CSE308 Operating Systems

Process Management

S.Rajarajan
SoC
SASTRA

- Early computers allowed **only one program** to be executed at a time
- This program had **complete control of the system** and had access to all the **system's resources**
- Current computers are **multi-programmed** and so allow **multiple programs** to be loaded and executed **concurrently**
- This resulted in the notion of a **process,** which is a program in execution
- A process is the unit of work in a modern **time-sharing system**

- Although main concern is the execution of user programs, OS also needs to take care of various **system tasks (e.g. Scheduling)**
- A system consists of a collection of processes:
  - **operating system processes** executing **system code**
  - and **user processes** executing **user code**
- Potentially, all these processes can execute **concurrently**, with the **CPU,** multiplexed among them
- By **switching the CPU** between processes, the operating system can make the computer more productive – **multiprogramming.**

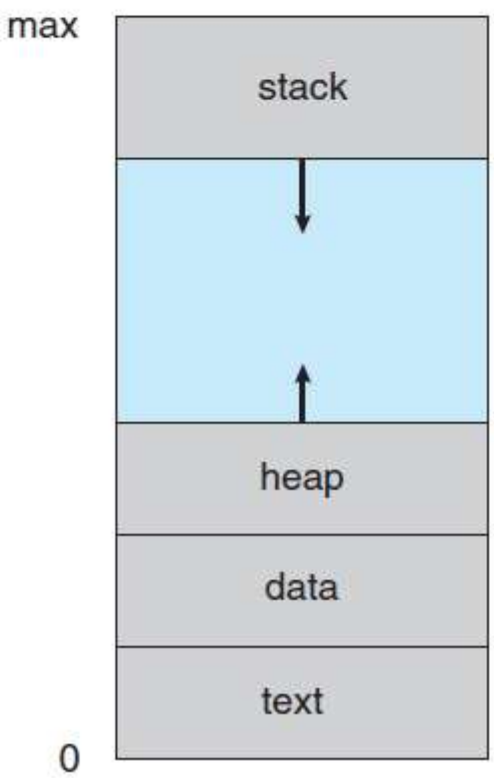
Process Concept

- A **batch** system executes **jobs**
- A **time-shared** system has **tasks**
- A user is able to run several programs at one time: a word processor, a Web browser, and an e-mail package.
- A **multi-programmed** system switches processes
- We call all of them processes

The Process

- A process is **more than the program code** (source code or text section)
- **It includes many other values like**
  - The current activity represented by the value of **program counter (PC)**
  - contents of the **processor's registers**
  - process **stack** which contains temporary data (such as **function parameters**, **return addresses**, and **local variables**)
  - **data section,** which contains **global variables**.
  - A process may also include a **heap,** which is memory that is **dynamically allocated** during process run time
- So a program by itself is not a process

Process in memory.



- 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 (**GUI**)
  - and entering the name of the executable file on the command line (as in **./a.out**).
- Even if two processes associated with the **same program**, they are considered **two separate processes**.
- Example parent and child processes created using **fork()**
- Although the **text sections** are equivalent, the **data, heap**, and **stack sections** vary.

- It is also common to have a process that **spawns** many processes as it runs (**parent-child**).
- A process itself can be an **execution environment** for other code.
- **Java programming environment** provides a example.
- An executable Java program is executed within the **Java virtual machine (JVM)**.
- The JVM **executes as a process** that **interprets** the **loaded Java code** and takes actions (**via native machine instructions**) on behalf of that code.
- For example, to run the compiled Java program Program.class, we would enter
  - java Program

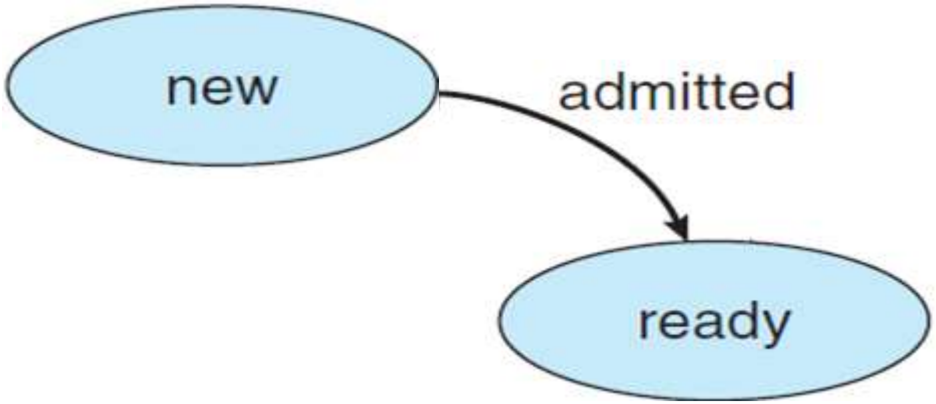
- The *java* command **runs the JVM as an ordinary process**, which in turns **executes the Java program** in the virtual machine

