PROCESS SCHEDULING AND TIME:

On a time sharing system, the kernel allocates the CPU to a process for a period of time called a time slice or time quantum, preempts the process and schedules another one when the time slice expires, and reschedules the process to continue execution at a later time.

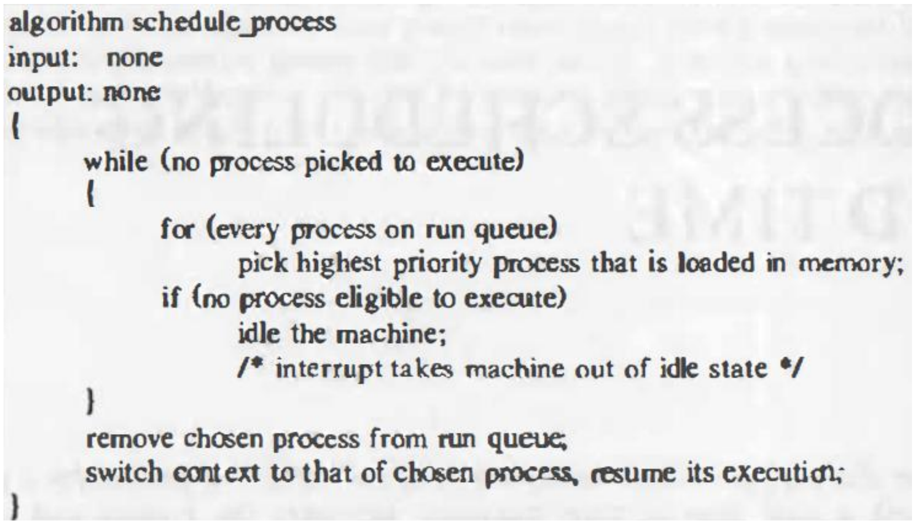
The scheduler function on the UNIX system uses relative time of execution as a parameter to determine which process to schedule next every active process has a scheduling priority; the kernel switches context to that of the process with the highest priority when it does a context switch.

The kernel recalculates the priority of the running process when it returns from kernel mode to user mode, and it periodically readjusts the priority of every "ready-to-run" process in user mode.

PROCESS SCHEDULING:

The scheduler on the UNIX system belongs to the general class of operating system schedulers known as round robin with multilevel feedback, meaning that the kernel allocates the CPU to a process for a time quantum, preempts a process that exceeds its time quantum, and feeds it back into one of several priority queues.

A process may need much iteration through the "feedback loop" before it finishes. When the kernel does a context switch and restores the context of a process, the process resumes execution from the point where it had been suspended.



```
algorithm schedule_process
input:  none
output: none
{
        while (no process picked to execute)
        {
                for (every process on run queue)
                        pick highest priority process that is loaded in memory;
                if (no process eligible to execute)
                        idle the machine;
                        /* interrupt takes machine out of idle state */
        }
        remove chosen process from run queue;
        switch context to that of chosen process, resume its execution;
}
```

Figure 7.1: Algorithm for Process Scheduling

Algorithm: At the conclusion of a context switch, the kernel executes the algorithm to schedule a process, selecting the highest priority process from those in the states "ready to run and loaded in memory" and “preempted".

It makes no sense to select a process if it is not loaded in memory, since it cannot execute until it is swapped in. If several processes tie for highest priority, the kernel picks the one that has been "ready to run" for the longest time, following a round robin scheduling policy.

If there are no processes eligible for execution, the processor idles until the next interrupt, which will happen in at most one clock tick; after handling that interrupt, the kernel again attempts to schedule a process to run.

Scheduling Parameters each process table entry contains a priority field for process scheduling. The priority of a process in user mode is a function of its recent CPU usage, with processes getting a lower priority if they have recently used the CPU. The range of process priorities can be partitioned into two classes: user priorities and kernel priorities.

Each class contains several priority values, and each priority has a queue of processes logically associated with it. Processes with user level priorities were preempted on their return from the kernel to user mode, and processes with kernel-level priorities achieved them in the sleep algorithm.

User level priorities are below a threshold value, and kernel-level priorities are above the threshold value. Kernel-level priorities are further subdivided: Processes with low kernel priority wake up on receipt of a signal, but processes with high kernel priority continue to sleep.

Figure 7.2 shows the threshold priority between user priorities and kernel priorities as the double .line between priorities "waiting for child exit" and “user level 0”.

The priorities called *"swapper," "waiting for disk I/O," "waiting for buffer," and "waiting for inode"* are high, non-interruptible system priorities, with *1, 3, 2, and 1* processes queued on the respective priority level, and the priorities called *"waiting for tty input," "waiting for tty output," and "waiting for child exit"* are low, interruptible system priorities with *4, 0, and 2* processes queued, respectively. The figure distinguishes user priorities, calling them "user level 0," "user level 1," to "user level n", containing 0, 4, and 1 processes, respectively.

