Interrupt Number	Interrupt Handler
0	clockintr
1	diskintr
2	ttyintr
3	devintr
4	softintr
5	otherintr

Figure 6.9. Sample Interrupt Vector

terminal interrupt handler *ttyintr*.

3. The kernel invokes the interrupt handler. The kernel stack for the new context layer is logically distinct from the kernel stack of the previous context layer. Some implementations use the kernel stack of the executing process to store the interrupt handler stack frames, and other implementations use a global interrupt stack to store the frames for interrupt handlers that are guaranteed to return without switching context.
4. The interrupt handler completes it work and returns. The kernel executes a machine-specific sequence of instructions that restores the register context and kernel stack of the previous context layer as they existed at the time of the interrupt and then resumes execution of the restored context layer. The behavior of the process may be affected by the interrupt handler, since the interrupt handler may have altered global kernel data structures and awakened sleeping processes. Usually, however, the process continues execution as if the interrupt had never happened.

```
algorithm inthand        /* handle interrupts */
input:  none
output: none
{
        save (push) current context layer;
        determine interrupt source;
        find interrupt vector;
        call interrupt handler;
        restore (pop) previous context layer;
}
```

Figure 6.10. Algorithm for Handling Interrupts

Figure 6.10 summarizes how the kernel handles interrupts. Some machines do part of the sequence of operations in hardware or microcode to get better performance than if all operations were done by software, but there are tradeoffs,

based on how much of the context layer must be saved and the speed of the hardware instructions doing the save. The specific operations required in a UNIX system implementation are therefore machine dependent.

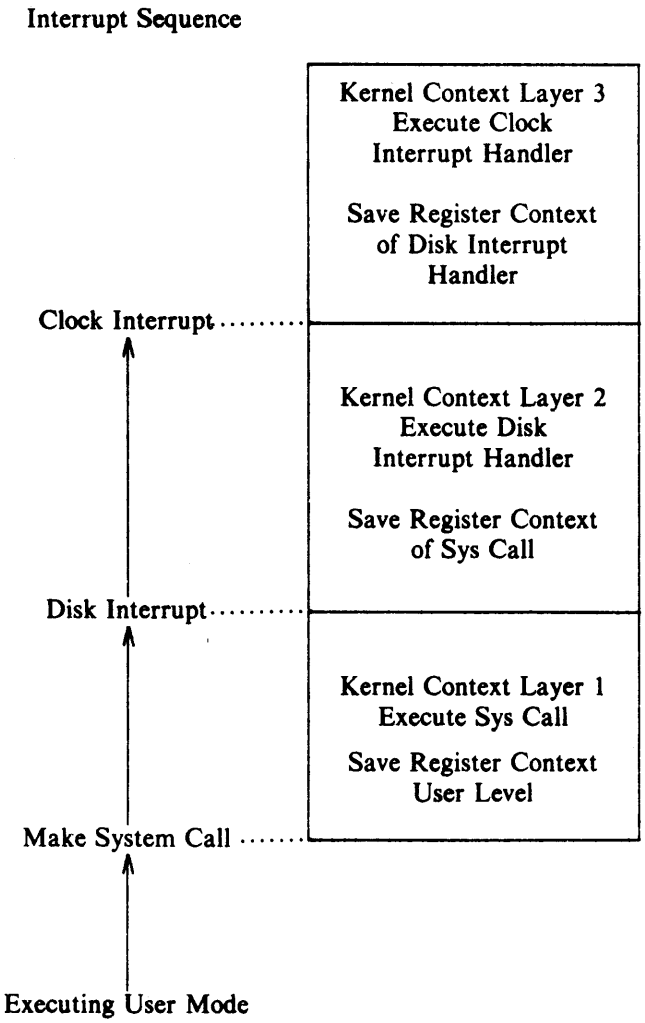

Figure 6.11. Example of Interrupts

Figure 6.11 shows an example where a process issues a system call (see the next section) and receives a disk interrupt while executing the system call. While executing the disk interrupt handler, the system receives a clock interrupt and

executes the clock interrupt handler. Every time the system receives an interrupt (or makes a system call), it creates a new context layer and saves the register context of the previous layer.

6.4.2 System Call Interface

The system call interface to the kernel has been described in previous chapters as though it were a normal function call. Obviously, the usual calling sequence cannot change the mode of a process from user to kernel. The C compiler uses a predefined library of functions (the C library) that have the names of the system calls, thus resolving the system call references in the user program to what would otherwise be undefined names. The library functions typically invoke an instruction that changes the process execution mode to kernel mode and causes the kernel to start executing code for system calls. The ensuing discussion refers to the instruction as an *operating system trap*. The library routines execute in user mode, but the system call interface is, in short, a special case of an interrupt handler. The library functions pass the kernel a unique number per system call in a machine-dependent way — either as a parameter to the operating system trap, in a particular register, or on the stack — and the kernel thus determines the specific system call the user is invoking.

```
algorithm syscall                /* algorithm for invocation of system call */
input:  system call number
output: result of system call
{
        find entry in system call table corresponding to system call number;
        determine number of parameters to system call;
        copy parameters from user address space to u area;
        save current context for abortive return (described in section 6.4.4);
        invoke system call code in kernel;
        if (error during execution of system call)
        {
                set register 0 in user saved register context to error number;
                turn on carry bit in PS register in user saved register context;
        }
        else
                set registers 0, 1 in user saved register context
                                to return values from system call;
}
```

Figure 6.12. Algorithm for System Calls

In handling the operating system trap, the kernel looks up the system call number in a table to find the address of the appropriate kernel routine that is the entry point for the system call and to find the number of parameters the system call expects (Figure 6.12). The kernel calculates the (user) address of the first parameter to the system call by adding (or subtracting, depending on the direction of stack growth) an offset to the user stack pointer, corresponding to the number of parameters to the system call. Finally, it copies the user parameters to the *u area* and calls the appropriate system call routine. After executing the code for the system call, the kernel determines whether there was error. If so, it adjusts register locations in the saved user register context, typically setting the "carry" bit for the PS register and copying the error number into the register 0 location. If there were no errors in the execution of the system call, the kernel clears the "carry" bit in the PS register and copies the appropriate return values from the system call into the locations for registers 0 and 1 in the saved user register context. When the kernel returns from the operating system trap to user mode, it returns to the library instruction after the trap. The library interprets the return values from the kernel and returns a value to the user program.

For example, consider the program that creates a file with read and write permission for all users (mode 0666) in the first part of Figure 6.13. The second part of the figure shows an edited portion of the generated output for the program, as compiled and disassembled on a Motorola 68000 system. Figure 6.14 depicts the stack configurations during the system call. The compiler generates code to push the two parameters onto the user stack, where the first parameter pushed is the permission mode setting, 0666, and the second parameter pushed is the variable *name*.[2] The process then calls the library function for the *creat* system call (address 7a) from address 64. The return address from the function call is 6a, and the process pushes this number onto the stack. The library function for *creat* moves the constant 8 into register 0 and executes a *trap* instruction that causes the process to change from user mode to kernel mode and handle the system call. The kernel recognizes that the user is making a system call and recovers the number 8 from register 0 to determine that the system call is *creat*. Looking up an internal table, the kernel finds that the *creat* system call takes two parameters; recovering the stack register of the previous context layer, it copies the parameters from user space into the *u area*. Kernel routines that need the parameters can find them in predictable locations in the *u area*. When the kernel completes executing the code for *creat*, it returns to the system call handler, which checks if the *u area* error field is set (meaning there was some error in the system call); if so, the handler sets the carry bit in the PS register, places the error code into register 0, and returns. If there is no error, the kernel places the system return code into registers 0 and 1.

2. The order that the compiler evaluates and pushes function parameters is implementation dependent.

```
char name[] = "file";
main()
{
        int fd;
        fd = creat(name, 0666);
}
```

Portions of Generated Motorola 68000 Assembler Code

```
Addr    Instruction
        .
        .
# code for main

58:     mov     &0x1b6,(%sp)        # move 0666 onto stack
5e:     mov     &0x204,-(%sp)       # move stack ptr
                                    # and move variable "name" onto stack
64:     jsr     0x7a                # call C library for creat
        .
        .
# library code for creat
7a:     movq    &0x8,%d0            # move data value 8 into data register 0
7c:     trap    &0x0                # operating system trap
7e:     bcc     &0x6 <86>           # branch to addr 86 if carry bit clear
80:     jmp     0x13c               # jump to addr 13c
86:     rts                         # return from subroutine
        .
        .
# library code for errors in system call
13c:    mov     %d0,&0x20e          # move data reg 0 to location 20e (errno)
142:    movq    &-0x1,%d0           # move constant -1 into data register 0
144:    mova    %d0,%a0
146:    rts                         # return from subroutine
```

Figure 6.13. Creat System Call and Generated Code for Motorola 68000

When returning from the system call handler to user mode, the C library checks the carry bit in the PS register at address 7e: If it is set, the process jumps to address 13c, takes the error code from register 0 and places it into the global variable *errno* at address 20e, places a -1 in register 0, and returns to the next instruction after the call at address 64. The return code for the function is -1, signifying an error in the system call. If, when returning from kernel mode to user mode, the carry bit in the PS register is clear, the process jumps from address 7e to address 86 and returns to the caller (address 64): Register 0 contains the return value from the system call.

