially **Digital Versatile Disk**. DVDs use the same general design as CDs, with 120-mm injection-molded polycarbonate disks containing pits and lands that are illuminated by a laser diode and read by a photodetector. What is new is the use of

1. Smaller pits (0.4 microns versus 0.8 microns for CDs).
2. A tighter spiral (0.74 microns between tracks versus 1.6 microns for CDs).
3. A red laser (at 0.65 microns versus 0.78 microns for CDs).

Together, these improvements raise the capacity sevenfold, to 4.7-GB. A 1x DVD drive operates at 1.4 MB/sec (versus 150 KB/sec for CDs). Unfortunately, the switch to the red lasers used in supermarkets means that DVD players will require a second laser or fancy conversion optics to be able to read existing CDs and CD-ROMs, something not all of them may provide. Also, reading CD-Rs and CD-RWs on a DVD drive may not be possible.

Is 4.7 GB enough? Maybe. Using MPEG-2 compression (standardized in IS 13346), a 4.7 GB DVD disk can hold 133 minutes of full-screen, full-motion video at high resolution (720×480), as well as soundtracks in up to eight languages and subtitles in 32 more. About 92 percent of all the movies Hollywood has ever made are under 133 minutes. Nevertheless, some applications such

as multimedia games or reference works may need more, and Hollywood would like to put multiple movies on the same disk, so four formats have been defined:

1. Single-sided, single-layer (4.7 GB).
2. Single-sided, dual-layer (8.5 GB).
3. Double-sided, single-layer (9.4 GB).
4. Double-sided, dual-layer (17 GB).

Why so many formats? In a word: politics. Philips and Sony wanted single-sided, dual-layer disks for the high capacity version, but Toshiba and Time Warner wanted double-sided, single-layer disks. Philips and Sony did not think people would be willing to turn the disks over, and Time Warner did not believe putting two layers on one side could be made to work. The compromise: all combinations, but the market will determine which ones survive.

The dual layering technology has a reflective layer at the bottom, topped with a semireflective layer. Depending on where the laser is focused, it bounces off one layer or the other. The lower layer needs slightly larger pits and lands to be read reliably, so its capacity is slightly smaller than the upper layer's.

Double-sided disks are made by taking two 0.6-mm single-sided disks and gluing them together back to back. To make the thicknesses of all versions the same, a single-sided disk consists of a 0.6-mm disk bonded to a blank substrate (or perhaps in the future, one consisting of 133 minutes of advertising, in the hope that people will be curious as to what is down there). The structure of the double-sided, dual-layer disk is illustrated in Fig. 5-23.

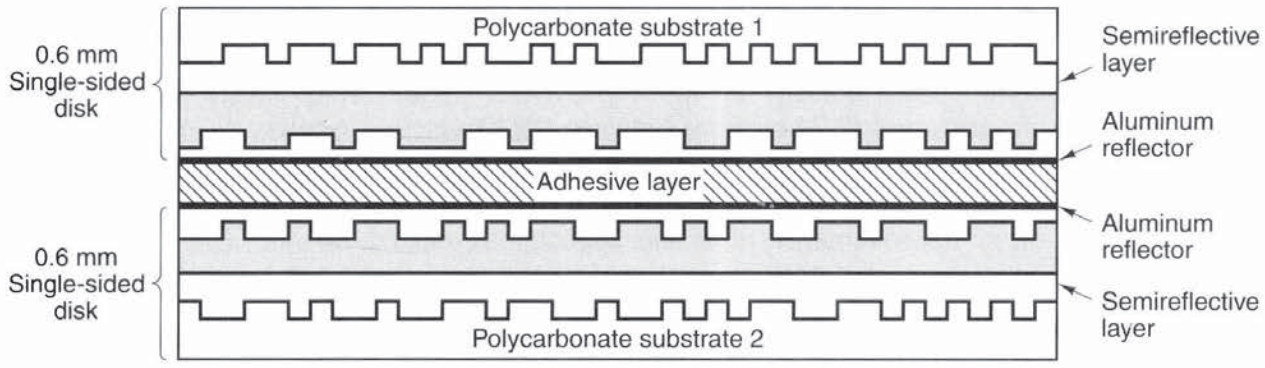


Figure 5-23. A double-sided, dual layer DVD disk.

DVD was devised by a consortium of 10 consumer electronics companies, seven of them Japanese, in close cooperation with the major Hollywood studios (some of which are owned by the Japanese electronics companies in the consortium). The computer and telecommunications industries were not invited to the

picnic, and the resulting focus was on using DVD for movie rental and sales shows. For example, standard features include real-time skipping of dirty scenes (to allow parents to turn a film rated NC17 into one safe for toddlers), six-channel sound, and support for Pan-and-Scan. The latter feature allows the DVD player to dynamically decide how to crop the left and right edges off movies (whose width:height ratio is 3:2) to fit on current television sets (whose aspect ratio is 4:3).

Another item the computer industry probably would not have thought of is an intentional incompatibility between disks intended for the United States and disks intended for Europe and yet other standards for other continents. Hollywood demanded this "feature" because new films are always released first in the United States and then shipped to Europe when the videos come out in the United States. The idea was to make sure European video stores could not buy videos in the U.S. too early, thereby reducing new movies' European theater sales. If Hollywood had been running the computer industry, we would have had 3.5-inch floppy disks in the United States and 9-cm floppy disks in Europe.

5.4.2 Disk Formatting

A hard disk consists of a stack of aluminum, alloy, or glass platters 5.25 inch or 3.5 inch in diameter (or even smaller on notebook computers). On each platter is deposited a thin magnetizable metal oxide. After manufacturing, there is no information whatsoever on the disk.

Before the disk can be used, each platter must receive a **low-level format** done by software. The format consists of a series of concentric tracks, each containing some number of sectors, with short gaps between the sectors. The format of a sector is shown in Fig. 5-24.



Figure 5-24. A disk sector.

The preamble starts with a certain bit pattern that allows the hardware to recognize the start of the sector. It also contains the cylinder and sector numbers and some other information. The size of the data portion is determined by the low-level formatting program. Most disks use 512-byte sectors. The ECC field contains redundant information that can be used to recover from read errors. The size and content of this field varies from manufacturer to manufacturer, depending on how must disk space the designer is willing to give up for higher reliability and how complex an ECC code the controller can handle. A 16-byte ECC field is not unusual. Furthermore, all hard disks have some number of spare sectors allocated to be used to replace sectors with a manufacturing defect.

The position of sector 0 on each track is offset from the previous track when the low-level format is laid down. This offset, called **cylinder skew**, is done to improve performance. The idea is to allow the disk to read multiple tracks in one continuous operation without losing data. The nature of the problem can be seen by looking at Fig. 5-18(a). Suppose that a request needs 18 sectors starting at sector 0 on the innermost track. Reading the first 16 sectors takes one disk rotation, but a seek is needed to move outward one track to get the 17th sector. By the time the head has moved one track, sector 0 has rotated past the head so an entire rotation is needed until it comes by again. That problem is eliminated by offsetting the sectors as shown in Fig. 5-25.

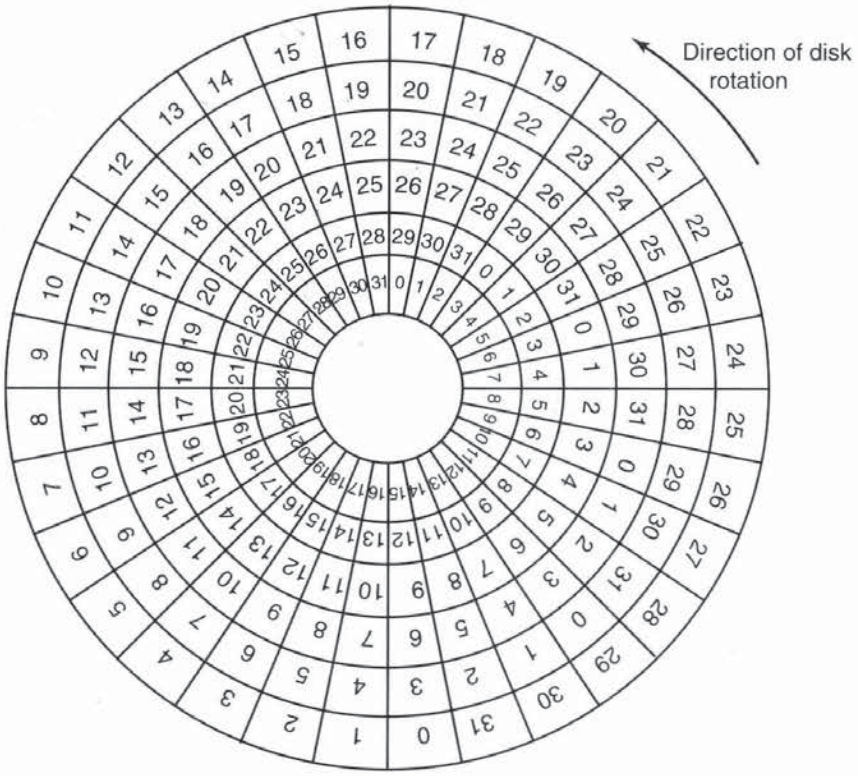


Figure 5-25. An illustration of cylinder skew.

The amount of cylinder skew depends on the drive geometry. For example, a 10,000-rpm drive rotates in 6 msec. If a track contains 300 sectors, a new sector passes under the head every 20 μsec. If the track-to-track seek time is 800 μsec, 40 sectors will pass by during the seek, so the cylinder skew should be 40 sectors, rather than the three sectors shown in Fig. 5-25. It is worth mentioning that switching between heads also takes a finite time, so there is **head skew** as well as cylinder skew, but head skew is not very large.

