

The kernel contains a process table with an entry that describes the state of every active process in the system. The `u` area contains additional information that controls the operation of a process. The process table entry and the `u` area are part of the context of a process. The aspect of the process context that most visibly distinguishes it from the context of another process is, of course, the contents of its address space.

PROCESS STATES AND TRANSITIONS:

The lifetime of a process can be conceptually divided into a set of states that describe the process. The following list contains the complete set of process states.

1. The process is executing in user mode.
2. The process is executing in kernel mode.
3. The process is not executing but is ready to run as soon as the kernel schedules it.
4. The process is sleeping and resides in main memory.
5. The process is ready to run, but the swapper (process 0) must swap the process into main memory before the kernel can schedule it to execute.
6. The process is sleeping, and the swapper has swapped the process to secondary storage to make room for other processes in main memory.
7. The process is returning from the kernel to user mode, but the kernel preempts it and does a context switch to schedule another process. The distinction between this state and state 3 ("ready to run") will be brought out shortly.
8. &. The process is newly created and is in a transition state; the process exists, but it is not ready to run, nor is it sleeping. This state is the start state for all processes except process 0.
9. The process executed the `exit` system call and is in the zombie state. The process no longer exists, but it leaves a record containing an exit code and some timing statistics for its parent process to collect. The zombie state is the final state of a process.

Figure 5.1 gives the complete process state transition diagram. The process enters the state model in the "created" state when the parent process executes the `fork` system call and eventually moves into a state where it is ready to run (3 or 5). For simplicity, assume the process enters the state "ready to run in memory".

The process scheduler will eventually pick the process to execute, and the process enters the state "kernel running," where it completes its part of the fork system call. When the process completes the system call, it may move to the state "user running," where it executes in user mode. After a period of time, the system clock may interrupt the processor, and the process enters state "kernel running" again.

When the clock interrupt handler finishes servicing the clock interrupt, the kernel may decide to schedule another process to execute, so the first process enters state "preempted" and the other process executes. The state "preempted" is really the same as the state "ready to run in memory" (the dotted line in the figure that connects the two states emphasizes their equivalence), but they are depicted separately to stress that a process executing in kernel mode can be preempted only when it is about to return to user mode.

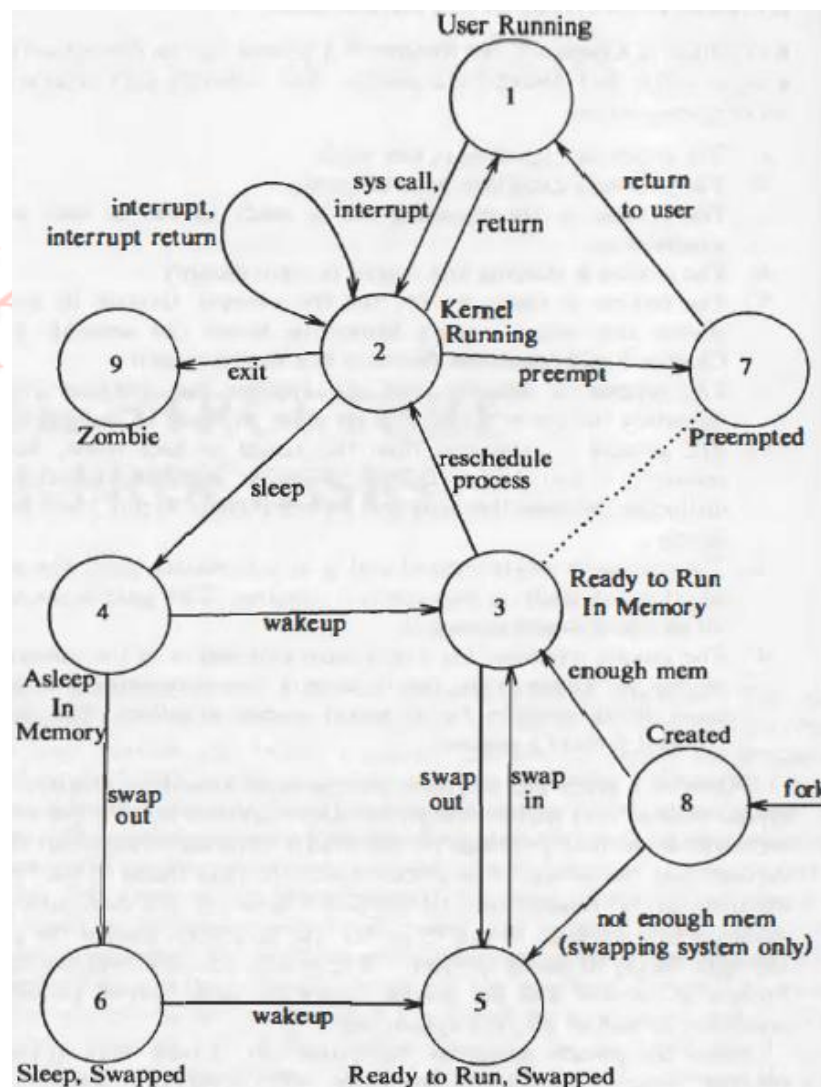


Figure 5.1: Process State Transition Diagram

Consequently, the kernel could swap a process from the state "preempted" if necessary. Eventually, the scheduler will choose the process to execute, and it returns to the state "user running," executing in user mode again.

When a process executes a system call, it leaves the state "user running" and enters the state "kernel running." Suppose the system call requires I/O from the disk, and the process must wait for the I/O to complete. It enters the state "asleep in memory," putting itself to sleep until it is notified that the I/O has completed.

When the I/O later completes, the hardware interrupts the CPU, and the interrupt handler awakens the process, causing it to enter the state "ready to run in memory."

Suppose the system is executing many processes that do not fit simultaneously into main memory, and the swapper (process 0) swaps out the process to make room for another process that is in the state "ready to run swapped" When evicted from main memory, the process enters the state "ready to run swapped."

Eventually, the swapper chooses the process as the most suitable to swap into main memory, and the process reenters the state "ready to run in memory". The scheduler will eventually choose to run the process, and it enters the state "kernel running" and proceeds. When a process completes, it invokes the exit system call, thus entering the states "kernel running" and, finally, the "zombie" state.

