

The page table, together with the program counter and the registers, specifies the state of a program. If we want to allow another program to use the CPU, we must save this state. Later, after restoring this state, the program can continue execution. We often refer to this state as a *process*. The process is considered *active* when it is in possession of the CPU and; otherwise, it is considered *inactive*. The operating system can make a process active by loading the process's state, including the program counter, which will initiate execution at the value of the saved program counter. The process's address space, and hence all the data it can access in memory, is defined by its page table, which resides in memory. Rather than save the entire page table, the operating system simply loads the page table register to point to the page table of the process it wants to make active.

Page Faults

If the valid bit for a virtual page is off, a page fault occurs. The operating system must be given control. This is done with the exception mechanism, the details of which we discuss later in this section. Once the operating system gets control, it must find the page in the next level of the hierarchy (usually magnetic disk) and decide where to place the requested page in physical memory. The virtual address alone does not immediately tell us where the page is on disk. Returning to our library analogy, we cannot find the location of a library book on the shelves just by knowing its title. Instead, we go to the catalog and look up the book, obtaining an address for the location on the shelves (for example, the Library of Congress number). Likewise, in a virtual memory system, we must keep track of the location on disk of each page in the virtual address space. Because we do not know ahead of time when a page will be chosen to be replaced, the operating system usually creates the space on disk for all the pages of a process when it creates the process. At that time, it also creates a data structure to record where each virtual page is stored on disk. This data structure may be part of the page table, or may be an auxiliary data structure indexed in the same way as the page table. Figure 7.17 shows the organization when a single table holds either the physical page number or the disk address.

Assuming that all the pages in physical memory are in use, the operating system must choose a page to replace. Because we want to minimize the number of page faults, most operating systems try to choose a page that they hy-

Implementing Protection with Virtual Memory

Although each process has its own virtual address space, the physical memory is shared among multiple user processes and also with the operating system. Nevertheless, we do not want a renegade process to be able to write into the address space of another user process or into the operating system. For example, if the program that maintains student grades is running on a machine at the same time as the programs of the students in the first programming course, we wouldn't want the errant program of a beginner to write over someone's grades. We also want to prevent one process from reading the data of another process. For example, we wouldn't want one student program to read the grades while they were in the processor's memory. Once we begin sharing physical memory, we must provide the ability for a process to protect its data both from reading and writing by another process.

Remember that each process has its own virtual address space. Thus, if the operating system keeps the page tables organized so that the independent virtual pages map to disjoint physical pages, one process will not be able to access another's data. Of course, this also requires that a user process not be able to change the page table mapping. The operating system can assure this if it prevents the user process from modifying its own page tables. Yet the operating system must be able to modify the page tables.

Hardware Software Interface

To enable the operating system to implement protection in the virtual memory system, the hardware must provide at least the three basic capabilities summarized below.

1. Support at least two modes that indicate whether the running process is a user process or an operating system process, variously called a *kernel* process, a *supervisor* process, or an *executive* process.
2. Provide a portion of the CPU state that a user process can read but not write. This includes the user/supervisor mode bit(s) and the page table pointer.
3. Provide mechanisms whereby the CPU can go from user mode to supervisor mode, and vice versa. The first direction is typically accomplished by a *system call* exception, implemented as a special instruction (*syscall* in the MIPS instruction set) that transfers control to a dedicated location in supervisor code space. As with any other exception, the program counter from the point of the system call is saved, and the CPU is placed in supervisor mode. The return to user mode from the exception will restore the state of the process that generated the exception.

With these mechanisms the operating system can change the page tables as well as prevent a user process from changing them, ensuring that a user process can access only the storage provided to it by the operating system.

A.4**Loading**

A program that links without an error can be run. Before being run, the program resides in a file on secondary storage, such as a disk. On Unix systems, the operating system kernel brings a program into memory and starts it running. To start a program, the operating system performs the following steps:

1. Reads the executable file's header to determine the size of the text and data segments.
2. Creates a new address space for the program. This address space is large enough to hold the text and data segments, along with a stack segment (see section A.5).
3. Copies instructions and data from the executable file into the new address space.
4. Copies arguments passed to the program onto the stack.
5. Initializes the machine registers. In general, most registers are cleared, but the stack pointer must be assigned the address of the first free stack location (see section A.5).
6. Jumps to a start-up routine that copies the program's arguments from the stack to registers and calls the program's `main` routine. If the `main` routine returns, the start-up routine terminates the program with the `exit` system call.

A.5**Memory Usage**

The next few sections elaborate the description of the MIPS architecture presented earlier in the book. Earlier chapters focused primarily on hardware and its relationship with low-level software. These sections focus primarily on how assembly language programmers use MIPS hardware. These sections describe a set of conventions followed on many MIPS systems. For the most part, the hardware does not impose these conventions. Instead, they represent an agreement among programmers to follow the same set of rules so that software written by different people can work together and make effective use of MIPS hardware.