As a result of the low-level formatting, disk capacity is reduced, depending on the sizes of the preamble, intersector gap, and ECC, as well as the number of

spare sectors reserved. Often the formatted capacity is 20% lower than the unformatted capacity. The spare sectors do not count toward the formatted capacity, so all disks of a given type have exactly the same capacity when shipped, independent of how many bad sectors they actually have (if the number of bad sectors exceeds the number of spares, the drive will be rejected and not shipped).

There is considerable confusion about disk capacity because some manufacturers advertised the unformatted capacity to make their drives look larger than they really are. For example, consider a drive whose unformatted capacity is 20×10^9 bytes. This might be sold as a 20-GB disk. However, after formatting, perhaps only $2^{34} \simeq 17.2 \times 10^9$ bytes are available for data. To add to the confusion, the operating system will probably report this capacity as 16.0 GB, not 17.2 GB because software considers 1 GB to be 2^{30} (1,073,741,824) bytes, not 10^9 (1,000,000,000) bytes.

To make things worse, in the data communication world, 1 Gbps means 1,000,000,000 bits/sec because the prefix *giga* really does mean 10^9 (a kilometer is 1000 meters, not 1024 meters, after all). Only with memory and disk sizes do kilo, mega, giga, and tera mean 2^{10}, 2^{20}, 2^{30}, and 2^{40}, respectively.

Formatting also affects performance. If a 10,000-rpm disk has 300 sectors per track of 512 bytes each, it takes 6 msec to read the 153,600 bytes on a track for a data rate of 25,600,000 bytes/sec or 24.4 MB/sec. It is not possible to go faster than this, no matter what kind of interface is present, even if it a SCSI interface at 80 MB/sec or 160 MB/sec.

Actually reading continuously at this rate requires a large buffer in the controller. Consider, for example, a controller with a one-sector buffer that has been given a command to read two consecutive sectors. After reading the first sector from the disk and doing the ECC calculation, the data must be transferred to main memory. While this transfer is taking place, the next sector will fly by the head. When the copy to memory is complete, the controller will have to wait almost an entire rotation time for the second sector to come around again.

This problem can be eliminated by numbering the sectors in an interleaved fashion when formatting the disk. In Fig. 5-26(a), we see the usual numbering pattern (ignoring cylinder skew here). In Fig. 5-26(b), we see **single interleaving**, which gives the controller some breathing space between consecutive sectors in order to copy the buffer to main memory.

If the copying process is very slow, the **double interleaving** of Fig. 5-27(c) may be needed. If the controller has a buffer of only one sector, it does not matter whether the copying from the buffer to main memory is done by the controller, the main CPU, or a DMA chip; it still takes some time. To avoid the need for interleaving, the controller should be able to buffer an entire track. Many modern controllers can do this.

After low-level formatting is completed, the disk is partitioned. Logically, each partition is like a separate disk. On the Pentium and most other computers, sector 0 contains the **master boot record**, which contains some boot code plus

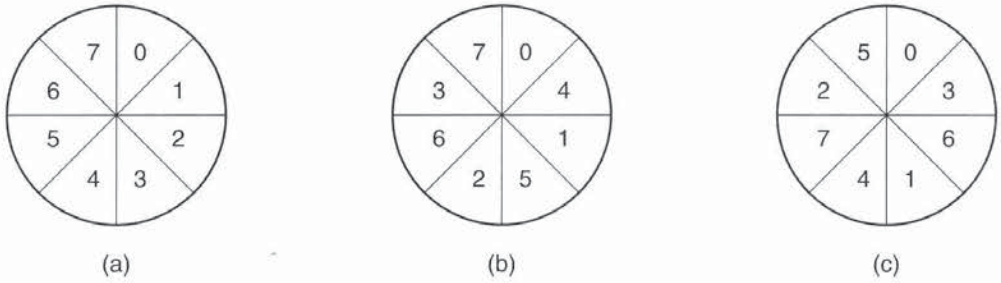


Figure 5-26. (a) No interleaving. (b) Single interleaving. (c) Double interleaving.

the partition table at the end. The partition table gives the starting sector and size of each partition. On the Pentium, the partition table has room for four partitions. If all of them are for Windows, they will be called C:, D:, E:, and F: and treated as separate drives. If three of them are for Windows and one is for UNIX, then Windows will call its partitions C:, D:, and E:. The first CD-ROM will then be F:. To be able to boot from the hard disk, one partition must be marked as active in the partition table.

The final step in preparing a disk for use is to perform a **high-level format** of each partition (separately). This operation lays down a boot block, the free storage administration (free list or bitmap), root directory, and an empty file system. It also puts a code in the partition table entry telling which file system is used in the partition because many operating systems support multiple incompatible file systems (for historical reasons). At this point the system can be booted.

When the power is turned on, the BIOS runs initially and then reads in the master boot record and jumps to it. This boot program then checks to see which partition is active. Then it reads in the boot sector from that partition and runs it. The boot sector contains a small program that searches the root directory for a certain program (either the operating system or a larger bootstrap loader). That program is loaded into memory and executed.

5.4.3 Disk Arm Scheduling Algorithms

In this section we will look at some issues related to disk drivers in general. First, consider how long it takes to read or write a disk block. The time required is determined by three factors:

1. Seek time (the time to move the arm to the proper cylinder).
2. Rotational delay (the time for the proper sector to rotate under the head).
3. Actual data transfer time.

For most disks, the seek time dominates the other two times, so reducing the mean seek time can improve system performance substantially.

If the disk driver accepts requests one at a time and carries them out in that order, that is, First-Come, First-Served (FCFS), little can be done to optimize seek time. However, another strategy is possible when the disk is heavily loaded. It is likely that while the arm is seeking on behalf of one request, other disk requests may be generated by other processes. Many disk drivers maintain a table, indexed by cylinder number, with all the pending requests for each cylinder chained together in a linked list headed by the table entries.

Given this kind of data structure, we can improve upon the first-come, first-served scheduling algorithm. To see how, consider an imaginary disk with 40 cylinders. A request comes in to read a block on cylinder 11. While the seek to cylinder 11 is in progress, new requests come in for cylinders 1, 36, 16, 34, 9, and 12, in that order. They are entered into the table of pending requests, with a separate linked list for each cylinder. The requests are shown in Fig. 5-27.

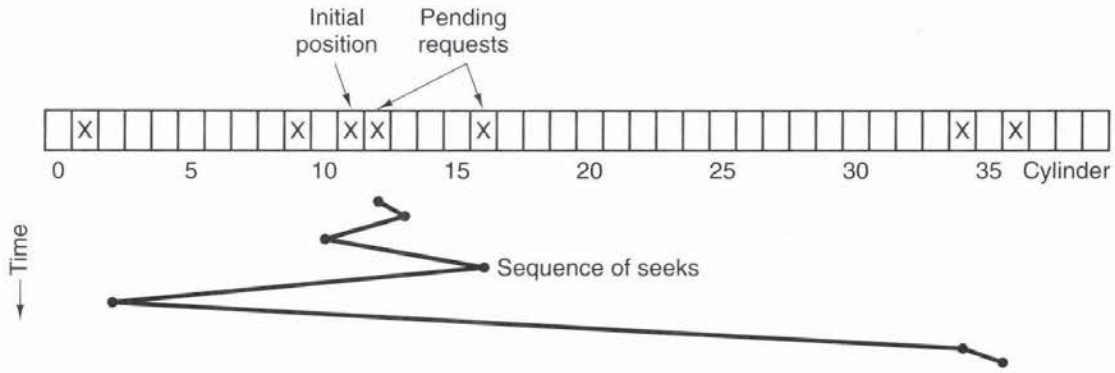


Figure 5-27. Shortest Seek First (SSF) disk scheduling algorithm.

When the current request (for cylinder 11) is finished, the disk driver has a choice of which request to handle next. Using FCFS, it would go next to cylinder 1, then to 36, and so on. This algorithm would require arm motions of 10, 35, 20, 18, 25, and 3, respectively, for a total of 111 cylinders.

Alternatively, it could always handle the closest request next, to minimize seek time. Given the requests of Fig. 5-27, the sequence is 12, 9, 16, 1, 34, and 36, as shown as the jagged line at the bottom of Fig. 5-27. With this sequence, the arm motions are 1, 3, 7, 15, 33, and 2, for a total of 61 cylinders. This algorithm, **Shortest Seek First** (SSF), cuts the total arm motion almost in half compared to FCFS.

Unfortunately, SSF has a problem. Suppose more requests keep coming in while the requests of Fig. 5-27 are being processed. For example, if, after going to cylinder 16, a new request for cylinder 8 is present, that request will have priority over cylinder 1. If a request for cylinder 13 then comes in, the arm will next go to 13, instead of 1. With a heavily loaded disk, the arm will tend to stay in the middle of the disk most of the time, so requests at either extreme will have to wait until a statistical fluctuation in the load causes there to be no requests near the

middle. Requests far from the middle may get poor service. The goals of minimal response time and fairness are in conflict here.

Tall buildings also have to deal with this trade-off. The problem of scheduling an elevator in a tall building is similar to that of scheduling a disk arm. Requests come in continuously calling the elevator to floors (cylinders) at random. The computer running the elevator could easily keep track of the sequence in which customers pushed the call button and service them using FCFS. It could also use SSF.

However, most elevators use a different algorithm to reconcile the conflicting goals of efficiency and fairness. They keep moving in the same direction until there are no more outstanding requests in that direction, then they switch directions. This algorithm, known both in the disk world and the elevator world as the **elevator algorithm**, requires the software to maintain 1 bit: the current direction bit, *UP* or *DOWN*. When a request finishes, the disk or elevator driver checks the bit. If it is *UP*, the arm or cabin is moved to the next highest pending request. If no requests are pending at higher positions, the direction bit is reversed. When the bit is set to *DOWN*, the move is to the next lowest requested position, if any.

