- The clock handler adjusts the priorities of all processes in user mode at 1 second intervals (on System V) and causes the kernel to go through the scheduling algorithm to prevent a process from monopolizing use of the CPU.

The clock may interrupt a process several times during its time quantum; at every clock interrupt, the clock handler increments a field in the process table that records the recent CPU usage of the process. Once a second, the clock handler also adjusts the recent CPU usage of each process according to a decay function,

```
decay(CPU) = CPU/2;
```

on System V. When it recomputes recent CPU usage, the clock handler also recalculates the priority of every process in the "preempted but ready-to-run" state according to the formula

```
priority=("recent CPU usage"/2) + (base level user priority)
```

where "base level user priority" is the threshold priority between kernel and user mode described above. A numerically low value implies a high scheduling priority. Examining the functions for recomputation of recent CPU usage and process priority, the slower the decay rate for recent CPU usage, the longer it will take for the priority of a process to reach its base level; consequently, processes in the "ready-to-run" state will tend to occupy more priority levels.

The effect of priority recalculation once a second is that processes with user-level priorities move between priority queues, as illustrated in Figure 8.3. Comparing this figure to Figure 8.2, one process has moved from the queue for user-level priority 1 to the queue for user-level priority 0. In a real system, all processes with user-level priorities in the figure would change priority queues, but only one has been depicted. The kernel does not change the priority of processes in kernel mode, nor does it allow processes with user-level priority to cross the threshold and attain kernel-level priority, unless they make a system call and go to sleep.

The kernel attempts to recompute the priority of all active processes once a second, but the interval can vary slightly. If the clock interrupt had come while the kernel was executing a critical region of code (that is, while the processor execution level was raised but, obviously, not raised high enough to block out the clock interrupt), the kernel does not recompute priorities, since that would keep the kernel in the critical region for too long a time. Instead, the kernel remembers that it should have recomputed process priorities and does so at a succeeding clock interrupt when the "previous" processor execution level is sufficiently low. Periodic recalculation of process priority assures a round-robin scheduling policy for processes executing in user mode. The kernel responds naturally to interactive requests such as for text editors or form entry programs: such processes have a high idle-time-to-CPU usage ratio, and consequently their priority value naturally rises when they are ready for execution (see page 1937 of [Thompson 78]). Other implementations of the scheduling mechanism vary the time quantum between 0 and 1 second dynamically, depending on system load. Such implementations can

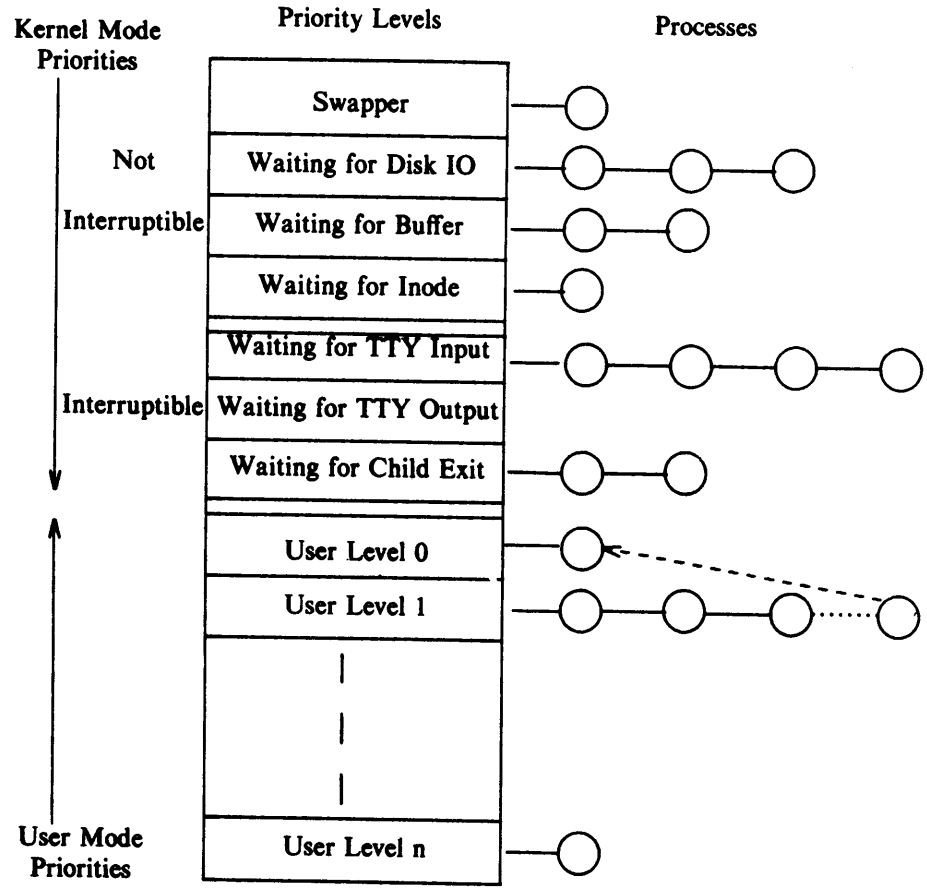


Figure 8.3. Movement of a Process on Priority Queues

thus give quicker response to processes, because they do not have to wait up to a second to run; on the other hand, the kernel has more overhead because of extra context switches.

8.1.3 Examples of Process Scheduling

Figure 8.4 shows the scheduling priorities on System V for 3 processes A, B, and C, under the following assumptions: They are created simultaneously with initial priority 60, the highest user-level priority is 60, the clock interrupts the system 60 times a second, the processes make no system calls, and no other processes are

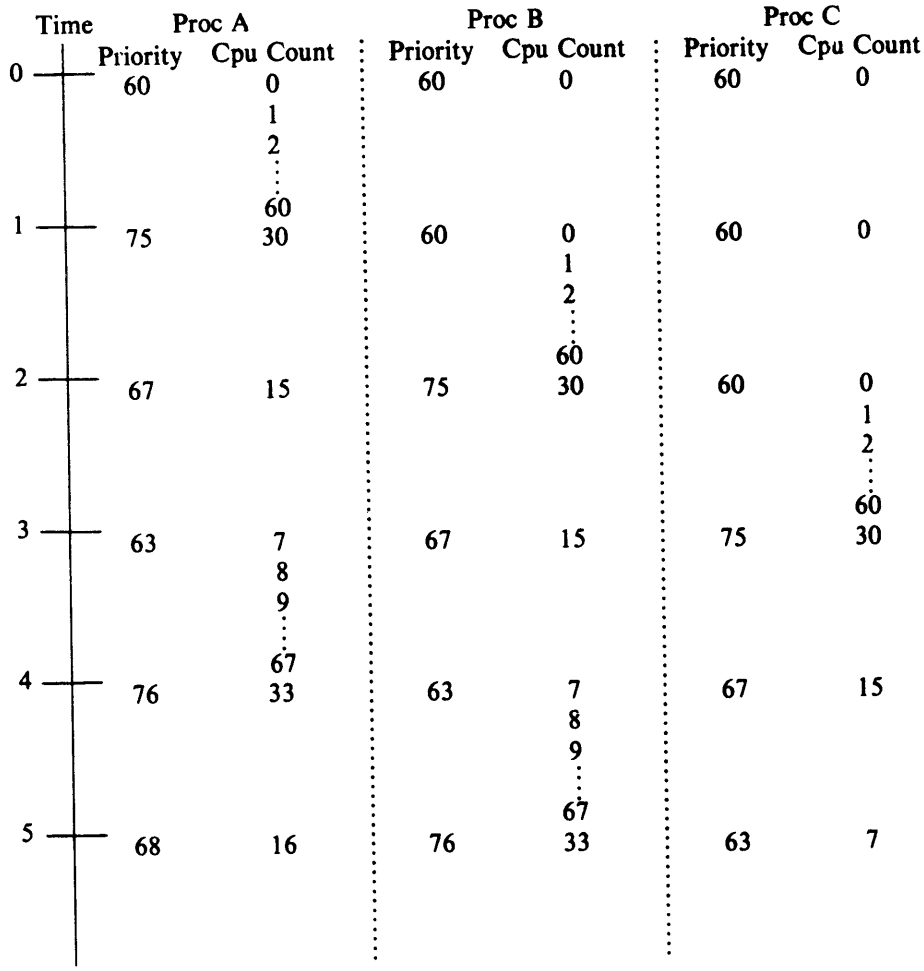


Figure 8.4. Example of Process Scheduling

ready to run. The kernel calculates the decay of the CPU usage by

```
CPU = decay(CPU) = CPU/2;
```

and the process priority as

```
priority = (CPU/2) + 60;
```

Assuming process A is the first to run and that it starts running at the beginning of a time quantum, it runs for 1 second: During that time the clock interrupts the system 60 times and the interrupt handler increments the CPU usage field of

process A 60 times (to 60). The kernel forces a context switch at the 1-second mark and, after recalculating the priorities of all processes, schedules process B for execution. The clock handler increments the CPU usage field of process B 60 times during the next second and then recalculates the CPU usage and priority of all processes and forces a context switch. The pattern repeats, with the processes taking turns to execute.

Now consider the processes with priorities shown in Figure 8.5, and assume other processes are in the system. The kernel may preempt process A, leaving it in the state "ready to run," after it had received several time quanta in succession on the CPU, and its user-level priority may therefore be low (Figure 8.5a). As time progresses, process B may enter the "ready-to-run" state, and its user-level priority may be higher than that of process A at that instant (Figure 8.5b). If the kernel does not schedule either process for a while (it schedules other processes), both processes could eventually be at the same user priority level, although process B would probably enter that level first since its starting level was originally closer (Figures 8.5c and 8.5d). Nevertheless, the kernel would choose to schedule process A ahead of process B because it was in the state "ready to run" for a longer time (Figure 8.5e): This is the tie-breaker rule for processes with equal priority.

Recall from Section 6.4.3 that the kernel schedules a process at the conclusion of a context switch: A process must do a context switch when it goes to sleep or *exit*s, and it has the *opportunity* to do a context switch when returning to user mode from kernel mode. The kernel preempts a process about to return to user mode if a process with higher priority is ready to run. Such a process exists if the kernel awakened a process with higher priority than the currently running process, or if the clock handler changed the priority of all "ready-to-run" processes. In the first case, the current process should not run in user mode given that a higher-priority kernel mode process is available. In the second case, the clock handler decides that the process used up its time quantum, and since many processes had their priorities changed, the kernel does a context switch to reschedule.

8.1.4 Controlling Process Priorities

Processes can exercise crude control of their scheduling priority by using the *nice* system call:

```
nice(value);
```

where *value* is added in the calculation of process priority:

priority=("recent CPU usage"/constant) + (base priority) + (nice value)

The *nice* system call increments or decrements the *nice* field in the process table by the value of the parameter, although only the superuser can supply *nice* values that increase the process priority. Similarly, only the superuser can supply a *nice* value below a particular threshold. Users who invoke the *nice* system call to lower their process priority when executing computation-intensive jobs are "nice" to other users

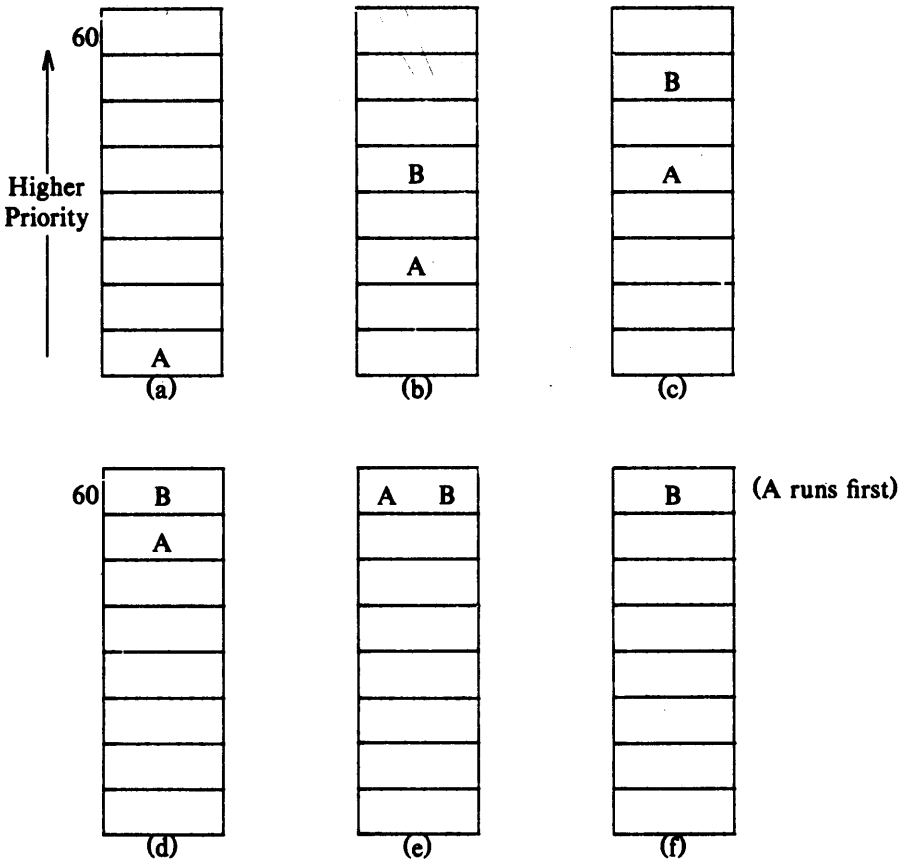


Figure 8.5. Round Robin Scheduling and Process Priorities

on the system, hence the name. Processes inherit the *nice* value of their parent during the *fork* system call. The *nice* system call works for the running process only; a process cannot reset the *nice* value of another process. Practically, this means that if a system administrator wishes to lower the priority values of various processes because they consume too much time, there is no way to do so short of *kill*ing them outright.