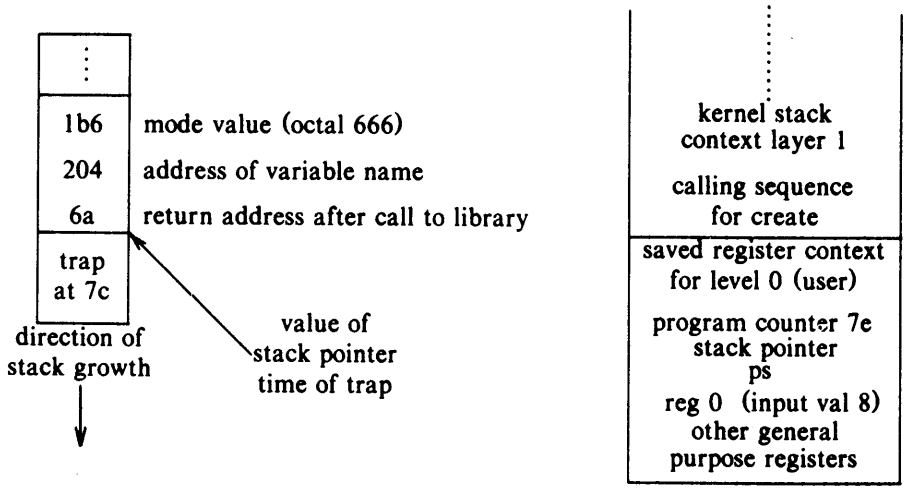

Figure 6.14. Stack Configuration for Creat System Call

Several library functions can map into one system call entry point. The system call entry point defines the true syntax and semantics for every system call, but the libraries frequently provide a more convenient interface. For example, there are several flavors of the *exec* system call, such as *execl* and *execle*, which provide slightly different interfaces for one system call. The libraries for these calls manipulate their parameters to implement the advertised features, but eventually, map into one kernel entry point.

6.4.3 Context Switch

Referring to the process state diagram in Figure 6.1, we see that the kernel permits a context switch under four circumstances: when a process puts itself to sleep, when it *exit*s, when it returns from a system call to user mode but is not the most eligible process to run, or when it returns to user mode after the kernel completes handling an interrupt but is not the most eligible process to run. The kernel ensures integrity and consistency of internal data structures by prohibiting arbitrary context switches, as explained in Chapter 2. It makes sure that the state of its data structures is consistent before it does a context switch: that is, that all appropriate updates are done, that queues are properly linked, that appropriate locks are set to prevent intrusion by other processes, that no data structures are left unnecessarily locked, and so on. For example, if the kernel allocates a buffer, *read*s a block in a file, and goes to sleep waiting for I/O transmission from the disk to complete, it keeps the buffer locked so that no other process can tamper with the buffer. But if

a process executes the *link* system call, the kernel releases the lock of the first inode before locking the second inode to avoid deadlocks.

The kernel must do a context switch at the conclusion of the *exit* system call, because there is nothing else for it to do. Similarly, the kernel allows a context switch when a process enters the sleep state, since a considerable amount of time may elapse until the process wakes up, and other processes can meanwhile execute. The kernel allows a context switch when a process is not the most eligible to run to permit fairer process scheduling: If a process completes a system call or returns from an interrupt and there is another process with higher priority waiting to run, it would be unfair to keep the high-priority process waiting.

The procedure for a context switch is similar to the procedures for handling interrupts and system calls, except that the kernel restores the context layer of a different process instead of the previous context layer of the same process. The reasons for the context switch are irrelevant. Similarly, the choice of which process to schedule next is a policy decision that does not affect the mechanics of the context switch.

1. Decide whether to do a context switch, and whether a context switch is permissible now.
2. Save the context of the "old" process.
3. Find the "best" process to schedule for execution, using the process scheduling algorithm in Chapter 8.
4. Restore its context.

Figure 6.15. Steps for a Context Switch

The code that implements the context switch on UNIX systems is usually the most difficult to understand in the operating system, because function calls give the appearance of not returning on some occasions and materializing from nowhere on others. This is because the kernel, in many implementations, saves the process context at one point in the code but proceeds to execute the context switch and scheduling algorithms in the context of the "old" process. When it later restores the context of the process, it resumes execution according to the previously saved context. To differentiate between the case where the kernel resumes the context of a new process and the case where it continues to execute in the old context after having saved it, the return values of critical functions may vary, or the program counter where the kernel executes may be set artificially.

Figure 6.16 shows a scenario for doing a context switch. The function *save_context* saves information about the context of the running process and returns the value 1. Among other pieces of information, the kernel saves the value of the current program counter (in the function *save_context*) and the value 0, to be used later as the return value in register 0 from *save_context*. The kernel continues to execute in the context of the old process (A), picking another process (B) to run

```
if (save_context())        /* save context of executing process */
{
        /* pick another process to run */
        .
        .
        .
        resume_context(new_process);
        /* never gets here ! */
}
/* resuming process executes from here */
```

Figure 6.16. Pseudo-Code for Context Switch

and calling *resume_context* to restore the new context (of B). After the new context is restored, the system is executing process B; the old process (A) is no longer executing but leaves its saved context behind (hence, the comment in the figure "never gets here"). Later, the kernel will again pick process A to run (except for the *exit* case, of course) when another process does a context switch, as just described. When process A's context is restored, the kernel will set the program counter to the value process A had previously saved in the function *save_context*, and it will also place the value 0, saved for the return value, into register 0. The kernel resumes execution of process A inside *save_context* even though it had executed the code up to the call to *resume_context* before the context switch. Finally, process A returns from the function *save_context* with the value 0 (in register 0) and resumes execution after the comment line "resuming process executes from here."

6.4.4 Saving Context for Abortive Returns

Situations arise when the kernel must abort its current execution sequence and immediately execute out of a previously saved context. Later sections dealing with *sleep* and *signals* describe the circumstances when a process must suddenly change its context; this section explains the mechanisms for executing a previous context. The algorithm to save a context is *setjmp* and the algorithm to restore the context is *longjmp*.[3] The method is identical to that described for the function *save_context* in the previous section, except that *save_context* pushes a new context layer, whereas *setjmp* stores the saved context in the *u area* and continues to execute in

3. These algorithms should not be confused with the library functions of the same name that users can call directly from their programs (see [SVID 85]). However, their functions are similar.

the old context layer. When the kernel wishes to resume the context it had saved in *setjmp*, it does a *longjmp*, restoring its context from the *u area* and returning a 1 from *setjmp*.

6.4.5 Copying Data between System and User Address Space

As presented so far, a process executes in kernel mode or in user mode with no overlap of modes. However, many system calls examined in the last chapter move data between kernel and user space, such as when copying system call parameters from user to kernel space or when copying data from I/O buffers in the *read* system call . Many machines allow the kernel to reference addresses in user space directly. The kernel must ascertain that the address being read or written is accessible as if it had been executing in user mode; otherwise, it could override the ordinary protection mechanisms and inadvertently read or write addresses outside the user address space (possibly kernel data structures). Therefore, copying data between kernel space and user space is an expensive proposition, requiring more than one instruction.

```
fubyte:                                  # move byte from user space
        prober  $3,$1,*4(ap)             # byte accessible?
        beql    eret                     # no
        movzbl  *4(ap),r0
        ret
eret:
        mnegl   $1,r0                    # error return (-1)
        ret
```

Figure 6.17. Moving Data from User to System Space on a VAX

Figure 6.17 shows sample VAX code for moving one character from user address space to kernel address space. The *prober* instruction checks if one byte at address *argument pointer register*+4 (*4(ap)) could be read in user mode (mode 3) and, if not, the kernel branches to address *eret*, stores −1 in register 0, and returns; the character move failed. Otherwise, the kernel moves one byte from the given user address to register 0 and returns that value to the caller. The procedure is expensive, requiring five instructions (with the function call to *fubyte*) to move 1 character.

6.5 MANIPULATION OF THE PROCESS ADDRESS SPACE

So far, this chapter has described how the kernel switches context between processes and how it pushes and pops context layers, viewing the user-level context as a static object that does not change during restoration of the process context.

However, various system calls manipulate the virtual address space of a process, as will be seen in the next chapter, doing so according to well defined operations on regions. This section describes the region data structure and the operations on regions; the next chapter deals with the system calls that use the region operations.

The region table entry contains the information necessary to describe a region. In particular, it contains the following entries:

- A pointer to the inode of the file whose contents were originally loaded into the region
- The region type (text, shared memory, private data or stack)
- The size of the region
- The location of the region in physical memory
- The status of a region, which may be a combination of
  - locked
  - in demand
  - in the process of being loaded into memory
  - valid, loaded into memory
- The reference count, giving the number of processes that reference the region.

The operations that manipulate regions are to lock a region, unlock a region, allocate a region, attach a region to the memory space of a process, change the size of a region, load a region from a file into the memory space of a process, free a region, detach a region from the memory space of a process, and duplicate the contents of a region. For example, the *exec* system call, which overlays the user address space with the contents of an executable file, detaches old regions, frees them if they were not shared, allocates new regions, attaches them, and loads them with the contents of the file. The remainder of this section describes the region operations in detail, assuming the memory management model described earlier (page tables and hardware register triples) and the existence of algorithms for allocation of page tables and pages of physical memory (Chapter 9).

6.5.1 Locking and Unlocking a Region

The kernel has operations to lock and unlock a region, independent of the operations to allocate and free a region, just as the file system has lock-unlock and allocate-release operations for inodes (algorithms *iget* and *iput*). Thus the kernel can lock and allocate a region and later unlock it without having to free the region. Similarly, if it wants to manipulate an allocated region, it can lock the region to prevent access by other processes and later unlock it.