The kernel calculates the priority of a process in specific process states.

- It assigns priority to a process about to go to sleep, correlating a fixed, and priority value with the reason for sleeping. The priority does not depend on the runtime characteristics of the process (I/O bound or CPU bound), but instead is a constant value that is hard-coded for each call to sleep, dependent on the reason the process is sleeping.

- The kernel adjusts the priority of a process that returns from kernel mode to user mode.
- 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.

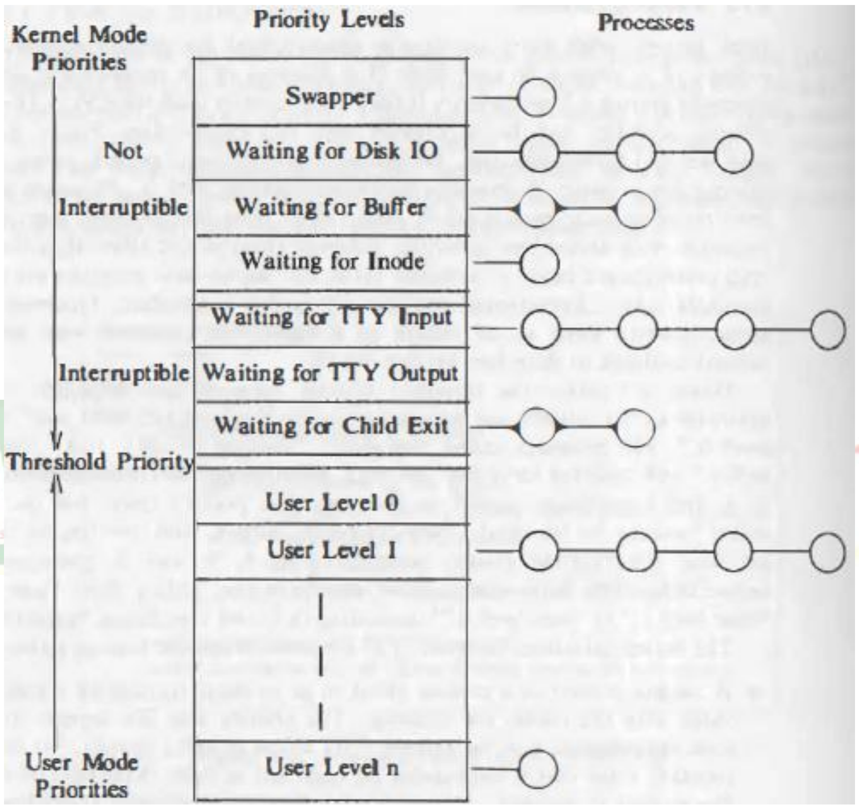


Figure 7.2: Range of Process Priorities

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; 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 7.3. Comparing this figure to Figure 7.2, 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.