Figure 5-28 shows the elevator algorithm using the same seven requests as Fig. 5-27, assuming the direction bit was initially *UP*. The order in which the cylinders are serviced is 12, 16, 34, 36, 9, and 1, which yields arm motions of 1, 4, 18, 2, 27, and 8, for a total of 60 cylinders. In this case the elevator algorithm is slightly better than SSF, although it is usually worse. One nice property that the elevator algorithm has is that given any collection of requests, the upper bound on the total motion is fixed: it is just twice the number of cylinders.

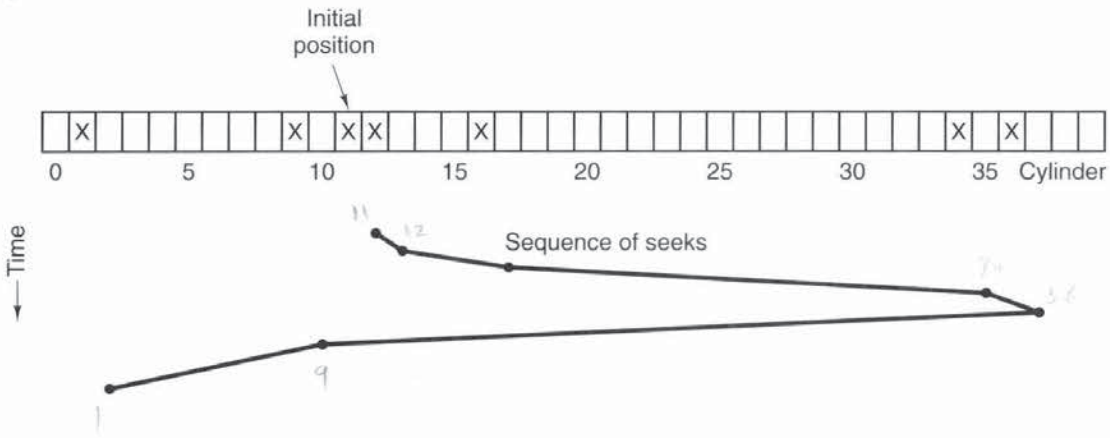


Figure 5-28. The elevator algorithm for scheduling disk requests.

A slight modification of this algorithm that has a smaller variance in response times (Teory, 1972) is to always scan in the same direction. When the highest numbered cylinder with a pending request has been serviced, the arm goes to the lowest-numbered cylinder with a pending request and then continues moving in an

upward direction. In effect, the lowest-numbered cylinder is thought of as being just above the highest-numbered cylinder.

Some disk controllers provide a way for the software to inspect the current sector number under the head. With such a controller, another optimization is possible. If two or more requests for the same cylinder are pending, the driver can issue a request for the sector that will pass under the head next. Note that when multiple tracks are present in a cylinder, consecutive requests can be for different tracks with no penalty. The controller can select any of its heads instantaneously, because head selection involves neither arm motion nor rotational delay.

If the disk has the property that the seek time is much faster than the rotational delay, then a different optimization strategy should be used. Pending requests should be sorted by sector number, and as a soon as the next sector is about to pass under the head, the arm should be zipped over to the right track to read or write it.

With a modern hard disk, the seek and rotational delays so dominate performance that reading one or two sectors at a time is very inefficient. For this reason, many disk controllers always read and cache multiple sectors, even when only one is requested. Typically any request to read a sector will cause that sector and much or all the rest of the current track to be read, depending upon how much space is available in the controller's cache memory. The disk described in Fig. 5-17 has a 2-MB or 4-MB cache, for example. The use of the cache is determined dynamically by the controller. In its simplest mode, the cache is divided into two sections, one for reads and one for writes. If a subsequent read can be satisfied out of the controller's cache, it can return the requested data immediately.

It is worth noting that the disk controller's cache is completely independent of the operating system's cache. The controller's cache usually holds blocks that have not actually been requested, but which were convenient the read because they just happened to pass under the head as a side effect of some other read. In contrast, any cache maintained by the operating system will consist of blocks that were explicitly read and which the operating system thinks might be needed again in the near future (e.g., a disk block holding a directory block).

When several drives are present on the same controller, the operating should maintain a pending request table for each drive separately. Whenever any drive is idle, a seek should be issued to move its arm to the cylinder where it will be needed next (assuming the controller allows overlapped seeks). When the current transfer finishes, a check can be made to see if any drives are positioned on the correct cylinder. If one or more are, the next transfer can be started on a drive that is already on the right cylinder. If none of the arms is in the right place, the driver should issue a new seek on the drive that just completed a transfer and wait until the next interrupt to see which arm gets to its destination first.

It is important to realize that all of the above disk scheduling algorithms tacitly assume that the real disk geometry is the same as the virtual geometry. If it is not, then scheduling disk requests makes no sense because the operating system

cannot really tell whether cylinder 40 or cylinder 200 is closer to cylinder 39. On the other hand, if the disk controller can accept multiple outstanding requests, it can use these scheduling algorithms internally. In that case, the algorithms are still valid, but one level down, inside the controller.

5.4.4 Error Handling

Disk manufacturers are constantly pushing the limits of the technology by increasing linear bit densities. A track midway out on a 5.25-inch disk has a circumference of about 300 mm. If the track holds 300 sectors of 512 bytes, the linear recording density may be about 5000 bits/mm taking into account the fact that some space is lost to preambles, ECCs, and intersector gaps. Recording 5000 bits/mm requires an extremely uniform substrate and a very fine oxide coating. Unfortunately, it is not possible to manufacture a disk to such specifications without defects. As soon as manufacturing technology has improved to the point where it is possible to operate flawlessly at such densities, disk designers will go to higher densities to increase the capacity. Doing so will probably reintroduce defects.

Manufacturing defects introduce bad sectors, that is, sectors that do not correctly read back the value just written to them. If the defect is very small, say, only a few bits, it is possible to use the bad sector and just let the ECC correct the errors every time. If the defect is bigger, the error cannot be masked.

There are two general approaches to bad blocks: deal with them in the controller or deal with them in the operating system. In the former approach, before the disk is shipped from the factory, it is tested and a list of bad sectors is written onto the disk. For each bad sector, one of the spares is substituted for it.

There are two ways to do this substitution. In Fig. 5-29(a), we see a single disk track with 30 data sectors and two spares. Sector 7 is defective. What the controller can do is remap one of the spares as sector 7 as shown in Fig. 5-29(b). The other way is to shift all the sectors up one, as shown in Fig. 5-29(c). In both cases the controller has to know which sector is which. It can keep track of this information through internal tables (one per track) or by rewriting the preambles to give the remapped sector numbers. If the preambles are rewritten, the method of Fig. 5-29(c) is more work (because 23 preambles must be rewritten) but ultimately gives better performance because an entire track can still be read in one rotation.

Errors can also develop during normal operation after the drive has been installed. The first line of defense upon getting an error that the ECC cannot handle is to just try the read again. Some read errors are transient, that is, are caused by specks of dust under the head and will go away on a second attempt. If the controller notices that it is getting repeated errors on a certain sector, it can switch to a spare before the sector has died completely. In this way, no data are lost and the operating system and user do not even notice the problem. Usually, the

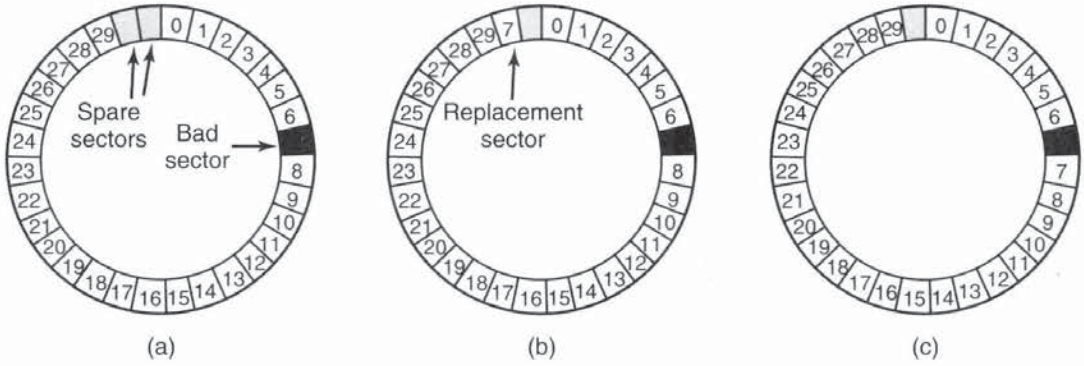


Figure 5-29. (a) A disk track with a bad sector. (b) Substituting a spare for the bad sector. (c) Shifting all the sectors to bypass the bad one.

method of Fig. 5-29(b) has to be used since the other sectors might now contain data. Using the method of Fig. 5-29(c) would require not only rewriting the preambles, but copying all the data as well.

Earlier we said there were two general approaches to handling errors: handle them in the controller or in the operating system. If the controller does not have the capability to transparently remap sectors as we have discussed, the operating system must do the same thing in software. This means that it must first acquire a list of bad sectors, either by reading them from the disk, or simply testing the entire disk itself. Once it knows which sectors are bad, it can build remapping tables. If the operating system wants to use the approach of Fig. 5-29(c), it must shift the data in sectors 7 through 29 up one sector.

If the operating system is handling the remapping, it must make sure that bad sectors do not occur in any files and also do not occur in the free list or bitmap. One way to do this is to create a secret file consisting of all the bad sectors. If this file is not entered into the file system, users will not accidentally read it (or worse yet, free it).