6.5.2 Allocating a Region

The kernel allocates a new region (algorithm *allocreg*, Figure 6.18) during *fork*, *exec*, and *shmget* (shared memory) system calls. The kernel contains a region

table whose entries appear either on a free linked list or on an active linked list. When it allocates a region table entry, the kernel removes the first available entry from the free list, places it on the active list, locks the region, and marks its type (shared or private). With few exceptions, every process is associated with an executable file as a result of a prior *exec* call, and *allocreg* sets the inode field in the region table entry to point to the inode of the executable file. The inode identifies the region to the kernel so that other processes can share the region if desired. The kernel increments the inode reference count to prevent other processes from removing its contents when *unlink*ing it, as will be explained in Section 7.5. *Allocreg* returns a locked, allocated region.

```
algorithm allocreg        /* allocate a region data structure */
input:  (1) inode pointer
        (2) region type
output: locked region
{
        remove region from linked list of free regions;
        assign region type;
        assign region inode pointer;
        if (inode pointer not null)
                increment inode reference count;
        place region on linked list of active regions;
        return(locked region);
}
```

Figure 6.18. Algorithm for Allocating a Region

6.5.3 Attaching a Region to a Process

The kernel attaches a region during the *fork*, *exec*, and *shmat* system calls to connect it to the address space of a process (algorithm *attachreg*, Figure 6.19). The region may be a newly allocated region or an existing region that the process will share with other processes. The kernel allocates a free pregion entry, sets its type field to text, data, shared memory, or stack, and records the virtual address where the region will exist in the process address space. The process must not exceed the system-imposed limit for the highest virtual address, and the virtual addresses of the new region must not overlap the addresses of existing regions. For example, if the system restricts the highest virtual address of a process to 8 megabytes, it would be illegal to attach a 1 megabyte-size region to virtual address 7.5M. If it is legal to attach the region, the kernel increments the size field in the process table entry according to the region size, and increments the region reference count.

```
algorithm attachreg        /* attach a region to a process */
input:  (1) pointer to (locked) region being attached
        (2) process to which region is being attached
        (3) virtual address in process where region will be attached
        (4) region type
output: per process region table entry
{
        allocate per process region table entry for process;
        initialize per process region table entry:
                set pointer to region being attached;
                set type field;
                set virtual address field;
        check legality of virtual address, region size;
        increment region reference count;
        increment process size according to attached region;
        initialize new hardware register triple for process;
        return(per process region table entry);
}
```

Figure 6.19. Algorithm for Attachreg

Attachreg then initializes a new set of memory management register triples for the process: If the region is not already attached to another process, the kernel allocates page tables for it in a subsequent call to *growreg* (next section); otherwise, it uses the existing page tables. Finally, *attachreg* returns a pointer to the pregion entry for the newly attached region. For example, suppose the kernel wants to attach an existing (shared) text region of size 7K bytes to virtual address 0 of a process (Figure 6.20): it allocates a new memory management register triple and initializes the triple with the address of the region page table, the process virtual address (0), and the size of the page table (9 entries).

6.5.4 Changing the Size of a Region

A process may expand or contract its virtual address space with the *sbrk* system call. Similarly, the stack of a process automatically expands (that is, the process does not make an explicit system call) according to the depth of nested procedure calls. Internally, the kernel invokes the algorithm *growreg* to change the size of a region (Figure 6.21). When a region expands, the kernel makes sure that the virtual addresses of the expanded region do not overlap those of another region and that the growth of the region does not cause the process size to become greater than the maximum allowed virtual memory space. The kernel never invokes *growreg* to increase the size of a shared region that is already attached to several processes; therefore, it does not have to worry about increasing the size of a region

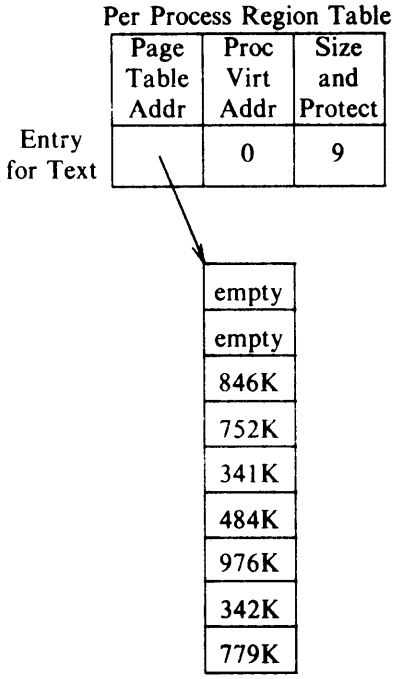

Figure 6.20. Example of Attaching to an Existing Text Region

for one process and causing another process to grow beyond the system limit for process size. The two cases where the kernel uses *growreg* on an existing region are *sbrk* on the data region of a process and automatic growth of the user stack. Both regions are private. Text regions and shared memory regions cannot grow after they are initialized. These cases will become clear in the next chapter.

The kernel now allocates page tables (or extends existing page tables) to accommodate the larger region and allocates physical memory on systems that do not support demand paging. When allocating physical memory, it makes sure such memory is available before invoking *growreg*; if the memory is unavailable, it resorts to other measures to increase the region size, as will be covered in Chapter 9. If the process contracts the region, the kernel simply releases memory assigned to the region. In both cases, it adjusts the process size and region size and reinitializes the pregion entry and memory management register triples to conform to the new mapping.

For example, suppose the stack region of a process starts at virtual address 128K and currently contains 6K bytes, and the kernel wants to extend the size of the region by 1K bytes (1 page). If the process size is acceptable and virtual

```
algorithm growreg        /* change the size of a region */
input:  (1) pointer to per process region table entry
        (2) change in size of region (may be positive or negative)
output: none
{
        if (region size increasing)
        {
                check legality of new region size;
                allocate auxiliary tables (page tables);
                if (not system supporting demand paging)
                {
                        allocate physical memory;
                        initialize auxiliary tables, as necessary;
                }
        }
        else        /* region size decreasing */
        {
                free physical memory, as appropriate;
                free auxiliary tables, as appropriate;
        }

        do (other) initialization of auxiliary tables, as necessary;
        set size field in process table;
}
```

Figure 6.21. Algorithm Growreg for Changing the Size of a Region

addresses 134K to 135K − 1 do not belong to another region attached to the process, the kernel extends the size of the region. It extends the page table, allocates a page of memory, and initializes the new page table entry. Figure 6.22 illustrates this case.

6.5.5 Loading a Region

In a system that supports demand paging, the kernel can "map" a file into the process address space during the *exec* system call, arranging to read individual physical pages later on demand, as will be explained in Chapter 9. If the kernel does not support demand paging, it must copy the executable file into memory, loading the process regions at virtual addresses specified in the executable file. It may attach a region at a different virtual address from where it loads the contents of the file, creating a gap in the page table (recall Figure 6.20). For example, this feature is used to cause memory faults when user programs access address 0 illegally. Programs with pointer variables sometimes use them erroneously without checking that their value is 0 and, hence, that they are illegal for use as a pointer

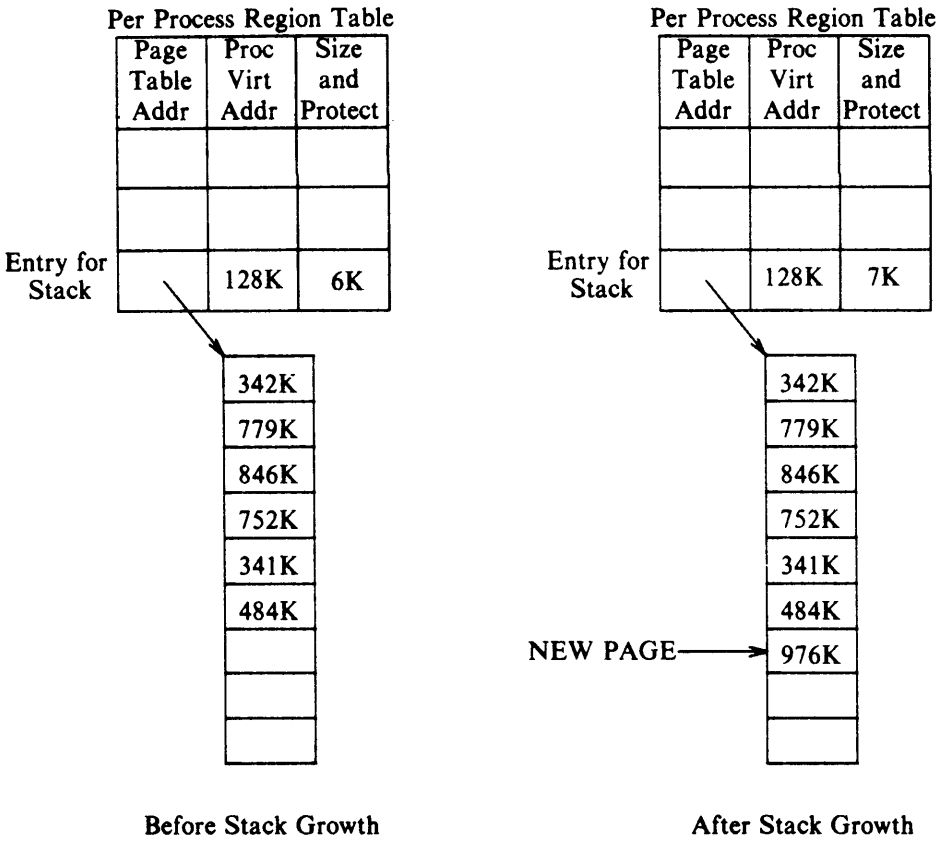

Figure 6.22. Growing the Stack Region by 1K Bytes

reference. By protecting the page containing address 0 appropriately, processes that errantly access address 0 incur a fault and abort, allowing programmers to discover such bugs more quickly.