Two kernel data structures describe the state of a process: the process table entry and the u area. The process table contains fields that must always be accessible to the kernel, but the u area contains fields that need to be accessible only to the running process. Therefore, the kernel allocates space for the u area only when creating a process: It does not need u areas for process table entries that do not have processes.

The fields in the process table are the following:

- The state field identifies the process state.
- The process table entry contains fields that allow the kernel to locate the process and its u area in main memory or in secondary storage. The kernel uses the information to do a context switch to the process when the process moves from state "ready to run in memory" to the state "kernel running" or from the state "preempted" to the state "user running."
- Several user identifiers (UIDs) determine various process privileges.
- Process identifiers (PIDs) specify the relationship of processes to each other. These ID fields are set up when the process enters the state "created" in the fork system call.

- The process table entry contains an event descriptor when the process is in the "sleep" state.
- Scheduling parameters allow the kernel to determine the order in which processes move to the states "kernel running" and "user running".
- A signal field enumerates the signals sent to a process but not yet handled.
- Various timers give process execution time and kernel resource utilization, used for process accounting and for the calculation of process scheduling priority. One field is a user-set timer used to send an alarm signal to a process.

The u area contains the following fields that further characterize the process states.

- A pointer to the process table identifies the entry that corresponds to the u area.
- The real and effective user IDs determine various privileges allowed the process, such as file access rights.
- Timer fields record the time the process spent executing in user mode and in kernel mode.
- An array indicates how the process wishes to react to signals.
- The control terminal field identifies the "login terminal" associated with the process, if one exists.
- Error field records errors encountered during a system call.
- A return value field contains the result of system calls.
- I/O parameters describe the amount of data to transfer, the address of the source (or target) data array in user space, file offsets for I/O, and so on.
- The current directory and current root describe the file system environment of the process.
- The user file descriptor table records the files the process has open.
- Limit fields restrict the size of a process and the size of a file it can write.
- A permission modes field masks mode settings on files the process creates.

LAYOUT OF SYSTEM MEMORY:

Assume that the physical memory of a machine is addressable, starting at byte offset 0 and going up to a byte offset equal to the amount of memory on the machine.

As we know that a process on the UNIX system consists of three logical sections: text, data, and stack.

The text section contains the set of instructions the machine executes for the process; addresses in the text section include text addresses (for branch instructions or subroutine calls), data addresses (for access to global data variables), or stack addresses (for access to data structures local to a subroutine).

If the machine were to treat the generated addresses as address locations in physical memory, it would be impossible for two processes to execute concurrently if their set of generated addresses overlapped.

The compiler could generate addresses that did not overlap between programs, but such a procedure is impractical for general-purpose computers because the amount of memory on a machine is finite and the set of all programs that could be compiled is infinite.

The compiler therefore generates addresses for a virtual address space with a given address range, and the machine's memory management unit translates the virtual addresses generated by the compiler into address locations in physical memory. The compiler does not have to know where in memory the kernel will later load the program for execution.

In fact, several copies of a program can coexist in memory: All execute using the same virtual addresses but reference different physical addresses. The subsystems of the kernel and the hardware that cooperate to translate virtual to physical addresses comprise the memory management subsystem.

Regions: The System V kernel divides the virtual address space of a process into logical regions. A region is a contiguous area of the virtual address space of a process that can be treated as a distinct object to be shared or protected.

Thus text, data, and stack usually form separate regions of a process. Several processes can share a region. For instance, several processes may execute the same program, and it is natural that they share one copy of the text region. Similarly, several processes may cooperate to share a common shared-memory region.

The kernel contains a region table and allocates an entry from the table for each active region in the system.

Let us assume the region table contains the information to determine where its contents are located in physical memory. Each process contains a private per process region table, called a pregion for short. Pregion entries may exist in the process table, the u area, or in a separately allocated area of memory, dependent on the implementation, but for simplicity, assume that they are part of the process table entry.

Each pregon entry points to a region table entry and contains the starting virtual address of the region in the process. Shared regions may have different virtual addresses in each process. The pregon entry also contains a permission field that indicates the type of access allowed the process: read-only, read-write, or read-execute.

The pregon and the region structure are analogous to the file table and the inode structure in the file system: Several processes can share parts of their address space via a region, much as they can share access to a file via an inode; each process accesses the region via a private pregon entry, much as it accesses the inode via private entries in its user file descriptor table and the kernel file table.

Figure 5.2 depicts two processes, A and B, showing their regions, pregon, and the virtual addresses where the regions are connected. The processes share text region 'a' at virtual addresses 8K and 4K respectively. If process A reads memory location 8K and process B reads memory location 4K, they read the identical memory location in region 'a'. The data regions and stack regions of the two processes are private.

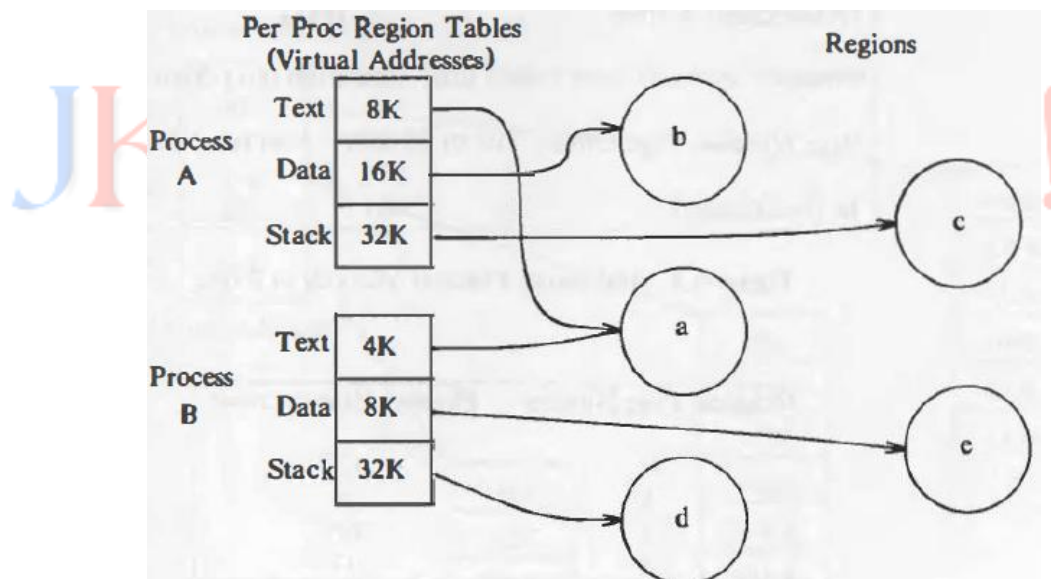


Figure 5.2: Processes and Regions

The concept of the region is independent of the memory management policies implemented by the operating system. Memory management policy refers to the actions the kernel takes to insure that processes share main memory fairly.