However, there is still another problem: backups. If the disk is backed up file by file, it is important that the backup utility not try to copy the bad block file. To prevent this, the operating system has to hide the bad block file so well that even a backup utility cannot find it. If the disk is backed up sector by sector rather than file by file, it will be difficult, if not impossible, to prevent read errors during backup. The only hope is that the backup program has enough smarts to give up after 10 failed reads and continue with the next sector.

Bad sectors are not the only source of errors. Seek errors caused by mechanical problems in the arm also occur. The controller keeps track of the arm position internally. To perform a seek, it issues a series of pulses to the arm motor, one pulse per cylinder, to move the arm to the new cylinder. When the arm gets to its destination, the controller reads the actual cylinder number from the preamble of the next sector. If the arm is in the wrong place, a seek error has occurred.

Most hard disk controllers correct seek errors automatically, but most floppy controllers (including the Pentium's) just set an error bit and leave the rest to the driver. The driver handles this error by issuing a recalibrate command, to move the arm as far out as it will go and reset the controller's internal idea of the current cylinder to 0. Usually this solves the problem. If it does not, the drive must be repaired.

As we have seen, the controller is really a specialized little computer, complete with software, variables, buffers, and occasionally, bugs. Sometimes an unusual sequence of events such as an interrupt on one drive occurring simultaneously with a recalibrate command for another drive will trigger a bug and cause the controller to go into a loop or lose track of what it was doing. Controller designers usually plan for the worst and provide a pin on the chip which, when asserted, forces the controller to forget whatever it was doing and reset itself. If all else fails, the disk driver can set a bit to invoke this signal and reset the controller. If that does not help, all the driver can do is print a message and give up.

Recalibrating a disk makes a funny noise but otherwise normally is not disturbing. However, there is one situation where recalibration is a serious problem: systems with real-time constraints. When a video is being played off a hard disk, or files from a hard disk are being burned onto a CD-ROM, it is essential that the bits arrive from the hard disk at a uniform rate. Under these circumstances, recalibrations insert gaps into the bit stream and are therefore unacceptable. Special drives, called **AV disks** (**Audio Visual disks**), which never recalibrate are available for such applications.

5.4.5 Stable Storage

As we have seen, disks sometimes make errors. Good sectors can suddenly become bad sectors. Whole drives can die unexpectedly. RAIDs protect against a few sectors going bad or even a drive falling out. However, they do not protect against write errors laying down bad data in the first place. They also do not protect against crashes during writes corrupting the original data without replacing them by newer data.

For some applications, it is essential that data never be lost or corrupted, even in the face of disk and CPU errors. Ideally, a disk should simply work all the time with no errors. Unfortunately, that is not achievable. What is achievable is a disk subsystem that has the following property: when a write is issued to it, the disk either correctly writes the data or it does nothing, leaving the existing data intact. Such as system is called **stable storage** and is implemented in software (Lampson and Sturgis, 1979). Below we will describe a slight variant of the original idea.

Before describing the algorithm, it is important to have a clear model of the possible errors. The model assumes that when a disk writes a block (one or more sectors), either the write is correct or it is incorrect and this error can be detected on a subsequent read by examining the values of the ECC fields. In principle,

guaranteed error detection is never possible because with a, say, 16-byte ECC field guarding a 512-byte sector, there are 2^{4096} data values and only 2^{144} ECC values. Thus if a block is garbled during writing but the ECC is not, there are billions upon billions of incorrect combinations that yield the same ECC. If any of them occur, the error will not be detected. On the whole, the probability of random data having the proper 16-byte ECC is about 2^{-144}, which is small enough that we will call it zero, even though it is really not.

The model also assumes that a correctly written sector can spontaneously go bad and become unreadable. However, the assumption is that such events are so rare that having the same sector go bad on a second (independent) drive during a reasonable time interval (e.g., 1 day) is small enough to ignore.

The model also assumes the CPU can fail, in which case it just stops. Any disk write in progress at the moment of failure also stops, leading to incorrect data in one sector and an incorrect ECC that can later be detected. Under all these conditions, stable storage can be made 100% reliable in the sense of writes either working correctly or leaving the old data in place. Of course, it does not protect against physical disasters, such as an earthquake happening and the computer falling 100 meters into a fissure and landing in a pool of boiling magma. It is tough to recover from this condition in software.

Stable storage uses a pair of identical disks with the corresponding blocks working together to form one error-free block. In the absence of errors, the corresponding blocks on both drives are the same. Either one can be read to get the same result. To achieve this goal, the following three operations are defined:

1. **Stable writes**. A stable write consists of first writing the block on drive 1, then reading it back to verify that it was written correctly. If it was not written correctly, the write and reread are done again up to n times until they work. After n consecutive failures, the block is remapped onto a spare and the operation repeated until it succeeds, no matter how many spares have to be tried. After the write to drive 1 has succeeded, the corresponding block on drive 2 is written and reread, repeatedly if need be, until it, too, finally succeeds. In the absence of CPU crashes, when a stable write completes, the block has correctly been written onto both drives and verified on both of them.

2. **Stable reads**. A stable read first reads the block from drive 1. If this yields an incorrect ECC, the read is tried again, up to n times. If all of these give bad ECCs, the corresponding block is read from drive 2. Given the fact that a successful stable write leaves two good copies of the block behind, and our assumption that the probability of the same block spontaneously going bad on both drives in a reasonable time interval is negligible, a stable read always succeeds.

3. **Crash recovery**. After a crash, a recovery program scans both disks comparing corresponding blocks. If a pair of blocks are both good and the same, nothing is done. If one of them has an ECC error, the bad block is overwritten with the corresponding good block. If a pair of blocks are both good but different, the block from drive 1 is written onto drive 2.

In the absence of CPU crashes, this scheme always works because stable writes always write two valid copies of every block and spontaneous errors are assumed never to occur on both corresponding blocks at the same time. What about in the presence of CPU crashes during stable writes? It depends on precisely when the crash occur. There are five possibilities, as depicted in Fig. 5-30.

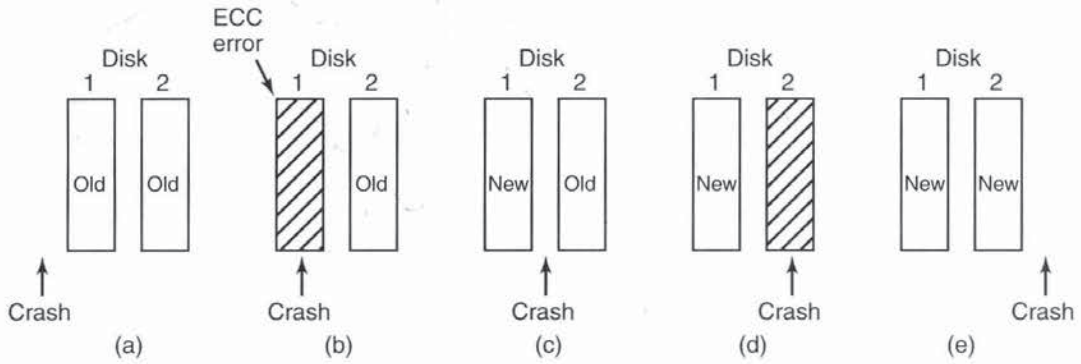


Figure 5-30. Analysis of the influence of crashes on stable writes.

In Fig. 5-30(a), the CPU crash happens before either copy of the block is written. During recovery, neither will be changed and the old value will continue to exist, which is allowed.

In Fig. 5-30(b), the CPU crashes during the write to drive 1, destroying the contents of the block. However the recovery program detects this error and restores the block on drive 1 from drive 2. Thus the effect of the crash is wiped out and the old state is fully restored.

In Fig. 5-30(c), the CPU crash happens after drive 1 is written but before drive 2 is written. The point of no return has been passed here: the recovery program copies the block from drive 1 to drive 2. The write succeeds.

Fig. 5-30(d) is like Fig. 5-30(b): during recovery, the good block overwrites the bad block. Again, the final value of both blocks is the new one.

Finally, in Fig. 5-30(e) the recovery program sees that both blocks are the same, so neither is changed and the write succeeds here too.

Various optimizations and improvements are possible to this scheme. For starters, comparing all the blocks pairwise after a crash is doable, but expensive. A huge improvement is to keep track of which block was being written during a

stable write so that only one block has to be checked during recovery. Some computers have a small amount of **nonvolatile RAM** which is a special CMOS memory powered by a lithium battery. Such batteries last for years, possibly even the whole life of the computer. Unlike main memory, which is lost after a crash, nonvolatile RAM is not lost after a crash. The time of day is normally kept here (and incremented by a special circuit), which is why computers still know what time it is even after having been unplugged.

Suppose that a few bytes of nonvolatile RAM are available for operating system purposes. The stable write can put the number of the block it is about to update in nonvolatile RAM before starting the write. After successfully completing the stable write, the block number in nonvolatile RAM is overwritten with an invalid block number, for example, −1. Under these conditions, after a crash the recovery program can check the nonvolatile RAM to see if a stable write happened to be in progress during the crash, and if so, which block was being written when the crashed happened. The two copies of the block can then be checked for correctness and consistency.

If nonvolatile RAM is not available, it can be simulated as follows. At the start of a stable write, a fixed disk block on drive 1 is overwritten with the number of the block to be stably written. This block is then read back to verify it. After getting it correct, the corresponding block on drive 2 is written and verified. When the stable write completes correctly, both blocks are overwritten with an invalid block number and verified. Again here, after a crash it is easy to determine whether or not a stable write was in progress during the crash. Of course, this technique requires eight extra disk operations to write a stable block, so it should be used exceedingly sparingly.