8.1.5 Fair Share Scheduler

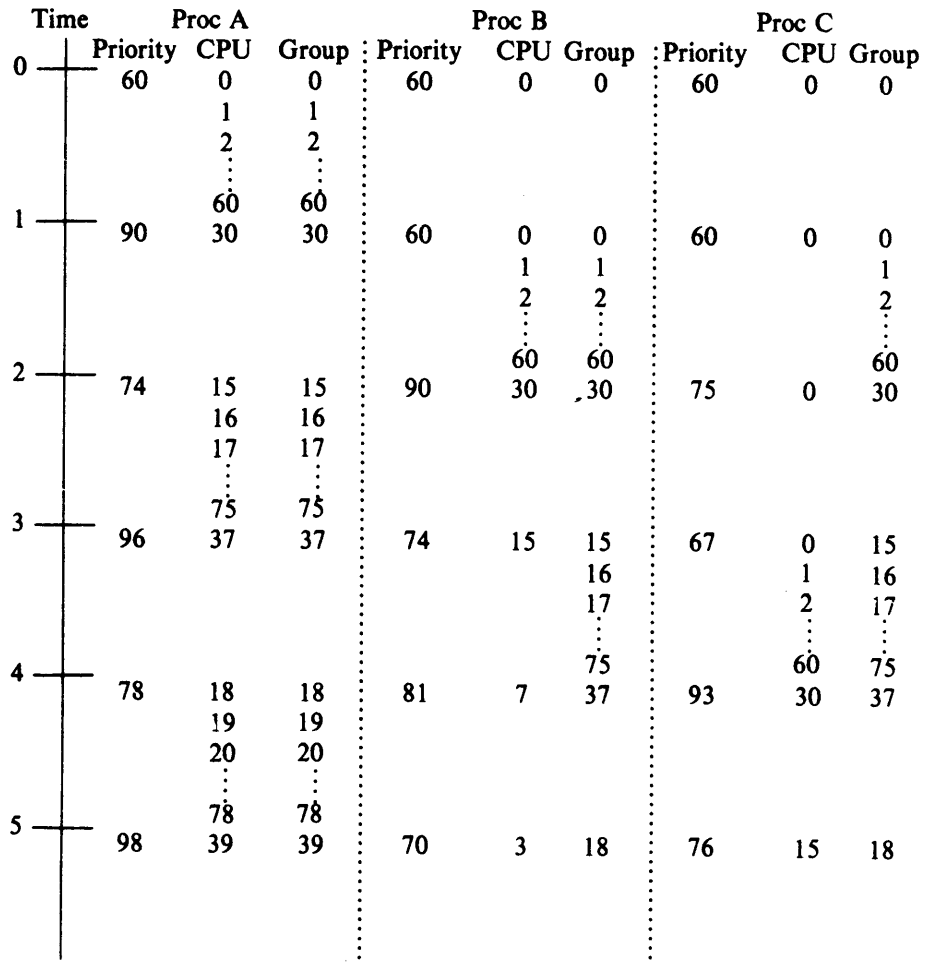
The scheduler algorithm described above does not differentiate between classes of users. That is, it is impossible to allocate half of the CPU time to a particular set

of processes, if desired. However, such considerations are important in a computer center environment, where a set of users may want to buy half of the CPU time of a machine on a guaranteed basis, to ensure a certain level of response. This section describes a scheme called the Fair Share Scheduler, implemented in the AT&T Bell Laboratories Indian Hill Computer Center [Henry 84].

The principle of the fair share scheduler is to divide the user community into a set of fair share groups, such that the members of each group are subject to the constraints of the regular process scheduler relative to other processes in the group. However, the system allocates its CPU time proportionally to each group, regardless of how many processes are in the groups. For example, suppose there are four fair share groups on a system, each with an allocated CPU share of 25%, and that the groups contain 1, 2, 3, and 4 CPU bound processes that never willingly give up the processor (they are in an infinite loop, for instance). Assuming there are no other processes on the system, each process in the four groups would get 10% of the CPU time (there are 10 processes) using the regular scheduling algorithm, because there is no way to distinguish them from each other. But using the fair share scheduler, the process in group 1 will receive twice as much CPU time as each process in group 2, 3 times as much CPU time as each process in group 3, and 4 times as much CPU time as each process in group 4. In this example, the CPU time of all processes in a group should be equal over time, because they are all in an infinite loop.

Implementation of this scheme is simple, a feature that makes it attractive: Another term is added to the formula for computation of process priority, namely, a "fair share group priority." Each process has a new field in its *u area* that points to a fair share CPU usage field, shared by all processes in the fair share group. The clock interrupt handler increments the fair share group CPU usage field for the running process, just as it increments the CPU usage field of the running process and decays the values of all fair share group CPU usage fields once a second. When calculating process priorities, a new component of the calculation is the group CPU usage, normalized according to the amount of CPU time allocated to the fair share group. The more CPU time processes in a group received recently, the higher the numerical value of the group CPU usage field is and, therefore, the lower the priority for all the processes in the fair share group.

For example, consider the three processes depicted in Figure 8.6 and suppose that process A is in one group and processes B and C are in another. Assuming the kernel schedules process A first, it will increment the CPU and group usage fields for process A over the next second. On recomputation of process priorities at the 1-second mark, processes B and C have the highest priority; assume the kernel schedules process B. During the next second, the CPU usage field of process B goes up to 60, as does the group usage field for processes B and C. Hence, on recomputation of process priorities at the 2-second mark, process C will have priority 75 (compare to Figure 8.4), and the kernel will schedule process A, with priority 74. The figure shows how the pattern repeats: the kernel schedules the processes in the order A, B, A, C, A, B, and so on.



Time	Proc A			Proc B			Proc C		
	Priority	CPU	Group	Priority	CPU	Group	Priority	CPU	Group
0	60	0	0	60	0	0	60	0	0
		1	1						
		2	2						
		⋮	⋮						
		60	60						
1	90	30	30	60	0	0	60	0	0
					1	1			1
					2	2			2
					⋮	⋮			⋮
					60	60			60
2	74	15	15	90	30	30	75	0	30
		16	16						
		17	17						
		⋮	⋮						
		75	75						
3	96	37	37	74	15	15	67	0	15
						16		1	16
						17		2	17
						⋮		⋮	⋮
						75		60	75
4	78	18	18	81	7	37	93	30	37
		19	19						
		20	20						
		⋮	⋮						
		78	78						
5	98	39	39	70	3	18	76	15	18

Figure 8.6. Example of Fair Share Scheduler — Three Processes, Two Groups

8.1.6 Real-Time Processing

Real-time processing implies the capability to provide immediate response to specific external events and, hence, to schedule particular processes to run within a specified time limit after occurrence of an event. For example, a computer may monitor the life-support systems of hospital patients to take instant action on a change in status of a patient. Processes such as text editors are not considered real-time processes: It is desirable that response to the user be quick, but it is not that critical that a user cannot wait a few extra seconds (although the user may

have other ideas). The scheduler algorithms described above were designed for use in a time-sharing environment and are inappropriate in a real-time environment, because they cannot guarantee that the kernel can schedule a particular process within a fixed time limit. Another impediment to the support of real-time processing is that the kernel is nonpreemptive; the kernel cannot schedule a real-time process in user mode if it is currently executing another process in kernel mode, unless major changes are made. Currently, system programmers must insert real-time processes into the kernel to achieve real-time response. A true solution to the problem must allow real-time processes to exist dynamically (that is, not be hard-coded in the kernel), providing them with a mechanism to inform the kernel of their real-time constraints. No standard UNIX system has this capability today.

8.2 SYSTEM CALLS FOR TIME

There are several time-related system calls, *stime*, *time*, *times*, and *alarm*. The first two deal with global system time, and the latter two deal with time for individual processes.

Stime allows the superuser to set a global kernel variable to a value that gives the current time:

```
stime(pvalue);
```

where *pvalue* points to a long integer that gives the time as measured in seconds from midnight before (00:00:00) January 1, 1970, GMT. The clock interrupt handler increments the kernel variable once a second. *Time* retrieves the time as set by *stime*:

```
time(tloc);
```

where *tloc* points to a location in the user process for the return value. *Time* returns this value from the system call, too. Commands such as *date* use *time* to determine the current time.

Times retrieves the cumulative times that the calling process spent executing in user mode and kernel mode and the cumulative times that all zombie children had executed in user mode and kernel mode. The syntax for the call is

```
times(tbuffer)
struct tms *tbuffer;
```

where the structure *tms* contains the retrieved times and is defined by

```
#include <sys/types.h>
#include <sys/times.h>
extern long times();

main()
{
        int i;
        /* tms is data structure containing the 4 time elements */
        struct tms pb1, pb2;
        long pt1, pt2;

        pt1 = times(&pb1);
        for (i = 0;  i < 10;  i++)
                if (fork() == 0)
                        child(i);

        for (i = 0;  i < 10;  i++)
                wait((int *) 0);
        pt2 = times(&pb2);
        printf("parent real %u user %u sys %u cuser %u csys %u\n",
                pt2 - pt1, pb2.tms_utime - pb1.tms_utime, pb2.tms_stime - pb1.tms_stime,
                pb2.tms_cutime - pb1.tms_cutime, pb2.tms_cstime - pb1.tms_cstime);
}

child(n)
        int n;
{
        int i;
        struct tms cb1, cb2;
        long t1, t2;

        t1 = times(&cb1);
        for (i = 0;  i < 10000;  i++)
                ;
        t2 = times(&cb2);
        printf("child %d: real %u user %u sys %u\n", n, t2 - t1,
                cb2.tms_utime - cb1.tms_utime, cb2.tms_stime - cb1.tms_stime);
        exit();
}
```

Figure 8.7. Program Using Times

```
struct tms {
        /* time_t is the data structure for time */
        time_t  tms_utime;          /* user time of process */
        time_t  tms_stime;          /* kernel time of process */
```

```
        time_t  tms_cutime;         /* user time of children */
        time_t  tms_cstime          /* kernel time of children */
};
```

Times returns the elapsed time "from an arbitrary point in the past," usually the time of system boot.

In the program in Figure 8.7, a process creates 10 child processes, and each child loops 10,000 times. The parent process calls *times* before creating the children and after they all *exit*, and the child processes call *times* before and after their loops. One would naively expect the parent *child user* and *child system* times to equal the respective sums of the child processes' *user* and *system* times, and the parent *real time* to equal the sum of the child processes' *real time*. However, the child times do not include time spent in the *fork* and *exit* system calls, and all times can be distorted by time spent handling interrupts or doing context switches.

User processes can schedule alarm signals using the *alarm* system call. For example, the program in Figure 8.8 checks the access time of a file every minute and prints a message if the file had been accessed. To do so, it enters an infinite loop: During each iteration, it calls *stat* to report the last time the file was accessed and, if accessed during the last minute, prints a message. The process then calls *signal* to catch alarm signals, calls *alarm* to schedule an alarm signal in 60 seconds, and calls *pause* to suspend its activity until receipt of a signal. After 60 seconds, the alarm signal goes off, the kernel sets up the process user stack to call the signal catcher function *wakeup*, the function returns to the position in the code after the *pause* call, and the process executes the loop again.

The common factor in all the time related system calls is their reliance on the system clock: the kernel manipulates various time counters when handling clock interrupts and initiates appropriate action.

8.3 CLOCK

The functions of the clock interrupt handler are to

- restart the clock,
- schedule invocation of internal kernel functions based on internal timers,
- provide execution profiling capability for the kernel and for user processes,
- gather system and process accounting statistics,
- keep track of time,
- send alarm signals to processes on request,
- periodically wake up the swapper process (see the next chapter),
- control process scheduling.

Some operations are done every clock interrupt, whereas others are done after several clock ticks. The clock handler runs with the processor execution level set high, preventing other events (such as interrupts from peripheral devices) from happening while the handler is active. The clock handler is therefore fast, so that

```
#include  <sys/types.h>
#include  <sys/stat.h>
#include  <sys/signal.h>

main(argc, argv)
      int argc;
      char *argv[];
{
      extern unsigned alarm();
      extern wakeup();
      struct stat statbuf;
      time_t axtime;

      if (argc != 2)
      {
            printf("only 1 arg\n");
            exit();
      }

      axtime = (time_t) 0;
      for (;;)
      {
            /* find out file access time */
            if (stat(argv[1], &statbuf) == -1)
            {
                  printf("file %s not there\n", argv[1]);
                  exit();
            }
            if (axtime != statbuf.st_atime)
            {
                  printf("file %s accessed\n", argv[1])
                  axtime = statbuf.st_atime;
            }
            signal(SIGALRM, wakeup);      /* reset for alarm */
            alarm(60);
            pause();                /* sleep until signal */
      }
}

wakeup()
{
}
```

Figure 8.8. Program Using Alarm Call

```
algorithm clock
input:  none
output: none
{
        restart clock;              /* so that it will interrupt again */
        if (callout table not empty)
        {
                adjust callout times;
                schedule callout function if time elapsed;
        }
        if (kernel profiling on)
                note program counter at time of interrupt;
        if (user profiling on)
                note program counter at time of interrupt;
        gather system statistics;
        gather statistics per process;
        adjust measure of process CPU utilitization;
        if (1 second or more since last here and interrupt not in critical
                                                region of code)
        {
                for (all processes in the system)
                {
                        adjust alarm time if active;
                        adjust measure of CPU utilization;
                        if (process to execute in user mode)
                                adjust process priority;
                }
                wakeup swapper process is necessary;
        }
}
```

Figure 8.9. Algorithm for the Clock Handler

the critical time periods when other interrupts are blocked is as small as possible. Figure 8.9 shows the algorithm for handling clock interrupts.

