MODULE 2

PROCESS MANAGEMENT

PROCESS

- Current-day computer systems allow multiple programs to be loaded into memory and executed concurrently.
- **Process is a program in execution.**
- **A process is the unit of work** in a modern time-sharing system.
- A system therefore consists of a collection of processes: **OS processes** executing system code **and user processes** executing user code.
- All these processes can **execute concurrently** with the CPU multiplexed among them.
- By **switching the CPU between processes**, OS can make the computer more productive.
- A process needs a program code, which is known as the **text section**.
- It also includes the current activity, as represented by the **value of the program counter** and the **contents of the processor's registers**.
- A process generally also includes the **process stack**, which contains temporary data (such as function parameters, return addresses, and local variables), and a **data section**, which contains global variables.
- A process may also include a **heap**, which is memory that is dynamically allocated during process run time.
- A **program is a *passive* entity**, such as a file containing a list of instructions stored on disk (often called an executable file); whereas a **process is an *active* entity**, with a program

counter specifying the next instruction to execute and a set of associated resources.

- A program becomes a process when an executable file is loaded into memory.
- **Two common techniques** for loading executable files are **double-clicking an icon** representing the executable file and **entering the name of the executable file** on the command line

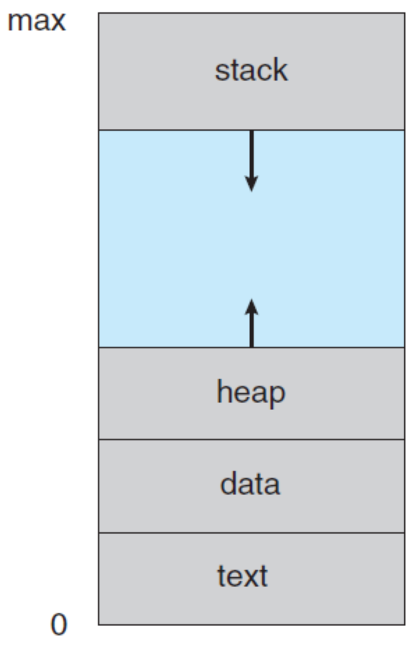

Figure 3.1 Process in memory.

- Two processes may be associated with the same program
- Eg: several users may be running different copies of the mail program, or the same user may invoke many copies of the Web browser program.
- Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary.

PROCESS STATES

- As a process executes, it changes its state.
- The **state** of a process is defined as the **current activity** of that process.
- Each process may be in one of the following states:
    1. **New**: The process is being created.
    2. **Running**: Instructions are being executed.
    3. **Waiting**: The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
    4. **Ready**: The process is waiting to be assigned to a processor.
    5. **Terminated**: The process has finished execution.
- Only one process can be **running** on any processor at any instant.
- Many processes may be **ready and waiting**.

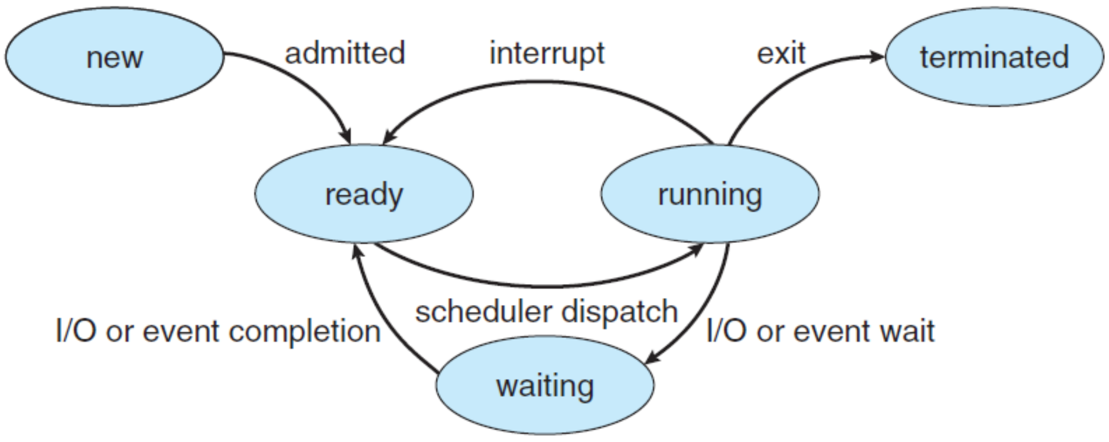

Figure 3.2 Diagram of process state.