Pages and Page Tables: In a memory management architecture based on pages, the memory management hardware divides physical memory into a set of equal-sized blocks called pages.

Typical page sizes range from 512 bytes to 4K bytes and are defined by the hardware. Every addressable location in memory is contained in a page and, consequently, every memory location can be addressed by a *(page number, byte offset in page)* pair.

When the kernel assigns physical pages of memory to a region, it need not assign the pages contiguously or in a particular order. The purpose of paged memory is to allow greater flexibility in assigning physical memory, analogous to the assignment of disk blocks to files in a file system; just as the kernel assigns blocks to a file to increase flexibility and to reduce the amount of unused space caused by block fragmentation, so it assigns pages of memory to a region.

Layout of the Kernel: Although the kernel executes in the context of a process, the virtual memory mapping associated with the kernel is independent of all processes. The code and data for the kernel reside in the system permanently, and all processes share it.

When the system is brought into service (booted), it loads the kernel code into memory and sets up the necessary tables and registers to *map its virtual addresses into physical memory addresses* (figure 5.3). The kernel page tables are analogous to the page tables associated with a process, and the mechanisms used to map kernel virtual addresses are similar to those used for user addresses.

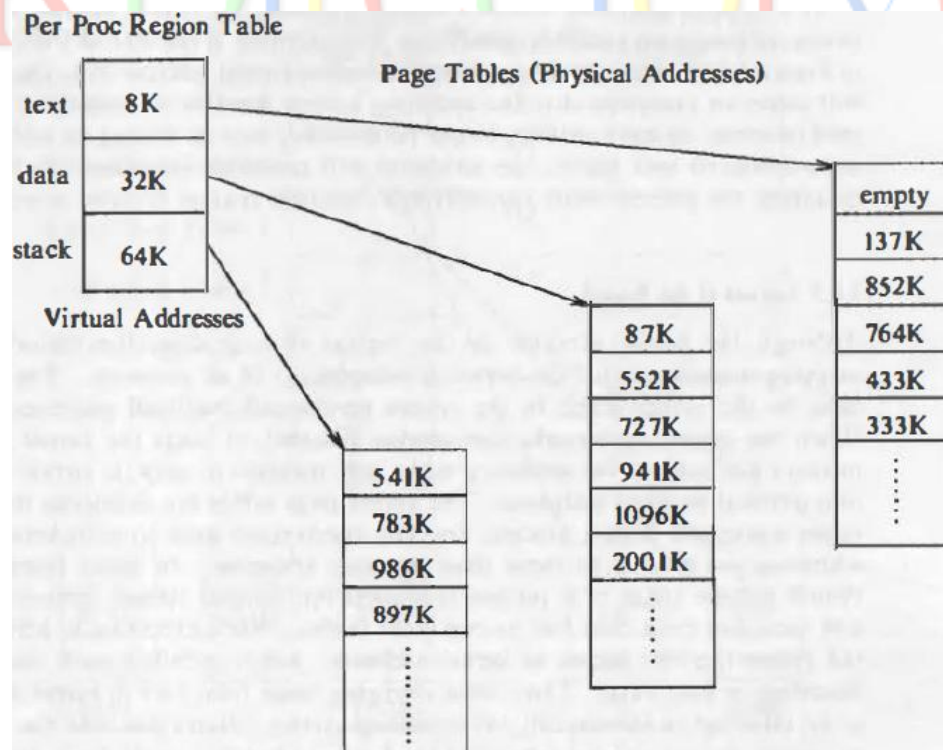


Figure 5.3: Mapping Virtual Addresses to Physical Addresses

In many machines, the virtual address space of a process is divided into several classes, including system and user, and each class has its own page tables. When executing in kernel mode, the system permits access to kernel addresses, but it prohibits such access when executing in user mode.

Thus, when changing mode from user to kernel as a result of an interrupt or system call, the operating system collaborates with the hardware to permit kernel address references, and when changing mode back to user, the operating system and hardware prohibit such references. Other machines change the virtual address translation by loading special registers when executing in kernel mode.

Figure 5.4 gives an example of the virtual addresses of the kernel and a process, where kernel virtual addresses range from 0 to 4M - 1 and user virtual addresses range from 4M up.

There are two sets of memory management triples, one for kernel addresses and one for user addresses, and each triple points to a page table that contains the physical page numbers corresponding to the virtual page addresses.

The system allows address references via the kernel register triples only when in kernel mode; hence, switching mode between kernel and user requires only that the system permit or deny address references via the kernel register triples.

