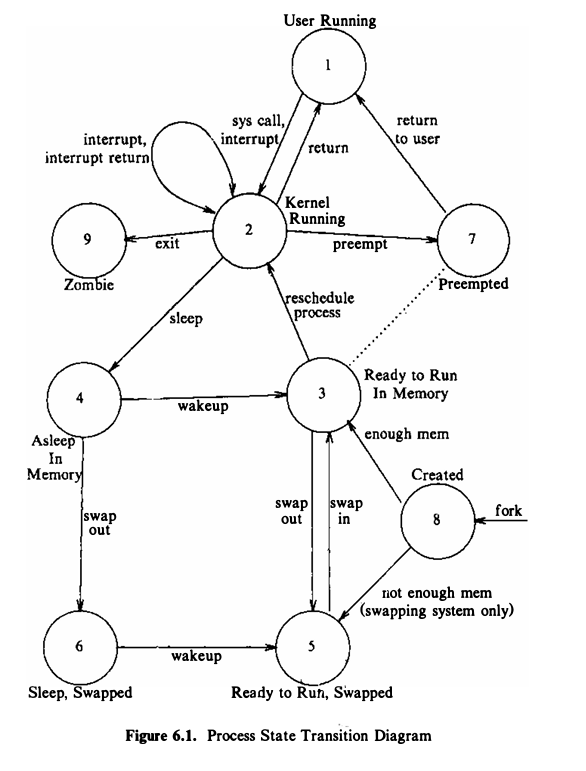
The kernel has a process table where it stores the state of the process and other information about the process. The information of the entry and the u-area of the process combined is the context of the process.

**Process States And Transitions**

The complete set of process states:

1. Executing in user mode.
2. Executing in kernel mode.
3. Ready to run.
4. Sleeping in memory.
5. Ready to run, but in swap space (covered later).
6. Sleeping in swap space.
7. Preempted. (the process is returning from kernel to user mode, but the kernel preempts it and does a context switch to schedule another process. Very similar to state 3)
8. Newly created. Not ready run, nor sleeping. This is the start state for all processes expect process 0.
9. The process executed 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.

Process state transition diagram



The process enters the created state when the parent process executes the fork system call model and eventually moves into a state where it is ready to run (3 or 5). The scheduler will eventually pick the process and the process enters the state kernel running, where it completes its part of fork system call. After the completion of system call, it may move to user running. When interrupts occur (such as system call), it again enters the state kernel running. After the servicing of the interrupt the kernel may decide to schedule another process to execute, so the first process enters the state preempted. The state preempted is really same as the state ready to run in memory, 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. Consequently, the kernel could swap a process from the state preempted if necessary. Eventually, it will return to user running again.

When a system call is executed, it leaves the state user running and enters the state kernel running. If in kernel mode, the process needs to sleep for some reason (such as waiting for I/O), it enters the state asleep in memory. When the event on it which it has slept, happens, the interrupt handler awakens the process, and it enters the state ready to run in memory.

Suppose the system is executing many processes that do not fit simultaneously into main memory, then the swapper (process 0) swaps out a 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, swapper chooses the process as most eligible to run and it re-enters the state ready to run in memory. And then when it is scheduled, it will enter the state kernel running. When a process completes and invokes exit system call, thus entering the states kernel running and finally, the zombie state.

Some state transitions can be controlled by the users, but not all. User can create a process. But the user has no control over when a process transitions to sleeping in memory to sleeping in swap, or ready to run in memory to ready to run in swap, etc. A process can make a system call to transition itself to kernel running state. But it has no control over when it will return from kernel mode. Finally, a process can exit whenever it wants, but that is not the only reason for exit to be called.

The process transitions follow a rigid model encoded in the kernel, reacting to events in a predictable way according to formulated rules (studied later). For example, no process can preempt another process executing in the kernel.

Two kernel data structures describe the state of a process: the process table entry and the u-area. The process table contains information that should be accessible to the kernel and the u-area contains the information that should be accessible to the process only when its running. Kernel allocates space for u-area only when creating a process. It does not need u-area for process table entries that do not have processes. For example, the process table entries that contain the information about the kernel context, do not have processes.

The fields in the process table are the following:

**State of the process**

1. Fields that allow the kernel to locate the process and its u-area in main memory or in secondary storage. This information is used 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 or when swapping the process. It contains a field that gives the size of the process so that the kernel knows how much space to allocate for the process.
2. Several user identifiers (user IDs or 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.
3. Event descriptor when the process is sleeping.
4. Scheduling parameters allow the kernel to determine the order in which processes move to the states kernel running and user running.
5. A signal fields enumerates the signals sent to a process but not yet handled.
6. Various timers give process execution time and kernel resource utilization. These are used for calculation of process scheduling priority. One field is a user-set timer used to send an alarm signal to a process.