To load a file into a region, *loadreg* (Figure 6.23) accounts for the gap between the virtual address where the region is attached to the process and the starting virtual address of the region data and expands the region according to the amount of memory the region requires. Then it places the region in the state "being loaded into memory" and reads the region data into memory from the file, using an internal variation of the *read* system call algorithm.

If the kernel is loading a text region that can be shared by several processes, it is possible that another process could find the region and attempt to use it before its contents were fully loaded, because the first process could sleep while reading the

```
algorithm loadreg        /* load a portion of a file into a region */
input:  (1) pointer to per process region table entry
        (2) virtual address to load region
        (3) inode pointer of file for loading region
        (4) byte offset in file for start of region
        (5) byte count for amount of data to load
output: none
{
        increase region size according to eventual size of region
                (algorithm growreg);
        mark region state: being loaded into memory;
        unlock region;
        set up u area parameters for reading file:
                target virtual address where data is read to,
                start offset value for reading file,
                count of bytes to read from file;
        read file into region (internal variant of read algorithm);
        lock region;
        mark region state: completely loaded into memory;
        awaken all processes waiting for region to be loaded;
}
```

Figure 6.23. Algorithm for Loadreg

file. The details of how this could happen and why locks cannot be used are left for the discussion of *exec* in the next chapter and in Chapter 9. To avoid a problem, the kernel checks a region state flag to see if the region is completely loaded and, if the region is not loaded, the process sleeps. At the end of *loadreg*, the kernel awakens processes that were waiting for the region to be loaded and changes the region state to valid and in memory.

For example, suppose the kernel wants to load text of size 7K into a region that is attached at virtual address 0 of a process but wants to leave a gap of 1K bytes at the beginning of the region (Figure 6.24). By this time, the kernel will have allocated a region table entry and will have attached the region at address 0 using algorithms *allocreg* and *attachreg*. Now it invokes *loadreg*, which invokes *growreg* twice — first, to account for the 1K byte gap at the beginning of the region, and second, to allocate storage for the contents of the region — and *growreg* allocates a page table for the region. The kernel then sets up fields in the *u area* to read the file: It reads 7K bytes from a specified byte offset in the file (supplied as a parameter by the kernel) into virtual address 1K of the process.

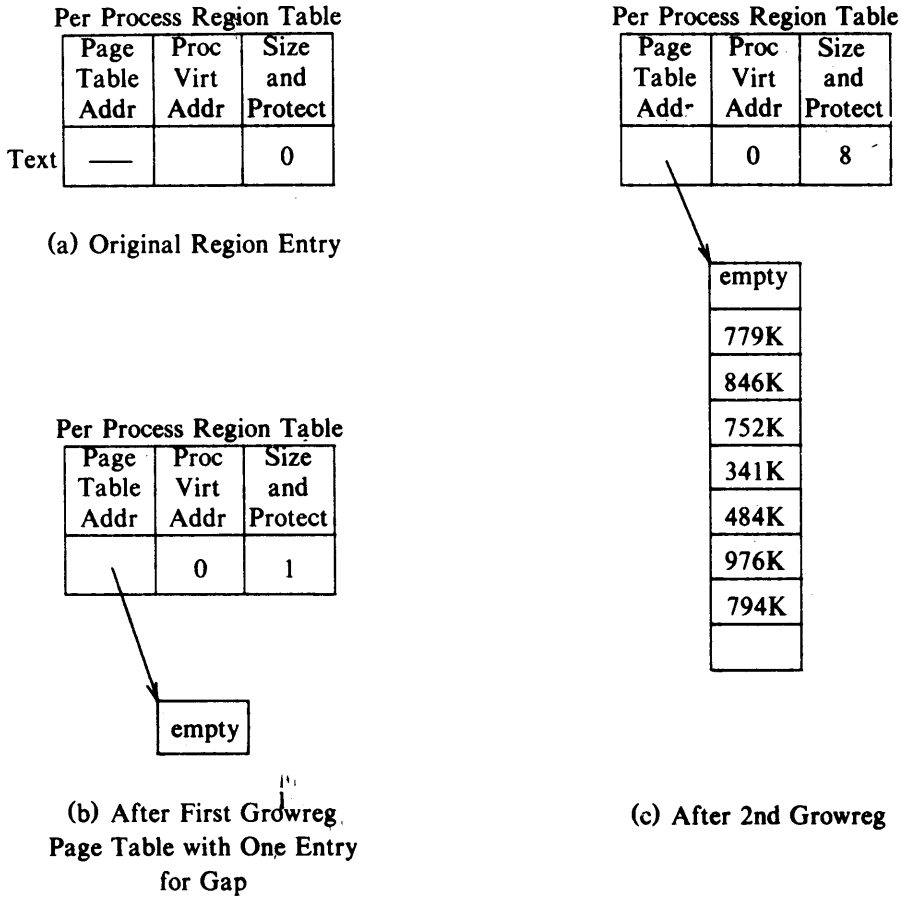

Figure 6.24. Loading a Text Region

6.5.6 Freeing a Region

When a region is no longer attached to any processes, the kernel can free the region and return it to the list of free regions (Figure 6.25). If the region is associated with an inode, the kernel releases the inode using algorithm *iput*, corresponding to the increment of the inode reference count in *allocreg*. The kernel releases physical resources associated with the region, such as page tables and memory pages. For example, suppose the kernel wants to free the stack region in Figure 6.22. Assuming the region reference count is 0, it releases the 7 pages of physical memory and the page table.

```
algorithm freereg       /* free an allocated region */
input:  pointer to a (locked) region
output: none
{
        if (region reference count non zero)
        {
                /* some process still using region */
                release region lock;
                if (region has an associated inode)
                        release inode lock;
                return;
        }
        if (region has associated inode)
                release inode (algorithm iput);
        free physical memory still associated with region;
        free auxiliary tables associated with region;
        clear region fields;
        place region on region free list;
        unlock region;
}
```

Figure 6.25. Algorithm for Freeing a Region

```
algorithm detachreg       /* detach a region from a process */
input:  pointer to per process region table entry
output: none
{
        get auxiliary memory management tables for process,
                        release as appropriate;
        decrement process size;
        decrement region reference count;
        if (region reference count is 0 and region not sticky bit)
                free region (algorithm freereg);
        else        /* either reference count non-0 or region sticky bit on */
        {
                free inode lock, if applicable (inode associated with region);
                free region lock;
        }
}
```

Figure 6.26. Algorithm Detachreg

6.5.7 Detaching a Region from a Process

The kernel detaches regions in the *exec*, *exit*, and *shmdt* (detach shared memory) system calls. It updates the pregion entry and severs the connection to physical memory by invalidating the associated memory management register triple (algorithm *detachreg*, Figure 6.26). The address translation mechanisms thus invalidated apply specifically to the *process*, not to the region (as in algorithm *freereg*). The kernel decrements the region reference count and the size field in the process table entry according to the size of the region. If the region reference count drops to 0 and if there is no reason to leave the region intact (the region is not a shared memory region or a text region with the *sticky bit* on, as will be described in Section 7.5), the kernel frees the region using algorithm *freereg*. Otherwise, it releases the region and inode locks, which had been locked to prevent race conditions as will be described in Section 7.5 but leaves the region and its resources allocated.

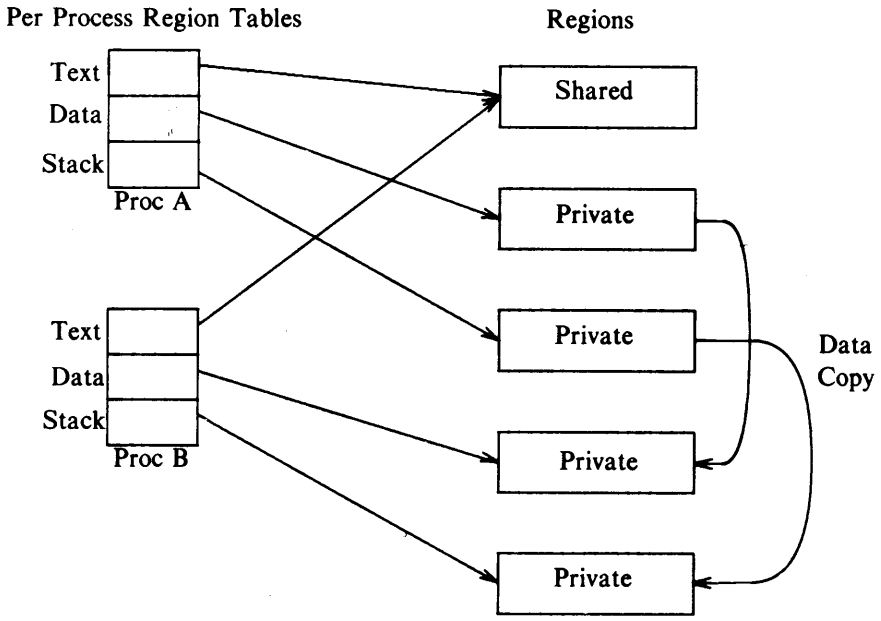

Figure 6.27. Duplicating a Region

```
algorithm dupreg        /* duplicate an existing region */
input:  pointer to region table entry
output: pointer to a region that looks identical to input region
{
        if (region type shared)
                /* caller will increment region reference count
                 * with subsequent attachreg call
                 */
                return(input region pointer);
        allocate new region (algorithm allocreg);
        set up auxiliary memory management structures, as currently
                exists in input region;
        allocate physical memory for region contents;
        "copy" region contents from input region to newly allocated
                region;
        return(pointer to allocated region);
}
```

Figure 6.28. Algorithm for Dupreg

6.5.8 Duplicating a Region

The *fork* system call requires that the kernel duplicate the regions of a process. If a region is shared (shared text or shared memory), however, the kernel need not physically copy the region; instead, it increments the region reference count, allowing the parent and child processes to share the region. If the region is not shared and the kernel must physically copy the region, it allocates a new region table entry, page table, and physical memory for the region. In Figure 6.27 for example, process A *fork*ed process B and duplicated its regions. The text region of process A is shared, so process B can share it with process A. But the data and stack regions of process A are private, so process B duplicates them by copying their contents to newly allocated regions. Even for private regions, a physical copy of the region is not always necessary, as will be seen (Chapter 9). Figure 6.28 shows the algorithm for *dupreg*.

