
FROM I/O PORTS TO PROCESS MANAGEMENT

Understanding the
**LINUX
KERNEL**



O'REILLY®

DANIEL P. BOVET & MARCO CESATI

Understanding the Linux Kernel

Daniel P. Bovet
Marco Cesati

Publisher: O'Reilly

First Edition October 2000
ISBN: 0-596-00002-2, 702 pages

Understanding the Linux Kernel helps readers understand how Linux performs best and how it meets the challenge of different environments. The authors introduce each topic by explaining its importance, and show how kernel operations relate to the utilities that are familiar to Unix programmers and users.

The `do_anonymous_page( )` function handles write and read requests separately:

```
if (write_access) {
    page = __get_free_page(GFP_USER);
    memset((void *) (page), 0, PAGE_SIZE);
    entry = pte_mkwrite(pte_mkdirty(mk_pte(page,
        vma->vm_page_prot)));
    vma->vm_mm->rss++;
    tsk->min_flt++;
    set_pte(pte, entry);
    return 1;
}
```

When handling a write access, the function invokes `__get_free_page( )` and fills the new page frame with zeros by using the `memset` macro. The function then increments the `min_flt` field of `tsk` to keep track of the number of *minor page faults* (those that require only a new page frame) caused by the process and the `rss` field of the `vma->vm_mm` process memory descriptor to keep track of the number of page frames allocated to the process.^[6] The Page Table entry is then set to the physical address of the page frame, which is marked as writable and dirty.

^[6] Linux records the number of minor page faults for each process. This information, together with several other statistics, may be used to tune the system. The value stored in the `rss` field of memory descriptors is also used by the kernel to select the region from which to steal page frames (see [Section 16.7](#) in [Chapter 16](#)).

Conversely, when handling a read access, the content of the page is irrelevant because the process is addressing it for the first time. It is safer to give to the process a page filled with zeros rather than an old page filled with information written by some other process. Linux goes one step further in the spirit of demand paging. There is no need to assign a new page frame filled with zeros to the process right away, since we might as well give it an existing page called *zero page*, thus deferring further page frame allocation. The zero page is allocated statically during kernel initialization in the `empty_zero_page` variable (an array of 1024 long integers filled with zeros); it is stored in the sixth page frame, starting from physical address `0x00005000`, and it can be referenced by means of the `ZERO_PAGE` macro.

The Page Table entry is thus set with the physical address of the zero page:

```
entry = pte_wrprotect(mk_pte(ZERO_PAGE, vma->vm_page_prot));
set_pte(pte, entry);
return 1;
```

Since the page is marked as nonwritable, if the process attempts to write in it, the Copy On Write mechanism will be activated. Then, and only then, will the process get a page of its own to write in. The mechanism is described in the next section.

7.4.4 Copy On Write

First-generation Unix systems implemented process creation in a rather clumsy way: when a `fork( )` system call was issued, the kernel duplicated the whole parent address space in the literal sense of the word and assigned the copy to the child process. This activity was quite time-consuming since it required:

- Allocating page frames for the page tables of the child process
- Allocating page frames for the pages of the child process
- Initializing the page tables of the child process
- Copying the pages of the parent process into the corresponding pages of the child process

This way of creating an address space involved many memory accesses, used up many CPU cycles, and entirely spoiled the cache contents. Last but not least, it was often pointless because many child processes start their execution by loading a new program, thus discarding entirely the inherited address space (see [Chapter 19](#)).

Modern Unix kernels, including Linux, follow a more efficient approach called *Copy On Write*, or *COW*. The idea is quite simple: instead of duplicating page frames, they are shared between the parent and the child process. However, as long as they are shared, they cannot be modified. Whenever the parent or the child process attempts to write into a shared page frame, an exception occurs, and at this point the kernel duplicates the page into a new page frame that it marks as writable. The original page frame remains write-protected: when the other process tries to write into it, the kernel checks whether the writing process is the only owner of the page frame; in such a case, it makes the page frame writable for the process.

The `count` field of the page descriptor is used to keep track of the number of processes that are sharing the corresponding page frame. Whenever a process releases a page frame or a Copy On Write is executed on it, its `count` field is decremented; the page frame is freed only when `count` becomes NULL.