8.3.1 Restarting the Clock

When the clock interrupts the system, most machines require that the clock be reprimed by software instructions so that it will interrupt the processor again after a suitable interval. Such instructions are hardware dependent and will not be discussed.

8.3.2 Internal System Timeouts

Some kernel operations, particularly device drivers and network protocols, require invocation of kernel functions on a real-time basis. For instance, a process may put a terminal into raw mode so that the kernel satisfies user *read* requests at fixed intervals instead of waiting for the user to type a carriage return (see Section 10.3.3). The kernel stores the necessary information in the *callout* table (Figure 8.9), which consists of the function to be invoked when time expires, a parameter for the function, and the time in clock ticks until the function should be called.

The user has no direct control over the entries in the callout table; various kernel algorithms make entries as needed. The kernel sorts entries in the callout table according to their respective "time to fire," independent of the order they are placed in the table. Because of the time ordering, the time field for each entry in the callout table is stored as the amount of time to fire after the previous element fires. The total time to fire for a given element in the table is the sum of the times to fire of all entries up to and including the element.

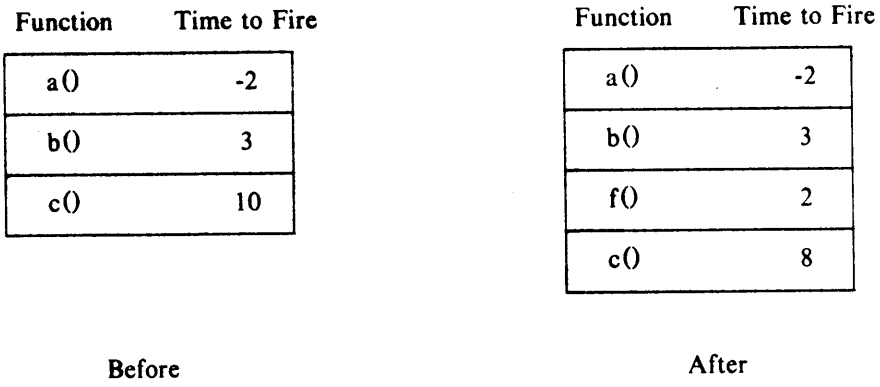


Figure 8.10. Callout Table and New Entry for f

Figure 8.10 shows an instance of the *callout* table before and after addition of a new entry for the function *f*. (The negative time field for function *a* will be explained shortly.) When making a new entry, the kernel finds the correct (timed) position for the new entry and appropriately adjusts the time field of the entry immediately after the new entry. In the figure, the kernel arranges to invoke function *f* after 5 clock ticks: it creates an entry for *f* after the entry for *b* with the value of its time field 2 (the sum of the time fields for *b* and *f* is 5), and changes the time field for *c* to 8 (*c* will still fire in 13 clock ticks). Kernel implementations can use a linked list for each entry of the callout table, or they can readjust position of the entries when changing the table. The latter option is not that expensive if the kernel does not use the callout table too much.

At every clock interrupt, the clock handler checks if there are any entries in the callout table and, if there are any, decrements the time field of the first entry. Because of the way the kernel keeps time in the callout table, decrementing the time field for the first entry effectively decrements the time field for all entries in the table. If the time field of the first entry in the list is less than or equal to 0, then the specified function should be invoked. The clock handler does not invoke the function directly so that it does not inadvertently block later clock interrupts: The processor priority level is currently set to block out clock interrupts, but the kernel has no idea how long the function will take to complete. If the function were to last longer than a clock tick, the next clock interrupt (and all other interrupts that occur) would be blocked. Instead, the clock handler typically schedules the function by causing a "software interrupt," sometimes called a "programmed interrupt" because it is caused by execution of a particular machine instruction. Because software interrupts are at a lower priority level than other interrupts, they are blocked until the kernel finishes handling all other interrupts. Many interrupts, including clock interrupts, could occur between the time the kernel is ready to call a function in the callout table and the time the software interrupt occurs and, therefore, the time field of the first callout entry can have a negative value. When the software interrupt finally happens, the interrupt handler removes entries from the callout table whose time fields have expired and calls the appropriate function.

Since it is possible that the time field of the first entries in the callout table are 0 or negative, the clock handler must find the first entry whose time field is positive and decrement it. In Figure 8.10 for example, the time field of the entry for function *a* is −2, meaning that the system took 2 clock interrupts after *a* was eligible to be called. Assuming the entry for *b* was in the table 2 ticks ago, the kernel skipped the entry for *a* and decremented the time field for *b*.

8.3.3 Profiling

Kernel profiling gives a measure of how much time the system is executing in user mode versus kernel mode, and how much time it spends executing individual routines in the kernel. The kernel profile driver monitors the relative performance of kernel modules by sampling system activity at the time of a clock interrupt. The profile driver has a list of kernel addresses to sample, usually addresses of kernel functions; a process had previously down-loaded these addresses by *writ*ing the profile driver. If kernel profiling is enabled, the clock interrupt handler invokes the interrupt handler of the profile driver, which determines whether the processor mode at the time of the interrupt was user or kernel. If the mode was user, the profiler increments a count for user execution, but if the mode was kernel, it increments an internal counter corresponding to the program counter. User processes can read the profile driver to obtain the kernel counts and do statistical measurements.

Algorithm	Address	Count
bread	100	5
breada	150	0
bwrite	200	0
brelse	300	2
getblk	400	1
user	—	2

Figure 8.11. Sample Addresses of Kernel Algorithms

For example, Figure 8.11 shows hypothetical addresses of several kernel routines. If the sequence of program counter values sampled over 10 clock interrupts is 110, 330, 145, address in user space, 125, 440, 130, 320, address in user space, and 104, the figure shows the counts the kernel would save. Examining these figures, one would conclude that the system spends 20% of its time in user mode and 50% of its time executing the kernel algorithm *bread*.

If kernel profiling is done for a long time period, the sampled pattern of program counter values converges toward a true proportion of system usage. However, the mechanism does not account for time spent executing the clock handler and code that blocks out clock-level interrupts, because the clock cannot interrupt such critical regions of code and therefore cannot invoke the profile interrupt handler there. This is unfortunate since such critical regions of kernel code are frequently those that are the most important to profile. Hence, results of kernel profiling must be taken with a grain of salt. Weinberger [Weinberger 84] describes a scheme for generating counters into basic blocks of code, such as the body of "if-then" and "else" statements, to provide exact counts of how many times they are executed. However, the method increases CPU time anywhere from 50% to 200%, so its use as a permanent kernel profiling mechanism is not practical.

Users can profile execution of processes at user-level with the *profil* system call:

```
profil(buff, bufsize, offset, scale);
```

where *buff* is the address of an array in user space, *bufsize* is the size of the array, *offset* is the virtual address of a user subroutine (usually, the first), and *scale* is a factor that maps user virtual addresses into the array. The kernel treats *scale* as a fixed-point binary fraction with the binary point at the extreme "left": The hexadecimal value 0xffff gives a one to one mapping of program counters to words in *buff*, 0x7fff maps pairs of program addresses into a single *buff* word, 0x3fff maps groups of 4 program addresses into a single *buff* word, and so on. The kernel stores the system call parameters in the process *u area*. When the clock interrupts the process while in user mode, the clock handler examines the user program counter at the time of the interrupt, compares it to *offset*, and increments a location in *buff* whose address is a function of *bufsize* and *scale*.

```
#include <signal.h>
int buffer[4096];
main()
{
        int offset, endof, scale, eff, gee, text;
        extern theend(), f(), g();
        signal(SIGINT, theend);
        endof = (int) theend;
        offset = (int) main;
        /* calculates number of words of program text */
        text = (endof - offset + sizeof(int) - 1)/sizeof(int);
        scale = 0xffff;
        printf("offset %d endof %d text %d\n", offset, endof, text);
        eff = (int) f;
        gee = (int) g;
        printf("f %d  g %d  fdiff %d gdiff %d\n", eff, gee, eff-offset, gee-offset);
        profil(buffer, sizeof(int)*text, offset, scale);
        for (;;)
        {
                f();
                g();
        }
}
f()
{
}
g()
{
}
theend()
{
        int i;
        for (i = 0; i < 4096;  i++)
                if (buffer[i])
                        printf("buf[%d] = %d\n", i, buffer[i]);
        exit();
}
```

Figure 8.12. Program Invoking Profil System Call

For example, consider the program in Figure 8.12, profiling execution of a program that calls the two functions *f* and *g* successively in an infinite loop. The process first invokes *signal* to arrange to call the function *theend* on occurrence of an interrupt signal and then calculates the range of text addresses it wishes to profile, extending from the address of the function *main* to the address of the function *theend*, and, finally, invokes *profil* to inform the kernel that it wishes to

```
offset 212 endof 440 text 57
f 416  g 428  fdiff 204 gdiff 216
buf[46] = 50
buf[48] = 8585216
buf[49] = 151
buf[51] = 12189799
buf[53] = 65
buf[54] = 10682455
buf[56] = 67
```

Figure 8.13. Sample Output for Profil Program

profile its execution. Running the program for about 10 seconds on a lightly loaded AT&T 3B20 computer gave the output shown in Figure 8.13. The address of *f* is 204 greater than the 0th profiling address; because the size of the text of *f* is 12 bytes and the size of an integer is 4 on an AT&T 3B20 computer, the addresses of *f* map into *buf* entries 51, 52, and 53. Similarly, the addresses of *g* map into *buf* entries 54, 55, and 56. The *buf* entries 46, 48, and 49 are for addresses in the loop in function *main*. In typical usage, the range of addresses to be profiled is determined by examination of the text addresses in the symbol table of the program being profiled. Users are discouraged from using the *profil* call directly because it is complicated; instead, an option on the C compiler directs the compiler to generate code to profile processes.

8.3.4 Accounting and Statistics

When the clock interrupts the system, the system may be executing in kernel mode, executing in user mode, or idle (not executing any processes). It is idle if all processes are sleeping, awaiting the occurrence of an event. The kernel keeps internal counters for each processor state and adjusts them during each clock interrupt, noting the current mode of the machine. User processes can later analyze the statistics gathered in the kernel.

Every process has two fields in its *u area* to keep a record of elapsed kernel and user time. When handling clock interrupts, the kernel updates the appropriate field for the executing process, depending on whether the process was executing in kernel mode or in user mode. Parent processes gather statistics for their child processes in the *wait* system call when accumulating execution statistics for *exit*ing child processes.

Every process has one field in its *u area* for the kernel to log its memory usage. When the clock interrupts a running process, the kernel calculates the total memory used by a process as a function of its private memory regions and its proportional usage of shared memory regions. For example, if a process shares a text region of size 50K bytes with four other processes and uses data and stack regions of size

25K and 40K bytes, respectively, the kernel charges the process for 75K bytes (50K/5 + 25K + 40K). For a paging system, it calculates the memory usage by counting the number of valid pages in each region. Thus, if the interrupted process uses two private regions and shares another region with another process, the kernel charges it for the number of valid pages in the private regions plus half the number of valid pages in the shared region. The kernel writes the information in an accounting record when the process *exit*s, and the information can be used for customer billing.

8.3.5 Keeping Time

The kernel increments a timer variable at every clock interrupt, keeping time in clock ticks from the time the system was booted. The kernel uses the timer variable to return a time value for the *time* system call, and to calculate the total (real time) execution time of a process. The kernel saves the process start time in its *u area* when a process is created in the *fork* system call, and it subtracts that value from the current time when the process *exit*s, giving the real execution time of the process. Another timer variable, set by the *stime* system call and updated once a second, keeps track of calendar time.

8.4 SUMMARY

This chapter has described the basic algorithm for process scheduling on the UNIX system. The kernel associates a scheduling priority with every process in the system, assigning the value when a process goes to sleep or, periodically, in the clock interrupt handler. The priority assigned when a process goes to sleep is a fixed value, dependent on the kernel algorithm the process was executing. The priority assigned in the clock handler (or when a process returns from kernel mode to user mode) depends on how much time the process has recently used the CPU: It receives a lower priority if it has used the CPU recently and a higher priority, otherwise. The *nice* system call allows a process to adjust one parameter used in computation of process priority.

This chapter also described system calls dealing with time: setting and retrieving kernel time, retrieving process execution times, and setting process alarm signals. Finally, it described the functions of the clock interrupt handler, which keeps track of system time, manages the callout table, gathers statistics, and arranges for invocation of the process scheduler, process swapper, and page stealer. The swapper and page stealer are the topics of the next chapter.

8.5 EXERCISES

1. In assigning priorities when a process goes to sleep, the kernel assigns a higher priority to a process waiting for a locked inode than to a process waiting for a locked buffer.