One last point is worth making. We assumed that only one spontaneous decay of a good block to a bad block happens per block pair per day. If enough days go by, the other one might go bad too. Therefore, once a day a complete scan of both disks must be done repairing any damage. That way, every morning both disks are always identical. Even if both blocks in a pair go bad within a period of a few days, all errors are repaired correctly.

5.5 CLOCKS

Clocks (also called **timers**) are essential to the operation of any multiprogrammed system for a variety of reasons. They maintain the time of day and prevent one process from monopolizing the CPU, among other things. The clock software can take the form of a device driver, even though a clock is neither a block device, like a disk, nor a character device, like a mouse. Our examination of clocks will follow the same pattern as in the previous section: first a look at clock hardware and then a look at the clock software.

5.5.1 Clock Hardware

Two types of clocks are commonly used in computers, and both are quite different from the clocks and watches used by people. The simpler clocks are tied to the 110- or 220-volt power line and cause an interrupt on every voltage cycle, at 50 or 60 Hz. These clocks used to dominate, but are rare nowadays.

The other kind of clock is built out of three components: a crystal oscillator, a counter, and a holding register, as shown in Fig. 5-31. When a piece of quartz crystal is properly cut and mounted under tension, it can be made to generate a periodic signal of very high accuracy, typically in the range of several hundred megahertz, depending on the crystal chosen. Using electronics, this base signal can be multiplied by a small integer to get frequencies up to 1000 MHz or even more. At least one such circuit is usually found in any computer, providing a synchronizing signal to the computer's various circuits. This signal is fed into the counter to make it count down to zero. When the counter gets to zero, it causes a CPU interrupt.

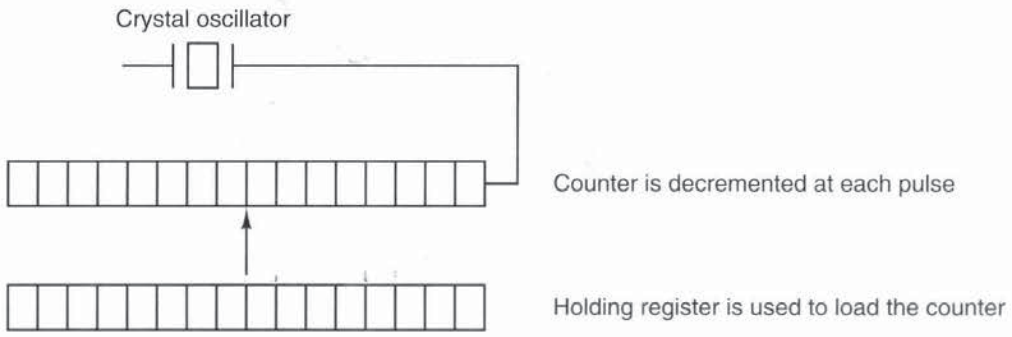


Figure 5-31. A programmable clock.

Programmable clocks typically have several modes of operation. In **one-shot mode**, when the clock is started, it copies the value of the holding register into the counter and then decrements the counter at each pulse from the crystal. When the counter gets to zero, it causes an interrupt and stops until it is explicitly started again by the software. In **square-wave mode**, after getting to zero and causing the interrupt, the holding register is automatically copied into the counter, and the whole process is repeated again indefinitely. These periodic interrupts are called **clock ticks**.

The advantage of the programmable clock is that its interrupt frequency can be controlled by software. If a 500-MHz crystal is used, then the counter is pulsed every 2 nsec. With (unsigned) 32-bit registers, interrupts can be programmed to occur at intervals from 2 nsec to 8.6 sec. Programmable clock chips usually contain two or three independently programmable clocks and have many other options as well (e.g., counting up instead of down, interrupts disabled, and more).

To prevent the current time from being lost when the computer's power is turned off, most computers have a battery-powered backup clock, implemented with the kind of low-power circuitry used in digital watches. The battery clock can be read at startup. If the backup clock is not present, the software may ask the user for the current date and time. There is also a standard way for a networked system to get the current time from a remote host. In any case the time is then translated into the number of clock ticks since 12 A.M. **UTC (Universal Coordinated Time)** (formerly known as Greenwich Mean Time) on Jan. 1, 1970, as UNIX does, or since some other benchmark. The origin of time for Windows is Jan. 1, 1980. At every clock tick, the real time is incremented by one count. Usually utility programs are provided to manually set the system clock and the backup clock and to synchronize the two clocks.

5.5.2 Clock Software

All the clock hardware does is generate interrupts at known intervals. Everything else involving time must be done by the software, the clock driver. The exact duties of the clock driver vary among operating systems, but usually include most of the following:

1. Maintaining the time of day.
2. Preventing processes from running longer than they are allowed to.
3. Accounting for CPU usage.
4. Handling the alarm system call made by user processes.
5. Providing watchdog timers for parts of the system itself.
6. Doing profiling, monitoring, and statistics gathering.

The first clock function, maintaining the time of day (also called the **real time**) is not difficult. It just requires incrementing a counter at each clock tick, as mentioned before. The only thing to watch out for is the number of bits in the time-of-day counter. With a clock rate of 60 Hz, a 32-bit counter will overflow in just over 2 years. Clearly the system cannot store the real time as the number of ticks since Jan. 1, 1970 in 32 bits.

Three approaches can be taken to solve this problem. The first way is to use a 64-bit counter, although doing so makes maintaining the counter more expensive since it has to be done many times a second. The second way is to maintain the time of day in seconds, rather than in ticks, using a subsidiary counter to count ticks until a whole second has been accumulated. Because 2^{32} seconds is more than 136 years, this method will work until the twenty-second century.

The third approach is to count in ticks, but to do that relative to the time the system was booted, rather than relative to a fixed external moment. When the

backup clock is read or the user types in the real time, the system boot time is calculated from the current time-of-day value and stored in memory in any convenient form. Later, when the time of day is requested, the stored time of day is added to the counter to get the current time of day. All three approaches are shown in Fig. 5-32.

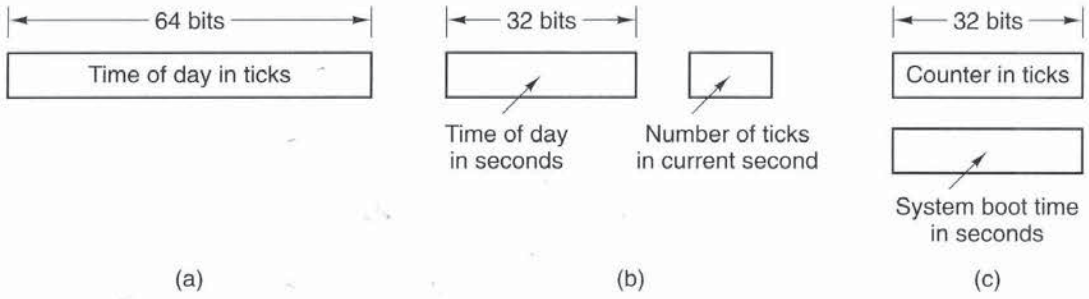


Figure 5-32. Three ways to maintain the time of day.

The second clock function is preventing processes from running too long. Whenever a process is started, the scheduler initializes a counter to the value of that process' quantum in clock ticks. At every clock interrupt, the clock driver decrements the quantum counter by 1. When it gets to zero, the clock driver calls the scheduler to set up another process.

The third clock function is doing CPU accounting. The most accurate way to do it is to start a second timer, distinct from the main system timer, whenever a process is started. When that process is stopped, the timer can be read out to tell how long the process has run. To do things right, the second timer should be saved when an interrupt occurs and restored afterward.

A less accurate, but much simpler, way to do accounting is to maintain a pointer to the process table entry for the currently running process in a global variable. At every clock tick, a field in the current process' entry is incremented. In this way, every clock tick is "charged" to the process running at the time of the tick. A minor problem with this strategy is that if many interrupts occur during a process' run, it is still charged for a full tick, even though it did not get much work done. Properly accounting for the CPU during interrupts is too expensive and is rarely done.

In many systems, a process can request that the operating system give it a warning after a certain interval. The warning is usually a signal, interrupt, message, or something similar. One application requiring such warnings is networking, in which a packet not acknowledged within a certain time interval must be retransmitted. Another application is computer-aided instruction, where a student not providing a response within a certain time is told the answer.

If the clock driver had enough clocks, it could set a separate clock for each request. This not being the case, it must simulate multiple virtual clocks with a single physical clock. One way is to maintain a table in which the signal time for

all pending timers is kept, as well as a variable giving the time of the next one. Whenever the time of day is updated, the driver checks to see if the closest signal has occurred. If it has, the table is searched for the next one to occur.

If many signals are expected, it is more efficient to simulate multiple clocks by chaining all the pending clock requests together, sorted on time, in a linked list, as shown in Fig. 5-33. Each entry on the list tells how many clock ticks following the previous one to wait before causing a signal. In this example, signals are pending for 4203, 4207, 4213, 4215, and 4216.

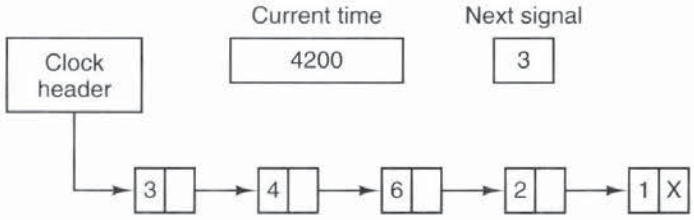


Figure 5-33. Simulating multiple timers with a single clock.