6.6 SLEEP

So far, this chapter has covered all the low-level functions that are executed for the transitions to and from the state "kernel running" except for the functions that move a process into the sleep state. It will conclude with a presentation of the algorithms for *sleep*, which changes the process state from "kernel running" to "asleep in memory," and *wakeup*, which changes the process state from "asleep" to "ready to run" in memory or swapped.

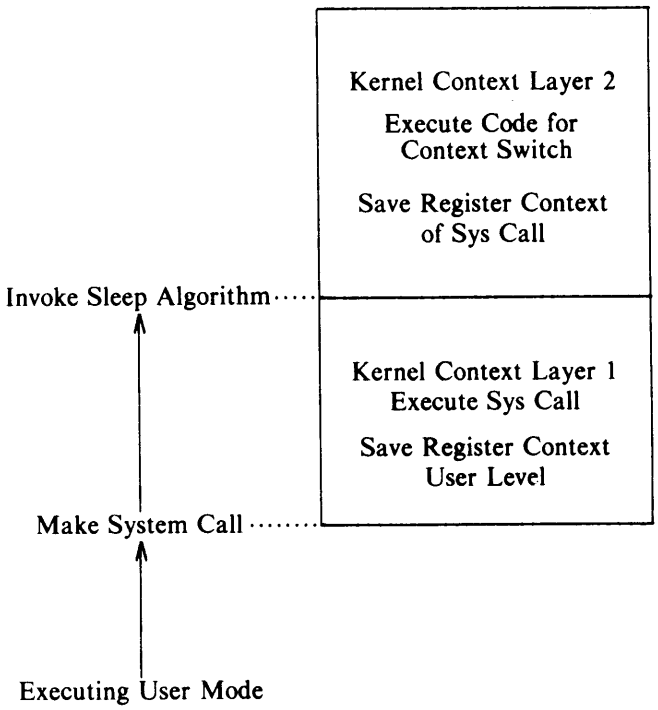

Figure 6.29. Typical Context Layers of a Sleeping Process

When a process goes to sleep, it typically does so during execution of a system call: The process enters the kernel (context layer 1) when it executes an operating system trap and goes to sleep awaiting a resource. When the process goes to sleep, it does a context switch, pushing its current context layer and executing in kernel context layer 2 (Figure 6.29). Processes also go to sleep when they incur page faults as a result of accessing virtual addresses that are not physically loaded; they sleep while the kernel reads in the contents of the pages.

6.6.1 Sleep Events and Addresses

Recall from Chapter 2 that processes are said to sleep on an event, meaning that they are in the sleep state until the event occurs, at which time they wake up and enter a "ready-to-run" state (in memory or swapped out). Although the system uses the abstraction of sleeping on an event, the implementation maps the set of events into a set of (kernel) virtual addresses. The addresses that represent the events are coded into the kernel, and their only significance is that the kernel

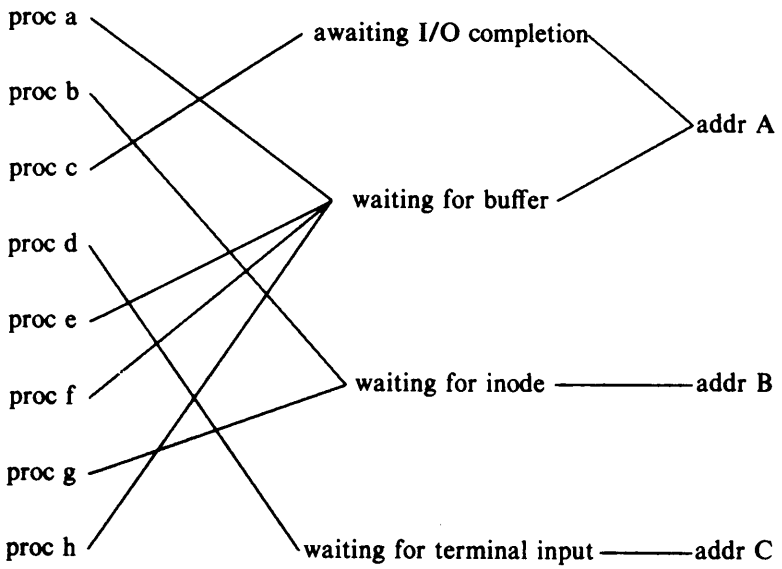

Figure 6.30. Processes Sleeping on Events and Events Mapping into Addresses

expects an event to map into a particular address. The abstraction of the event does not distinguish how many processes are awaiting the event, nor does the implementation. As a result, two anomalies arise. First, when an event occurs and a wakeup call is issued for processes that are sleeping on the event, they *all* wake up and move from a sleep state to a ready-to-run state. The kernel does not wake up one process at a time, even though they may contend for a single locked structure, and many may go back to sleep after a brief visit to the kernel running state (recall the discussion in Chapters 2 and 3). Figure 6.30 shows several processes sleeping on events.

The second anomaly in the implementation is that several events may map into one address. In Figure 6.30, for example, the events "waiting for the buffer" to become free and "awaiting I/O completion" map into the address of the buffer ("addr A"). When I/O for the buffer completes, the kernel wakes up all processes sleeping on both events. Since a process waiting for I/O keeps the buffer locked, other processes waiting for the buffer to become free will go back to sleep if the buffer is still locked when they execute. It would be more efficient if there would be a one-to-one mapping of events to addresses. In practice, however, performance is not hurt, because the mapping of multiple events into one address is rare and because the running process usually frees the locked resource before the other processes are scheduled to run. Stylistically, however, it would make the kernel a little easier to understand if the mapping were one-to-one.

```
algorithm sleep
input:  (1) sleep address
        (2) priority
output: 1 if process awakened as a result of a signal that process catches,
        longjump algorithm if process awakened as a result of a signal
                        that it does not catch,
        0 otherwise;
{
      raise processor execution level to block all interrupts;
      set process state to sleep;
      put process on sleep hash queue, based on sleep address;
      save sleep address in process table slot;
      set process priority level to input priority;
      if (process sleep is NOT interruptible)
      {
            do context switch;
            /* process resumes execution here when it wakes up */
            reset processor priority level to allow interrupts as when
                  process went to sleep;
            return(0);
      }

      /* here, process sleep is interruptible by signals */
      if (no signal pending against process)
      {
            do context switch;
            /* process resumes execution here when it wakes up */
            if (no signal pending against process)
            {
                  reset processor priority level to what it was when
                        process went to sleep;
                  return(0);
            }
      }
      remove process from sleep hash queue, if still there;

      reset processor priority level to what it was when process went to sleep;
      if (process sleep priority set to catch signals)
            return(1)
      do longjmp algorithm;
}
```

Figure 6.31. Sleep Algorithm

6.6.2 Algorithms for Sleep and Wakeup

Figure 6.31 shows the algorithm for *sleep*. The kernel first raises the processor execution level to block out all interrupts so that there can be no race conditions when it manipulates the sleep queues, and it saves the old processor execution level so that it can be restored when the process later wakes up. It marks the process state "asleep," saves the sleep address and priority in the process table, and puts it onto a hashed queue of sleeping processes. In the simple case (sleep cannot be interrupted), the process does a context switch and is safely asleep. When a sleeping process wakes up, the kernel later schedules it to run: The process returns from its context switch in the *sleep* algorithm, restores the processor execution level to the value it had when the process entered the algorithm, and returns.

```
algorithm wakeup                /* wake up a sleeping process */
input:  sleep address
output: none
{
        raise processor execution level to block all interrupts;
        find sleep hash queue for sleep address;
        for (every process asleep on sleep address)
        {
                remove process from hash queue;
                mark process state "ready to run";
                put process on scheduler list of processes ready to run;
                clear field in process table entry for sleep address;
                if (process not loaded in memory)
                        wake up swapper process (0);
                else if (awakened process is more elligible to run than
                                        currently running process)
                        set scheduler flag;
        }
        restore processor execution level to original level;
}
```

Figure 6.32. Algorithm for Wakeup

To wake up sleeping processes, the kernel executes the *wakeup* algorithm (Figure 6.32), either during the usual system call algorithms or when handling an interrupt. For instance, the algorithm *iput* releases a locked inode and awakens all processes waiting for the lock to become free. Similarly, the disk interrupt handler awakens a process waiting for I/O completion. The kernel raises the processor execution level in *wakeup* to block out interrupts. Then for every process sleeping on the input sleep address, it marks the process state field "ready to run," removes the process from the linked list of sleeping processes, places it on a linked list of processes eligible for scheduling, and clears the field in the process table that

marked its sleep address. If a process that woke up was not loaded in memory, the kernel awakens the swapper process to swap the process into memory (assuming the system is one that does not support demand paging); otherwise, if the awakened process is more eligible to run than the currently executing process, the kernel sets a scheduler flag so that it will go through the process scheduling algorithm when the process returns to user mode (Chapter 8). Finally, the kernel restores the processor execution level. It cannot be stressed enough: *wakeup* does *not* cause a process to be scheduled immediately; it only makes the process eligible for scheduling.