Systems based on MIPS processors typically divide memory into three parts (see Figure A.8). The first part, near the bottom of the address space

Most of the bookkeeping associated with a call is centered around a block of memory called a *procedure call frame*. This memory is used for a variety of purposes:

- To hold values passed to a procedure as arguments.
- To save registers that a procedure may modify, but which the procedure's caller does not want changed.
- To provide space for variables local to a procedure.

In most programming languages, procedure calls and returns follow a strict last-in, first-out (LIFO) order so this memory can be allocated and deallocated on a stack, which is why these blocks of memory are sometimes called *stack frames*.

Elaboration: A programming language that does not permit recursive procedures—procedures that call themselves either directly or indirectly through a chain of calls—need not allocate frames on a stack. In a nonrecursive language, each procedure's frame may be statically allocated since only one invocation of a procedure can be active at a time. Older versions of Fortran prohibited recursion because statically-allocated frames produced faster code on some older machines. However, on load-store architectures like MIPS, stack frames may be just as fast because a frame pointer register points directly to the active stack frame, which permits a single load or store instruction to access values in the frame. In addition, recursion is a valuable programming technique.

Figure A.10 shows a typical stack frame. The frame consists of the memory between the frame pointer (`$fp`), which points to the first word of the frame, and the stack pointer (`$sp`), which points to the first stack word after the frame. The stack grows down from higher memory addresses, so the frame pointer points above the stack pointer. The executing procedure uses the frame pointer to quickly access values in its stack frame. For example, an argument in the stack frame can be loaded into register `$2` with the instruction:

```
lw $2, 0($fp)
```

A stack frame may be built in many different ways; however, the caller and callee must agree on the sequence of steps. The steps below describe the calling convention used on most MIPS machines. This convention comes into play at three points during a procedure call: immediately before the caller invokes the callee, just as the callee starts executing, and immediately before the callee

Example

Figure A.11 shows the stack at the call: `fact(7)`. `main` runs first, so its frame is deepest on the stack. `main` calls `fact(10)`, whose stack frame is next on the stack. Each invocation recursively invokes `fact` to compute the next-lowest factorial. The stack frames parallel the LIFO order of these calls. What does the stack look like when the call to `fact(6)` returns?