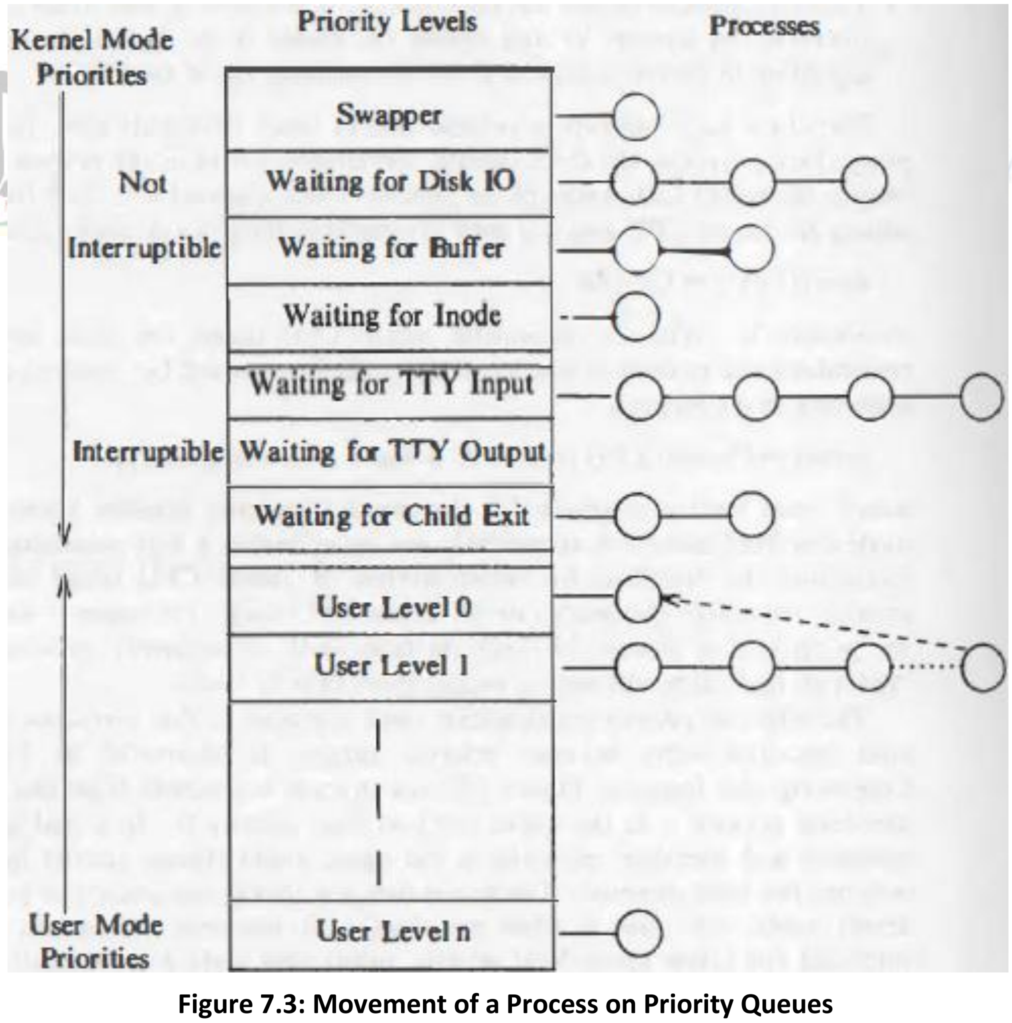
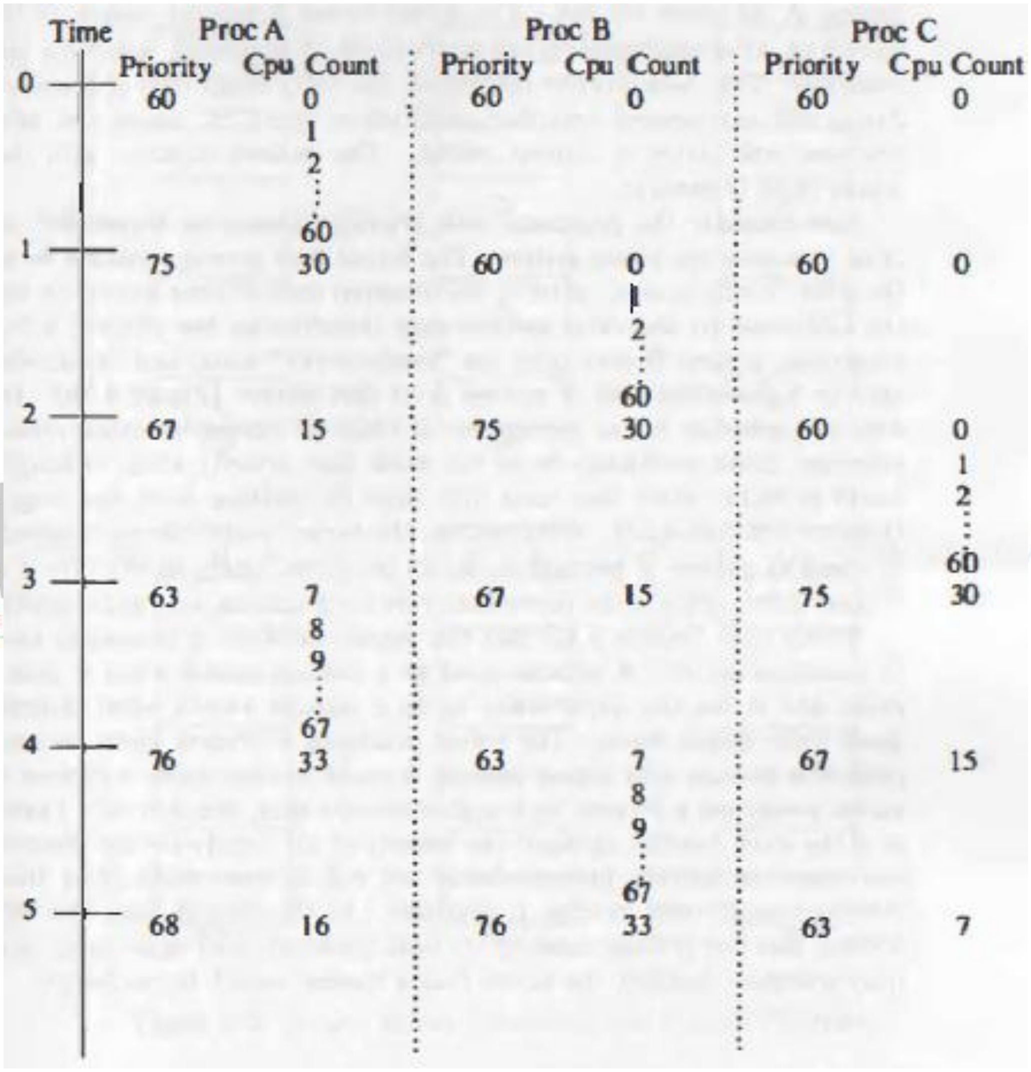


Figure 7.3: Movement of a Process on Priority Queues

Examples of Process Scheduling:

Figure 7.4 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 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;***



Time	Proc A Priority	Proc A Cpu Count	Proc B Priority	Proc B Cpu Count	Proc C Priority	Proc C Cpu Count
0	60	0 1 2 … 60	60	0	60	0
1	75	30	60	0 1 2 … 60	60	0
2	67	15	75	30	60	0 1 2 … 60
3	63	7 8 9 … 67	67	15	75	30
4	76	33	63	7 8 9 … 67	67	15
5	68	16	76	33	63	7

Figure 7.4: Example of Process Scheduling

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 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 killing them outright.