The discussion above is the simple case of the *sleep* and *wakeup* algorithms, because it assumes that the process sleeps until the proper event occurs. Processes frequently sleep on events that are "sure" to happen, such as when awaiting a locked resource (inodes or buffers) or when awaiting completion of disk I/O. The process is sure to wake up because the use of such resources is designed to be temporary. However, a process may sometimes sleep on an event that is not sure to happen, and if so, it must have a way to regain control and continue execution. For such cases, the kernel "interrupts" the sleeping process immediately by sending it a *signal*. The next chapter explains *signals* in great detail; for now, assume that the kernel can (selectively) wake up a sleeping process as a result of the signal, and that the process can recognize that it has been sent a signal.

For instance, if a process issues a *read* system call to a terminal, the kernel does not satisfy the call until a user types data on the terminal keyboard (Chapter 10). However, the user that started the process may leave the terminal for an all-day meeting, leaving the process asleep and waiting for input, and another user may want to use the terminal. If the second user resorts to drastic measures (such as turning the terminal off), the kernel needs a way to recover the disconnected process: As a first step, it must awaken the process from its sleep as the result of a signal. Parenthetically, there is nothing wrong with processes sleeping for a long time. Sleeping process occupy a slot in the process table and could thus lengthen the search times for certain algorithms, but they do not use CPU time, so their overhead is small.

To distinguish the types of sleep states, the kernel sets the scheduling priority of the sleeping process when it enters the sleep state, based on the sleep priority parameter. That is, it invokes the *sleep* algorithm with a priority value, based on its knowledge that the sleep event is sure to occur or not. If the priority is above a threshold value, the process will not wake up prematurely on receipt of a signal but will sleep until the event it is waiting for happens. But if the priority value is below the threshold value, the process will awaken immediately on receipt of the signal.[4]

4. The terms "above" and "below" refer to the normal usage of the terms high priority and low priority. However, the kernel implementation uses integers to measure the priority value, with lower values implying higher priority.

If a signal is already set against a process when it enters the *sleep* algorithm, the conditions just stated determine whether the process ever gets to sleep. For instance, if the sleep priority is above the threshold value, the process goes to sleep and waits for an explicit wakeup call. If the sleep priority is below the threshold value, however, the process does not go to sleep but responds to the signal as if the signal had arrived while it was asleep. If the kernel did not check for signals before going to sleep, the signal may not arrive again and the process would never wake up.

When a process is awakened as a result of a signal (or if it never gets to sleep because of existence of a signal), the kernel may do a *longjmp*, depending on the reason the process originally went to sleep. The kernel does a *longjmp* to restore a previously saved context if it has no way to complete the system call it is executing. For instance, if a terminal *read* call is interrupted because a user turns the terminal off, the *read* should not complete but should return with an error indication. This holds for all system calls that can be interrupted while they are asleep. The process should not continue normally after waking up from its sleep, because the sleep event was not satisfied. The kernel saves the process context at the beginning of most system calls using *setjmp* in anticipation of the need for a later *longjmp*.

There are occasions when the kernel wants the process to wake up on receipt of a signal but not do a *longjmp*. The kernel invokes the *sleep* algorithm with a special priority parameter that suppresses execution of the *longjmp* and causes the *sleep* algorithm to return the value 1. This is more efficient than doing a *setjmp* immediately before the *sleep* call and then a *longjmp* to restore the context of the process as it was before entering the sleep state. The purpose is to allow the kernel to clean up local data structures. For example, a device driver may allocate private data structures and then go to sleep at an interruptible priority; if it wakes up because of a signal, it should free the allocated data structures, then *longjmp* if necessary. The user has no control over whether a process does a *longjmp*; that depends on the reason the process was sleeping and whether kernel data structures need modification before the process returns from the system call.

6.7 SUMMARY

This chapter has defined the context of a process. Processes in the UNIX system move between various logical states according to well-defined transition rules, and state information is saved in the process table and the *u area*. The context of a process consists of its user-level context and its system-level context. The user-level context consists of the process text, data, (user) stack, and shared memory regions, and the system-level context consists of a static part (process table entry, *u area*, and memory mapping information) and a dynamic part (kernel stack and saved registers of previous system context layer) that is pushed and popped as the process executes system calls, handles interrupts, and does context switches. The user-level context of a process is divided into separate regions, comprising contiguous ranges of virtual addresses that are treated as distinct objects for protection and sharing.

The memory management model used to describe the virtual address layout of a process assumes the use of a page table for each process region. The kernel contains various algorithms that manipulate regions. Finally, the chapter described the algorithms for *sleep* and *wakeup*. The following chapters use the low-level structures and algorithms described here, in the explanation of the system calls for process management, process scheduling, and the implementation of memory management policies.

6.8 EXERCISES

1. Design an algorithm that translates virtual addresses to physical addresses, given the virtual address and the address of the pregion entry.
2. The AT&T 3B2 computer and the NSC Series 32000 use a two-tiered (segmented) translation scheme to translate virtual addresses to physical addresses. That is, the system contains a pointer to a table of page table pointers, and each entry in the table can address a fixed portion of the process address space, according to its offset in the table. Compare the algorithm for virtual address translation on these machines to the algorithm discussed for the memory model in the text. Consider issues of performance and the space needed for auxiliary tables.
3. The VAX-11 architecture contains two sets of base and limit registers that the machine uses for user address translation. The scheme is the same as that described in the previous problem, except that the number of page table pointers is two. Given that processes have three regions, text, data, and stack, what is a good way of mapping the regions into page tables and using the two sets of registers? The stack in the VAX-11 architecture grows towards lower virtual addresses. What should the stack region look like? Chapter 11 will describe another region for shared memory: How should it fit into the VAX-11 architecture?
4. Design an algorithm for allocating and freeing memory pages and page tables. What data structures would allow best performance or simplest implementation?
5. The MC68451 memory management unit for the Motorola 68000 Family of Microprocessors allows allocation of memory segments with sizes ranging from 256 bytes to 16 megabytes in powers of 2. Each (physical) memory management unit contains 32 segment descriptors. Describe an efficient method for memory allocation. What should the implementation of regions look like?
6. Consider the virtual address map in Figure 6.5. Suppose the kernel swaps the process out (in a swapping system) or swaps out many pages in the stack region (in a paging system). If the process later reads (virtual) address 68,432, must it read the identical location in physical memory that it would have read before the swap or paging operation? If the lower levels of memory management were implemented with page tables, must the page tables be located in the same locations of physical memory?
* 7. It is possible to implement the system such that the kernel stack grows on top of the user stack. Discuss the advantages and disadvantages of such an implementation.
8. When attaching a region to a process, how can the kernel check that the region does not overlap virtual addresses in regions already attached to the process?
9. Consider the algorithm for doing a context switch. Suppose the system contains only one process that is ready to run. In other words, the kernel picks the process that just saved its context to run. Describe what happens.

10. Suppose a process goes to sleep and the system contains no processes ready to run. What happens when the (about to be) sleeping process does its context switch?
11. Suppose that a process executing in user mode uses up its time slice and, as a result of a clock interrupt, the kernel schedules a new process to run. Show that the context switch takes place at kernel context layer 2.
12. In a paging system, a process executing in user mode may incur a page fault because it is attempting to access a page that is not loaded in memory. In the course of servicing the interrupt, the kernel reads the page from a swap device and goes to sleep. Show that the context switch (during the sleep) takes place at kernel context layer 2.
13. A process executes the system call

    ```
    read(fd, buf, 1024);
    ```

    on a paging system. Suppose the kernel executes algorithm *read* to the point where it has read the data into a system buffer, but it incurs a page fault when trying to copy the data into the user address space because the page containing *buf* was paged out. The kernel handles the interrupt by reading the offending page into memory. What happens in each kernel context layer? What happens if the page fault handler goes to sleep while waiting for the page to be written into main memory?
14. When copying data from user address space to the kernel in Figure 6.17, what would happen if the user supplied address was illegal?
* 15. In algorithms *sleep* and *wakeup*, the kernel raises the processor execution level to prevent interrupts. What bad things could happen if it did not raise the processor execution level? (Hint: The kernel frequently awakens sleeping processes from interrupt handlers.)
* 16. Suppose a process attempts to go to sleep on event A but has not yet executed the code in the *sleep* algorithm to block interrupts; suppose an interrupt occurs before the process raises the processor execution level in *sleep*, and the interrupt handler attempts to awaken all processes asleep on event A. What will happen to the process attempting to go to sleep? Is this a dangerous situation? If so, how can the kernel avoid it?
17. What happens if the kernel issues a *wakeup* call for all processes asleep on address A, but no processes are asleep on that address at the time?
18. Many processes can sleep on an address, but the kernel may want to wake up selected processes that receive a signal. Assume the signal mechanism can identify the particular processes. Describe how the *wakeup* algorithm should be changed to wake up one process on a sleep address instead of all the processes.
19. The Multics system contains algorithms for *sleep* and *wakeup* with the following syntax:

    ```
    sleep(event);
    wakeup(event, priority);
    ```

    That is, the *wakeup* algorithm assigns a priority to the process it is awakening Compare these calls to the *sleep* and *wakeup* calls in the UNIX system.