Let us now describe how Linux implements COW. When `handle_pte_fault( )` determines that the "Page fault" exception was caused by a request to write into a write-protected page present in memory, it executes the following instructions:

```
if (pte_present(pte)) {
    entry = pte_mkyoung(entry);
    set_pte(pte, entry);
    flush_tlb_page(vma, address);
    if (write_access) {
        if (!pte_write(entry))
            return do_wp_page(tsk, vma, address, pte);
        entry = pte_mkdirty(entry);
        set_pte(pte, entry);
        flush_tlb_page(vma, address);
    }
    return 1;
}
```

First, the `pte_mkyoung( )` and `set_pte( )` functions are invoked in order to set the Accessed bit in the Page Table entry of the page that caused the exception. This setting makes the page "younger" and reduces its chances of being swapped out to disk (see [Chapter 16](#)). If the exception was caused by a write-protection violation, `handle_pte_fault( )` returns the value yielded by the `do_wp_page( )` function; otherwise, some error condition has been detected (for instance, a page inside the User Mode process address space with the User/Supervisor flag equal to 0), and the function returns the value 1.

The `do_wp_page( )` function starts by loading the `pte` local variable with the Page Table entry referenced by the `page_table` parameter and by getting a new page frame:

```
pte = *page_table;
new_page = __get_free_page(GFP_USER);
```

Since the allocation of a page frame can block the process, the function performs the following consistency checks on the Page Table entry once the page frame has been obtained:

- Whether the page has been swapped out while the process waited for a free page frame (`pte` and `*page_table` do not have the same value)
- Whether the page is no longer in RAM (the page's `Present` flag is in its Page Table entry)
- Whether the page can now be written (the page's `Read/Write` flag is 1 in its Page Table entry)

If any of these conditions occurs, `do_wp_page( )` releases the page frame obtained previously and returns the value 1.

Now the function updates the number of minor page faults and stores in the `page_map` local variable a pointer to the page descriptor of the page that caused the exception:

```
tsk->minflt++;
page_map = mem_map + MAP_NR(old_page);
```

Next, the function must determine whether the page must really be duplicated. If only one process owns the page, Copy On Write does not apply and the process should be free to write the page. Thus, the page frame is marked as writable so that it will not cause further "Page fault" exceptions when writes are attempted, the previously allocated new page frame is released, and the function terminates with a return value of 1. This check is made by reading the value of the `count` field of the page descriptor:^[7]

^[7] Actually, the check is slightly more complicated, since the `count` field is also incremented when the page is inserted into the swap cache (see [Section 16.3 in Chapter 16](#)).

```
if (page_map->count == 1) {
    set_pte(page_table, pte_mkdirty(pte_mkwrite(pte)));
    flush_tlb_page(vma, address);
    if (new_page)
        free_page(new_page);
    return 1;
}
```

Conversely, if the page frame is shared among two or more processes, the function copies the content of the old page frame (`old_page`) into the newly allocated one (`new_page`):

```

if (old_page == ZERO_PAGE)
    memset((void *) new_page, 0, PAGE_SIZE);
else
    memcpy((void *) new_page, (void *) old_page, PAGE_SIZE);
set_pte(page_table, pte_mkwrite(pte_mkdirty(
    mk_pte(new_page, vma->vm_page_prot))));
flush_tlb_page(vma, address);
__free_page(page_map);
return 1;

```

If the old page is the zero page, the new frame is efficiently filled with zeros by using the `memset` macro. Otherwise, the page frame content is copied using the `memcpy` macro. Special handling for the zero page is not strictly required, but it improves the system performance since it preserves the microprocessor hardware cache by making fewer address references.

The Page Table entry is then updated with the physical address of the new page frame, which is also marked as writable and dirty. Finally, the function invokes `__free_page( )` to decrement the usage counter of the old page frame.

7.5 Creating and Deleting a Process Address Space

Out of the six typical cases mentioned in [Section 7.1](#) in which a process gets new memory regions, the first one—issuing a `fork( )` system call—requires the creation of a whole new address space for the child process. Conversely, when a process terminates, the kernel destroys its address space. In this section we'll discuss how these two activities are performed by Linux.

7.5.1 Creating a Process Address Space

We have mentioned in [Section 3.3.1](#) in [Chapter 3](#), that the kernel invokes the `copy_mm( )` function while creating a new process. This function takes care of the process address space creation by setting up all page tables and memory descriptors of the new process.

Each process usually has its own address space, but lightweight processes can be created by calling `__clone( )` with the `CLONE_VM` flag set. These share the same address space; that is, they are allowed to address the same set of pages.

Following the COW approach described earlier, traditional processes inherit the address space of their parent: pages stay shared as long as they are only read. When one of the processes attempts to write one of them, however, the page is duplicated; after some time, a forked process usually gets its own address space different from that of the parent process. Lightweight processes, on the other hand, use the address space of their parent process: Linux implements them simply by not duplicating address space. Lightweight processes can be created considerably faster than normal processes, and the sharing of pages can also be considered a benefit so long as the parent and children coordinate their accesses carefully.

If the new process has been created by means of the `__clone( )` system call and if the `CLONE_VM` flag of the `flag` parameter is set, `copy_mm( )` gives the clone the address space of its parent: