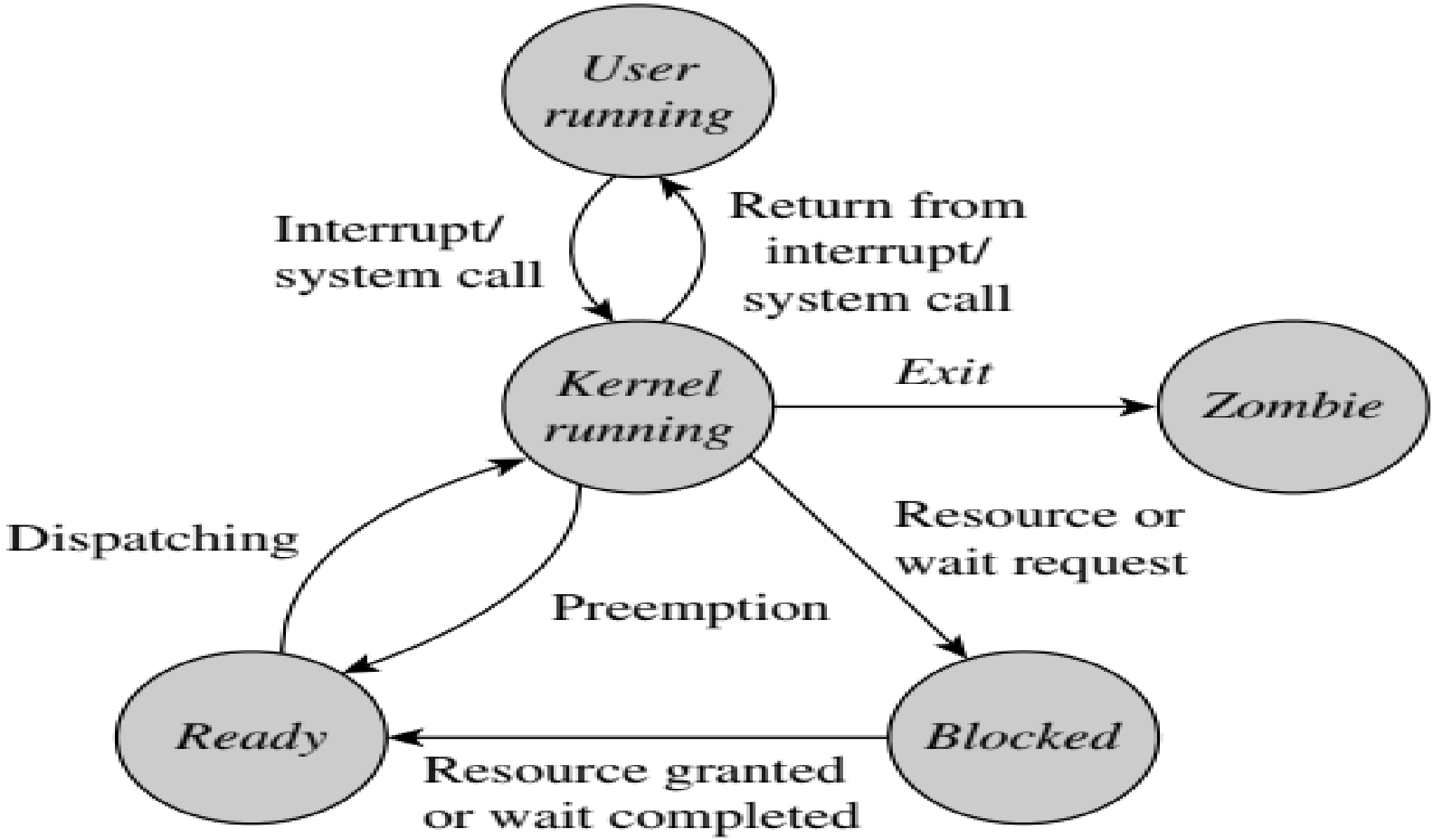


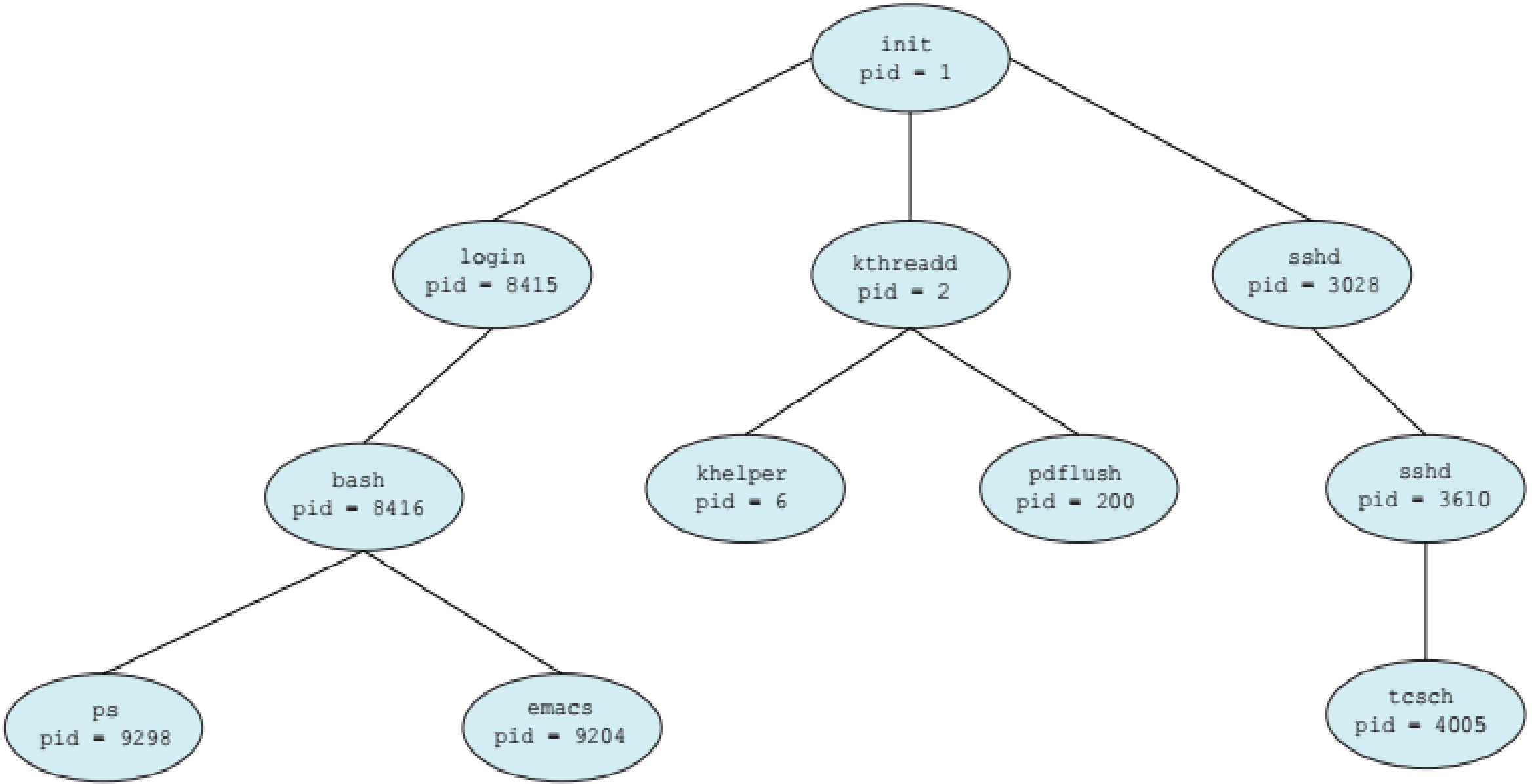
Process state transitions in Unix.

Process States and State Transitions

- There is one conceptual difference between the process model described earlier and that used in Unix.
- In the previous model, a process in the *running* state is put in the *ready* state the moment its execution is interrupted.
- A system process then handles the event that caused the interrupt.
- If the running process had itself caused a software interrupt by executing an <*SI_instrn*>, its state may further change to *blocked* if its request cannot be granted immediately.
- In this model a user process executes only user code; it does not need any special privileges.
- A system process may have to use privileged instructions like I/O initiation and setting of memory protection information, so the system process executes with the CPU in the kernel mode.

- Processes behave differently in the Unix model.
- When a process makes a system call, the process itself proceeds to execute the kernel code meant to handle the system call.
- To ensure that it has the necessary privileges, it needs to execute with the CPU in the kernel mode.
- A mode change is thus necessary every time a system call is made.
- The opposite mode change is necessary after processing a system call.
- Similar mode changes are needed when a process starts executing the interrupt servicing code in the kernel because of an interrupt, and when it returns after servicing an interrupt.
- Unix uses two distinct *running* states.
- These states are called *user running* and *kernel running* states.
- A user process executes user code while in the *user running* state, and kernel code while in the *kernel running* state.