**6.5.1 Allocating a Region**

* The kernel allocates a new region (algorithm allocreg) during fork, exec, and shmget (shared memory) system calls.
* The kernel contains a region table whose entries appear either on a free linked list or on an active linked list.
* When it allocates a region table entry, the kernel removes the first available entry from the free list, places it on the active list, locks the region, and marks its type (shared or private).
* With few exceptions, every process is associated with an executable file as a result of a prior exec call, and allocreg sets the inode field in the region table entry to point to the inode of the executable file.
* The inode identifies the region to the kernel so that other processes can share the region if desired.
* The kernel increments the inode reference count to prevent other processes from removing its contents when unlinking it.

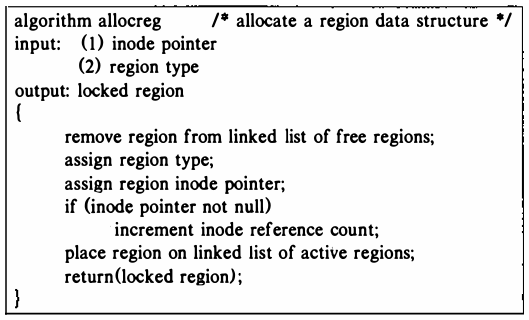


Figure 6.18. Algorithm for Allocating a Region

**what is context switch? explain the steps for context switch. 184**

The kernel permits a context switch under four circumstances:

1. when a process puts itself to sleep,

2. when it exits,

3. when it returns from a system call to user mode but is not the most eligible process to run, or

3. when it returns to user mode after the kernel completes handling an interrupt but is not the most eligible process to run.

The kernel ensures integrity and consistency of internal data structures by prohibiting arbitrary context switches.

It makes sure that the state of its data structures is consistent before it does a context switch: that is, all appropriate updates are done, that queues are properly linked, that appropriate locks are set to prevent intrusion by other processes, that no data structures are left unnecessarily locked, and so on.

For example, if the kernel allocates a buffer, reads a block in a file, and goes to sleep waiting for I/O transmission from the disk to complete, it keeps the buffer locked so that no other process can tamper with the buffer. But if a process executes the link system call, the kernel releases the lock of the first inode before locking the second inode to avoid deadlocks.

Sometimes the kernel has to switch tasks, like when a task finishes or goes to sleep. It also switches when a task isn't the most important one to make sure other important tasks get their turn. The way it switches tasks is similar to how it handles interrupts and system calls, except it brings back the context of a different task instead of the same one. The reasons for switching tasks don't matter, and the choice of which task to switch to is a policy decision.

Steps for a Context Switch:

1. Decide whether to do a context switch and whether a context switch is permissible now.

2. Save the context of the "old" process.

3. Find the "best" process to schedule for execution, using the process scheduling algorithm.

4. Restore its context.

The code for switching tasks in UNIX systems can be hard to understand because it might seem like it doesn't always return from functions or appears out of nowhere.

This happens because the kernel saves one task's context but keeps executing in the current context, then later goes back to the saved context.

To tell if the kernel is switching to a new task or staying with the current one, the return values of key functions might change, or the program counter might be set artificially.

Figure 6.16 shows how a context switch works. The function save\_context saves details about the current task and gives back the number 1. One of the things the kernel saves is where the program was (in save\_context), and it also saves 0 to use later as a return value in register 0 from save\_context. The kernel keeps working with the current task (A) while choosing another task (B) to run. It then calls resume\_context to switch to the new task's details (B). Now, the system is working on task B, and task A isn't running anymore, but its details are saved for later (that's why it says "never gets here" in the figure). Later, when the system needs to switch back to task A (unless it's exiting), the kernel sets the program to where task A left off when it called save\_context, and it puts 0 back into register 0. So, even though the kernel was working on something else, it goes back to working on task A from where it left off. Finally, task A finishes save\_context and goes back to work after the "resuming process executes from here" line.

if (save\_context()): /\* save context of executing process \*/

{

/\* pick another process to run \*/

-

-

-

resume\_context(new\_process);