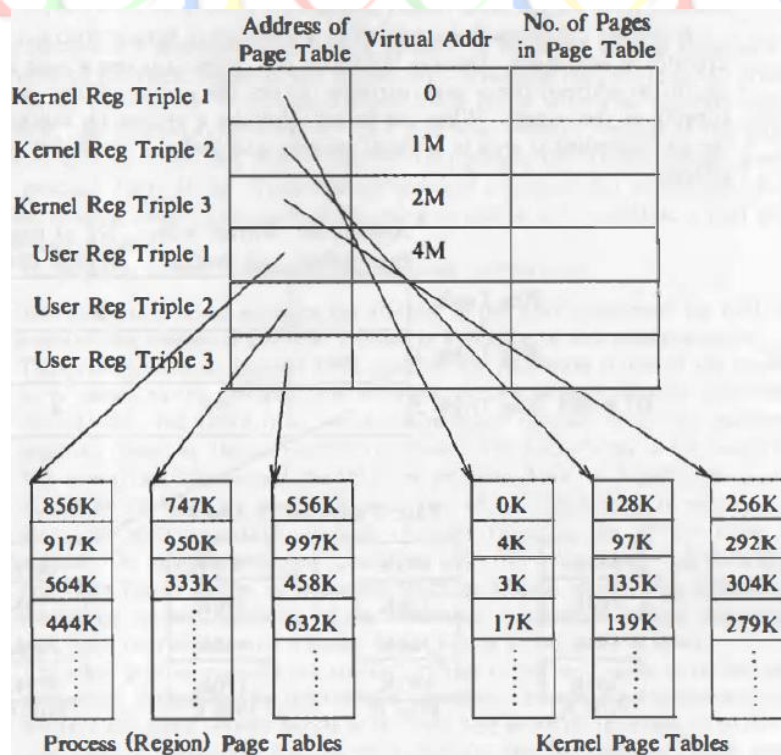


Figure 5.4: Changing the mode from User to Kernel

The U Area: Every process has a private u area, yet the kernel accesses it as if there were only one u area in the system, that of the running process.

The kernel changes its virtual address translation map according to the executing process to access the correct u area.

When compiling the operating system, the loader assigns the variable u, the name of the u area, a fixed virtual address.

The value of the u area virtual address is known to other parts of the kernel, in particular, the module that does the context switch.

The kernel knows where in its memory management tables the virtual address translation for the u area is done, and it can dynamically change the address mapping of the u area to another physical address.

The two physical addresses represent the u areas of two processes, but the kernel accesses them via the same virtual address.

A process can access its u area when it executes in kernel mode but not when it executes in user mode. Because the kernel can access only one u area at a time by its virtual address, the u area partially defines the context of the process that is running on the system.

When the kernel schedules a process for execution, it finds the corresponding u area in physical memory and makes it accessible by its virtual address.

Figure 5.5 shows a sample memory layout, where the first two register triples refer to kernel text and data (the addresses and pointers are not shown), and the third triple refers to the u area for process D.

If the kernel wants to access the u area of process A, it copies the appropriate page table information for the u area into the third register triple.

At any instant, the third kernel register triple refers to the u area of the currently running process, but the kernel can refer to the u area of another process by overwriting the entries for the u area page table address with a new address.

The entries for register triples 1 and 2 do not change for the kernel, because all processes share kernel text and data.

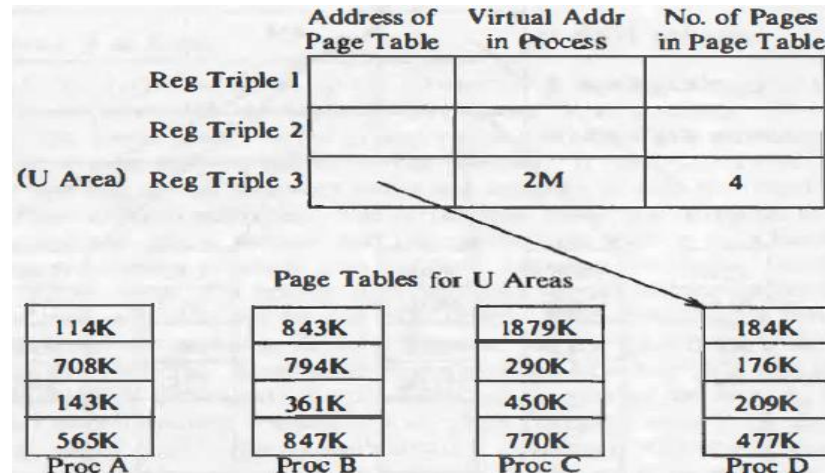


Figure 5.5: Memory Map of U Area in the Kernel

THE CONTEXT OF A PROCESS:

The context of a process consists of the contents of its (user) address space and the contents of hardware registers and kernel data structures that relate to the process.

Formally, the context of a process is the union of its user-level context, register context, and system-level context. The user-level context consists of the process text, data, user stack, and shared memory that occupy the virtual address space of the process.

Parts of the virtual address space of a process that periodically do not reside in main memory because of swapping or paging still constitute a part of the user-level context.

The register context consists of the following components.

- The program counter specifies the address of the next instruction the CPU will execute; the address is a virtual address in kernel or in user memory space.
- The processor status register (PS) specifies the hardware status of the machine as it relates to the process.
- The stack pointer contains the current address of the next entry in the kernel or user stack, determined by the mode of execution.
- The general-purpose registers contain data generated by the process during its execution.

The system-level context of a process has a "static part" (first three items of the following list) and a "dynamic part" (last two items).

A process has one static part of the system-level context throughout its lifetime, but it can have a variable number of dynamic parts.

The dynamic part of the system-level context should be viewed as a stack of context layers that the kernel pushes and pops on occurrence of various events. The system-level context consists of the following components.

- The process table entry of a process defines the state of a process, and contains control information that is always accessible to the kernel.
- The u area of a process contains process control information that need be accessed only in the context of the process.
- Pregion entries, region tables and page tables define the mapping from virtual to physical addresses and therefore define the text, data, stack, and other regions of a process.
- The kernel stack contains the stack frames of kernel procedures as a process executes in kernel mode. Although all processes execute the identical kernel code, they have a private copy of the kernel stack that specifies their particular invocation of the kernel functions.
- The dynamic part of the system-level context of a process consists of a set of layers, visualized as a last-in-first-out stack. Each system-level context layer contains the necessary information to recover the previous layer, including the register context of the previous level.

The kernel pushes a context layer when an interrupt occurs, when a process makes a system call, or when a process does a context switch. It pops a context layer when the kernel returns from handling an interrupt, when a process returns to user mode after the kernel completes execution of a system call, or when a process does a context switch.

Figure 5.6 depicts the components that form the context of a process. The left side of the figure shows the static portion of the context. It consists of the user level context, containing the process text (instructions), data, stack, and shared memory (if the process has any), and the static part of the system-level context, containing the process table entry, the u area, and the pregon entries (the virtual address mapping information for the user-level context).

The right side of the figure shows the dynamic portion of the context. It consists of several stack frames, where each frame contains the saved register context of the previous layer, and the kernel stack as the kernel executes in that layer.