Similarly, it assigns higher priority to processes waiting to read terminal input than to processes waiting to write terminal output. Justify both cases.

* 2. The algorithm for the clock interrupt handler recalculates process priorities and reschedules processes in 1-second intervals. Discuss an algorithm that dynamically changes the interval depending on system load. Is the gain worth the added complexity?

3. The Sixth Edition of the UNIX system uses the following formula to adjust the recent CPU usage of a process:

```
decay(CPU) = max(threshold priority, CPU - 10);
```

and the Seventh Edition uses the formula:

```
decay(CPU) = .8 * CPU;
```

Both systems calculate process priority by the formula

```
priority = CPU/16 + (base level priority);
```

Try the example in Figure 8.4 using these decay functions.

4. Repeat the example in Figure 8.4 with seven processes instead of three. Repeat the example assuming there are 100 clock interrupts per second instead of 60. Comment.

5. Design a scheme such that the system puts a time limit on how long a process executes, forcing it to exit if it exceeds the time limit. How should the user distinguish such processes from processes that should run for ever? If the only requirement was to run such a scheme from the shell, what would have to be done?

6. When a process executes the *wait* system call and finds a zombie process, the kernel adds the child's CPU usage field to the parent's. What is the rationale for penalizing the parent?

7. The command *nice* causes the subsequent command to be invoked with the given nice value, as in

```
nice 6 nroff -mm big_memo > output
```

Write C code for the *nice* command.

8. Trace the scheduling of the processes in Figure 8.4 given that the *nice* value of process A is 5 or −5.

9. Implement a system call, *renice x y*, where *x* is a process ID (of an active process) and *y* is the value that its *nice* value should take.

10. Reconsider the example in Figure 8.6 for the fair share scheduler. Suppose the group containing process A pays for 33% of the CPU and the group containing processes B and C pays for 66% of the CPU time. What should the sequence of scheduled processes look like? Generalize the computation of process priorities so that it normalizes the value of the group CPU usage field.

11. Implement the command *date*: with no arguments, the command prints the system's opinion of the current date; using a parameter. as in

```
date mmddhhmmyy
```

a (super) user can set the system's opinion of the current date to the corresponding month, day, year, hour, and minute. For example,

```
date 0911205084
```

sets the system date to September 11, 1984, 8:50 p.m.

12. Programs can use a user-level *sleep* function

```
sleep(seconds);
```

to suspend execution for the indicated number of seconds. Implement the function using the *alarm* and *pause* system calls. What should happen if the process had called *alarm* before calling *sleep*? Consider two possibilities: that the previous *alarm* call would expire while the process was sleeping, and that it would expire after the *sleep* completed.

* 13. Refering to the last problem, the kernel could do a context switch between the *alarm* and *pause* calls in the *sleep* function, and the process could receive the *alarm* signal before it calls *pause*. What would happen? How can this race condition be fixed?

9

MEMORY MANAGEMENT POLICIES

The CPU scheduling algorithm described in the last chapter is strongly influenced by memory management policies. At least part of a process must be contained in primary memory to run; the CPU cannot execute a process that exists entirely in secondary memory. However, primary memory is a precious resource that frequently cannot contain all active processes in the system. For instance, if a system contains 8 megabytes of primary memory, nine 1-megabyte processes will not fit there simultaneously. The memory management subsystem decides which processes should reside (at least partially) in main memory, and manages the parts of the virtual address space of a process that are not core resident. It monitors the amount of available primary memory and may periodically write processes to a secondary memory device called the *swap device* to provide more space in primary memory. At a later time, the kernel reads the data from the swap device back to main memory.

Historically, UNIX systems transferred entire processes between primary memory and the swap device, but did not transfer parts of a process independently, except for shared text. Such a memory management policy is called *swapping*. It made sense to implement such a policy on the PDP 11, where the maximum process size was 64K bytes. For this policy, the size of a process is bounded by the amount of physical memory available on a system. The BSD system (release 4.0) was the first major implementation of a *demand paging* policy, transferring memory pages instead of processes to and from a secondary device; recent releases

of UNIX System V also support demand paging. The entire process does not have to reside in main memory to execute, and the kernel loads pages for a process on demand when the process references the pages. The advantage of a demand paging policy is that it permits greater flexibility in mapping the virtual address space of a process into the physical memory of a machine, usually allowing the size of a process to be greater than the amount of available physical memory and allowing more processes to fit simultaneously in main memory. The advantage of a swapping policy is that it is easier to implement and results in less system overhead. This chapter discusses the two memory management policies, swapping and paging.

9.1 SWAPPING

There are three parts to the description of the swapping algorithm: managing space on the swap device, swapping processes out of main memory, and swapping processes into main memory.

9.1.1 Allocation of Swap Space

The swap device is a block device in a configurable section of a disk. Whereas the kernel allocates space for files one block at a time, it allocates space on the swap device in groups of contiguous blocks. Space allocated for files is used statically; since it will exist for a long time, the allocation scheme is flexible to reduce the amount of fragmentation and, hence, unallocatable space in the file system. But the allocation of space on the swap device is transitory, depending on the pattern of process scheduling. A process that resides on the swap device will eventually migrate back to main memory, freeing the space it had occupied on the swap device. Since speed is critical and the system can do I/O faster in one multiblock operation than in several single block operations, the kernel allocates contiguous space on the swap device without regard for fragmentation.

Because the allocation scheme for the swap device differs from the allocation scheme for file systems, the data structures that catalog free space differ too. The kernel maintains free space for file systems in a linked list of free blocks, accessible from the file system super block, but it maintains the free space for the swap device in an in-core table, called a *map*. Maps, used for other resources besides the swap device (some device drivers, for example), allow a first-fit allocation of contiguous "blocks" of a resource.

A map is an array where each entry consists of an address of an allocatable resource and the number of resource units available there; the kernel interprets the address and units according to the type of map. Initially, a map contains one entry that indicates the address and the total number of resources. For instance, the kernel treats each unit of the swap map as a group of disk blocks, and it treats the address as a block offset from the beginning of the swap area. Figure 9.1 illustrates an initial swap map that consists of 10,000 blocks starting at address 1.

Address	Units
1	10000

Figure 9.1. Initial Swap Map

```
algorithm malloc        /* algorithm to allocate map space */
input:  (1) map address         /* indicates which map to use */
        (2) requested number of units
output: address, if successful
        0, otherwise
{
      for (every map entry)
      {
            if (current map entry can fit requested units)
            {
                  if (requested units == number of units in entry)
                        delete entry from map;
                  else
                        adjust start address of entry;
                  return (original address of entry);
            }
      }
      return(0);
}
```

Figure 9.2. Algorithm for Allocating Space from Maps

As the kernel allocates and frees resources, it updates the map so that it continues to contain accurate information about free resources.

Figure 9.2 gives the algorithm *malloc* for allocating space from maps. The kernel searches the map for the first entry that contains enough space to accommodate the request. If the request consumes all the resources of the map entry, the kernel removes the entry from the array and compresses the map (that is, the map has one fewer entries). Otherwise, it adjusts the address and unit fields of the entry according to the amount of resources allocated. Figure 9.3 shows the sequence of swap map configurations after allocating 100 units, 50 units, then 100 units again. The kernel adjusts the swap map to show that the first 250 units have been allocated, and that it now contains 9750 free units starting at address 251.

When freeing resources, the kernel finds their proper position in the map by address. Three cases are possible:

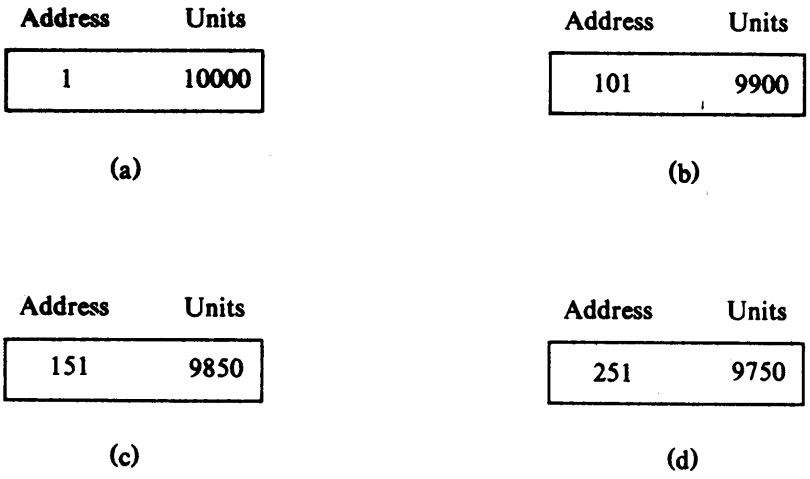


Figure 9.3. Allocating Swap Space

1. The freed resources completely fill a hole in the map: they are contiguous to the entries whose addresses would immediately precede them and follow them in the map. In this case, the kernel combines the newly freed resources and the existing (two) entries into one entry in the map.
2. The freed resources partially fill a hole in the map. If the address of the freed resources are contiguous with the map entry that would immediately precede them or with the entry that would immediately follow them (but not both), the kernel adjusts the address and units fields of the appropriate entry to account for the resources just freed. The number of entries in the map remains the same.
3. The freed resources partially fill a hole but are not contiguous to any resources in the map. The kernel creates a new entry for the map and inserts it in the proper position.

Returning to the previous example, if the kernel frees 50 units of the swap resource starting at address 101, the swap map contains a new entry for the freed resources, since the returned resources are not contiguous to existing entries in the map. If the kernel then frees 100 units of the swap resource starting at address 1, it adjusts the first entry of the swap map since the freed resources are contiguous to those in the first entry. Figure 9.4 shows the sequence of swap map configurations corresponding to these events.

Suppose the kernel now requests 200 units of swap space. Because the first entry in the swap map only contains 150 units, the kernel satisfies the request from the second entry (see Figure 9.5). Finally, suppose the kernel frees 350 units of

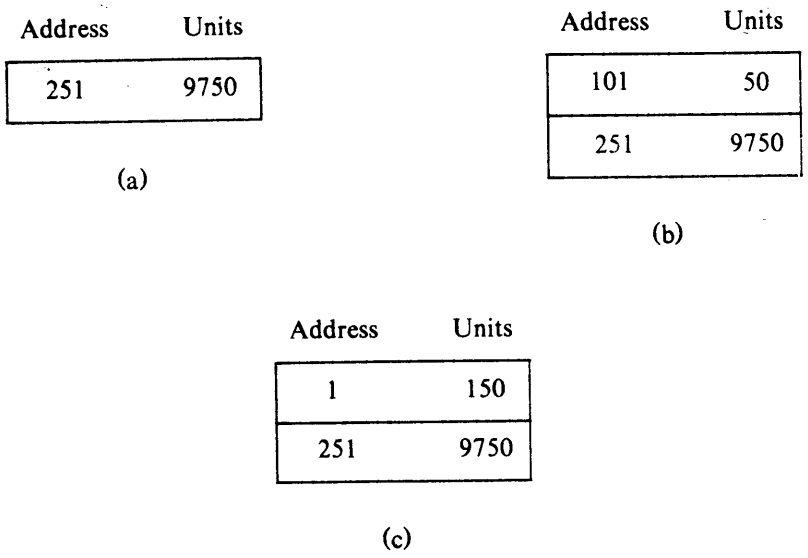


Figure 9.4. Freeing Swap Space

Figure 9.5. Allocating Swap Space from the Second Entry in the Map

swap space starting at address 151. Although the 350 units were allocated separately, there is no reason the kernel could not free them at once. (It does not do so for swap space, since requests for swap space are independent of each other.) The kernel realizes that the freed resources fit neatly into the hole between the first and second entries in the swap map and creates one entry for the former two (and the freed resources).

Traditional implementations of the UNIX system use one swap device, but the latest implementations of System V allow multiple swap devices. The kernel

chooses the swap device in a round robin scheme, provided it contains enough contiguous memory. Administrators can create and remove swap devices dynamically. If a swap device is being removed, the kernel does not swap data to it, as data is swapped from it, it ennuics out until it is free and can be removed.

9.1.2 Swapping Processes Out

The kernel swaps a process out if it needs space in memory, which may result from any of the following.

1. The *fork* system call must allocate space for a child process,
2. The *brk* system call increases the size of a process,
3. A process becomes larger by the natural growth of its stack,
4. The kernel wants to free space in memory for processes it had previously swapped out and should now swap in.

The case of *fork* stands out, because it is the only case where the in-core memory previously occupied by the process is *not* relinquished.

When the kernel decides that a process is eligible for swapping from main memory, it decrements the reference count of each region in the process and swaps the region out if its reference count drops to 0. The kernel allocates space on a swap device and locks the process in memory (for cases 1–3), preventing the swapper from swapping it out (see exercise 9.12) while the current swap operation is in progress. The kernel saves the swap address of the region in the region table entry.

The kernel swaps as much data as possible per I/O operation directly between the swap device and user address space, bypassing the buffer cache. If the hardware cannot transfer multiple pages in one operation, the kernel software must iteratively transfer one page of memory at a time. The exact rate of data transfer and its mechanics therefore depend on the capabilities of the disk controller and the implementation of memory management, among other factors. For instance, if memory is organized in pages, the data to be swapped out is likely to be discontiguous in physical memory. The kernel must gather the page addresses of data to be swapped out, and the disk driver may use the collection of page addresses to set up the I/O. The swapper waits for each I/O operation to complete before swapping out other data.