Transitions are:

- **Admitted**: new to ready
- **Schedule or dispatch**: ready to running
- **Interrupt**: running to ready
- **Wait** (by I/O or event): running to waiting

- **I/O or event completion**: waiting to ready
- **Exit**: running to terminated

PROCESS CONTROL BLOCK (PCB)

- Each process in the OS is represented by a Process Control Block (PCB)
- Also called **task control block**
- PCB stores all information regarding a process

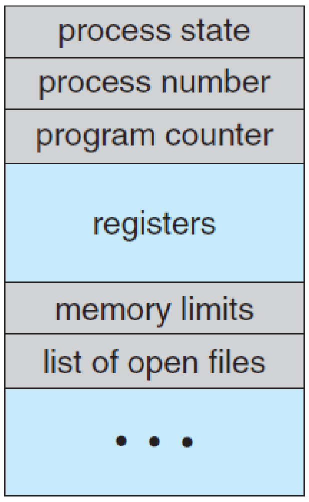

Figure 3.3 Process control block (PCB).

- **Process state**: The state may be new, ready, running, waiting, halted, and so on.
- **Process number**: Specific number to identify the process
- **Program counter**: The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers**: The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, etc
- **Memory-management information**: This information may include such information as the value of the base and

limit registers, the page tables, or the segment tables, depending on the memory system used by OS

- **I/O status information**: This information includes the list of I/O devices allocated to the process, a list of open files, and so on.
- **CPU-scheduling information**: This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Accounting information**: This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

- PCBs are used in **CPU switching** from process to process as shown below:

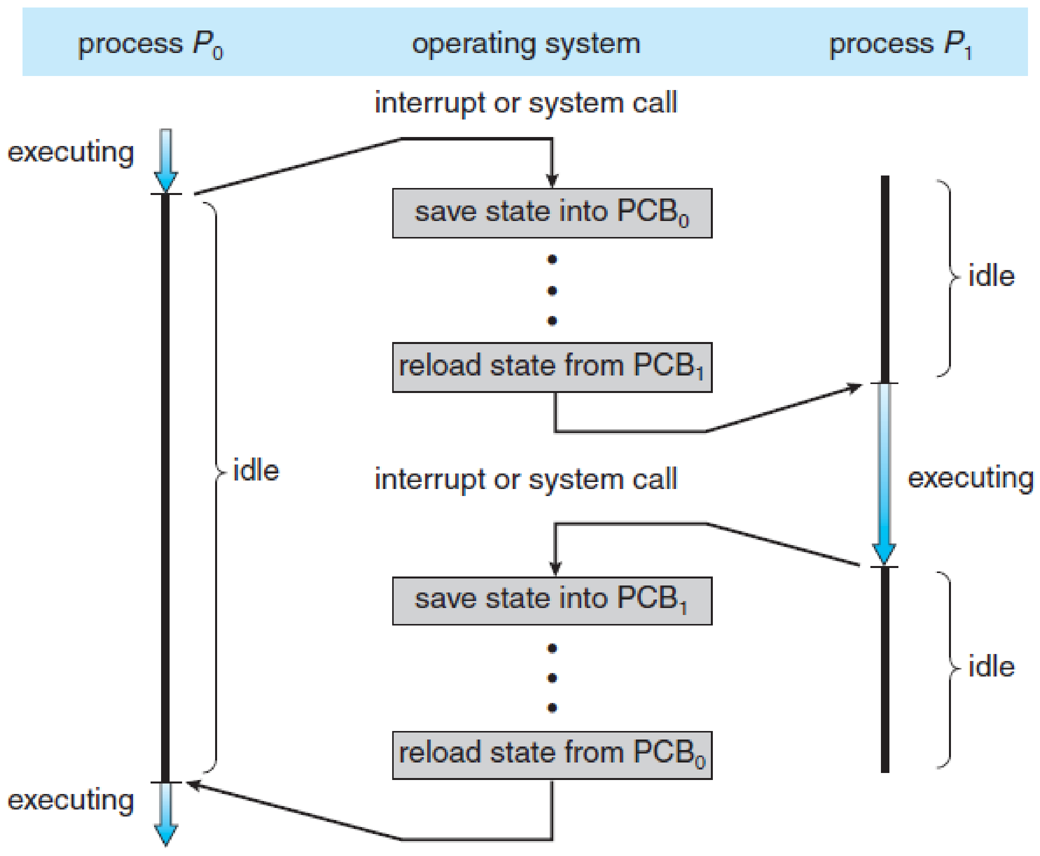

Figure 3.4 Diagram showing CPU switch from process to process.

THREADS

- Usually, a process is a program that performs a single **thread** of execution.
- Many modern OS have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time.
- For example, when a process is running a word-processor program, if a single thread of instructions is being executed, this single thread of control allows the process to perform only one task at one time. The user cannot simultaneously type in characters and run the spell checker within the same process.
- **A *thread* is a single sequence within a process that can be managed independently.**
- **A *thread* is the smallest unit of processing that can be performed in an *OS*.**
- On a system that supports threads, the PCB is expanded to include information for each thread. Other changes throughout the system are also needed to support threads.

PROCESS SCHEDULING

- The objective of **multiprogramming** is to have some process running at all times, to maximize CPU utilization.
- The objective of **time sharing** is to switch the CPU among processes so frequently that users can interact with each program while it is running.

- To meet these objectives, the **process scheduler selects an available process** (possibly from a set of several available processes) for program execution on the CPU.
- For a single-processor system, there will never be more than one running process.
- If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

Scheduling Queues

- As processes enter the system, they are put into a job queue, which consists of all processes in the system.
- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**.
- This queue is generally stored as a **linked list**. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.
- The system also includes other queues.
- The list of processes waiting for a particular I/O device is called a **device queue**.
- Each device has its own device queue.

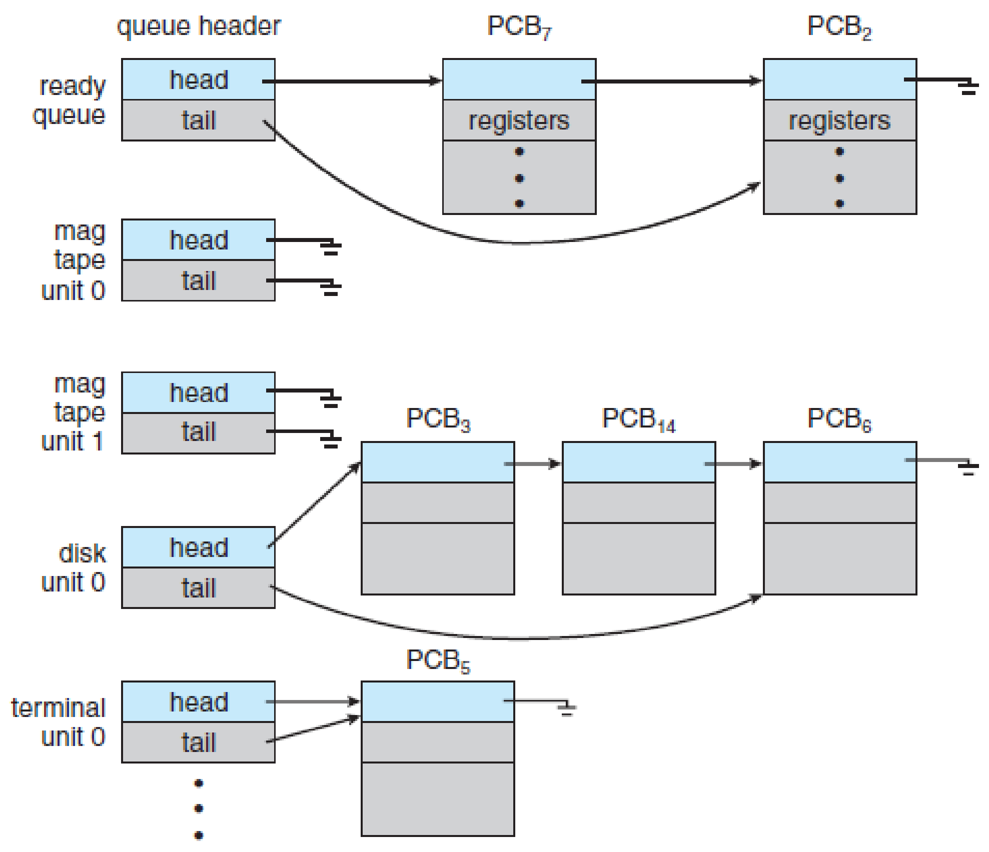

Figure 3.5 The ready queue and various I/O device queues.

- A common representation of process scheduling is a **queueing diagram**
- **Rectangular box represents a queue.**
- Two types of queues are present: the ready queue and a set of device queues.
- The **circles represent the resources** that serve the queues, and the **arrows indicate the flow of processes** in the system.

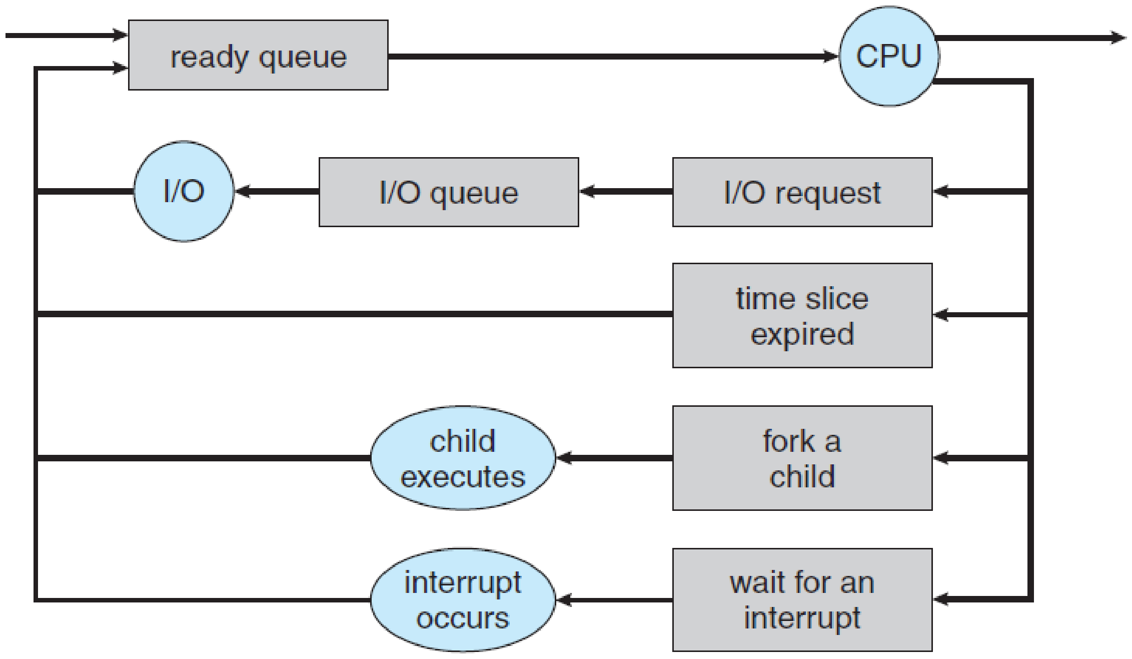

Figure 3.6 Queueing-diagram representation of process scheduling.

- A new process is initially put in the ready queue. It waits there until it is selected for execution, or is **dispatched**.
- Once the process is allocated the CPU and is executing, one of several events could occur:
  - Issue an I/0 request and then be placed in an I/0 queue.
  - Create a new child process and wait for child process's termination.
  - Be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.
  - Expire the time slot
- A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

Schedulers

- OS must select processes for execution from scheduling queues in some fashion. The selection process is carried out by the appropriate **scheduler**.
- **3 types of schedulers**
    1. **Long-term schedulers**
    2. **Short-term schedulers**
    3. **Medium- term schedulers**
- Often, in a batch system, more processes are submitted than can be executed immediately. These processes are **spooled** to a mass-storage device (typically a disk), where they are kept for later execution.
- The **long-term scheduler, or job scheduler**, selects processes from this pool and loads them into memory for execution.
- The **short-term scheduler, or CPU scheduler**, selects from among the processes that are ready to execute and allocates the CPU to one of them.
- The primary distinction between these two schedulers lies in frequency of execution. The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/0 request. Often, the short-term scheduler executes at least once every 100 milliseconds. Because of the short time between executions, the short-term scheduler must be fast.
- The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next. The long-term scheduler controls the degree of multiprogramming (the number of processes in memory).

- If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.
- Thus, the long-term scheduler may need to be invoked only when a process leaves the system. Because of the longer interval between executions, the long-term scheduler can afford to take more time to decide which process should be selected for execution. It is important that the long-term scheduler make a careful selection.
- Processes can be divided into two types: **I/ 0 bound or CPU bound**.
- An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations.
- A CPU-bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations.
- It is important that the long-term scheduler select a good process mix of I/O-bound and CPU-bound processes.
- If all processes are I/O bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do.
- If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced.
- The system with the best performance will thus have a combination of CPU-bound and I/O-bound processes.
- On some systems, the long-term scheduler may be absent or minimal.
- For example, time-sharing systems such as UNIX and Microsoft Windows systems often have no long-term scheduler but simply put every new process in memory for the short-term scheduler.

- The stability of these systems depends either on a physical limitation (such as the number of available terminals) or on the self-adjusting nature of human users. If performance declines to unacceptable levels on a multiuser system, some users will simply quit.
- Some OS, such as time-sharing systems, may introduce an additional, intermediate level of scheduling: **Medium-term scheduler**

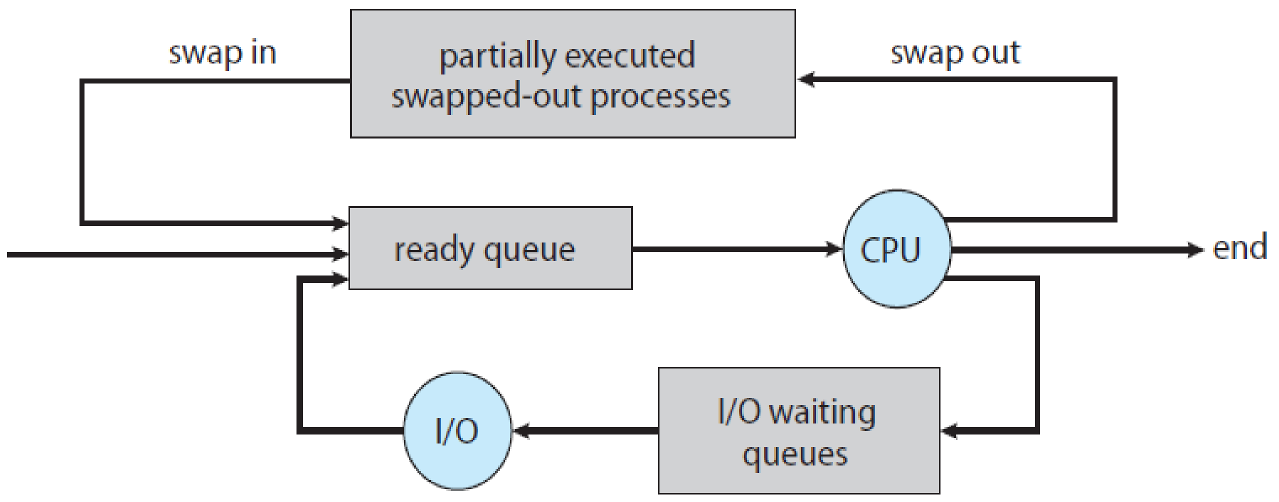

Figure 3.7 Addition of medium-term scheduling to the queueing diagram.

- The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes from memory (and from active contention for the CPU) and thus reduce the degree of multi-programming.
- Later, the process can be reintroduced into memory, and its execution can be continued where it left off. **This scheme is called swapping**. The process is swapped out, and is later swapped in, by the medium-term scheduler.
- Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.