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 deals 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

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

User processes can schedule alarm signals using the alarm system call. The common factor in all the time related system calls i:s their reliance on the system clock: the kernel manipulates various time counters when handling clock interrupts and initiates appropriate action.

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
- Control process scheduling.

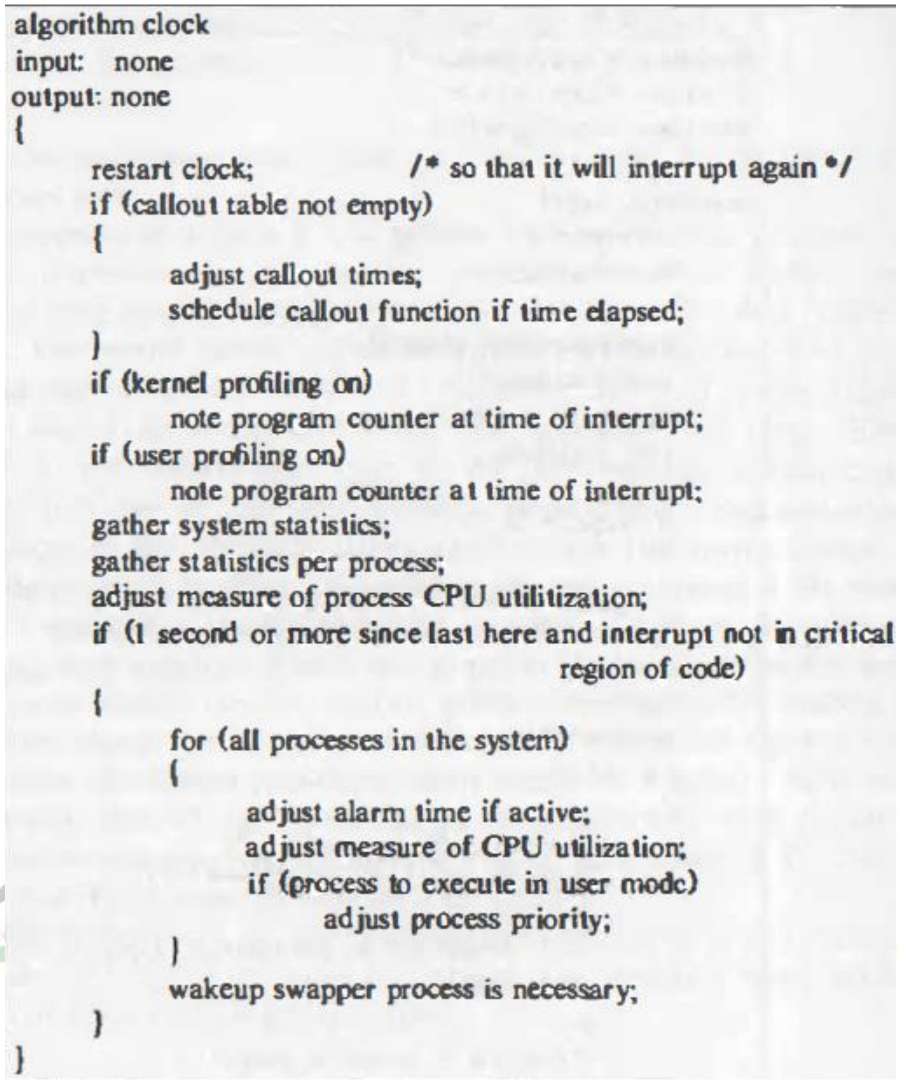
Some operations are done every clock interrupt, whereas others are done after several clock ticks. The clock handler runs with the processor execution level set high, preventing other events (such as interrupts from peripheral devices) from happening while the handler is active.

The clock handler is therefore fast, so that the critical time periods when other interrupts are blocked is as small as possible. Figure 7.5 shows the algorithm for handling clock interrupts.

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.