System context layer 0 is a dummy layer that represents the user-level context; growth of the stack here is in the user address space, and the kernel stack is null.

The arrow pointing from the static part of the system-level context to the top layer of the dynamic portion of the context represents the logical information stored in the process table entry to enable the kernel to recover the current context layer of the process.

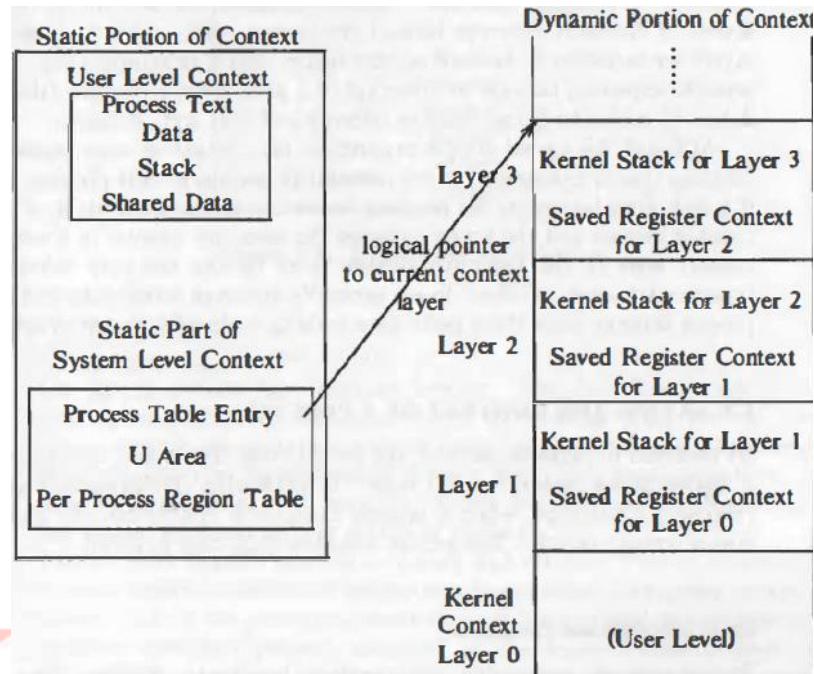


Figure 5.6: Components of the Context of a Process

A process runs within its context or, more precisely, within its current context layer. The number of context layers is bounded by the number of interrupt levels the machine supports.

SAVING THE CONTEXT OF A PROCESS:

The kernel saves the context of a process whenever it pushes a new system context layer. In particular, this happens when the system receives an interrupt, when a process executes a system call, or when the kernel does a context switch.

Interrupts and Exceptions:

The system is responsible for handling interrupts, whether they result from hardware (such as from the clock or from peripheral devices), from a programmed interrupt (execution of instructions designed to cause "software interrupts"), or from exceptions (such as page faults).

If the CPU is executing at a lower processor execution level than the level of the interrupt, it accepts the interrupt before decoding the next instruction and raises the processor execution level, so that no other interrupts of that level (or lower) can happen while it handles the current interrupt, preserving the integrity of kernel data structures.

The kernel handles the interrupt with the following sequence of operations:

- It saves the current register context of the executing process and creates (pushes) a new context layer.
- It determines the "source" or cause of the interrupt, identifying the type of interrupt (such as clock or disk) and the unit number of the interrupt, if applicable (such as which disk drive caused the interrupt).
 - When the system receives an interrupt, it gets a number from the machine that it uses as an offset into a table, commonly called an interrupt vector.
- The kernel invokes the interrupt handler. The kernel stack for the new context layer is logically distinct from the kernel stack of the previous context layer.
- The interrupt handler (*example for the table of interrupt handler is in figure 5.7*) completes its work and returns.
 - The kernel executes a machine-specific sequence of instructions that restores the register context and kernel stack of the previous context layer as they existed at the time of the interrupt and then resumes execution of the restored context layer.
 - The behavior of the process may be affected by the interrupt handler, since the interrupt handler may have altered global kernel data structures and awakened sleeping processes.

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

Figure 5.7: Sample Interrupt Vector

Figure 5.8 summarizes how the kernel handles interrupts. Some machines do part of the sequence of operations in hardware or microcode to get better performance than if all operations were done by software, but there are tradeoffs, based on how much of the context layer must be saved and the speed of the hardware instructions doing the save. The specific operations required in a UNIX system implementation are therefore machine dependent.

```

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

```

Figure 5.8: Algorithm for Handling Interrupts

Example of interrupt is represented in figure 5.9 as follows:

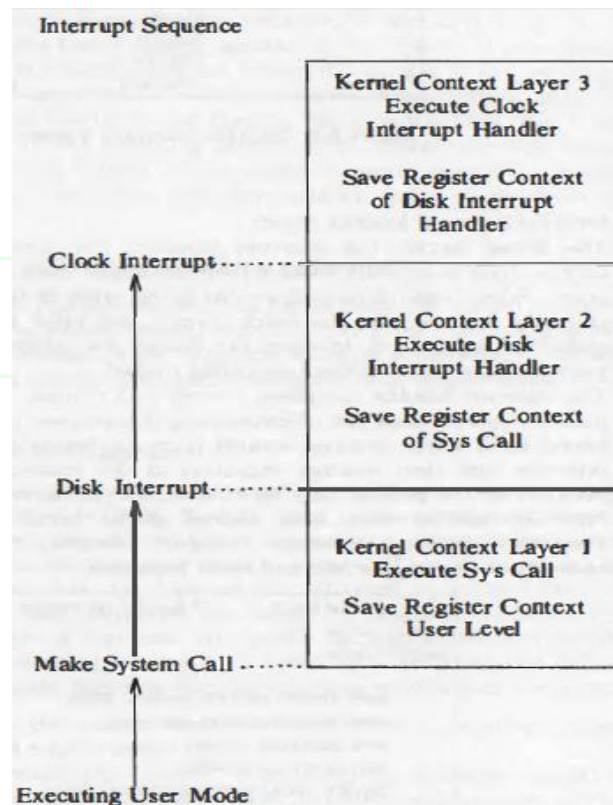


Figure 5.9: Example of Interrupts

Context Switch:

Referring to the process state diagram in Figure 5.1, we see that the kernel permits a context switch under four circumstances: when a process puts itself to sleep, when it exits, when it returns from a system call to user mode but is not the most eligible process to run, or when it returns to user mode after the kernel completes handling an interrupt but is not the most eligible process to run.