Context Switch

- When an interrupt occurs, the system needs to save the current **context** of the process running on the CPU so that it can restore that context when its processing is done.
- Context is important during suspending the process and then resuming it.
- **The context is represented in the PCB of the process**; it includes the value of the CPU registers, the process state, and memory-management information.
- Perform a **state save** of the current state of the CPU and then a **state restore** to resume operations.
- Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process.
- **This task is known as a context switch**.
- When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
- Context-switch time is pure ***overhead***, because the system does no useful work while switching.
- Context switching time varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers), etc.
- Typical speeds are a few milliseconds.
- Context-switch times are highly dependent on hardware support.
- For instance, some processors provide multiple sets of registers.

- A context switch here simply requires changing the pointer to the current register set.
- If there are more active processes than there are register sets, the system needs to copy register data to and from memory, as normal.

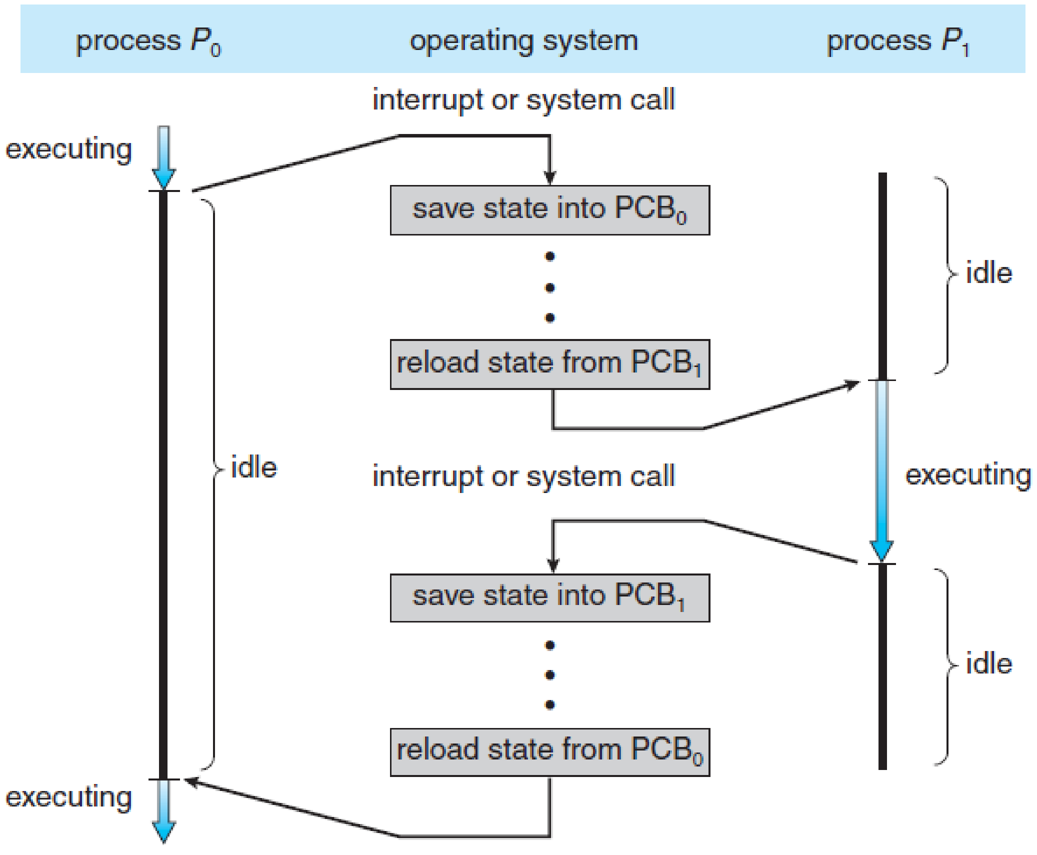

Figure 3.4 Diagram showing CPU switch from process to process.

OPERATIONS ON PROCESSES

1. PROCESS CREATION

- A process may create several new processes, via a create-process system call, during the course of execution.
- The creating process is called a **parent process**, and the new processes are called the **children** of that process.

- Each of these new processes may in turn create other processes, forming a **tree of processes.**
- Most OS identify processes according to a unique **process identifier (or pid),** which is typically an integer number.