- It makes the transition from *user running* to *kernel running* when it makes a system call, or when an interrupt occurs.
- It may get blocked while in the *kernel running* state because of an I/O operation or non availability of a resource.
- When the I/O operation completes or its resource request is granted, the process returns to the *kernel running* state and completes the execution of the kernel code that it was executing.
- It now leaves the kernel mode and returns to the user mode.
- Accordingly, its state is changed from *kernel running* to *user running*.
- Because of this arrangement, a process does not get blocked or preempted in the *user running* state—it first makes a transition to the *kernel running* state and then gets blocked or preempted.
- In fact, *user running* → *kernel running* is the only transition out of the *user running* state.
- Process termination occurs when a process is in the *kernel running* state. This happens because the process executes the system call *exit* while in the *user running* state.
- This call changes its state to *kernel running*. The process actually terminates and becomes a *zombie* process as a result of processing this call.


init
pid = 1
login
pid = 8415
kthreadd
pid = 2
sshd
pid = 3028
bash
pid = 8416
khelper
pid = 6
pdflush
pid = 200
sshd
pid = 3610
ps
pid = 9298
emacs
pid = 9204
tcsch
pid = 4005

Process Creation

- Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique **process identifier** (or **pid**), which is typically an integer number.
- The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel.
- The init process (which always has a pid of 1) serves as the root parent process for all user processes.
- Once the system has booted, the init process can also create various user processes, such as a web or print server, an ssh server, and the like.
- In Figure there are two children of init—kthreadd and sshd.
- The kthreadd process is responsible for creating additional processes that perform tasks on behalf of the kernel
- The sshd process is responsible for managing clients that connect to the system by using ssh (which is short for ***secure shell***).
- The login process is responsible for managing clients that directly log onto the system.

- When a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task.
- A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.
- The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children.
- Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes.
- In addition to supplying various physical and logical resources, the parent process may pass along initialization data (input) to the child process.
- Alternatively, some operating systems pass resources to child processes.
- When a process creates a new process, two possibilities for execution exist:
  1. The parent continues to execute concurrently with its children.
  2. The parent waits until some or all of its children have terminated.

- There are also two address-space possibilities for the new process:
  1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
  2. The child process has a new program loaded into it.

Process Termination

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call.
- At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call).
- All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.
- Termination can occur in other circumstances as well. Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs.
- Note that a parent needs to know the identities of its children if it is to terminate them.
- Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.
- A parent may terminate the execution of one of its children for a variety of reasons, such as these:
  - The child has exceeded its usage of some of the resources that it has been allocated.
  - The task assigned to the child is no longer required.
  - The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Process Synchronization in Unix

- Unix system V provides a kernel-level implementation of semaphores.
- The name of a semaphore is called a *key*.
- The key is actually associated with an array of semaphores, and individual semaphores in the array are distinguished with the help of subscripts.
- Processes share a semaphore by using the same key.
- A process wishing to use a semaphore obtains access to it by making a *semget* system call with a key as a parameter.
- If a semaphore array with matching key already exists, the kernel makes that array accessible to the process making the *semget* call; otherwise, it creates a new semaphore array, assigns the key to it and makes it accessible to the process.
- The kernel provides a single system call *semop* for *wait* and *signal* operations.

Scheduling in Unix

- Unix is a pure time-sharing operating system.
- It uses a multilevel adaptive scheduling policy in which process priorities are varied to ensure good system performance and also to provide good user service.
- Processes are allocated numerical priorities, where a larger numerical value implies a lower effective priority.
- In Unix 4.3 BSD, the priorities are in the range 0 to 127.
- Processes in the user mode have priorities between 50 and 127, while those in the kernel mode have priorities between 0 and 49.
- When a process is blocked in a system call, its priority is changed to a value in the range 0–49, depending on the cause of blocking.
- When it becomes active again, it executes the remainder of the system call with this priority.
- This arrangement ensures that the process would be scheduled as soon as possible, complete the task it was performing in the kernel mode and release kernel resources.
- When it exits the kernel mode, its priority reverts to its previous value, which was in the range 50–127.

- Unix uses the following formula to vary the priority of a process:
  Process priority = base priority for user processes + *f (*CPU time used recently*)* + nice value (i**)**
- The scheduler maintains the CPU time used by a process in its process table entry.
- This field is initialized to 0.
- The real-time clock raises an interrupt 60 times a second, and the clock handler increments the count in the CPU usage field of the running process.
- The scheduler re computes process priorities every second in a loop.
- For each process, it divides the value in the CPU usage field by 2, stores it back, and also uses it as the value of *f*.
- A large numerical value implies a lower effective priority, so the second factor in Eq. (i) lowers the priority of a process.
- The division by 2 ensures that the effect of CPU time used by a process *decays*; i.e., it wears off over a period of time, to avoid the problem of starvation faced in the least completed next (LCN) policy.
- A process can vary its own priority through the last factor in Eq. (i).
- The system call "*nice*(*<priority value>*);" sets the *nice value* of a user process.
- It takes a zero or positive value as its argument. Thus, a process can only decrease its effective priority to be nice to other processes. It would typically do this when it enters a CPU-bound phase.

Fair Share Scheduling

- To ensure a fair share of CPU time to groups of processes, Unix schedulers add the term *f* (CPU time used by processes in the group) to Eq. (i).
- Thus, priorities of all processes in a group reduce when any of them consumes CPU time.
- This feature ensures that processes of a group would receive favored treatment if none of them has consumed much CPU time recently.
- The effect of the new factor also decays over time.