In Fig. 5-33, the next interrupt occurs in 3 ticks. On each tick, *Next signal* is decremented. When it gets to 0, the signal corresponding to the first item on the list is caused, and that item is removed from the list. Then *Next signal* is set to the value in the entry now at the head of the list, in this example, 4.

Note that during a clock interrupt, the clock driver has several things to do—increment the real time, decrement the quantum and check for 0, do CPU accounting, and decrement the alarm counter. However, each of these operations has been carefully arranged to be very fast because they have to be repeated many times a second.

Parts of the operating system also need to set timers. These are called **watchdog timers**. For example, floppy disks do not rotate when not in use, to avoid wear and tear on the medium and disk head. When data are needed from a floppy disk, the motor must first be started. Only when the floppy disk is rotating at full speed can I/O begin. When a process attempts to read from an idle floppy disk, the floppy disk driver starts the motor and then sets a watchdog timer to cause an interrupt after a sufficiently long time interval (because there is no up-to-speed interrupt from the floppy disk itself).

The mechanism used by the clock driver to handle watchdog timers is the same as for user signals. The only difference is that when a timer goes off, instead of causing a signal, the clock driver calls a procedure supplied by the caller. The procedure is part of the caller's code. The called procedure can do whatever is necessary, even causing an interrupt, although within the kernel interrupts are often inconvenient and signals do not exist. That is why the watchdog mechanism is provided. It is worth nothing that the watchdog mechanism works only when the clock driver and the procedure to be called are in the same address space.

The last thing in our list is profiling. Some operating systems provide a mechanism by which a user program can have the system build up a histogram of its program counter, so it can see where it is spending its time. When profiling is a possibility, at every tick the driver checks to see if the current process is being profiled, and if so, computes the bin number (a range of addresses) corresponding to the current program counter. It then increments that bin by one. This mechanism can also be used to profile the system itself.

5.5.3 Soft Timers

Most computers have a second programmable clock that can be set to cause timer interrupts at whatever rate a program needs. This timer is in addition to the main system timer whose functions were described above. As long as the interrupt frequency is low, there is no problem using this second timer for application-specific purposes. The trouble arrives when the frequency of the application-specific timer is very high. Below we will briefly describe a software-based timer scheme that works well under many circumstances, even at high frequencies. The idea is due to Aron and Druschel (1999). For more details, please see their paper.

Generally, there are two ways to manage I/O: interrupts and polling. Interrupts have low latency, that is, they happen immediately after the event itself with little or no delay. On the other hand, with modern CPUs, interrupts have a substantial overhead due to the need for context switching and their influence on the pipeline, TLB and cache.

The alternative to interrupts is to have the application poll for the event expected itself. Doing this avoids interrupts, but there may be substantial latency because an event may happen directly after a poll, in which case it waits almost a whole polling interval. On the average, the latency is half the polling interval.

For some applications, neither the overhead of interrupts nor the latency of polling is acceptable. Consider, for example, a high-performance network such as Gigabit Ethernet. This network is capable of accepting or delivering a full-size packet every 12 μsec. To run at optimal performance on output, one packet should be sent every 12 μsec.

One way to achieve this rate is to have the completion of a packet transmission cause an interrupt or to set the second timer to interrupt every 12 μsec. The problem is that this interrupt has been measured to take 4.45 μsec on a 300 MHz Pentium II (Aron and Druschel, 1999). This overhead is barely better than that of computers in the 1970s. On most minicomputers, for example, an interrupt took four bus cycles: to stack the program counter and PSW and to load a new program counter and PSW. Nowadays dealing with the pipeline, MMU, TLB, and cache, add a great deal to the overhead. These effects are likely to get worse rather than better in time, thus canceling out faster clock rates.

Soft timers avoid interrupts. Instead, whenever the kernel is running for some other reason, just before it returns to user mode it checks the real time clock

to see if a soft timer has expired. If the timer has expired, the scheduled event (e.g., packet transmission or checking for an incoming packet) is performed, with no need to switch into kernel mode since the system is already there. After the work has been performed, the soft timer is reset to go off again. All that has to be done is copy the current clock value to the timer and add the timeout interval to it.

Soft timers stand or fall with the rate at which kernel entries are made for other reasons. These reasons include

1. System calls.
2. TLB misses.
3. Page faults.
4. I/O interrupts.
5. The CPU going idle.

To see how often these events happen, Aron and Druschel made measurements with several CPU loads, including a fully-loaded Web server, a Web server with a compute-bound background job, playing real-time audio from the Internet, and recompiling the UNIX kernel. The average entry rate into the kernel varied from 2 μsec to 18 μsec, with about half of these entries being system calls. Thus to a first-order approximation, having a soft timer go off every 12 μsec is doable, albeit with an occasional missed deadline. For applications like sending packets or polling for incoming packets, being 10 μsec late from time to time is better than having interrupts eat up 35% of the CPU.

Of course, there will be periods when there are no system calls, TLB misses, or page faults, in which case no soft timers will go off. To put an upper bound on these intervals, the second hardware timer can be set to go off, say, every 1 msec. If the application can live with only 1000 packets/sec for occasional intervals, then the combination of soft timers and a low-frequency hardware timer may be better than either pure interrupt-driven I/O or pure polling.

5.6 CHARACTER-ORIENTED TERMINALS

Every general-purpose computer has at least one keyboard and one display (monitor or flat screen) used to communicate with it. Although the keyboard and display on a personal computer are technically separate devices, they work closely together. On mainframes, there are frequently many remote users, each with a device containing a keyboard and an attached display. These devices have historically been called **terminals**. We will continue to use that term, even when discussing personal computers (mostly for lack of a better term).

Terminals come in many forms. Three of the types most commonly encountered in practice nowadays are

1. Standalone terminals with RS-232 serial interfaces for use on mainframes.
2. Personal computer displays with graphical user interfaces.
3. Network terminals.

Each type of terminal has its own ecological niche. In the following sections we will describe each of these types in turn.

5.6.1 RS-232 Terminal Hardware

RS-232 terminals are hardware devices containing both a keyboard and a display and which communicate using a serial interface, one bit at a time (see Fig. 5-34). These terminals use a 9-pin or 25-pin connector, of which one pin is used for transmitting data, one pin is for receiving data, and one pin is ground. The other pins are for various control functions, most of which are not used. Lines in which characters are sent one bit at a time (as opposed to 8 bits in parallel the way printers are interfaced to PCs) are called **serial lines**. All modems also use this interface. On UNIX, serial lines have names like */dev/tty1* and */dev/tty2*. On Windows they have names like *COM1* and *COM2*.

To send a character over a serial line to an RS-232 terminal or modem, the computer must transmit it 1 bit at a time, prefixed by a start bit, and followed by 1 or 2 stop bits to delimit the character. A parity bit which provides rudimentary error detection may also be inserted preceding the stop bits, although this is commonly required only for communication with mainframe systems.

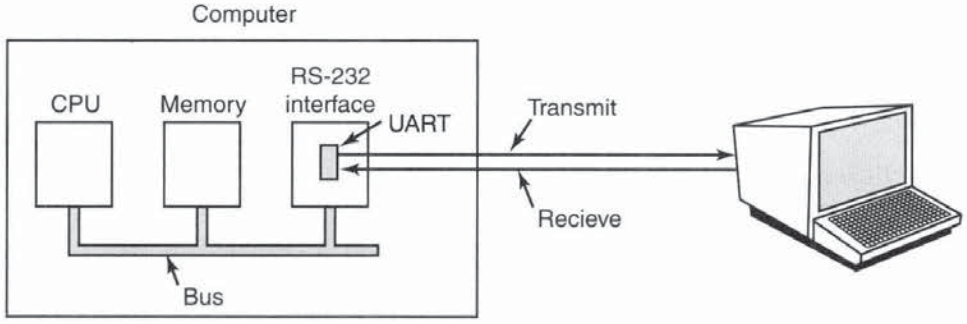


Figure 5-34. An RS-232 terminal communicates with a computer over a communication line, one bit at a time.

RS-232 terminals are still commonly used in the mainframe world to allow a remote user to communicate with the mainframe, sometimes using a modem and a telephone line. They are found in the airline, banking, and other industries. Even

when they are replaced by personal computers, the PCs often simply emulate the old RS-232 terminals to avoid having to change the mainframe software.

These terminals also used to dominate the minicomputer world. A great deal of software for systems that grew up in this period are based on these terminals. For example, all UNIX systems support this kind of device.

However, even more important, many current UNIX systems (and other systems) provide the option of creating a window consisting of some number of lines of text. Many programmers work almost exclusively in text mode in such windows, even on personal computers or high-end workstations. These windows usually simulate some RS-232 terminal (or the ANSI standard for this type of terminal) so they can run the large existing software base that was written for such terminals. In the course of the years, this software, such as the *vi* and *emacs* editors, has become completely debugged and extremely stable, properties programmers value highly.

The keyboard and terminal software for these terminal emulation windows is the same as for the real terminals. Since these terminal emulators are in widespread use, the software is still important, so we will describe it in the following two sections.

RS-232 terminals are character oriented. What this means is that the screen or window displays a certain number of lines of text, each of a maximum size. A typical size is 25 lines of 80 characters each. While a few special characters are sometimes supported, these terminals (and the emulators) are basically text only.

Since both computers and terminals work internally with whole characters but must communicate over a serial line a bit at a time, chips have been developed to do the character-to-serial and serial-to-character conversions. They are called **UART**s (Universal Asynchronous Receiver Transmitters). UARTs are attached to the computer by plugging RS-232 interface cards into the bus as illustrated in Fig. 5-34. On many computers, one or two serial ports are built into the parentboard.