It is not necessary that the kernel write the entire virtual address space of a process to a swap device. Instead, it copies the physical memory assigned to a process to the allocated space on the swap device, ignoring unassigned virtual addresses. When the kernel swaps the process back into memory, it knows the virtual address map of the process, so it can reassign the process to the correct virtual addresses. The kernel eliminates an extra copy from a data buffer to physical memory by reading the data into the physical memory locations that were previously set up to conform to the virtual address locations.

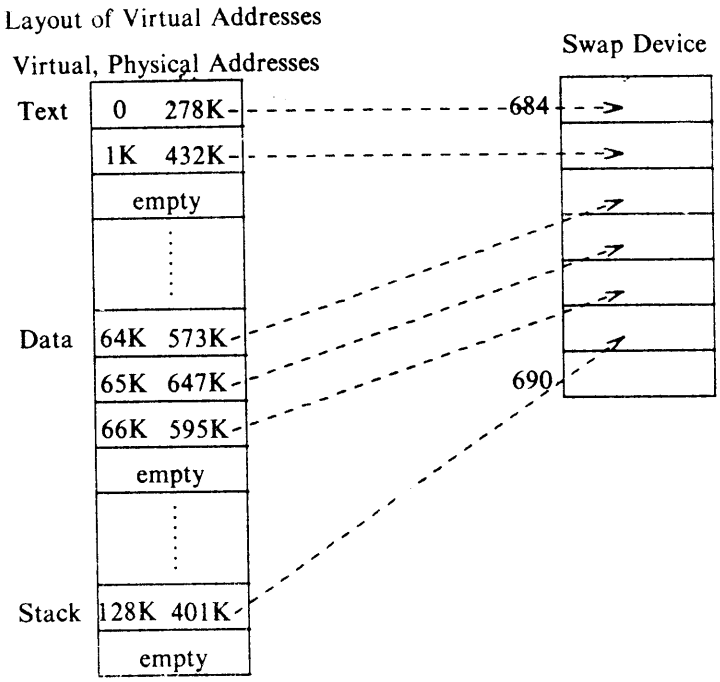


Figure 9.6. Mapping Process Space onto the Swap Device

Figure 9.6 gives an example of mapping the in-core image of a process onto a swap device.[1] The process contains three regions for text, data, and stack: the text region ends at virtual address 2K, and the data region starts at virtual address 64K, leaving a gap of 62K bytes in the virtual address space. When the kernel swaps the process out, it swaps the pages for virtual addresses 0, 1K, 64K, 65K, 66K, and 128K; it does not allocate swap space for the empty 62K bytes between the text and data regions or the empty 61K bytes between the data and stack regions but fills the swap space contiguously. When the kernel swaps the process back in, it knows that the process has a 62K-byte hole by consulting the process memory map, and it assigns physical memory accordingly. Figure 9.7 demonstrates this case. Comparison of Figures 9.6 and 9.7 shows that the physical addresses occupied by

1. For simplicity, the virtual address space of a process is depicted as a linear array of page table entries in this and in later figures, disregarding the fact that each region usually has a separate page table.

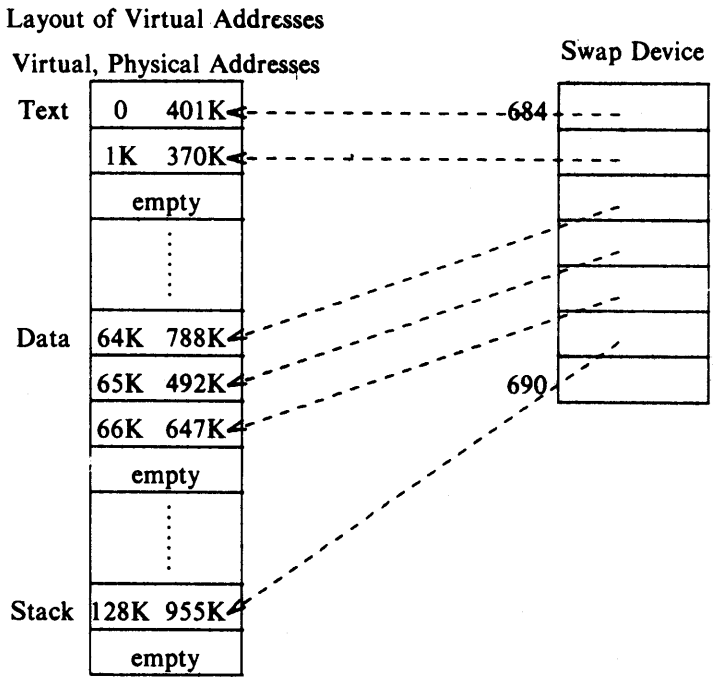


Figure 9.7. Swapping a Process into Memory

the process before and after the swap are not the same; however, the process does not notice a change at user-level, because the contents of its virtual space are the same.

Theoretically, all memory space occupied by a process, including its *u area* and kernel stack, is eligible to be swapped out, although the kernel may temporarily lock a region into memory while a sensitive operation is underway. Practically, however, kernel implementations do not swap the *u area* if the *u area* contains the address translation tables for the process. The implementation also dictates whether a process can swap itself out or whether it must request another process to swap it out (see exercise 9.4).

9.1.2.1 Fork Swap

The description of the *fork* system call (Section 7.1) assumed that the parent process found enough memory to create the child context. Otherwise, the kernel swaps the process out without freeing the memory occupied by the in-core (parent) copy. When the swap is complete, the child process exists on the swap device; the

parent places the child in the "ready-to-run" state (see Figure 6.1) and returns to user mode. Since the child is in the "ready-to-run" state, the swapper will eventually swap it into memory, where the kernel will schedule it; the child will complete its part of the *fork* system call and return to user mode.

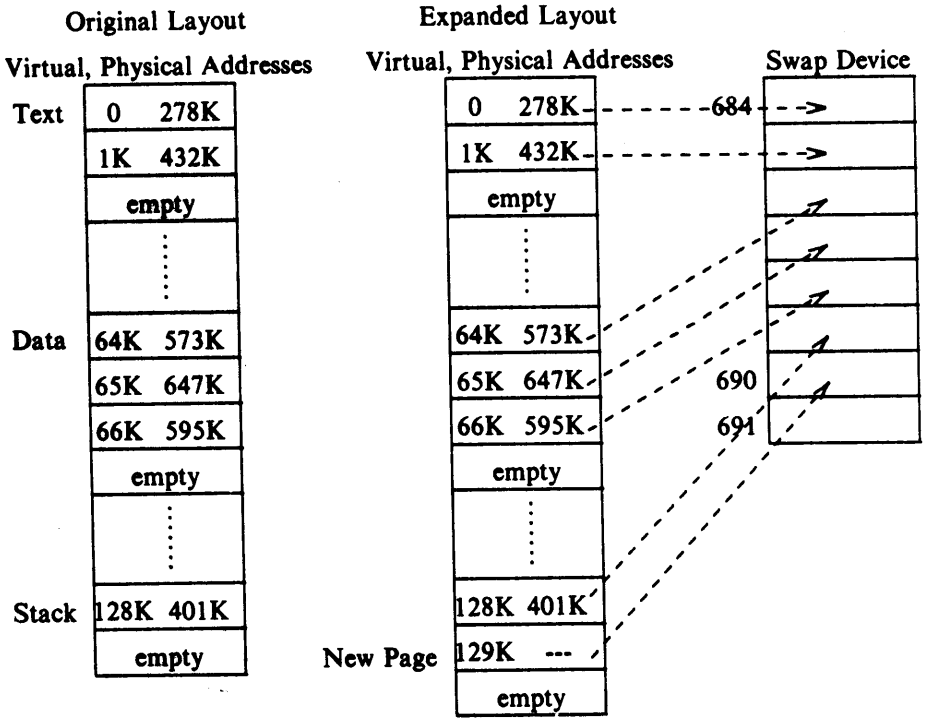


Figure 9.8. Adjusting Memory Map for Expansion Swap

9.1.2.2 Expansion Swap

If a process requires more physical memory than is currently allocated to it, either as a result of user stack growth or invocation of the *brk* system call and if it needs more memory than is currently available, the kernel does an *expansion swap* of the process. It reserves enough space on the swap device to contain the memory space of the process, including the newly requested space. Then, it adjusts the address translation mapping of the process to account for the new virtual memory but does not assign physical memory (since none was available). Finally, it swaps the process out in a normal swapping operation, zeroing out the newly allocated space

on the swap device (see Figure 9.8). When the kernel later swaps the process into memory, it will allocate physical memory according to the new (augmented size) address translation map. When the process resumes execution, it will have enough memory.

9.1.3 Swapping Processes In

Process 0, the swapper, is the only process that swaps processes into memory from swap devices. At the conclusion of system initialization, the swapper goes into an infinite loop, where its only task is to do process swapping, as mentioned in Section 7.9. It attempts to swap processes in from the swap device, and it swaps processes out if it needs space in main memory. The swapper sleeps if there is no work for it to do (for example, if there are no processes to swap in) or if it is unable to do any work (there are no processes eligible to swap out); the kernel periodically wakes it up, as will be seen. The kernel schedules the swapper to execute just as it schedules other processes, albeit at higher priority, but the swapper executes only in kernel mode. The swapper makes no system calls but uses internal kernel functions to do swapping; it is the archetype of all kernel processes.

As mentioned briefly in Chapter 8, the clock handler measures the time that each process has been in core or swapped out. When the swapper wakes up to swap processes in, it examines all processes that are in the state "ready to run but swapped out" and selects one that has been swapped out the longest (see Figure 9.9). If there is enough free memory available, the swapper swaps the process in, reversing the operation done for swapping out: It allocates physical memory, reads the process from the swap device, and frees the swap space.

If the swapper successfully swaps in a process, it searches the set of "ready-to-run but swapped out" processes for others to swap in and repeats the above procedure. One of the following situations eventually arises:

- No "ready-to-run" processes exist on the swap device: The swapper goes to sleep until a process on the swap device wakes up or until the kernel swaps out a process that is "ready to run." (Recall the state diagram in Figure 6.1.)
- The swapper finds an eligible process to swap in but the system does not contain enough memory: The swapper attempts to swap another process out and, if successful, restarts the swapping algorithm, searching for a process to swap in.

If the swapper must swap a process out, it examines every process in memory: Zombie processes do not get swapped out, because they do not take up any physical memory; processes locked in memory, doing region operations, for example, are also not swapped out. The kernel swaps out sleeping processes rather than those "ready to run," because "ready-to-run" processes have a greater chance of being scheduled soon. The choice of which sleeping process to swap out is a function of the process priority and the time the process has been in memory. If there are no sleeping processes in memory, the choice of which "ready-to-run" process to swap out is a function of the process *nice* value and the time the process has been in memory.

```
algorithm swapper       /* swap in swapped out processes,
                         * swap out other processes to make room */
input:  none
output: none
{
    loop:
        for (all swapped out processes that are ready to run)
                pick process swapped out longest;
        if (no such process)
        {
                sleep (event must swap in);
                goto loop;
        }
        if (enough room in main memory for process)
        {
                swap process in;
                goto loop;
        }
    /* loop2: here in revised algorithm (see page 285) */
        for (all processes loaded in main memory, not zombie and not locked in memory)
        {
                if (there is a sleeping process)
                        choose process such that priority + residence time
                                is numerically highest;
                else  /* no sleeping processes */
                        choose process such that residence time + nice
                                is numerically highest;
        }
        if (chosen process not sleeping or residency requirements not
                                satisfied)
                sleep (event must swap process in);
        else
                swap out process;
        goto loop;      /* goto loop2 in revised algorithm */
}
```

Figure 8.8. Algorithm for the Swapper

A "ready-to-run" process must be core resident for at least 2 seconds before being swapped out, and a process to be swapped in must have been swapped out for at least 2 seconds. If the swapper cannot find any processes to swap out or if neither the process to be swapped in nor the process to be swapped out have accumulated more than 2 seconds[2] residence time in their environment, then the

swapper sleeps on the event that it wants to swap a process into memory but cannot find room for it. The clock will awaken the swapper once a second in that state. The kernel also awakens the swapper if another process goes to sleep, since it may be more eligible for swapping out than the processes previously considered by the swapper. If the swapper swaps out a process or if it sleeps because it could not swap out a process, it will resume execution at the beginning of the swapping algorithm, attempting to swap in eligible processes.

Figure 9.10 depicts five processes and the time they spend in memory or on the swap device as they go through a sequence of swapping operations. For simplicity, assume that all processes are CPU intensive and that they do not make any system calls; hence, a context switch happens only as a result of clock interrupts at 1-second intervals. The swapper runs at highest scheduling priority, so it always runs briefly at 1-second intervals if it has work to do. Further, assume that the processes are the same size and the system can contain at most two processes simultaneously in main memory. Initially, processes A and B are in main memory and the other processes are swapped out. The swapper cannot swap any processes during the first 2 seconds, because none have been in memory or on the swap device for 2 seconds (the residency requirement), but at the 2-second mark, it swaps out processes A and B and swaps in processes C and D. It attempts to swap in process E, too, but fails because there is no more room in main memory. At the 3 second mark, process E is eligible for swapping because it has been on the swap device for 3 seconds, but the swapper cannot swap processes out of main memory because their residency time is under 2 seconds. At the 4-second mark, the swapper swaps out processes C and D and swaps in processes E and A.