MANIPULATION OF THE PROCESS ADDRESS SPACE:

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

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

The operations that manipulate regions are to lock a region, unlock a region, allocate a region, attach a region to the memory space of a process, change the size of a region, load a region from a file into the memory space of a process, free a region, detach a region from the memory space of a process, and duplicate the contents of a region.

For example, the `exec` system call, which overlays the user address space with the contents of an executable file, detaches old regions, frees them if they were not shared, allocates new regions, attaches them, and loads them with the contents of the file.

Locking and Unlocking a Region:

The kernel has operations to lock and unlock a region, independent of the operations to allocate and free a region, just as the file system has lock-unlock and allocate-release operations for inodes (algorithms `iget` and `iput`).

Thus the kernel can lock and allocate a region and later unlock it without having to free the region. Similarly, if it wants to manipulate an allocated region, it can lock the region to prevent access by other processes and later unlock it.

Allocating a Region:

The kernel allocates a new region (algorithm allocreg, Figure 5.10) during fork, exec, and shmget (shared memory) system calls. The kernel contains a region table whose entries appear either on a free linked list or on an active linked list.

When it allocates a region table entry, the kernel removes the first available entry from the free list, places it on the active list, locks the region, and marks its type (shared or private). With few exceptions, every process is associated with an executable file as a result of a prior exec call, and *allocreg* sets the inode field in the region table entry to point to the inode of the executable file.

The inode identifies the region to the kernel so that other processes can share the region if desired. The kernel increments the inode reference counts to prevent other processes from removing its contents when unlinking it; *allocreg* returns a locked, allocated region.

Algorithm allocreg /*allocate a region data structure*/

Input: (1) inode pointer

(2) Region type

Output: locked region

{

remove region from linked list of free regions;

assign region type;

assign region inode pointer;

if (inode pointer not null)

increment inode reference count;

place region on linked list of active regions;

return(locked region);

}

Figure 5.10: Algorithm for Allocating a Region

Attaching a Region to a Process:

The kernel attaches a region during the fork, exec, and shmat system calls to connect it to the address space of a process (algorithm attachreg, Figure 5.11).


```

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

```

Figure 5.11: Algorithm for Attachreg

The region may be a newly allocated region or an existing region that the process will share with other processes.

The kernel allocates a free pregon entry, sets its type field to text, data, shared memory, or stack, and records the virtual address where the region will exist in the process address space.

The process must not exceed the system-imposed limit for the highest virtual address, and the virtual addresses of the new region must not overlap the addresses of existing regions.

For example, if the system restricts the highest virtual address of a process to 8 megabytes, it would be illegal to attach a 1 megabyte-size region to virtual address 7.SM. If it is legal to attach the region, the kernel increments the size field in the process table entry according to the region size, and increments the region reference count.

Attachreg then initializes a new set of memory management register triples for the process: If the region is not already attached to another process, the kernel allocates page tables for it in a subsequent call to *growreg*; otherwise, it uses the existing page tables. Finally, attachreg returns a pointer to the pregon entry for the newly attached region.

Changing the Size of a Region:

A process may expand or contract its virtual address space with the *sbrk* system call. Similarly, the stack of a process automatically expands (that is, the process does not make an explicit system call) according to the depth of nested procedure calls. Internally, the kernel invokes the algorithm *growreg* to change the size of a region (Figure 5.12).

```

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

```

Figure 5.12: Example of Attaching to an Existing Text Region

When a region expands, the kernel makes sure that the virtual addresses of the expanded region do not overlap those of another region and that the growth of the region does

not cause the process size to become greater than the maximum allowed virtual memory space.

The kernel never invokes `growreg` to increase the size of a shared region that is already attached to several processes; therefore, it does not have to worry about increasing the size of a region for one process and causing another process to grow beyond the system limit for process size.

The two cases where the kernel uses `growreg` on an existing region are `sbrk` on the data region of a process and automatic growth of the user stack. Both regions are private. Text regions and shared memory regions cannot grow after they are initialized.

The kernel now allocates page tables (or extends existing page tables) to accommodate the larger region and allocates physical memory on systems that do not support demand paging.

Loading a Region:

In a system that supports demand paging, the kernel can "map" a file into the process address space during the `exec` system call, arranging to read individual physical pages later on demand.

If the kernel does not support demand paging, it must copy the executable file into memory, loading the process regions at virtual addresses specified in the executable file.

It may attach a region at a different virtual address from where it loads the contents of the file, creating a gap in the page table.

To load a file into a region, `loadreg` (Figure 5.13) accounts for the gap between the virtual address where the region is attached to the process and the starting virtual address of the region data and expands the region according to the amount of memory the region requires.

Then it places the region in the state "being loaded into memory" and reads the region data into memory from the file, using an internal variation of the read system call algorithm.

Algorithm loadreg /*load a portion of a file into a region*/

Input: (1) pointer to per process region table entry

(2) virtual address to load region

(3) inode pointer of file for loading region

(4) byte offset in file for start of region

(5) byte count for amount of data to load

Output: none

(

increase region size according to eventual size of region

(algorithm growreg);

mark region state: being loaded into memory;

unlock region;

set up u area parameters for reading file:

target virtual address where data is read to,

start offset t value for reading file,

count of bytes to read from file;

read file into region (internal variant of read algorithm);

lock region;

mark region state: completely loaded into memory;

awaken all processes waiting for region to be loaded;

}

Figure 5.13: Algorithm for Loadreg

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

Freeing a Region:

When a region is no longer attached to any processes, the kernel can free the region and return it to the list of free regions (Figure 5.14).

If the region is associated with an inode, the kernel releases the inode using algorithm iput, corresponding to the increment of the inode reference count in allocreg.

```

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

```

Figure 5.14: Algorithm for Freeing a Region

The kernel releases physical resources associated with the region, such as page tables and memory pages.