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.



```
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 7.5: Algorithm for the Clock Handler

The kernel stores the necessary information in the callout table (Figure 7.5), 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.

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 writing 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.

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.

Keeping Time

The kernel increments a timer variable at every clock interrupts, 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 exits, giving the real execution time of the process. Another timer variable set by the slime system call and updated once a second, keeps track of calendar time.

MEMORY MANAGEMENT POLICIES:

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.

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.

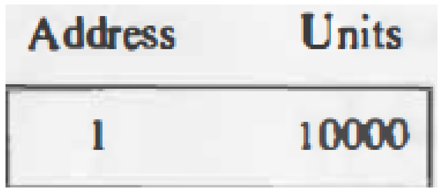
A process that resides on the swap device will eventually migrate back to main memory, freeing the space it had occupied on the swap device. Since speed is critical and the system can do I/O faster in one multiblock operation than in several single block operations, the kernel allocates contiguous space 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, 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.

Figure 7.6 illustrates an initial swap map that consists of 10,000 blocks starting at address 1.



Address	Units
1	10000

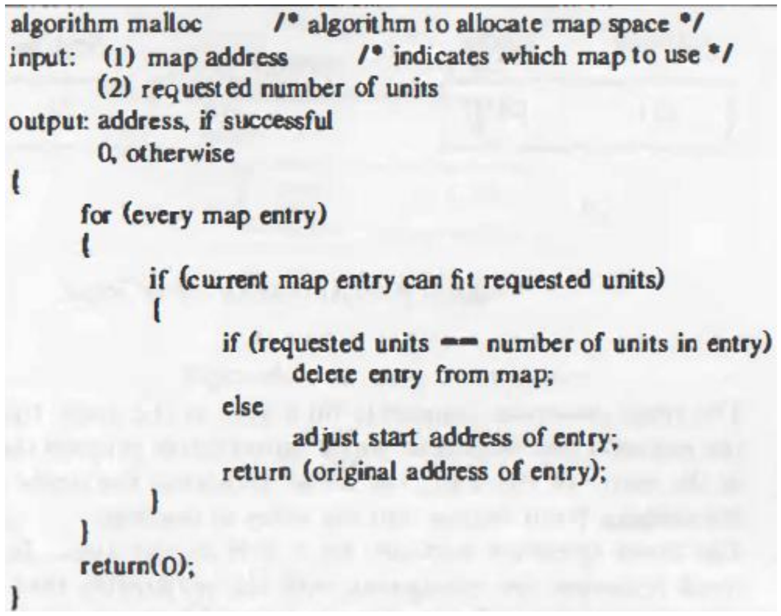
Figure 7.6: Initial Swap Map

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

Figure 7.7 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.

Otherwise, it adjusts the address and unit fields of the entry according to the amount of resources allocated.



```
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 7.7: Algorithm for Allocating Space from Maps

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 preventing the swapper from swapping it out 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.

Fork Swap:

The description of the fork system call 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 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.

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.

Finally, it swaps the process out in a normal swapping operation, zeroing out the newly allocated space on the swap device (see Figure 7.8).

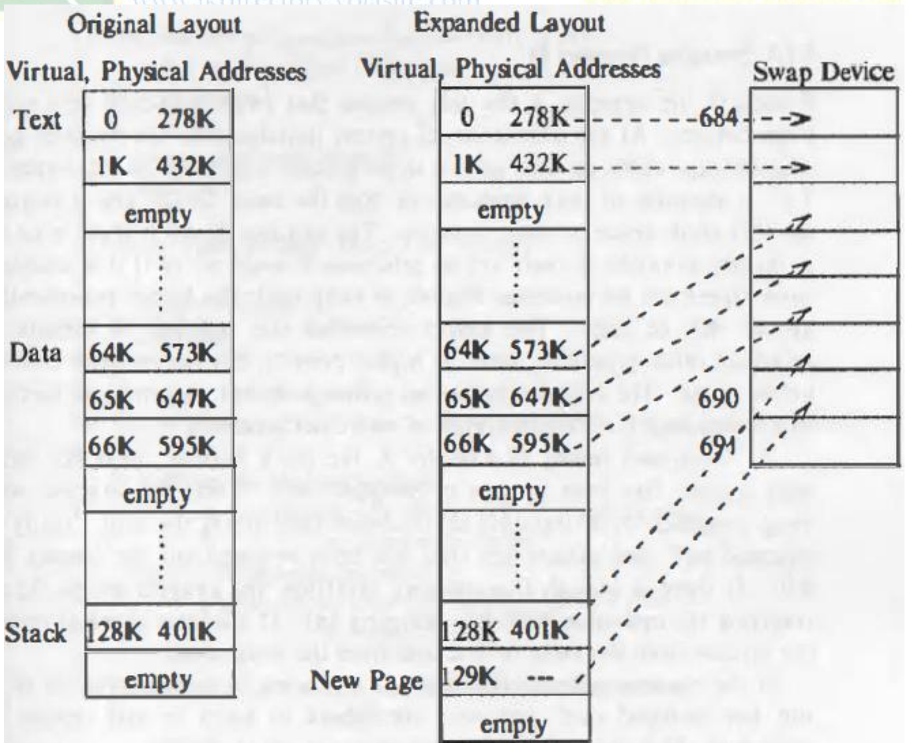


Figure 7.8: Adjusting Memory Map for Expansion Swap

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.