7

PROCESS CONTROL

The last chapter defined the context of a process and explained the algorithms that manipulate it; this chapter will describe the use and implementation of the system calls that control the process context. The *fork* system call creates a new process, the *exit* call terminates process execution, and the *wait* call allows a *parent* process to synchronize its execution with the *exit* of a *child* process. Signals inform processes of asynchronous events. Because the kernel synchronizes execution of *exit* and *wait* via signals, the chapter presents signals before *exit* and *wait*. The *exec* system call allows a process to invoke a "new" program, overlaying its address space with the executable image of a file. The *brk* system call allows a process to allocate more memory dynamically; similarly, the system allows the user stack to grow dynamically by allocating more space when necessary, using the same mechanisms as for *brk*. Finally, the chapter sketches the construction of the major loops of the shell and of *init*.

Figure 7.1 shows the relationship between the system calls described in this chapter and the memory management algorithms described in the last chapter. Almost all calls use *sleep* and *wakeup*, not shown in the figure. Furthermore, *exec* interacts with the file system algorithms described in Chapters 4 and 5.

System Calls Dealing with Memory Management			System Calls Dealing with Synchronization				Miscellaneous	
fork	exec	brk	exit	wait	signal	kill	setpgrp	setuid
dupreg attachreg	detachreg allocreg attachreg growreg loadreg mapreg	growreg	detachreg					

Figure 7.1. Process System Calls and Relation to Other Algorithms

7.1 PROCESS CREATION

The only way for a user to create a new process in the UNIX operating system is to invoke the *fork* system call. The process that invokes *fork* is called the *parent* process, and the newly created process is called the *child* process. The syntax for the *fork* system call is

```
pid = fork();
```

On return from the *fork* system call, the two processes have identical copies of their user-level context except for the return value *pid*. In the parent process, *pid* is the child process ID; in the child process, *pid* is 0. Process 0, created internally by the kernel when the system is booted, is the only process not created via *fork*.

The kernel does the following sequence of operations for *fork*.

1. It allocates a slot in the process table for the new process.
2. It assigns a unique ID number to the child process.
3. It makes a logical copy of the context of the parent process. Since certain portions of a process, such as the text region, may be shared between processes, the kernel can sometimes increment a region reference count instead of copying the region to a new physical location in memory.
4. It increments file and inode table counters for files associated with the process.
5. It returns the ID number of the child to the parent process, and a 0 value to the child process.

The implementation of the *fork* system call is not trivial, because the child process appears to start its execution sequence out of thin air. The algorithm for *fork* varies slightly for demand paging and swapping systems; the ensuing discussion is

based on traditional swapping systems but will point out the places that change for demand paging systems. It also assumes that the system has enough main memory available to store the child process. Chapter 9 considers the case where not enough memory is available for the child process, and it also describes the implementation of *fork* on a paging system.

```
algorithm fork
input:  none
output: to parent process, child PID number
        to child process, 0
{
      check for available kernel resources;
      get free proc table slot, unique PID number;
      check that user not running too many processes;
      mark child state "being created;"
      copy data from parent proc table slot to new child slot;
      increment counts on current directory inode and changed root (if applicable);
      increment open file counts in file table;
      make copy of parent context (u area, text, data, stack) in memory;
      push dummy system level context layer onto child system level context;
                  dummy context contains data allowing child process
                  to recognize itself, and start running from here
                  when scheduled;
      if (executing process is parent process)
      {
            change child state to "ready to run;"
            return(child ID);       /* from system to user */
      }
      else      /* executing process is the child process */
      {
            initialize u area timing fields;
            return(0);        /* to user */
      }
}
```

Figure 7.2. Algorithm for Fork

Figure 7.2 shows the algorithm for *fork*. The kernel first ascertains that it has available resources to complete the *fork* successfully. On a swapping system, it needs space either in memory or on disk to hold the child process; on a paging system, it has to allocate memory for auxiliary tables such as page tables. If the resources are unavailable, the *fork* call fails. The kernel finds a slot in the process table to start constructing the context of the child process and makes sure that the user doing the *fork* does not have too many processes already running. It also picks a unique ID number for the new process, one greater than the most recently

assigned ID number. If another process already has the proposed ID number, the kernel attempts to assign the next higher ID number. When the ID numbers reach a maximum value, assignment starts from 0 again. Since most processes execute for a short time, most ID numbers are not in use when ID assignment wraps around.

The system imposes a (configurable) limit on the number of processes a user can simultaneously execute so that no user can steal many process table slots, thereby preventing other users from creating new processes. Similarly, ordinary users cannot create a process that would occupy the last remaining slot in the process table, or else the system could effectively deadlock. That is, the kernel cannot guarantee that existing processes will *exit* naturally and, therefore, no new processes could be created, because all the process table slots are in use. On the other hand, a superuser can execute as many processes as it likes, bounded by the size of the process table, and a superuser process *can* occupy the last available slot in the process table. Presumably, a superuser could take drastic action and spawn a process that forces other processes to *exit* if necessary (see Section 7.2.3 for the *kill* system call).

The kernel next initializes the child's process table slot, copying various fields from the parent slot. For instance, the child "inherits" the parent process real and effective user ID numbers, the parent process group, and the parent *nice* value, used for calculation of scheduling priority. Later sections discuss the meaning of these fields. The kernel assigns the parent process ID field in the child slot, putting the child in the process tree structure, and initializes various scheduling parameters, such as the initial priority value, initial CPU usage, and other timing fields. The initial state of the process is "being created" (recall Figure 6.1).

The kernel now adjusts reference counts for files with which the child process is automatically associated. First, the child process resides in the current directory of the parent process. The number of processes that currently access the directory increases by 1 and, accordingly, the kernel increments its inode reference count. Second, if the parent process or one of its ancestors had ever executed the *chroot* system call to change its root, the child process inherits the changed root and increments its inode reference count. Finally, the kernel searches the parent's user file descriptor table for open files known to the process and increments the global file table reference count associated with each open file. Not only does the child process inherit access rights to open files, but it also shares access to the files with the parent process because both processes manipulate the same file table entries. The effect of *fork* is similar to that of *dup* vis-a-vis open files: A new entry in the user file descriptor table points to the entry in the global file table for the open file. For *dup*, however, the entries in the user file descriptor table are in one process; for *fork*, they are in different processes.

The kernel is now ready to create the user-level context of the child process. It allocates memory for the child process *u area*, regions, and auxiliary page tables, duplicates every region in the parent process using algorithm *dupreg*, and attaches every region to the child process using algorithm *attachreg*. In a swapping system,

it copies the contents of regions that are not shared into a new area of main memory. Recall from Section 6.2.4 that the *u area* contains a pointer to its process table slot. Except for that field, the contents of the child *u area* are initially the same as the contents of the parent process *u area*, but they can diverge after completion of the *fork*. For instance, the parent process may *open* a new file after the *fork*, but the child process does not have automatic access to it.

So far, the kernel has created the static portion of the child context; now it creates the dynamic portion. The kernel copies the parent context layer 1, containing the user saved register context and the kernel stack frame of the *fork* system call. If the implementation is one where the kernel stack is part of the *u area*, the kernel automatically creates the child kernel stack when it creates the child *u area*. Otherwise, the parent process must copy its kernel stack to a private area of memory associated with the child process. In either case, the kernel stacks for the parent and child processes are identical. The kernel then creates a dummy context layer (2) for the child process, containing the saved register context for context layer (1). It sets the program counter and other registers in the saved register context so that it can "restore" the child context, even though it had never executed before, and so that the child process can recognize itself as the child when it runs. For instance, if the kernel code tests the value of register 0 to decide if the process is the parent or the child, it writes the appropriate value in the child saved register context in layer 1. The mechanism is similar to that discussed for a context switch in the previous chapter.

When the child context is ready, the parent completes its part of *fork* by changing the child state to "ready to run (in memory)" and by returning the child process ID to the user. The kernel later schedules the child process for execution via the normal scheduling algorithm, and the child process "completes" *its* part of the *fork*. The context of the child process was set up by the parent process; to the kernel, the child process appears to have awakened after awaiting a resource. The child process executes part of the code for the *fork* system call, according to the program counter that the kernel restored from the saved register context in context layer 2, and returns a 0 from the system call.

Figure 7.3 gives a logical view of the parent and child processes and their relationship to other kernel data structures immediately after completion of the *fork* system call. To summarize, both processes share files that the parent had open at the time of the *fork*, and the file table reference count for those files is one greater than it had been. Similarly, the child process has the same current directory and changed root (if applicable) as the parent, and the inode reference count of those directories is one greater than it had been. The processes have identical copies of the text, data, and (user) stack regions; the region type and the system implementation determine whether the processes can share a physical copy of the text region.