Detaching a Region from a Process:

The kernel detaches regions in the `exec`, `exit`, and `shmdt` (detach shared memory) system calls.

It updates the `pregion` entry and severs the connection to physical memory by invalidating the associated memory management register triple (algorithm `detachreg`, Figure 5.15).

```

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

```

Figure 5.15: Algorithm Detachreg

The address translation mechanisms thus invalidated apply specifically to the process, not to the region (as in algorithm freereg). The kernel decrements the region reference count and the size field in the process table entry according to the size of the region.

If the region reference count drops to 0 and if there is no reason to leave the region intact, the kernel frees the region using algorithm freereg. Otherwise, it releases the region and inode locks, which had been locked to prevent race conditions but leaves the region and its resources allocated.

Duplicating a Region:

The fork system call requires that the kernel duplicate the regions of a process. If a region is shared (shared text or shared memory), however, the kernel need not physically copy the region; instead, it increments the region reference count, allowing the parent and child processes to share the region.

If the region is not shared and the kernel must physically copy the region, it allocates a new region table entry, page table, and physical memory for the region. In *Figure 5.16* for example, process A forked process B and duplicated its regions. The text region of process A is shared, so process B can share it with process A. But the data and stack regions of process A are private, so process B duplicates them by copying their contents to newly allocated regions. Even for private regions, a physical copy of the region is not always necessary

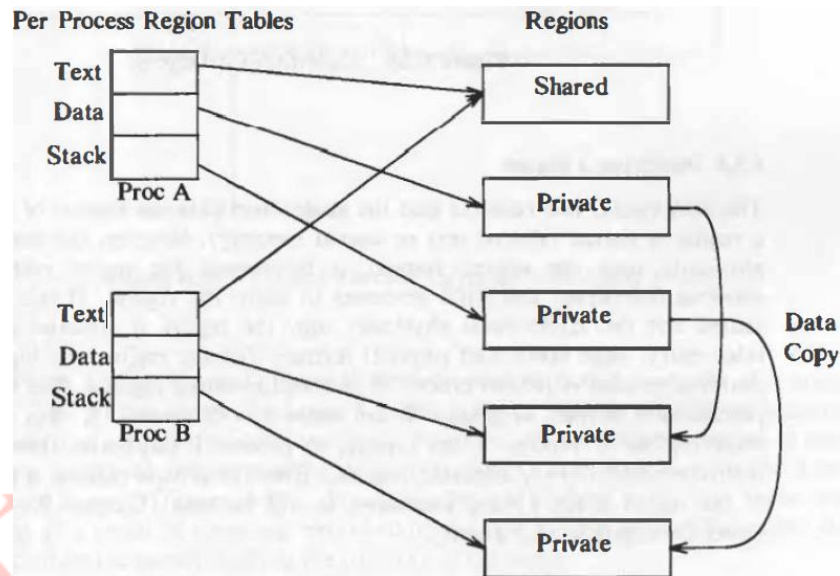


Figure 5.16: Duplicating a Region

Figure 5.17 shows the algorithm for dupreg.

```

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

```

Figure 5.17: Algorithm for Dupreg

SLEEP:

The algorithms for sleep, which changes the process state from "kernel running" to "asleep in memory," and wakeup, which changes the process state from "asleep" to "ready to run" in memory or swapped.

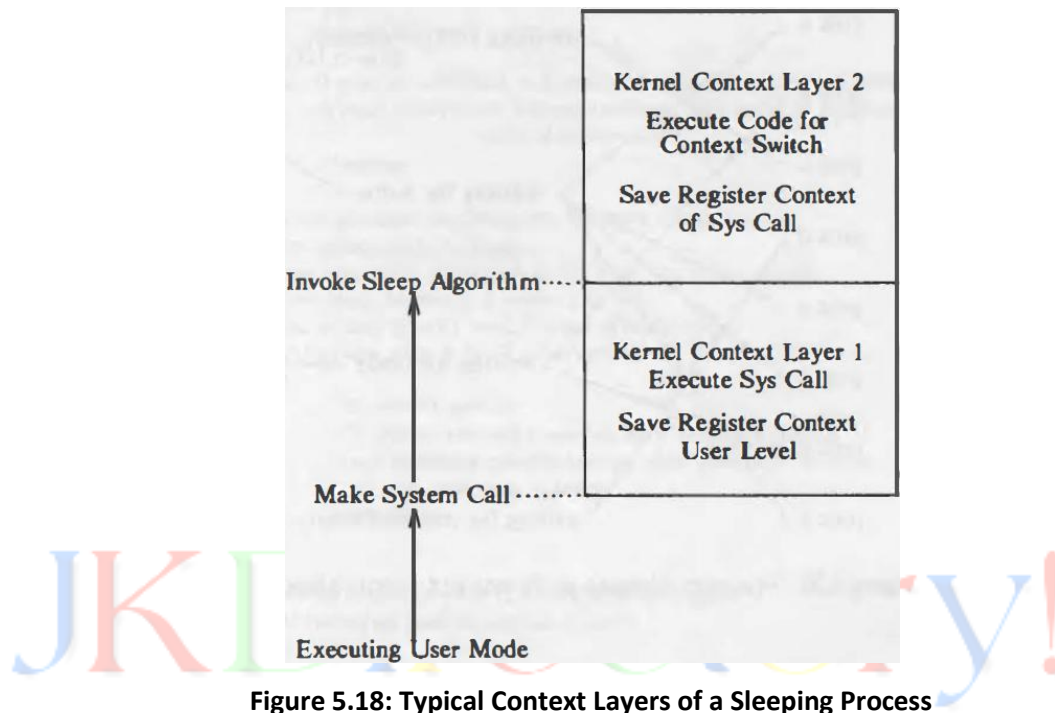


Figure 5.18: Typical Context Layers of a Sleeping Process

When a process goes to sleep, it typically does so during execution of a system call: The process enters the kernel (context layer 1) when it executes an operating system trap and goes to sleep awaiting a resource. When the process goes to sleep, it does a context switch, pushing its current context layer and executing in kernel context layer 2 (Figure 5.18).

Processes also go to sleep when they incur page faults as a result of accessing virtual addresses that are not physically loaded; they sleep while the kernel reads in the contents of the pages.