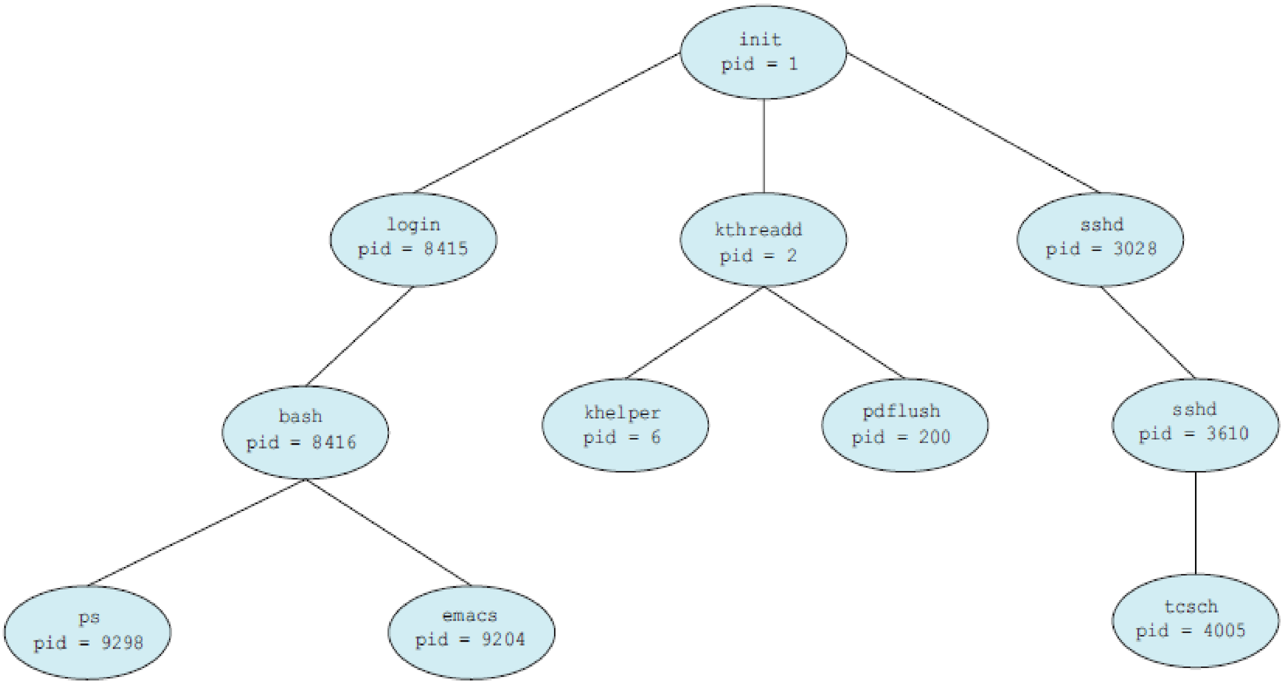

Figure 3.8 A tree of processes on a typical Linux system.

- 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
- 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
- The '*login*' process is responsible for managing clients that directly log onto the system.
- On UNIX, we can obtain a listing of processes by using the ***ps command***.

- For example, the command *'ps –el'* will list complete information for all processes currently active in the system.
- A process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task.
- When a process creates a subprocess, that subprocess may be able to obtain its resources directly from OS, 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.
- When a process is created, initialization data (input) may be passed along by the parent process to the child process.
- For example, consider a process whose function is to display the contents of a file. When it is created, it will get, as an input from its parent process, the name of the file
- When a process creates a new process, two possibilities exist in terms of execution:
    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 possibilities in terms of the address space of the new process:
    1. The child process is a duplicate of the parent process (it has the same program and data)
    2. The child process has a new program loaded into it.
- **In UNIX, a new process is created by the fork() system call**.