The swapper chooses processes to swap in based on the amount of time the processes had been swapped out. Another criterion could have been to swap in the highest-priority process that is ready to run, since such processes deserve a better chance to execute. It has been demonstrated that such a policy results in "slightly" better throughput under heavy system load (see [Peachey 84]).

The algorithm for choosing a process to swap out to make room in memory has more serious flaws, however. First, the swapper swaps out a process based on its priority, memory-residence time, and *nice* value. Although it swaps out a process only to make room for a process being swapped in, it may swap out a process that does not provide enough memory for the incoming process. For instance, if the swapper attempts to swap in a process that occupies 1 megabyte of memory and the system contains no free memory, it is futile to swap out a process that occupies only 2K bytes of memory. An alternative strategy would be to swap out groups of

2. The Version 6 implementation of the UNIX system did not swap a process out to make room for an incoming process until the incoming process had been disk resident for 3 seconds. The outgoing process had to reside in memory at least 2 seconds. The choice of the time interval cuts down on thrashing and increases system throughput.

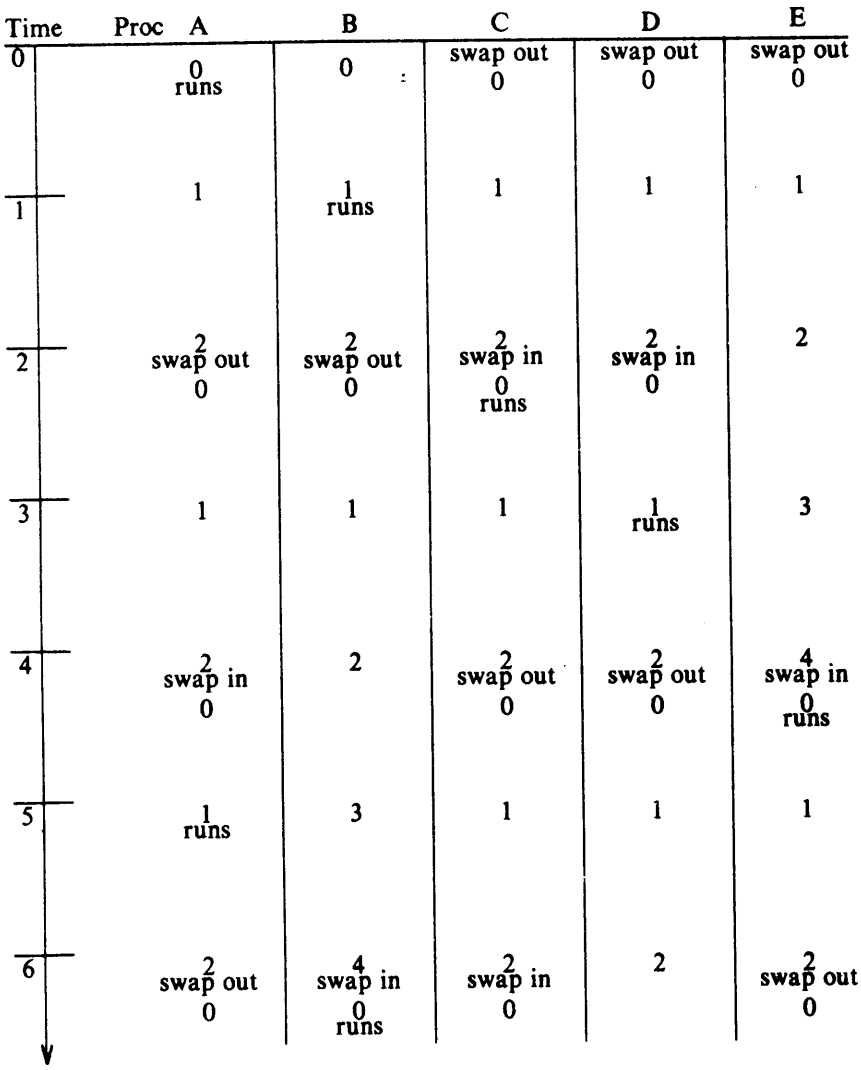


Figure 9.10. Sequence of Swapping Operations

Time	Proc A	B	C	D	E
0	0 runs	0	swap out 0	nice 25 swap out 0	swap out 0
1	1	1 runs	1	1	1
2	2 swap out 0	2 swap out 0	2 swap in 0 runs	2 swap in 0	2
3	1	1	1	1 swap out 0	3 swap in 0 runs
4	2 swap in 0 runs	2	2 swap out 0	1	1
5	1	3 swap in 0 runs	1	2	2 swap out 0
6	2 swap out 0	1	2	3 swap in 0 runs	1

Figure 9.11. Thrashing due to Swapping

processes only if they provide enough memory for the incoming process. Experiments using a PDP 11/23 computer have shown that such a strategy can increase system throughput by about 10 percent under heavy loads (see [Peachey 84]).

Second, if the swapper sleeps because it could not find enough memory to swap in a process, it searches again for a process to swap in although it had previously chosen one. The reason is that other swapped processes may have awakened in the meantime and they may be more eligible for swapping in than the previously chosen process. But that is small solace to the original process still trying to be swapped in. In some implementations, the swapper tries to swap out many smaller processes to make room for the big process to be swapped in before searching for another process to swap in; this is the revision in the swapper algorithm shown by the comments in Figure 9.9.

Third, if the swapper chooses a "ready-to-run" process to swap out, it is possible that the process had not executed since it was previously swapped in. Figure 9.11 depicts such a case, where the kernel swaps in process D at the 2-second mark, schedules process C, and then swaps out process D at the 3-second mark in favor of process E (because of the interaction of the *nice* value) even though process D had never run. Such thrashing is clearly undesirable.

One final danger is worthy of mention. If the swapper attempts to swap out a process but cannot find space on the swap device, a system deadlock could arise if the following four conditions are met: All processes in main memory are asleep, all "ready-to-run" processes are swapped out, there is no room on the swap device for new processes, and there is no room in main memory for incoming processes. Exercise 9.5 explores this situation. Interest in fixing problems with the swapper has declined in recent years as demand paging algorithms have been implemented for UNIX systems.

9.2 DEMAND PAGING

Machines whose memory architecture is based on pages and whose CPU has restartable instructions[3] can support a kernel that implements a demand paging algorithm, swapping pages of memory between main memory and a swap device. Demand paging systems free processes from size limitations otherwise imposed by the amount of physical memory available on a machine. For instance, machines that contain 1 or 2 megabytes of physical memory can execute processes whose sizes are 4 or 5 megabytes. The kernel still imposes a limit on the virtual size of a process, dependent on the amount of virtual memory the machine can address. Since a process may not fit into physical memory, the kernel must load its relevant portions into memory dynamically and execute it even though other parts are not loaded. Demand paging is transparent to user programs except for the virtual size

3. If a machine executes "part" of an instruction and incurs a page fault, the CPU must restart the instruction after handling the fault, because intermediate computations done before the page fault may have been lost.

permissible to a process

Processes tend to execute instructions in small portions of their text space, such as program loops and frequently called subroutines, and their data references tend to cluster in small subsets of the total data space of the process. This is known as the principle of "locality." Denning [Denning 68] formalized the notion of the *working set* of a process, which is the set of pages that the process has referenced in its last *n* memory references; the number *n* is called the *window* of the working set. Because the working set is a fraction of the entire process, more processes may fit simultaneously into main memory than in a swapping system, potentially increasing system throughput because of reduced swapping traffic. When a process addresses a page that is not in its working set, it incurs a page fault; in handling the fault, the kernel updates the working set, reading in pages from a secondary device if necessary.

Figure 9.12 shows a sequence of page references a process could make, depicting the working sets for various window sizes and following a least recently used replacement policy. As a process executes, its working set changes, depending on the pattern of memory references the process makes; a larger window size yields a larger working set, implying that a process will not fault as often. It is impractical to implement a pure working set model, because it is expensive to remember the order of page references. Instead, systems approximate a working set model by setting a *reference* bit whenever a process accesses a page and by sampling memory references periodically: If a page was recently referenced, it is part of a working set; otherwise, it "ages" in memory until it is eligible for swapping.

When a process accesses a page that is not part of its working set, it incurs a *validity page fault*. The kernel suspends execution of the process until it reads the page into memory and makes it accessible to the process. When the page is loaded in memory, the process restarts the instruction it was executing when it incurred the fault. Thus, the implementation of a paging subsystem has two parts: swapping rarely used pages to a swapping device and handling page faults. This general description of paging schemes extends to non-UNIX systems, too. The rest of this chapter examines the paging scheme for UNIX System V in detail.

9.2.1 Data Structures for Demand Paging

The kernel contains 4 major data structures to support low-level memory management functions and demand paging: page table entries, *disk block descriptors*, the *page frame data table* (called *pfdata* for short), and the swap-use table. The kernel allocates space for the pfdata table once for the lifetime of the system but allocates memory pages for the other structures dynamically.

Recall from Chapter 6 that a region contains page tables to access physical memory. Each entry of a page table (Figure 9.13) contains the physical address of the page, protection bits indicating whether processes can read, write or execute from the page, and the following bit fields to support demand paging:

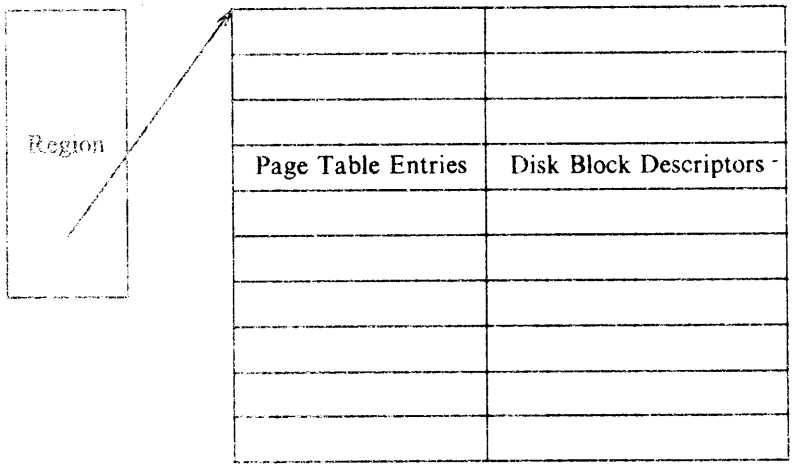
Sequence of Page References	Working Sets 2	3	Window Sizes 4	5
24	24	24	24	24
15	15 24	15 24	15 24	15 24
18	18 15	18 15 24	18 15 24	18 15 24
23	23 18	23 18 15	23 18 15 24	23 18 15 24
24	24 23	24 23 18	⋮	⋮
17	17 24	17 24 23	17 24 23 18	17 24 23 18 15
18	18 17	18 17 24	⋮	⋮
24	24 18	⋮	⋮	⋮
18	18 24	⋮	⋮	⋮
17	17 18	⋮	⋮	⋮
17	17	⋮	⋮	⋮
15	15 17	15 17 18	15 17 18 24	⋮
24	24 15	24 15 17	⋮	⋮
17	17 24	⋮	⋮	⋮
24	24 17	⋮	⋮	⋮
18	18 24	18 24 17	⋮	⋮

Figure 9.12. Working Set of a Process

- Valid
- Reference
- Modify
- Copy on write
- Age

The kernel turns on the *valid* bit to indicate that the contents of a page are legal, but the page reference is not necessarily illegal if the *valid* bit is off, as will be seen. The *reference* bit indicates whether a process recently referenced a page, and the *modify* bit indicates whether a process recently modified the contents of a page. The *copy on write* bit, used in the *fork* system call, indicates that the kernel must create a new copy of the page when a process modifies its contents. Finally, the kernel manipulates the *age* bits to indicate how long a page has been a member of the working set of a process. Assume the kernel manipulates the valid copy on

write, and age bits, and the hardware sets the reference and modify bits of the page table entry; Section 9.2.4 will consider hardware that does not have these capabilities.



Page Table Entry

Page (Physical) Address	Age	Cp/Wrt	Mod	Ref	Val	Prot

Disk Block Descriptor

Swap Dev	Block Num	Type (swap, file, fill 0, demand fill)

Figure 9.13. Page Table Entries and Disk Block Descriptors

Each page table entry is associated with a disk block descriptor, which describes the disk copy of the virtual page (Figure 9.13). Processes that share a region therefore access common page table entries and disk block descriptors. The contents of a virtual page are either in a particular block on a swap device, in an executable file, or not on a swap device. If the page is on a swap device, the disk block descriptor contains the logical device number and block number containing the page contents. If the page is contained in an executable file, the disk block descriptor contains the logical block number in the file that contains the page; the kernel can quickly map this number into its disk address. The disk block descriptor also indicates two special conditions set during *exec*: that a page is "demand fill"

or "demand zero." Section 9.2.1.2 will explain these conditions.

The pfdata table describes each page of *physical* memory and is indexed by page number. The fields of an entry are

- The page state, indicating that the page is on a swap device or executable file, that DMA is currently underway for the page (reading data from a swap device), or that the page can be reassigned.
- The number of processes that reference the page. The reference count equals the number of valid page table entries that reference the page. It may differ from the number of processes that share regions containing the page, as will be described below when reconsidering the algorithm for *fork*.
- The logical device (swap or file system) and block number that contains a copy of the page.
- Pointers to other pfdata table entries on a list of free pages and on a hash queue of pages.