Sleep Events and Addresses:

Processes are said to sleep on an event, meaning that they are in the sleep state until the event occurs, at which time they wake up and enter a "ready-to-run" state (in memory or swapped out). Although the system uses the abstraction of sleeping on an event, the implementation maps the set of events into a set of (kernel) virtual addresses.

The addresses that represent the events are coded into the kernel, and their only significance is that the kernel expects an event to map into a particular address.

The abstraction of the event does not distinguish how many processes are awaiting the event, nor does the implementation. As a result, two anomalies arise. First, when an event occurs and a wakeup call is issued for processes that are sleeping on the event, they all wake up and move from a sleep state to a ready-to-run state.

The kernel does not wake up one process at a time, even though they may contend for a single locked structure, and many may go back to sleep after a brief visit to the kernel running state.

Figure 5.19 shows several processes sleeping on events. The second anomaly in the implementation is that several events may map into one address. In Figure 5.19, for example, the events "waiting for the buffer" to become free and "awaiting I/O completion" map into the address of the buffer ("addr A"). When I/O for the buffer completes, the kernel wakes up all processes sleeping on both events.

Since a process waiting for I/O keeps the buffer locked, other processes waiting for the buffer to become free will go back to sleep if the buffer is still locked when they execute. It would be more efficient if there would be a one-to-one mapping of events to addresses.

In practice, however, performance is not hurt, because the mapping of multiple events into one address is rare and because the running process usually frees the locked resource before the other processes are scheduled to run. Stylistically, however, it would make the kernel a little easier to understand if the mapping were one-to-one.

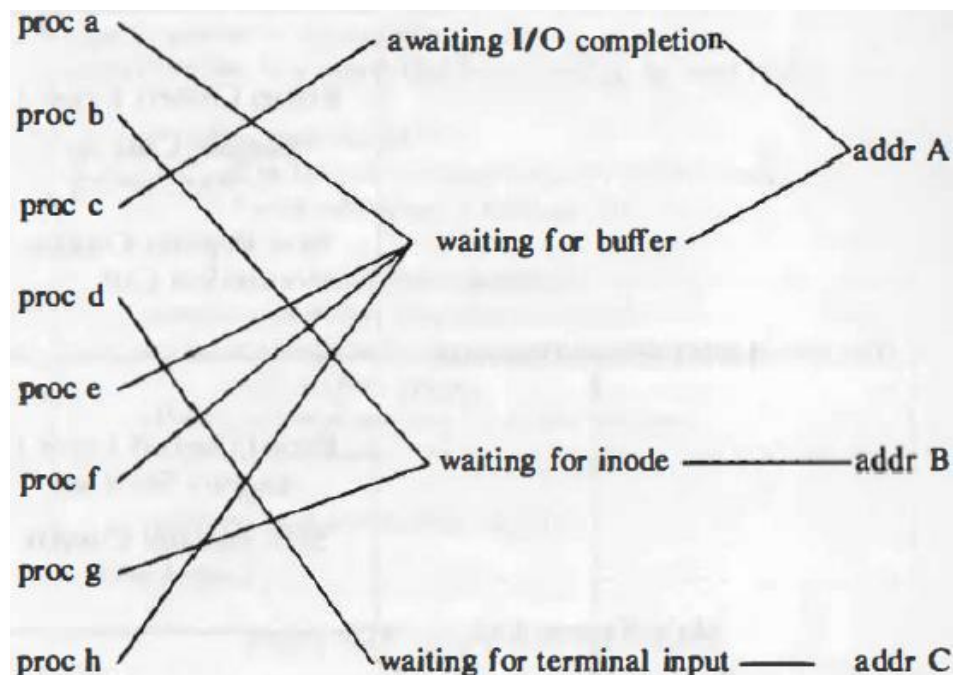


Figure 5.19: Processes Sleeping on Events and Events Mapping into Addresses

Algorithms for Sleep and Wakeup:

```

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

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

    remove process from sleep hash queue, if still there;

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

```

Figure 5.20: Sleep Algorithm

Figure 5.20 shows the algorithm for sleep. The kernel first raises the process or execution level to block out all interrupts so that there can be no race conditions when it manipulates the sleep queues, and it saves the old processor execution level so that it can be restored when the process later wakes up.

It marks the process state "asleep," saves the sleep address and priority in the process table, and puts it onto a hashed queue of sleeping processes. In the simple case (sleep cannot be interrupted), the process does a context switch and is safely asleep.

When a sleeping process wakes up, the kernel later schedules it to run: The process returns from its context switch in the sleep algorithm, restores the processor execution level to the value it had when the process entered the algorithm, and returns.

```

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

```

Figure 5.21: Algorithm for Wakeup

To wake up sleeping processes, the kernel executes the wakeup algorithm (Figure 5.21), either during the usual system call algorithms or when handling an interrupt.

For instance, the algorithm iput releases a locked inode and awakens all processes waiting for the lock to become free.

Similarly, the disk interrupt handler awakens a process waiting for I/O completion. The kernel raises the processor execution level in wakeup to block out interrupts.

Then for every process sleeping on the input sleep address, it marks the process state field “ready to run”, removes the process from the linked list of sleeping processes, places it on a linked list of processes eligible for scheduling, and clears the field in the process table that marked its sleep address.

If a process that woke up was not loaded in memory, the kernel awakens the swapper process to swap the process into memory (assuming the system is one that does not support demand paging); otherwise, if the awakened process is more eligible to run than the currently executing process, the kernel sets a scheduler flag so that it will go through the process scheduling algorithm when the process returns to user mode.

Finally, the kernel restores the processor execution level. It cannot be stressed enough: wakeup does not cause a process to be scheduled immediately; it only makes the process eligible for scheduling.

To distinguish the types of sleep states, the kernel sets the scheduling priority of the sleeping process when it enters the sleep state, based on the sleep priority parameter. That is, it invokes the sleep algorithm with a priority value, based on its knowledge that the sleep event is sure to occur or not.

If the priority is above a threshold value, the process will not wake up prematurely on receipt of a signal but will sleep until the event it is waiting for happens. But if the priority value is below the threshold value, the process will awaken immediately on receipt of the signal.