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 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 or if it is unable to do any work; the kernel periodically wakes it up, as will be seen. The kernel schedules the swapper to execute just as it schedules other processes, even though 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.

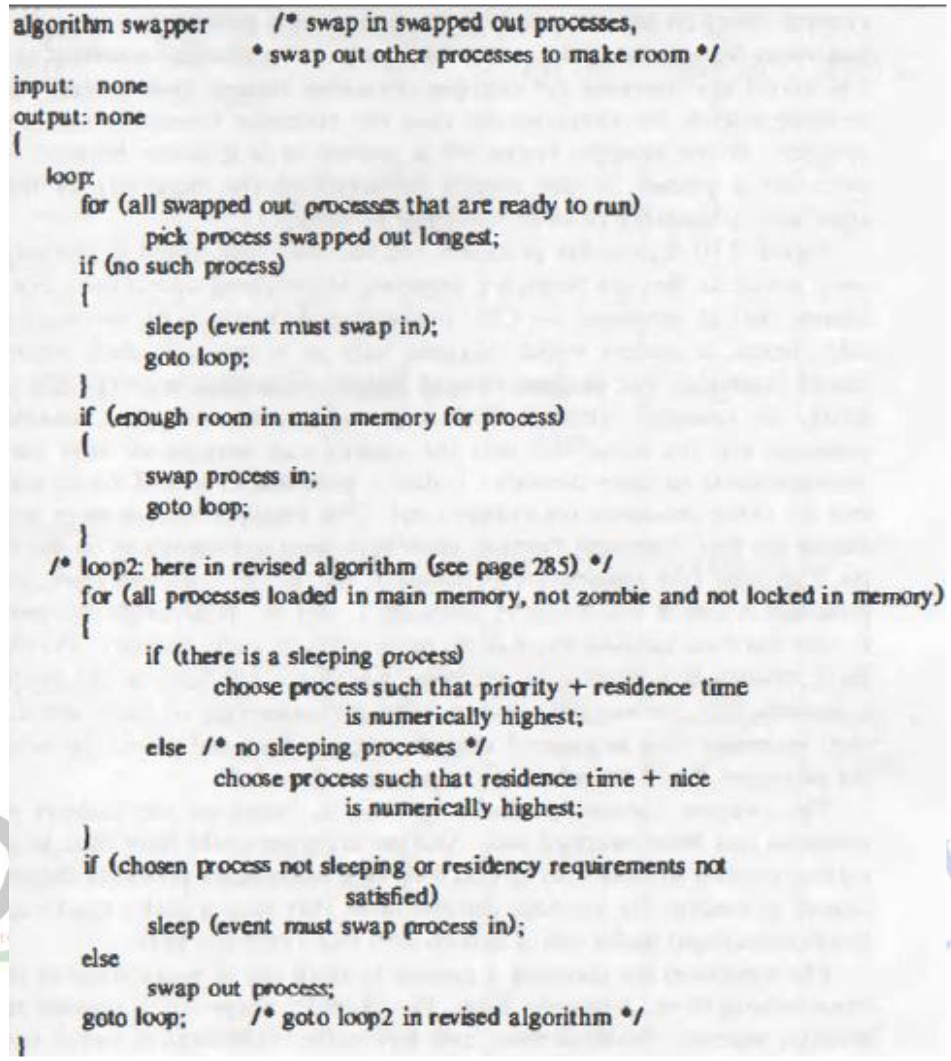
The clock handler measures the time that each process has been in core or swapped out. When the swapper wakes up to swap processes in, it examines all processes that are in the state "ready to run but swapped out" and selects one that has been swapped out the longest (see Figure 7.9).