To display a character, the terminal driver writes the character to the interface card, where it is buffered and then shifted out over the serial line one bit at a time by the UART. For example, for an analog modem operating at 56,000 bps, it takes just over 179 μsec to send a character. As a result of this slow transmission rate, the driver generally outputs a character to the RS-232 card and blocks, waiting for the interrupt generated by the interface when the character has been transmitted and the UART is able to accept another character. The UART can send and receive characters simultaneously. An interrupt is also generated when a character is received, and usually a small number of input characters can be buffered. The terminal driver must check a register when an interrupt is received to determine the cause of the interrupt. Some interface cards have a CPU and memory and can handle multiple lines, taking over much of the I/O load from the main CPU.

RS-232 terminals can be subdivided into three categories. The simplest ones are hardcopy (i.e., printing) terminals. Characters typed on the keyboard are

transmitted to the computer. Characters sent by the computer are printed on the paper. These terminals are obsolete and rarely seen any more except as low-end printers.

Dumb CRT terminals work the same way, only with a screen instead of paper. These are often called "glass ttys" because they are functionally the same as hardcopy ttys. (The term "tty" is an abbreviation for Teletype®, a former company that pioneered in the computer terminal business; "tty" has come to mean any terminal.) Glass ttys are also obsolete.

Intelligent CRT terminals are in fact miniature, specialized computers. They have a CPU and memory and contain software, usually in ROM. From the operating system's viewpoint, the main difference between a glass tty and an intelligent terminal is that the latter understands certain escape sequences. For example, by sending the ASCII ESC character (0x1B), followed by various other characters, it may be possible to move the cursor to any position on the screen, insert text in the middle of the screen, and so forth. Intelligent terminals are the ones used in mainframe systems and are the ones emulated by other operating systems. It is their software that we will discuss below.

5.6.2 Input Software

The keyboard and display are almost independent devices, so we will treat them separately here. They are not quite independent, however, since typed characters generally are displayed on the screen.

The basic job of the keyboard driver is to collect input from the keyboard and pass it to user programs when they read from the terminal. Two possible philosophies can be adopted for the driver. In the first one, the driver's job is just to accept input and pass it upward unmodified. A program reading from the terminal gets a raw sequence of ASCII codes. (Giving user programs the key numbers is too primitive, as well as being highly machine dependent.)

This philosophy is well suited to the needs of sophisticated screen editors such as *emacs*, which allow the user to bind an arbitrary action to any character or sequence of characters. It does, however, mean that if the user types *dste* instead of *date* and then corrects the error by typing three backspaces and *ate*, followed by a carriage return, the user program will be given all 11 ASCII codes typed, as follows:

d s t e ← ← ← a t e CR

Not all programs want this much detail. Often they just want the corrected input, not the exact sequence of how it was produced. This observation leads to the second philosophy: the driver handles all the intraline editing, and just delivers corrected lines to the user programs. The first philosophy is character-oriented; the second one is line oriented. Originally they were referred to as **raw mode** and **cooked mode**, respectively. The POSIX standard uses the less-picturesque term

canonical mode to describe line-oriented mode. **Noncanonical mode** is equivalent to raw mode, although many details of terminal behavior can be changed. POSIX-compatible systems provide several library functions that support selecting either mode and changing many aspects of terminal configuration.

The first task of the keyboard driver is to collect characters. If every keystroke causes an interrupt, the driver can acquire the character during the interrupt. If interrupts are turned into messages by the low-level software, it is possible to put the newly acquired character in the message. Alternatively, it can be put in a small buffer in memory and the message used to tell the driver that something has arrived. The latter approach is actually safer if a message can be sent only to a waiting process and there is some chance that the keyboard driver might still be busy with the previous character.

If the terminal is in canonical (cooked) mode, characters must be stored until an entire line has been accumulated, because the user may subsequently decide to erase part of it. Even if the terminal is in raw mode, the program may not yet have requested input, so the characters must be buffered to allow type ahead. (System designers who do not allow users to type far ahead ought to be tarred and feathered, or worse yet, be forced to use their own system.)

Two approaches to character buffering are common. In the first one, the driver contains a central pool of buffers, each buffer holding perhaps 10 characters. Associated with each terminal is a data structure, which contains, among other items, a pointer to the chain of buffers for input collected from that terminal. As more characters are typed, more buffers are acquired and hung on the chain. When the characters are passed to a user program, the buffers are removed and put back in the central pool.

The other approach is to do the buffering directly in the terminal data structure itself, with no central pool of buffers. Since it is common for users to type a command that will take a little while (say, recompiling and linking a large binary program) and then type a few lines ahead, to be safe the driver should allocate something like 200 characters per terminal. In a large-scale timesharing system with 100 terminals, allocating 20K all the time for type ahead is clearly overkill, so a central buffer pool with space for perhaps 5K is probably enough. On the other hand, a dedicated buffer per terminal makes the driver simpler (no linked list management) and is to be preferred on personal computers with only one keyboard. Figure 5-35 shows the difference between these two methods.

Although the keyboard and display are logically separate devices, many users have grown accustomed to seeing the characters they have just typed appear on the screen. Some (older) terminals oblige by automatically displaying (in hardware) whatever has just been typed, which is not only a nuisance when passwords are being entered but greatly limits the flexibility of sophisticated editors and other programs. Fortunately, with most terminals, nothing is automatically displayed when a key is struck. It is entirely up to the software in the computer to display the character, if desired. This process is called **echoing**.

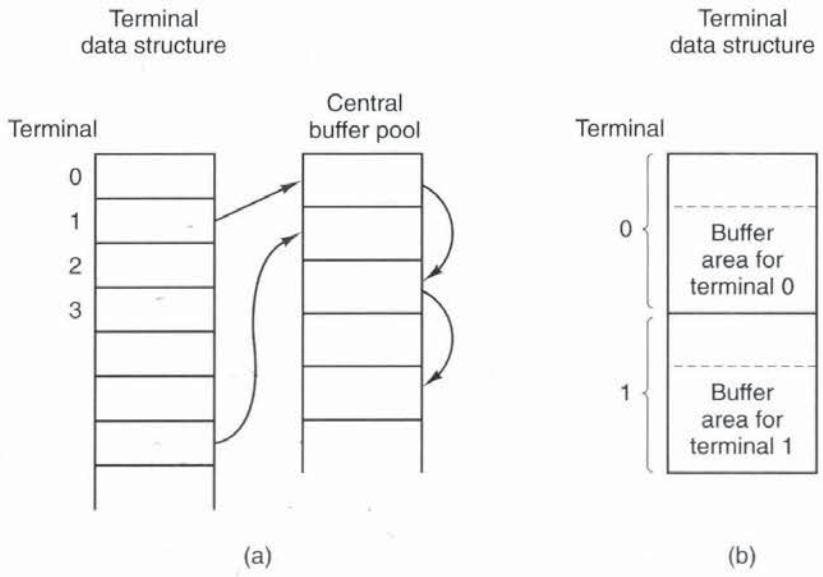


Figure 5-35. (a) Central buffer pool. (b) Dedicated buffer for each terminal.

Echoing is complicated by the fact that a program may be writing to the screen while the user is typing. At the very least, the keyboard driver has to figure out where to put the new input without it being overwritten by program output.

Echoing also gets complicated when more than 80 characters have to be displayed on a screen with 80-character lines (or some other number). Depending on the application, wrapping around to the next line may be appropriate. Some drivers just truncate lines to 80 characters by throwing away all characters beyond column 80.

Another problem is tab handling. It is usually up to the driver to compute where the cursor is currently located, taking into account both output from programs and output from echoing, and compute the proper number of spaces to be echoed.

Now we come to the problem of device equivalence. Logically, at the end of a line of text, one wants a carriage return, to move the cursor back to column 1, and a linefeed, to advance to the next line. Requiring users to type both at the end of each line would not sell well (although some terminals have a key which generates both, with a 50 percent chance of doing so in the order that the software wants them). It is up to the driver to convert whatever comes in to the standard internal format used by the operating system.

If the standard form is just to store a linefeed (the UNIX convention), then carriage returns should be turned into linefeeds. If the internal format is to store both (the Windows convention), then the driver should generate a linefeed when it gets

a carriage return and a carriage return when it gets a linefeed. No matter what the internal convention, the terminal may require both a linefeed and a carriage return to be echoed in order to get the screen updated properly. Since a large computer may well have a wide variety of different terminals connected to it, it is up to the keyboard driver to get all the different carriage return/linefeed combinations converted to the internal system standard and arrange for all echoing to be done right.

When operating in canonical mode, a number of input characters have special meanings. Figure 5-36 shows all of the special characters required by POSIX. The defaults are all control characters that should not conflict with text input or codes used by programs, but all except the last two can be changed under program control.

Character	POSIX name	Comment
CTRL-H	ERASE	Backspace one character
CTRL-U	KILL	Erase entire line being typed
CTRL-V	LNEXT	Interpret next character literally
CTRL-S	STOP	Stop output
CTRL-Q	START	Start output
DEL	INTR	Interrupt process (SIGINT)
CTRL-\	QUIT	Force core dump (SIGQUIT)
CTRL-D	EOF	End of file
CTRL-M	CR	Carriage return (unchangeable)
CTRL-J	NL	Linefeed (unchangeable)

Figure 5-36. Characters that are handled specially in canonical mode.

The *ERASE* character allows the user to rub out the character just typed. It is usually the backspace (CTRL-H). It is not added to the character queue but instead removes the previous character from the queue. It should be echoed as a sequence of three characters, backspace, space, and backspace, in order to remove the previous character from the screen. If the previous character was a tab, erasing it depends on how it was processed when it was typed. If it is immediately expanded into spaces, some extra information is needed to determine how far to back up. If the tab itself is stored in the input queue, it can be removed and the entire line just output again. In most systems, backspacing will only erase characters on the current line. It will not erase a carriage return and back up into the previous line.