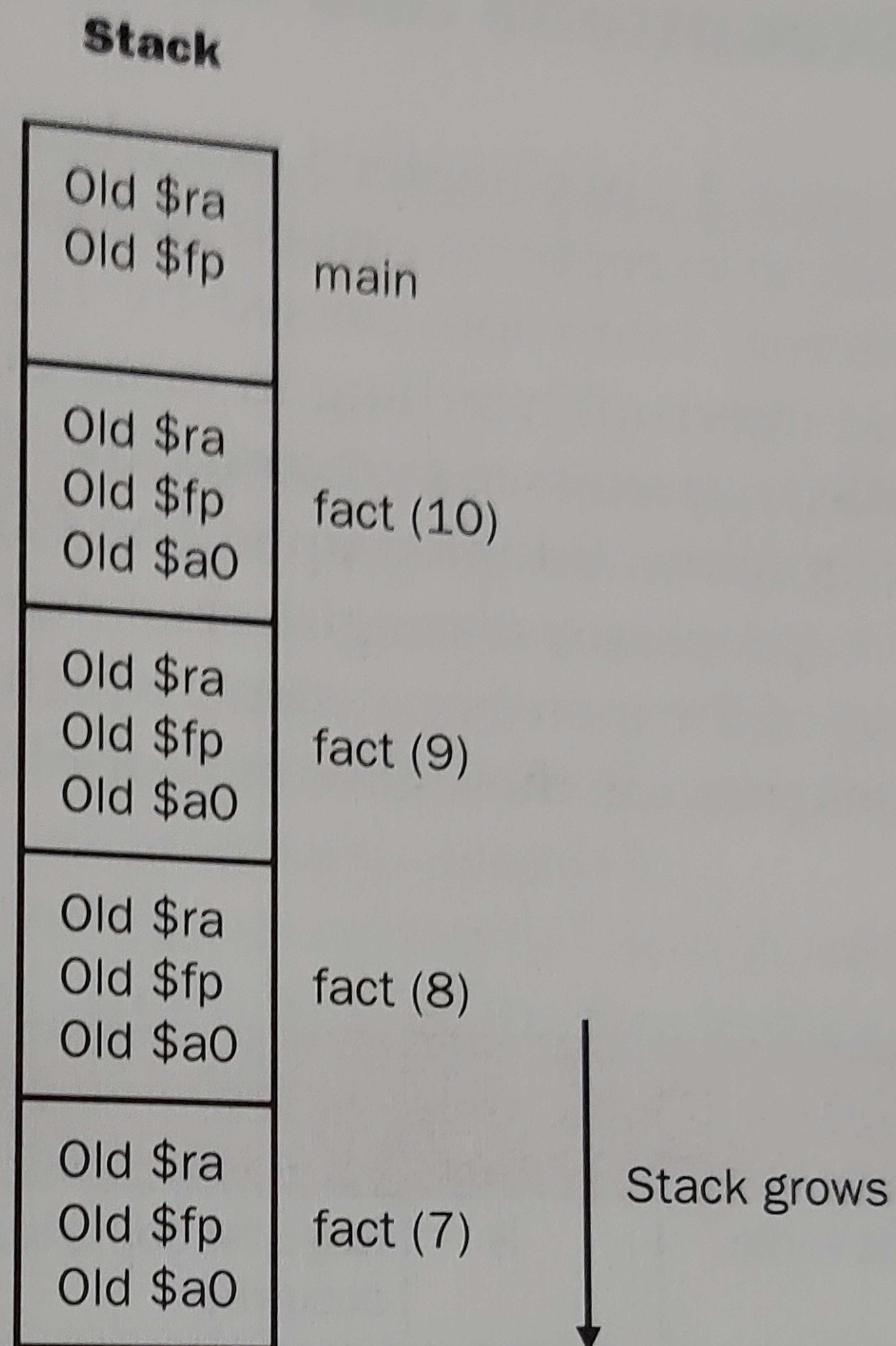
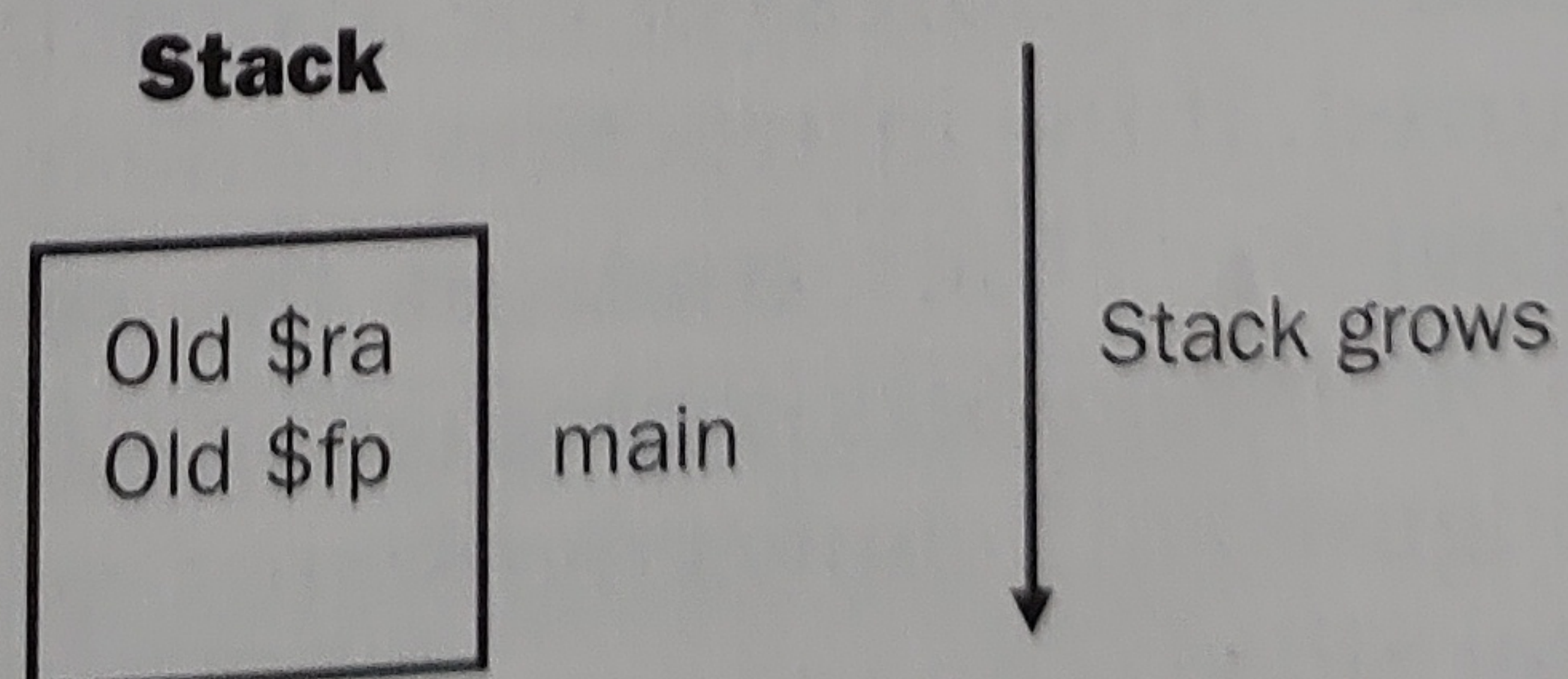


FIGURE A.11 Stack frames during the call of `fact(7)`.

Answer

Elaboration: The difference between the MIPS compiler and the gcc compiler is that the MIPS compiler usually does not use a frame pointer, so this register is available as