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".
- 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.



```
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 7.9: Algorithm for the Swapper

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.

DEMAND PAGING:

Machines whose memory architecture is based on pages and whose CPU has restartable instructions 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 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."

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.

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.
- 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.

Page Faults:

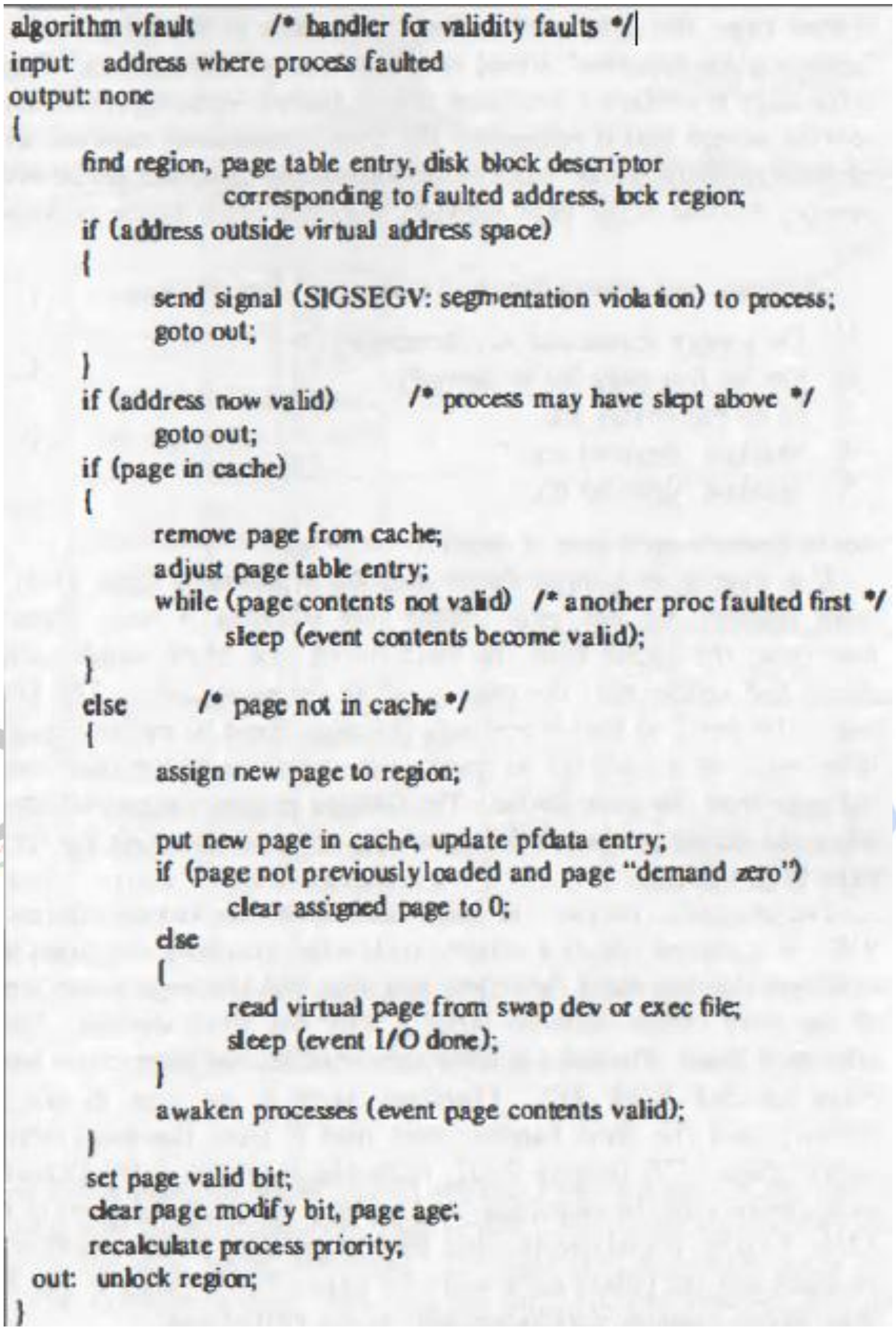
The system can incur two types of page faults: validity faults and protection faults. Because the fault handlers may have to read a page from disk to memory and sleep during the I/O operation, fault handlers are an exception to the general rule that interrupt handlers cannot sleep. However, because the 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.

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 7.10). 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 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"