- The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process.
- Both processes (the parent and the child) **continue parallel execution** at the instruction after the fork ()
- **The return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.**
- The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a wait() system call to move itself off the ready queue until the termination of the child.
- The child process inherits privileges and scheduling attributes from the parent, as well certain resources, such as open files.

```
main()
        {
        int pid;
        /* fork a child process */
        pid =fork();
        if (pid < 0)            //error
                {
                printf( "Fork Failed");
                }
        else if (pid == 0)          // child process
                {
                //functions of child process
                }
        else                        //parent process
                {
```

```
        /* parent will wait for the child to complete */
        wait () ;
        //functions of parent process
        }
}
```

Process creation in UNIX using fork() system call

- The **exec() system call** can be used after a fork() system call by one of the two processes to replace the process's memory space with a new program.
- The exec() system call destroys the memory image of the program containing the exec() system call and starts its execution.
- In this manner, the two processes can go their separate ways.

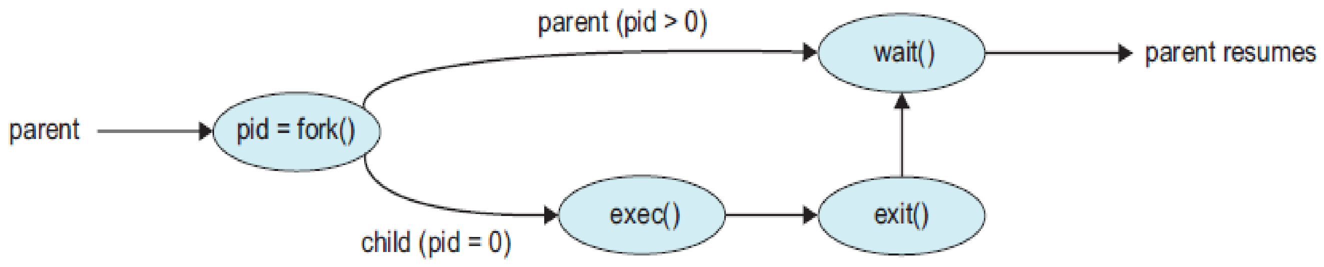

Figure 3.10 Process creation using the `fork()` system call.

- **In Windows, processes are created in the Win32 API using the CreateProcess () function which is similar to fork ()** in that a parent creates a new child process.
- Whereas fork() has the child process inheriting the address space of its parent, but CreateProcess() requires loading a specified program into the address space of the child process at process creation.

- Whereas fork() is passed no parameters, CreateProcess () expects no fewer than ten parameters.
- **Two important parameters** passed to CreateProcess () are instances of the **STARTUPINFO and PROCESS_INFORMATION structures.**
- STARTUPINFO specifies many properties of the new process, such as window size and appearance and handles to standard input and output files.
- The PROCESS_INFORMATION structure contains a handle and the identifiers to the newly created process and its thread.
- We invoke the **ZeroMemory ()** function to allocate memory for each of these structures before proceeding with CreateProcess ().
- The parent process waits for the child to complete by invoking the wait() system call. The equivalent of this in Windows is WaitForSingleObject()

<u>2. PROCESS TERMINATION</u>

- A process terminates when it finishes executing its final statement and asks OS 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 OS

- Otherwise, a process can forcefully cause the termination of another process via an appropriate system call (for example, **TerminateProcess () in Win32**).
- 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.
- A parent needs to know the identities of its children. 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:
  1. The child has exceeded its usage of some of the resources that it has been allocated.
  2. The task assigned to the child is no longer required.
  3. The parent is exiting, and the OS does not allow a child to continue if its parent terminates.
- Some systems do not allow a child to exist if its parent has terminated.
- In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated.
- This phenomenon, referred to as **cascading termination**, is normally initiated by OS.
- **In UNIX**, we can terminate a process by using the **exit() system call**
- Its parent process may wait for the termination of a child process by using the **wait() system call**.
- The wait() system call returns the process identifier of a terminated child so that the parent can tell which of its children has terminated.

- When a process terminates, its resources are deallocated by OS.
- However, its entry in the process table must remain there until the parent calls wait(), because the process table contains the process's exit status.
- A process that has terminated, but whose parent has not yet called wait(), is known as a **zombie process**.
- All processes transition to this state when they terminate, but generally they exist as zombies only briefly.
- Once the parent calls wait(), the process identifier of the zombie process and its entry in the process table are released.
- Now consider what would happen if a parent did not invoke wait() and instead terminated, thereby leaving its child processes as **orphans**.
- Linux and UNIX address this scenario by assigning the '*init'* process as the new parent to orphan processes.