/\* never gets here ! \*/

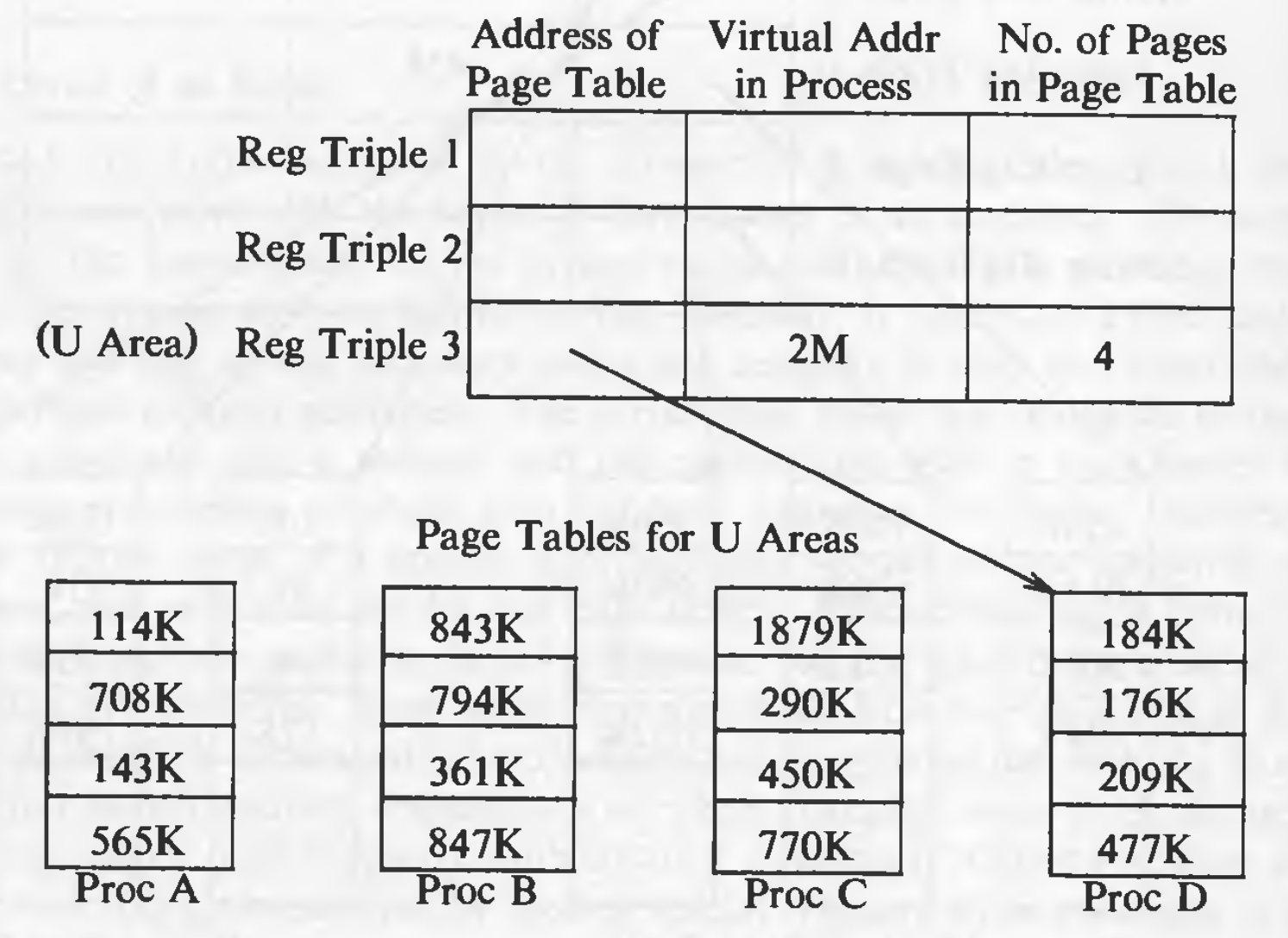
}

/\* resuming process executes from here \*/

**The U Area**

Even if every process has a u-area, the kernel accesses them through its u variable. It needs to access only one u-area at a time, of the currently executing process. The kernel knows where the page table entry of the u-area is located, therefore, when a process is scheduled, the physical address of its u-area is loaded into kernel page tables.

Example:



The first two register triples point to text and data and the third triple refers to the u-area of currently executing process (in this case, process D). When a context switch happens, the entry in this fields changes and points to the u-area of the newly scheduled process. Entries 1 and 2 do not change as all the process share the kernel text and data.

The u-area contains these fields (some are covered previously as well) :

1. A pointer in the process table identifies the entry that corresponds to the u-area.
2. The real and effective user IDs determine various privileges allowed the process, such as file access rights.
3. Timer fields record the time the process spent executing in user mode and in kernel mode.
4. An array indicates how the process wishes to react to signals.
5. The control terminal field identifies the "login terminal" associated with the process, if one exists.
6. An error field records errors encountered during a system call.
7. A return value field contains the result of system calls.
8. 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.
9. The current directory and current root describe the file system environment of the process.
10. The user file descriptor table records the files the process has open.
11. Limit fields restrict the size of a process and the size of a file it can write.
12. A permission modes field masks mode settings on files the process creates.