```
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;
}
```

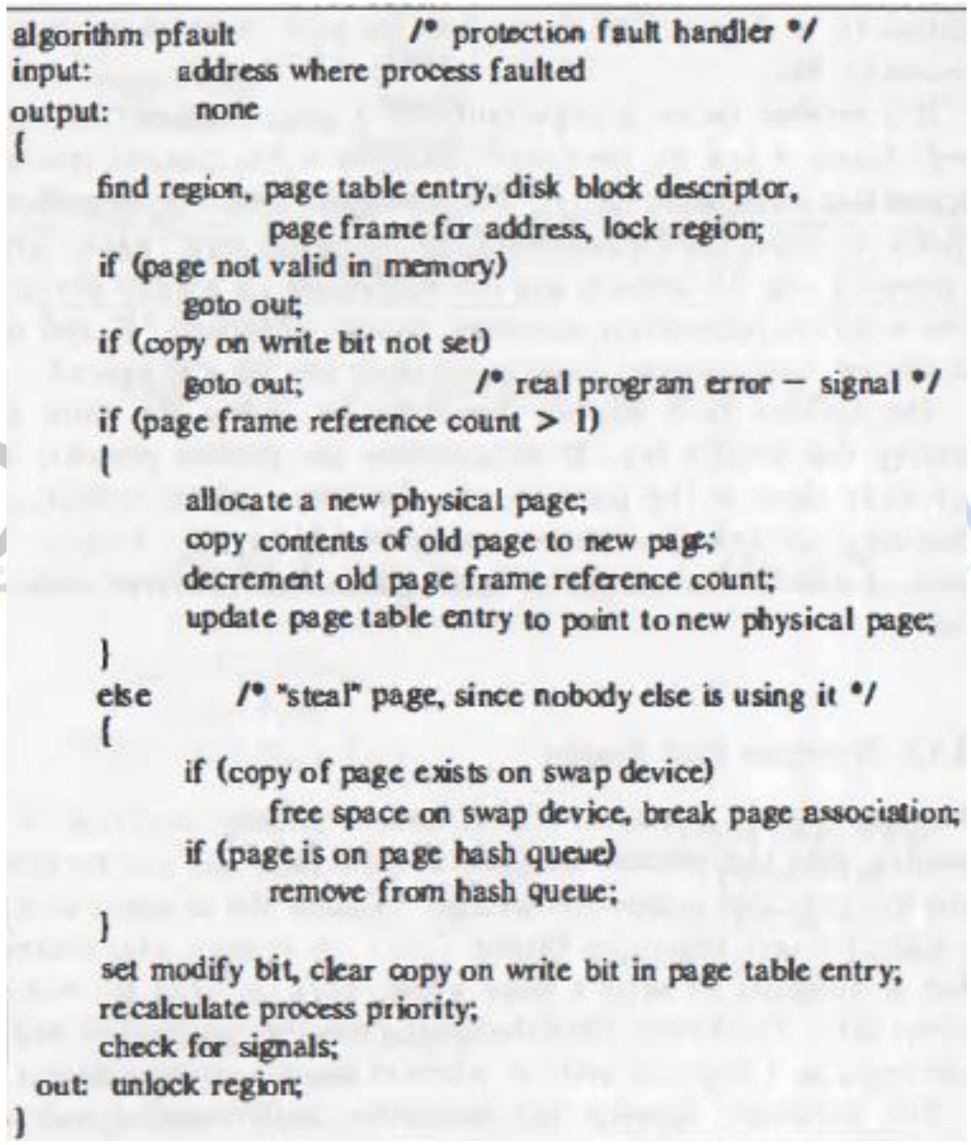
Figure 7.10: Algorithm for Validity Fault Handle

Protection Fault Handler

The second kind of memory fault that a process can incur is a protection fault, meaning that the process accessed a valid page but the permission bits associated with the page did not permit access.

A process also incurs a protection fault when it attempts to write a page whose copy on write bit was set during the fork system call. The kernel must determine whether permission was denied because the page requires a copy on write or whether something truly illegal happened.

The hardware supplies the protection fault handler with the virtual address where the fault occurred, and the fault handler finds the appropriate region and page table entry (Figure 7.11).



```
algorithm pfault              /* protection fault handler */
input:      address where process faulted
output:     none
{
      find region, page table entry, disk block descriptor,
                  page frame for address, lock region;
      if (page not valid in memory)
            goto out;
      if (copy on write bit not set)
            goto out;                /* real program error - signal */
      if (page frame reference count > 1)
      {
            allocate a new physical page;
            copy contents of old page to new page;
            decrement old page frame reference count;
            update page table entry to point to new physical page;
      }
      else        /* "steal" page, since nobody else is using it */
      {
            if (copy of page exists on swap device)
                  free space on swap device, break page association;
            if (page is on page hash queue)
                  remove from hash queue;
      }
      set modify bit, clear copy on write bit in page table entry;
      recalculate process priority;
      check for signals;
  out: unlock region;
}
```

Figure 7.11: Algorithm for Protection Fault Handler

It locks the region so that the page stealer cannot steal the page while the protection fault handler operates on it. If the fault handler determines that the fault was caused because the copy on write bit was set, and if the page is shared with other processes, the kernel allocates a new page and copies the contents of the old page to it; the other processes retain their references to the old page. After copying the page and updating the page table entry with the new page number, the kernel decrements the reference count of the old pfdata table entry.

If the copy on write bit is set but no other processes share the page, the kernel allows the process to reuse the physical page. It turns off the copy on write bit and disassociates the page from its disk copy, if one exists, because other processes may share the disk copy.

A HYBRID SYSTEM WITH SWAPPING AND DEMAND PAGING:

Although demand paging systems treat memory more flexibly than swapping systems, situations can arise where the page stealer and validity fault handler thrash because of a shortage of memory.

If the sum of the working sets of all processes is greater than the physical memory on a machine. The fault handler will usually sleep, because it cannot allocate pages for a process. The page stealer will not be able to steal pages fast enough, because all pages are in a working set. System throughput suffers because the kernel spends too much time in overhead, rearranging memory at a frantic pace.

The System V kernel runs swapping and demand paging algorithms to avoid thrashing problems. When the kernel cannot allocate pages for a process, it wakes up the swapper and puts the calling process into a state that is the equivalent of "ready to run but swapped." Several processes may be in this state simultaneously.

The swapper swaps out entire processes until available memory exceeds the high-water mark. For each process swapped out, it makes one “ready-to-run but swapped” process ready to run. It does not swap those processes in via the normal swapping algorithm but Jets them fault in pages as needed.