The kernel links entries of the pfdata table onto a free list and a hashed list, analogous to the linked lists of the buffer cache. The free list is a cache of pages that are available for reassignment, but a process may fault on an address and still find the corresponding page intact on the free-list. The free list thus allows the kernel to avoid unnecessary read operations from the swap device. The kernel allocates new pages from the list in least recently used order. The kernel also hashes the pfdata table entry according to its (swap) device number and block number. Thus, given a device and block number, the kernel can quickly locate a page if it is in memory. To assign a physical page to a region, the kernel removes a free page frame entry from the head of the free list, updates its swap device and block numbers, and puts it onto the correct hash queue.

The swap-use table contains an entry for every page on a swap device. The entry consists of a reference count of how many page table entries point to a page on a swap device.

Figure 9.14 shows the relationship between page table entries, disk block descriptors, pfdata table entries, and the swap-use count table. Virtual address 1493K of a process maps into a page table entry that points to physical page 794; the disk block descriptor for the page table entry shows that a copy of the page exists at disk block 2743 on swap device 1. The pfdata table entry for physical page 794 also shows that a copy of the page exists at disk block 2743 on swap device 1, and its in-core reference count is 1. Section 9.2.4.1 will explain why the disk block number is duplicated in the pfdata table and the disk block descriptor. The swap use count for the virtual page is 1, meaning that one page table entry points to the swap copy.

9.2.1.1 Fork in a Paging System

As explained in Section 7.1, the kernel duplicates every region of the parent process during the *fork* system call and attaches it to the child process. Traditionally, the

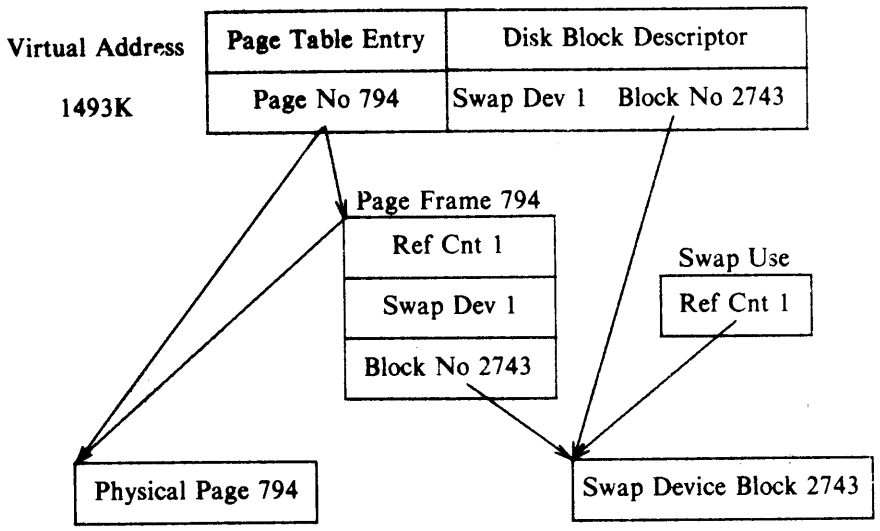


Figure 9.14. Relationship of Data Structures for Demand Paging

kernel of a swapping system makes a physical copy of the parent's address space, usually a wasteful operation, because processes often call *exec* soon after the *fork* call and immediately free the memory just copied. On the System V paging system, the kernel avoids copying the page by manipulating the region tables, page table entries, and pfdata table entries: It simply increments the region reference count of shared regions. For private regions such as data and stack, however, it allocates a new region table entry and page table and then examines each parent page table entry: If a page is valid, it increments the reference count in the pfdata table entry, indicating the number of processes that share the page via *different* regions (as opposed to the number that share the page by sharing the region). If the page exists on a swap device, it increments the swap-use table reference count for the page.

The page can now be referenced through both regions, which share the page until a process writes to it. The kernel then copies the page so that each region has a private version. To do this, the kernel turns on the "copy on write" bit for every page table entry in private regions of the parent and child processes during *fork*. If either process writes the page, it incurs a protection fault, and in handling the fault, the kernel makes a new copy of the page for the faulting process. The physical copying of the page is thus deferred until a process really needs it.

Figure 9.15 shows the data structures when a process *forks*. The processes share access to the page table of the shared text region T, so the region reference count is 2 and the pfdata reference count for pages in the text region is 1. The

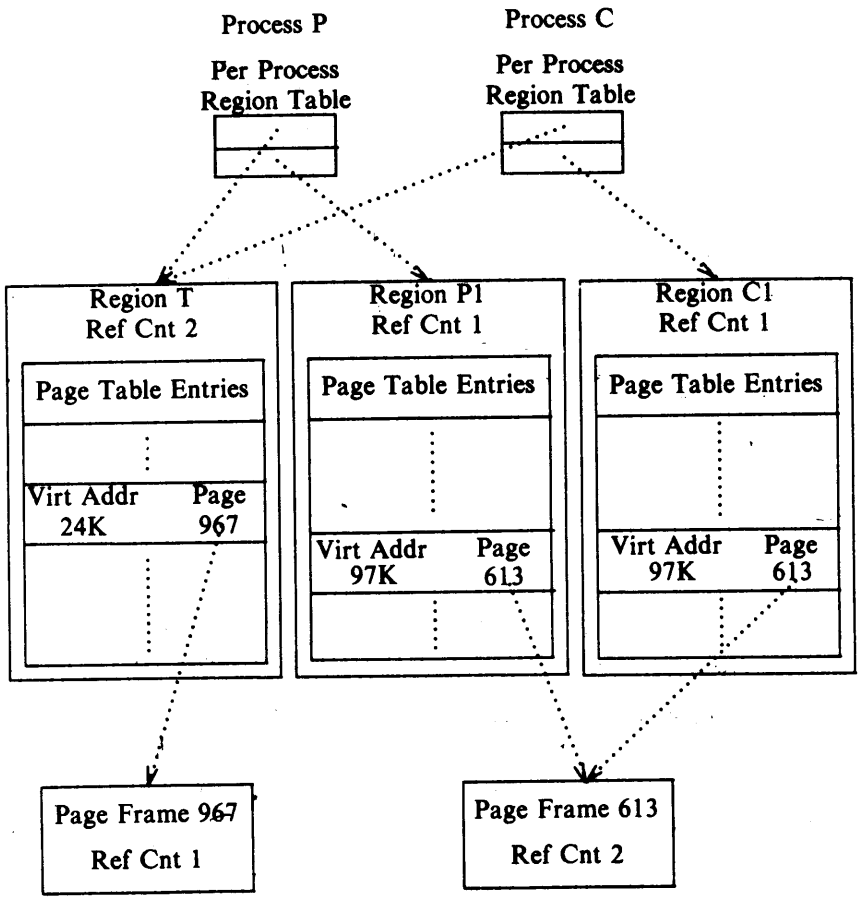


Figure 9.15. A Page in a Process that Forks

kernel allocates a new child data region, *C1*, a copy of region *P1* in the parent process. The page table entries of the two regions are identical, as illustrated by the entry for virtual address 97K. The page table entries point to pfdata table entry 613, whose reference count is 2, indicating that two regions reference the page.

The implementation of the *fork* system call in the BSD system makes a physical copy of the pages of the parent process. Recognizing the performance improvement gained by not having to do the copy, however, the BSD system also contains the *vfork* system call, which assumes that a child process will immediately invoke *exec* on return from the *vfork* call. *Vfork* does not copy page tables so it is faster than the System V *fork* implementation. But the child process executes in the same

physical address space as the parent process (until an *exec* or *exit*) and can thus overwrite the parent's data and stack. A dangerous situation could arise if a programmer uses *vfork* incorrectly, so the onus for calling *vfork* lies with the programmer. The difference between the System V approach and the BSD approach is philosophical: Should the kernel hide idiosyncrasies of its implementation from users, or should it allow sophisticated users the opportunity to take advantage of the implementation to do a logical function more efficiently?

```
int global;
main()
{
        int local;

        local = 1;
        if (vfork() == 0)
        {
                /* child */
                global = 2;       /* write parent data space */
                local = 3;        /* write parent stack */
                _exit();
        }
        printf("global %d local %d\n", global, local);
}
```

Figure 9.16. Vfork and Corruption of Process Memory

For example, consider the program in Figure 9.16. After the *vfork* call, the child process does not *exec*, but resets the variables *global* and *local* and exits.[4] The system guarantees that the parent process is suspended until the child process *exec*s or *exit*s. When the parent process finally resumes execution, it finds that the values of the two variables are not the same as they were before the *vfork*! More spectacular effects can occur if the child process returns from the function that had called *vfork* (see exercise 9.8).

4. The call to *_exit* is used, because *exit* "cleans up" the standard I/O (user-level) data structures for the parent *and* child processes, preventing the parent's *printf* statement from working correctly — another unfortunate side effect of *vfork*.

9.2.1.2 Exec in a Paging System

When a process invokes the *exec* system call, the kernel reads the executable file into memory from the file system, as described in Chapter 7. On a demand paged system, however, the executable file may be too large to fit in the available main memory. The kernel, therefore, does not preassign memory to the executable file but "faults" it in, assigning memory as needed. It first assigns the page tables and disk block descriptors for the executable file, marking the page table entries "demand fill" (for non-bss data) or "demand zero" (for bss data). Following a variant of the *read* algorithm for reading the file into memory, the process incurs a validity fault as it reads each page. The fault handler notes whether the page is "demand fill," meaning its contents will immediately be overwritten with the contents of the executable file so it need not be cleared, or that it is "demand zero," meaning that its contents should be cleared. The description of the validity fault handler in Section 9.2.3 will show how this is done. If the process cannot fit into memory, the page-stealer process periodically swaps pages from memory, making room for the incoming file.

There are obvious inefficiencies in this scheme. First, a process incurs a page fault when reading each page of the executable file, even though it may never access the page. Second, the page stealer may swap pages from memory before the *exec* is done, resulting in two extra swap operations per page if the process needs the page early. To make *exec* more efficient, the kernel can demand page directly from the executable file if the data is properly aligned, as indicated by a special magic number. However, use of standard algorithms (such as *bmap*, in Chapter 4) to access a file would make it expensive to demand page from indirect blocks because of the multiple buffer cache accesses necessary to read a block. Furthermore, consistency problems could arise because *bmap* is not reentrant. The kernel sets various I/O parameters in the *u area* during the *read* system call. If a process incurs a page fault during a *read* system call when attempting to copy data to user space, it would overwrite these fields in the *u area* to read the page from the file system. Therefore, the kernel cannot use the regular algorithms to fault in pages from the file system. The algorithms are, of course, reentrant in regular cases, because each process has a separate *u area* and a process cannot simultaneously execute multiple system calls.

To page directly from an executable file, the kernel finds all the disk block numbers of the executable file when it does the *exec* and attaches the list to the file inode. When setting up the page tables for such an executable file, the kernel marks the disk block descriptor with the logical block number (starting from block 0 in the file) containing the page; the validity fault handler later uses this information to load the page from the file. Figure 9.17 shows a typical arrangement, where the disk block descriptor indicates that the page is at logical block offset 84 in the file. The kernel follows the pointer from the region to the inode and looks up the appropriate disk block number (279).

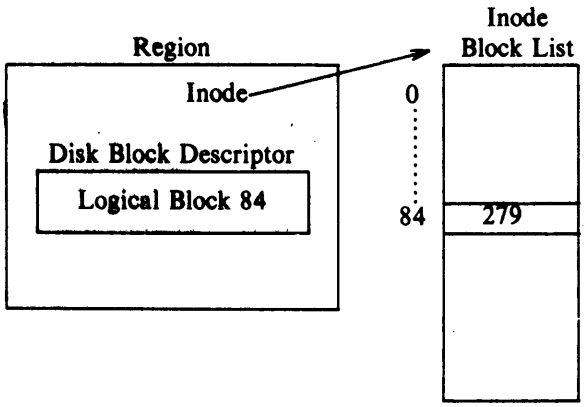


Figure 9.17. Mapping a File into a Region

9.2.2 The Page-Stealer Process

The page stealer is a kernel process that swaps out memory pages that are no longer part of the working set of a process. The kernel creates the page stealer during system initialization and invokes it throughout the lifetime of the system when low on free pages. It examines every active, unlocked region, skipping locked regions in the expectation of examining them during its next pass through the region list, and increments the age field of all valid pages. The kernel locks a region when a process faults on a page in the region, so that the page stealer cannot steal the page being faulted in.

There are two paging states for a page in memory: The page is aging and is not yet eligible for swapping, or the page is eligible for swapping and is available for reassignment to other virtual pages. The first state indicates that a process recently accessed the page, and the page is therefore in its working set. Some machines set a *reference* bit when they reference a page, but software methods can be substituted if the hardware does not have this feature (Section 9.2.4). The page stealer turns off the *reference* bit for such pages but remembers how many examinations have passed since the page was last referenced. The first state thus consists of several substates, corresponding to the number of passes the page stealer makes before the page is eligible for swapping (see Figure 9.18). When the number exceeds a threshold value, the kernel puts the page into the second state, ready to be swapped. The maximum period that a page can age before it is eligible to be swapped is implementation dependent, constrained by the number of bits available in the page table entry.