When the user notices an error at the start of the line being typed in, it is often convenient to erase the entire line and start again. The *KILL* character erases the entire line. Most systems make the erased line vanish from the screen, but a few echo it plus a carriage return and linefeed because some users like to see the old

line. Consequently, how to echo *KILL* is a matter of taste. As with *ERASE* it is usually not possible to go further back than the current line. When a block of characters is killed, it may or may not be worth the trouble for the driver to return buffers to the pool, if one is used.

Sometimes the *ERASE* or *KILL* characters must be entered as ordinary data. The *LNEXT* character serves as an **escape character**. In UNIX CTRL-V is the default. As an example, older UNIX systems often used the @ sign for *KILL*, but the Internet mail system uses addresses of the form *linda@cs.washington.edu*. Someone who feels more comfortable with older conventions might redefine *KILL* as @, but then need to enter an @ sign literally to address email. This can be done by typing CTRL-V @. The CTRL-V itself can be entered literally by typing CTRL-V CTRL-V. After seeing a CTRL-V, the driver sets a flag saying that the next character is exempt from special processing. The *LNEXT* character itself is not entered in the character queue.

To allow users to stop a screen image from scrolling out of view, control codes are provided to freeze the screen and restart it later. In UNIX these are *STOP*, (CTRL-S) and *START*, (CTRL-Q), respectively. They are not stored but are used to set and clear a flag in the terminal data structure. Whenever output is attempted, the flag is inspected. If it is set, no output occurs. Usually, echoing is also suppressed along with program output.

It is often necessary to kill a runaway program being debugged. The *INTR* (DEL) and *QUIT* (CTRL-\) characters can be used for this purpose. In UNIX, DEL sends the SIGINT signal to all the processes started up from the terminal. Implementing DEL can be quite tricky. The hard part is getting the information from the driver to the part of the system that handles signals, which, after all, has not asked for this information. CTRL-\ is similar to DEL, except that it sends the SIGQUIT signal, which forces a core dump if not caught or ignored. When either of these keys is struck, the driver should echo a carriage return and linefeed and discard all accumulated input to allow for a fresh start. The default value for *INTR* is often CTRL-C instead of DEL, since many programs use DEL interchangeably with the backspace for editing.

Another special character is *EOF* (CTRL-D), which in UNIX causes any pending read requests for the terminal to be satisfied with whatever is available in the buffer, even if the buffer is empty. Typing CTRL-D at the start of a line causes the program to get a read of 0 bytes, which is conventionally interpreted as end-of-file and causes most programs to act the same way as they would upon seeing end-of-file on an input file.

Some terminal drivers allow much fancier intraline editing than we have sketched here. They have special control characters to erase a word, skip backward or forward characters or words, go to the beginning or end of the line being typed, inserting text in the middle of the line, and so forth. Adding all these functions to the terminal driver makes it much larger and, furthermore, is wasted when using fancy screen editors that work in raw mode anyway.

5.6.3 Output Software

Output is simpler than input. For the most part, the computer sends characters to the terminal and they are displayed there. Usually, a block of characters, for example, a line, is written to the terminal in one system call. The method that is commonly used for RS-232 terminals is to have output buffers associated with each terminal. The buffers can come from the same pool as the input buffers, or be dedicated, as with input. When a programs writes to the terminal, the output is first copied to the buffer. Similarly, output from echoing is also copied to the buffer. After all the output has been copied to the buffer, the first character is output, and the driver goes to sleep. When the interrupt comes in, the next character is output, and so on.

Screen editors and many other sophisticated programs need to be able to update the screen in complex ways such as replacing one line in the middle of the screen. To accommodate this need, most terminals support a a series of commands to move the cursor, insert and delete characters or lines at the cursor, etc. These commands are often called **escape sequences**. In the heyday of the RS-232 terminal, there were hundreds of terminal types, each with its own escape sequences. As a consequence, it was difficult to write software that worked on more than one terminal type.

One solution, which was introduced in Berkeley UNIX, was a terminal database called **termcap**. This software package defined a number of basic actions, such as moving the cursor to (*row*, *column*). To move the cursor to a particular location, the software, say, an editor, used a generic escape sequence which was then converted to the actual escape sequence for the terminal being written to. In this way, the editor worked on any terminal that had an entry in the termcap database.

Eventually, the industry saw the need for standardization of the escape sequence, so an ANSI standard was developed. A few of the values are shown in Fig. 5-37.

Consider how these escape sequences might be used by a text editor. Suppose that the user types a command telling the editor to delete all of line 3 and then close up the gap between lines 2 and 4. The editor might send the following escape sequence over the serial line to the terminal:

```
ESC [ 3 ; 1 H ESC [ 0 K ESC [ 1 M
```

(where the spaces are used above only to separate the symbols; they are not transmitted). This sequence moves the cursor to the start of line 3, erases the entire line, and then deletes the now-empty line, causing all the lines starting at 5 to move up 1 line. Then what was line 4 becomes line 3; what was line 5 becomes line 4, and so on. Analogous escape sequences can be used to add text to the middle of the display. Words and be added or removed in a similar way.

Escape sequence	Meaning
ESC [*n* A	Move up *n* lines
ESC [*n* B	Move down *n* lines
ESC [*n* C	Move right *n* spaces
ESC [*n* D	Move left *n* spaces
ESC [*m* ; *n* H	Move cursor to (*m*,*n*)
ESC [*s* J	Clear screen from cursor (0 to end, 1 from start, 2 all)
ESC [*s* K	Clear line from cursor (0 to end, 1 from start, 2 all)
ESC [*n* L	Insert *n* lines at cursor
ESC [*n* M	Delete *n* lines at cursor
ESC [*n* P	Delete *n* chars at cursor
ESC [*n* @	Insert *n* chars at cursor
ESC [*n* m	Enable rendition *n* (0=normal, 4=bold, 5=blinking, 7=reverse)
ESC M	Scroll the screen backward if the cursor is on the top line

Figure 5-37. The ANSI escape sequences accepted by the terminal driver on output. ESC denotes the ASCII escape character (0x1B), and *n*, *m*, and *s* are optional numeric parameters.

5.7 GRAPHICAL USER INTERFACES

PCs can use character-based interfaces. In fact, for years MS-DOS, which is character based, dominated the scene. However, nowadays most personal computers use a **GUI** (**Graphical User Interface**). The acronym GUI is pronounced "gooey."

The GUI was invented by Douglas Engelbart and his research group at the Stanford Research Institute. It was then copied by researchers at Xerox PARC. One fine day, Steve Jobs, cofounder of Apple, was touring PARC and saw a GUI on a Xerox computer. This gave him the idea for a new computer, which became the Apple Lisa. The Lisa was too expensive and was a commercial failure, but its successor, the Macintosh, was a huge success. The Macintosh was the inspiration for Microsoft Windows and other GUI-based systems.

A GUI has four essential elements, denoted by the characters WIMP. These letters stand for Windows, Icons, Menus, and Pointing device, respectively. Windows are rectangular blocks of screen area used to run programs. Icons are little symbols that can be clicked on to cause some action to happen. Menus are lists of actions from which one can be chosen. Finally, a pointing device is a mouse, trackball, or other hardware device used to move a cursor around the screen to select items.

The GUI software can be implemented in either user-level code, as is done in UNIX systems, or in the operating system itself, as in the case in Windows. In the

following sections we will look at the hardware, input software, and output software associated with personal computer GUIs with an emphasis on Windows, but the general ideas hold for other GUIs as well.

5.7.1 Personal Computer Keyboard, Mouse, and Display Hardware

Modern personal computers all have a keyboard and a bit-oriented memory-mapped display. These components are an integral part of the computer itself. However, the keyboard and screen are completely separated, each with their own driver.

The keyboard may be interfaced via a serial port, a parallel port, or a USB port. On every key action the CPU is interrupted, and the keyboard driver extracts the character typed by reading an I/O port. Everything else happens in software, mostly in the keyboard driver.

On a Pentium, the keyboard contains an embedded microprocessor which communicates through a specialized serial port with a controller chip on the parentboard. An interrupt is generated whenever a key is struck and also when one is released. Furthermore, all that the keyboard hardware provides is the key number, not the ASCII code. When the *A* key is struck, the key code (30) is put in an I/O register. It is up to the driver to determine whether it is lower case, upper case, CTRL-A, ALT-A, CTRL-ALT-A, or some other combination. Since the driver can tell which keys have been struck but not yet released (e.g., SHIFT), it has enough information to do the job.

For example, the key sequence

```
DEPRESS SHIFT, DEPRESS A, RELEASE A, RELEASE SHIFT
```

indicates an upper case A. However, the key sequence

```
DEPRESS SHIFT, DEPRESS A, RELEASE SHIFT, RELEASE A
```

also indicates an upper case A. Although this keyboard interface puts the full burden on the software, it is extremely flexible. For example, user programs may be interested in whether a digit just typed came from the top row of keys or the numeric key pad on the side. In principle, the driver can provide this information.

Most PCs have a mouse, or sometimes a trackball, which is just a mouse lying on its back. The most common type of mouse has a rubber ball inside that protrudes through a hole in the bottom and rotates as the mouse is moved over a rough surface. As the ball rotates, it rubs against rubber rollers placed on orthogonal shafts. Motion in the east-west direction causes the shaft parallel to the *y*-axis to rotate; motion in the north-south direction causes the shaft parallel to the *x*-axis to rotate. Whenever the mouse has moved a certain minimum distance in either direction or a button is depressed or released, a message is sent to the computer. The minimum distance is about 0.1 mm (although it can be set in software). Some people call this unit a **mickey**. Mice (or occasionally, mouses) can have