Consider the program in Figure 7.4, an example of sharing file access across a *fork* system call. A user should invoke the program with two parameters, the name of an existing file and the name of a new file to be created. The process *opens* the

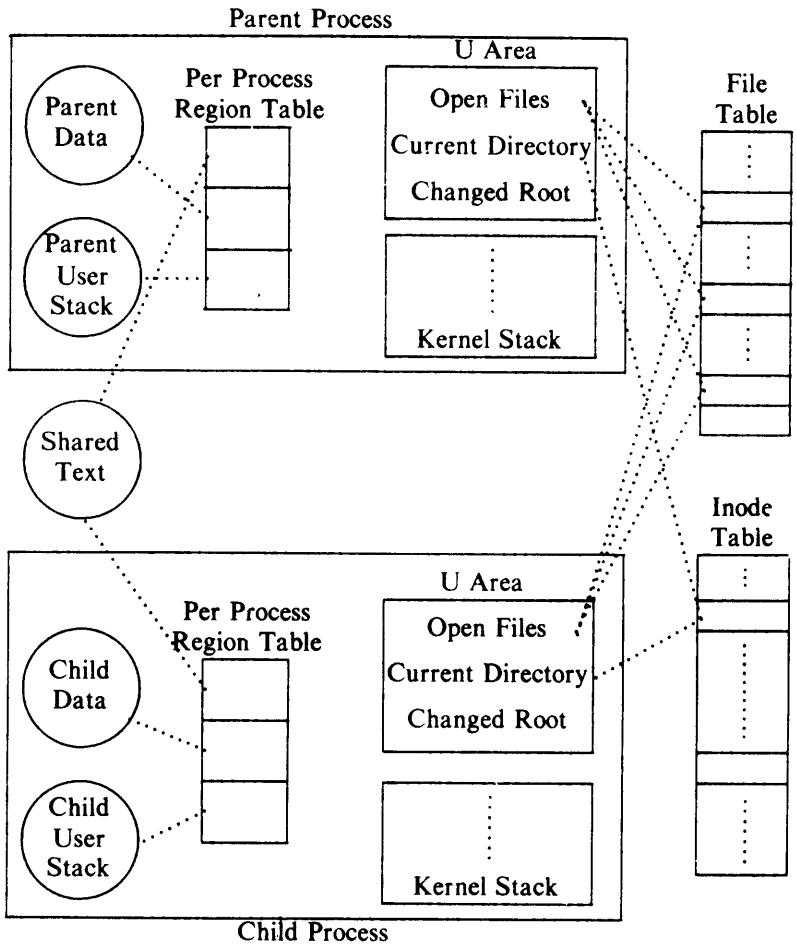

Figure 7.3. Fork Creating a New Process Context

existing file, *creat*s the new file, and — assuming it encounters no errors — *fork*s and creates a child process. Internally, the kernel makes a copy of the parent context for the child process, and the parent process executes in one address space and the child process executes in another. Each process can access private copies of the global variables *fdrd*, *fdwt*, and *c* and private copies of the stack variables *argc* and *argv*, but neither process can access the variables of the other process. However, the kernel copied the *u area* of the original process to the child process during the *fork*, and the child thus inherits access to the parent files (that is, the files the parent originally *open*ed and *creat*ed) using the same file descriptors.

```
#include <fcntl.h>
int fdrd, fdwt;
char c;

main(argc, argv)
        int argc;
        char *argv[];
{
        if (argc != 3)
                exit(1);
        if ((fdrd = open(argv[1], O_RDONLY)) == -1)
                exit(1);
        if ((fdwt = creat(argv[2], 0666)) == -1)
                exit(1);

        fork();
        /* both procs execute same code */
        rdwrt();
        exit(0);
}

rdwrt()
{
        for (;;)
        {
                if (read(fdrd, &c, 1) != 1)
                        return;
                write(fdwt, &c, 1);
        }
}
```

Figure 7.4. Program where Parent and Child Share File Access

The parent and child processes call the function *rdwrt*, independently, of course, and execute a loop, *read*ing one byte from the source file and *writ*ing it to the target file. The function *rdwrt* returns when the *read* system call encounters the end of file. The kernel had incremented the file table counts of the source and target files, and the file descriptors in both processes refer to the *same* file table entries. That is, the file descriptors *fdrd* for both processes refer to the file table entry for the source file, and the file descriptors *fdwt* for both processes refer to the file table entry for the target file. Therefore, the two processes never *read* or *write* the same file offset values, because the kernel increments them after each *read* and *write* call. Although the processes appear to copy the source file twice as fast because they share the work load, the contents of the target file depend on the order that the kernel scheduled the processes. If it schedules the processes such

that they alternate execution of their system calls, or even if they alternate the execution of pairs of *read-write* system calls, the contents of the target file would be identical to the contents of the source file. But consider the following scenario where the processes are about to *read* the two character sequence "ab" in the source file. Suppose the parent process *read*s the character 'a', and the kernel does a context switch to execute the child process before the parent does the *write*. If the child process *read*s the character 'b' and *write*s it to the target file before the parent is rescheduled, the target file will not contain the string "ab" in the proper place, but "ba". The kernel does not guarantee the relative rates of process execution.

Now consider the program in Figure 7.5, which inherits file descriptors 0 and 1 (standard input and standard output) from its parent. The execution of each *pipe* system call allocates two more file descriptors in the arrays *to_par* and *to_chil*, respectively. The process *fork*s and makes a copy of its context: each process can access its own data, as in the previous example. The parent process *close*s its standard output file (file descriptor 1), and *dup*s the write descriptor returned for the pipe *to_chil*. Because the first free slot in the parent file descriptor table is the slot just cleared by the *close*, the kernel copies the pipe write descriptor to slot 1 in the file descriptor table, and the standard output file descriptor becomes the pipe write descriptor for *to_chil*. The parent process does a similar operation to make its standard input descriptor the pipe read descriptor for *to_par*. Similarly, the child process closes its standard input file (descriptor 0) and *dup*s the pipe read descriptor for *to_chil*. Since the first free slot in the file descriptor table is the previous standard input slot, the child standard input becomes the pipe read descriptor for *to_chil*. The child does a similar set of operations to make its standard output the pipe write descriptor for *to par*. Both processes *close* the file descriptors returned from *pipe*— good programming practice, as will be explained. As a result, when the parent *write*s its standard output, it is *writ*ing the pipe *to_chil* and sending data to the child process, which *read*s the pipe on its standard input. When the child *write*s its standard output, it is *writ*ing the pipe *to_par* and sending data to the parent process, which *read*s the pipe on its standard input. The processes thus exchange messages over the two pipes.

The results of this example are invariant, regardless of the order that the processes execute their respective system calls. That is, it makes no difference whether the parent returns from the *fork* call before the child or afterwards. Similarly, it makes no difference in what relative order the processes execute the system calls until they enter their loops: The kernel structures are identical. If the child process executes its *read* system call before the parent does its *write*, the child process will sleep until the parent *write*s the pipe and awakens it. If the parent process *write*s the pipe before the child *read*s the pipe, the parent will not complete its *read* of standard input until the child *read*s *its* standard input and *write*s its standard output. From then on, the order of execution is fixed: Each process completes a *read* and *write*, system call and cannot complete its next *read* system call until the other process completes a *read* and *write* system call. The parent

```
#include <string.h>
char string[] = "hello world";
main()
{
        int count, i;
        int to_par[2], to_chil[2];      /* for pipes to parent, child */
        char buf[256];
        pipe(to_par);
        pipe(to_chil);
        if (fork() == 0)
        {
                /* child process executes here */
                close(0);           /* close old standard input */
                dup(to_chil[0]);    /* dup pipe read to standard input */
                close(1);           /* close old standard output */
                dup(to_par[1]);     /* dup pipe write to standard out */
                close(to_par[1]);   /* close unnecessary pipe descriptors */
                close(to_chil[0]);
                close(to_par[0]);
                close(to_chil[1]);
                for (;;)
                {
                        if ((count = read(0, buf, sizeof(buf))) == 0)
                                exit();
                        write(1, buf, count);
                }
        }
        /* parent process executes here */
        close(1);       /* rearrange standard in, out */
        dup(to_chil[1]);
        close(0);
        dup(to_par[0]);
        close(to_chil[1]);
        close(to_par[0]);
        close(to_chil[0]);
        close(to_par[1]);
        for (i = 0;  i < 15;  i++)
        {
                write(1, string, strlen(string));
                read(0, buf, sizeof(buf));
        }
}
```

Figure 7.5. Use of Pipe, Dup, and Fork

*exit*s after 15 iterations through the loop; the child then *read*s "end-of-file" because the pipe has no writer processes and *exit*s. If the child were to *write* the pipe after the parent had *exit*ed, it would receive a signal for writing a pipe with no reader processes.

We mentioned above that it is good programming practice to *close* superfluous file descriptors. This is true for three reasons. First, it conserves file descriptors in view of the system-imposed limit. Second, if a child process *exec*s, the file descriptors remain assigned in the new context, as will be seen. Closing extraneous files before an *exec* allows programs to execute in a clean, surprise-free environment, with only standard input, standard output, and standard error file descriptors open. Finally, a *read* of a pipe returns end-of-file only if no processes have the pipe open for writing. If a reader process keeps the pipe write descriptor open, it will never know when the writer processes *close* their end of the pipe. The example above would not work properly unless the child *close*s its write pipe descriptors before entering its loop.

7.2 SIGNALS

Signals inform processes of the occurrence of asynchronous events. Processes may send each other *signals* with the *kill* system call, or the kernel may send signals internally. There are 19 signals in the System V (Release 2) UNIX system that can be classified as follows (see the description of the signal system call in [SVID 85]):

- Signals having to do with the termination of a process, sent when a process *exit*s or when a process invokes the *signal* system call with the *death of child* parameter;
- Signals having to do with process induced exceptions such as when a process accesses an address outside its virtual address space, when it attempts to write memory that is read-only (such as program text), or when it executes a privileged instruction or for various hardware errors;
- Signals having to do with the unrecoverable conditions during a system call, such as running out of system resources during *exec* after the original address space has been released (see Section 7.5)
- Signals caused by an unexpected error condition during a system call, such as making a nonexistent system call (the process passed a system call number that does not correspond to a legal system call), writing a pipe that has no reader processes, or using an illegal "reference" value for the *lseek* system call. It would be more consistent to return an error on such system calls instead of generating a signal, but the use of signals to abort misbehaving processes is more pragmatic;[1]