Figure 9.19 depicts the interaction between processes accessing a page and examinations by the page stealer. The page starts out in main memory, and the figure shows the number of examinations by the page stealer between memory

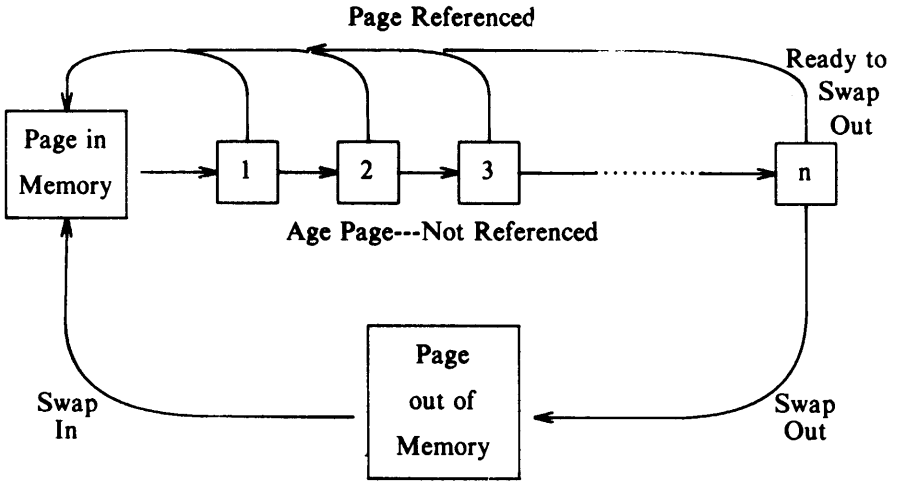


Figure 9.18. State Diagram for Page Aging

references. A process referenced the page after the second examination, dropping its age to 0. Similarly, a process referenced the page again after one more examination. Finally, the page stealer examined the page three times without an intervening reference and swapped the page out.

If two or more processes share a region, they update the *reference* bits of the same set of page table entries. Pages can thus be part of the working set of more than one process, but that does not matter to the page stealer. If a page is part of the working set of any process, it remains in memory; if it is not part of the working set of any process, it is eligible for swapping. It does not matter if one region has more pages in memory than others: the page stealer does not attempt to swap out equal numbers of pages from all active regions.

The kernel wakes up the page stealer when the available free memory in the system is below a low-water mark, and the page stealer swaps out pages until the available free memory in the system exceeds a high-water mark. The use of high- and low-water marks reduces thrashing: If the kernel were only to use one threshold, it would swap out enough pages to get above the threshold (of free pages), but as a result of faulting pages back into memory, the number would soon drop below the threshold. The page stealer would effectively thrash about the threshold. By swapping out pages until the number of free pages exceeds a high-water mark, it takes longer until the number of free pages drops below the low-water mark, so the page stealer does not run as often. Administrators can configure the values of the high- and low-water marks for best performance.

When the page stealer decides to swap out a page, it considers whether a copy of the page is on a swap device. There are three possibilities.

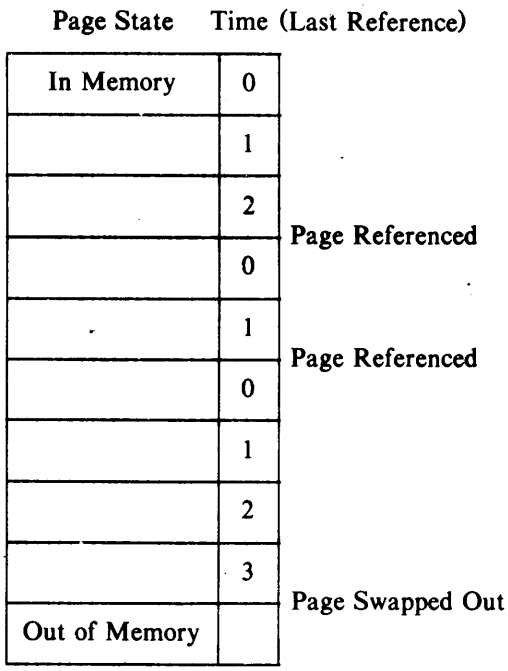


Figure 9.19. Example of Aging a Page

1. If no copy of the page is on a swap device, the kernel "schedules" the page for swapping: The page stealer places the page on a list of pages to be swapped out and continues; the swap is logically complete. When the list of pages to be swapped reaches a limit (dependent on the capabilities of the disk controller), the kernel writes the pages to the swap device.
2. If a copy of the page is already on a swap device and no process had modified its in-core contents (the page table entry *modify* bit is clear), the kernel clears the page table entry *valid* bit, decrements the reference count in the pfdata table entry, and puts the entry on the free list for future allocation.
3. If a copy of the page is on a swap device but a process had modified its contents in memory, the kernel schedules the page for swapping, as above, and frees the space it currently occupies on the swap device.

The page stealer copies the page to a swap device if case 1 or case 3 is true.

To illustrate the differences between the last two cases, suppose a page is on a swap device and is swapped into main memory after a process incurs a validity fault. Assume the kernel does not automatically remove the disk copy. Eventually, the page stealer decides to swap the page out again. If no process has written the

page since it was swapped in, the memory copy is identical to the disk copy and there is no need to write the page to the swap device. If a process has written the page, however, the memory copy differs from the disk copy, so the kernel must write the page to the swap device, after freeing the space on the swap device previously occupied by the page. It does not reuse the space on the swap device immediately, so that it can keep swap space contiguous for better performance.

The page stealer fills a list of pages to be swapped, possibly from different regions, and swaps them to a swap device when the list is full. Every page of a process need not be swapped: Some pages may not have aged sufficiently, for example. This differs from the policy of the swapping process, which swaps every page of a process from memory, but the method for writing data to the swap device is identical to that described in Section 9.1.2 for a swapping system. If no swap device contains enough contiguous space, the kernel swaps out one page at a time, which is clearly more costly. There is more fragmentation of a swap device in the paging scheme than in a swapping scheme, because the kernel swaps out blocks of pages but swaps in only one page at a time.

When the kernel writes a page to a swap device, it turns off the *valid* bit in its page table entry and decrements the use count of its pfdata table entry. If the count drops to 0, it places the pfdata table entry at the end of the free list, caching it until reassignment. If the count is not 0, several processes are sharing the page as a result of a previous *fork* call, but the kernel still swaps the page out. Finally, the kernel allocates swap space, saves the swap address in the disk block descriptor, and increments the swap-use table count for the page. If a process incurs a page fault while the page is on the free list, however, the kernel can rescue the page from memory instead of having to retrieve it from the swap device. However, the page is still swapped if it is on the swap list.

For example, suppose the page stealer swaps out 30, 40, 50 and 20 pages from processes A, B, C, and D, respectively, and that it writes 64 pages to the swap device in one disk write operation. Figure 9.20 shows the sequence of page-swapping operations that would occur if the page stealer examines pages of the processes in the order A, B, C, and D. The page stealer allocates space for 64 pages on the swap device and swaps out the 30 pages of process A and 34 pages of process B. It then allocates more space on the swap device for another 64 pages and swaps out the remaining 6 pages of process B, the 50 pages of process C, and 8 pages of process D. The two areas of the swap device for the two write operations need not be contiguous. The page stealer keeps the remaining 12 pages of process D on the list of pages to be swapped but does not swap them until the list is full. As processes fault in pages from the swap device or when the pages are no longer in use (processes *exit*), free space develops on the swap device.

To summarize, there are two phases to swapping a page from memory. First, the page stealer finds the page eligible for swapping and places the page number on a list of pages to be swapped. Second, the kernel copies the page to a swap device when convenient, turns off the *valid* bit in the page table entry, decrements the pfdata table entry reference count, and places the pfdata table entry at the end of

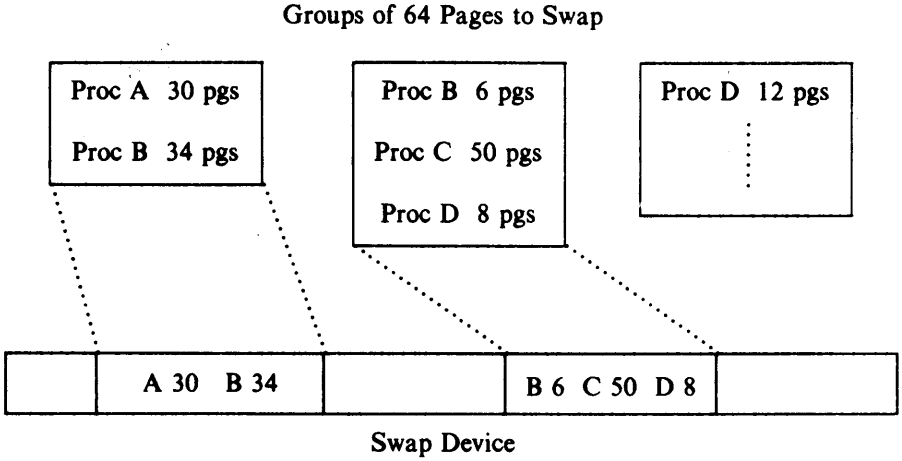


Figure 9.20. Allocation of Swap Space in Paging Scheme

the free list if its reference count is 0. The contents of the physical page in memory are valid until the page is reassigned.

9.2.3 Page Faults

The system can incur two types of page faults: validity faults and protection faults. Because the fault handlers may have to read a page from disk to memory and sleep during the I/O operation, fault handlers are an exception to the general rule that interrupt handlers cannot sleep. However, because the fault handler sleeps in the context of the process that caused the memory fault, the fault relates to the running process; hence, no arbitrary processes are put to sleep.

9.2.3.1 Validity Fault Handler

If a process attempts to access a page whose *valid* bit is not set, it incurs a validity fault and the kernel invokes the validity fault handler (Figure 9.21). The *valid* bit is not set for pages outside the virtual address space of a process, nor is it set for pages that are part of the virtual address space but do not currently have a physical page assigned to them. The hardware supplies the kernel with the virtual address that was accessed to cause the memory fault, and the kernel finds the page table entry and disk block descriptor for the page. The kernel locks the region containing the page table entry to prevent race conditions that would occur if the page stealer attempted to swap the page out. If the disk block descriptor has no record of the

```
algorithm vfault        /* handler for validity faults */
input:  address where process faulted
output: none
{
      find region, page table entry, disk block descriptor
                  corresponding to faulted address, lock region;
      if (address outside virtual address space)
      {
            send signal (SIGSEGV: segmentation violation) to process;
            goto out;
      }
      if (address now valid)        /* process may have slept above */
            goto out;
      if (page in cache)
      {
            remove page from cache;
            adjust page table entry;
            while (page contents not valid)  /* another proc faulted first */
                  sleep (event contents become valid);
      }
      else        /* page not in cache */
      {
            assign new page to region;

            put new page in cache, update pfdata entry;
            if (page not previously loaded and page "demand zero")
                  clear assigned page to 0;
            else
            {
                  read virtual page from swap dev or exec file;
                  sleep (event I/O done);
            }
            awaken processes (event page contents valid);
      }
      set page valid bit;
      clear page modify bit, page age;
      recalculate process priority;
out:  unlock region;
}
```

Figure 9.21. Algorithm for Validity Fault Handler

faulted page, the attempted memory reference is invalid and the kernel sends a "segmentation violation" signal to the offending process (recall Figure 7.25). This is the same procedure a swapping system follows when a process accesses an invalid address, except that it recognizes the error immediately because all legal pages are memory resident. If the memory reference was legal, the kernel allocates a page of memory to read in the page contents from the swap device or from the executable file.

The page that caused the fault is in one of five states:

1. On a swap device and not in memory,
2. On the free page list in memory,
3. In an executable file,
4. Marked "demand zero,"
5. Marked "demand fill."

Let us consider each case in detail.

If a page is on a swap device and not in memory (case 1), it once resided in main memory but the page stealer had swapped it out. From the disk block descriptor, the kernel finds the swap device and block number where the page is stored and verifies that the page is not in the page cache. The kernel updates the page table entry so that it points to the page about to be read in, places the pfdata table entry on a hash list to speed later operation of the fault handler, and reads the page from the swap device. The faulting process sleeps until the I/O completes, when the kernel awakens other processes who were waiting for the contents of the page to be read in.

For example, consider the page table entry for virtual address 66K in Figure 9.22. If a process incurs a validity fault when accessing the page, the fault handler examines the disk block descriptor and sees that the page is contained in block 847 of the swap device (assume there is only one swap device): Hence, the virtual address is legal. The fault handler then searches the page cache but fails to find an entry for disk block 847. Therefore, there is no copy of the virtual page in memory, and the fault handler must read it from the swap device. The kernel assigns page 1776 (Figure 9.23), reads the contents of the virtual page from the swap device into the new page, and updates the page table entry to refer to page 1776. Finally, it updates the disk block descriptor to indicate that the page is still swapped and the pfdata table entry for page 1776 to indicate that block 847 of the swap device contains a duplicate copy of the virtual page.

The kernel does not always have to do an I/O operation when it incurs a validity fault, even though the disk block descriptor indicates that the page is swapped (case 2). It is possible that the kernel had never reassigned the physical page after swapping it out, or that another process had faulted the virtual page into another physical page. In either case, the fault handler finds the page in the page cache, keying off the block number in the disk block descriptor. It reassigns the page table entry to point to the page just found, increments its page reference count, and removes the page from the free list, if necessary. For example, suppose