Process State

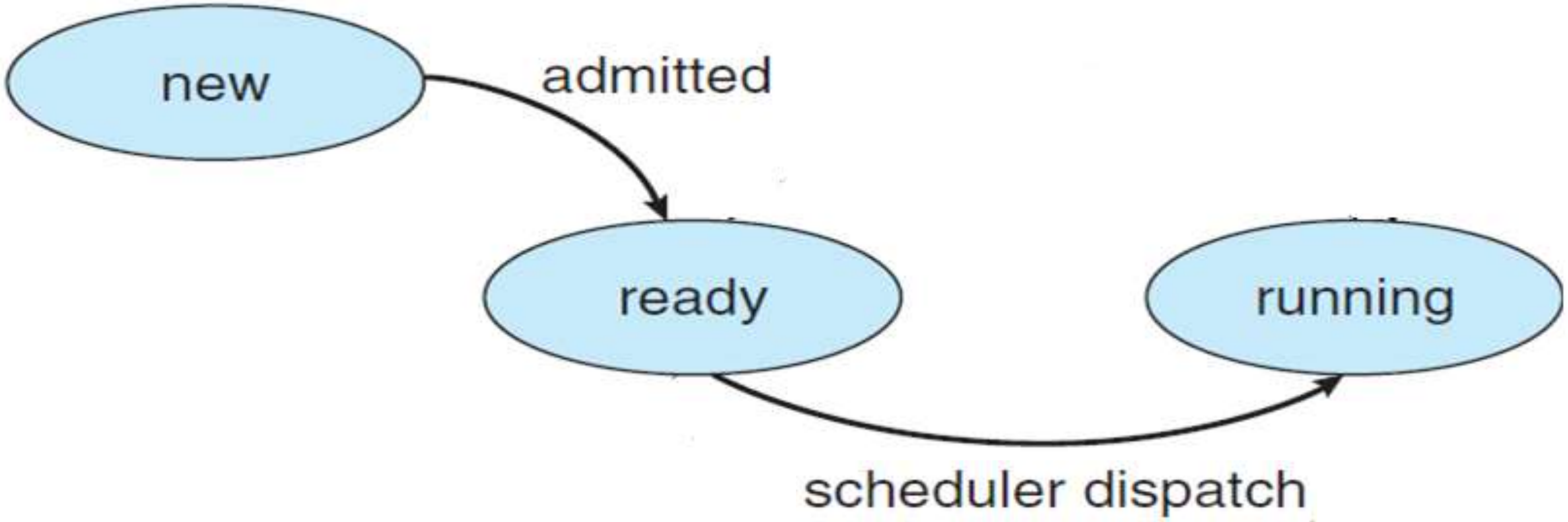
- As a process executes, it changes **state.**
- The state of a process is defined in part by the current activity of that process.
  - **New.** The process is being created.
  - **Running**. Instructions are being executed.
  - **Waiting.** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
  - **Ready.** The process is waiting to be assigned to a processor.
  - **Terminated.** The process has finished execution

New state – A program is submitted for execution e.g ./a.out

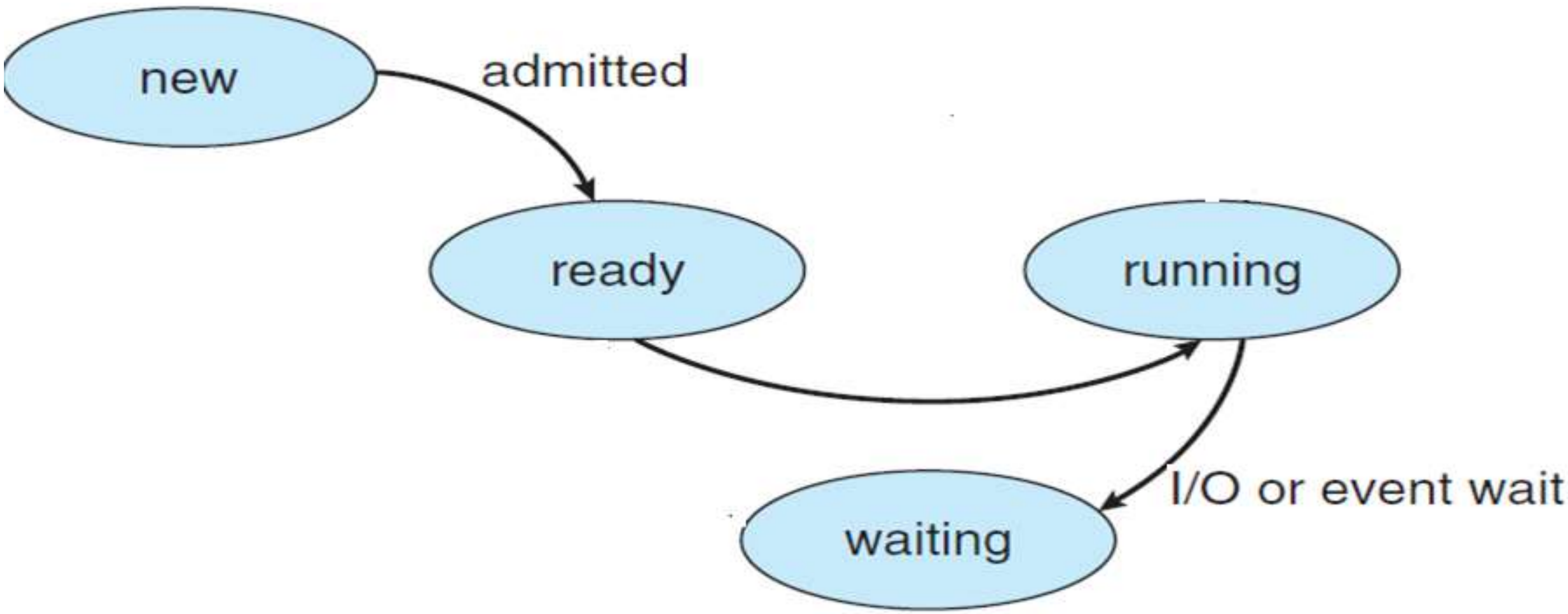
Ready state – New process is loaded into memory and queued for CPU



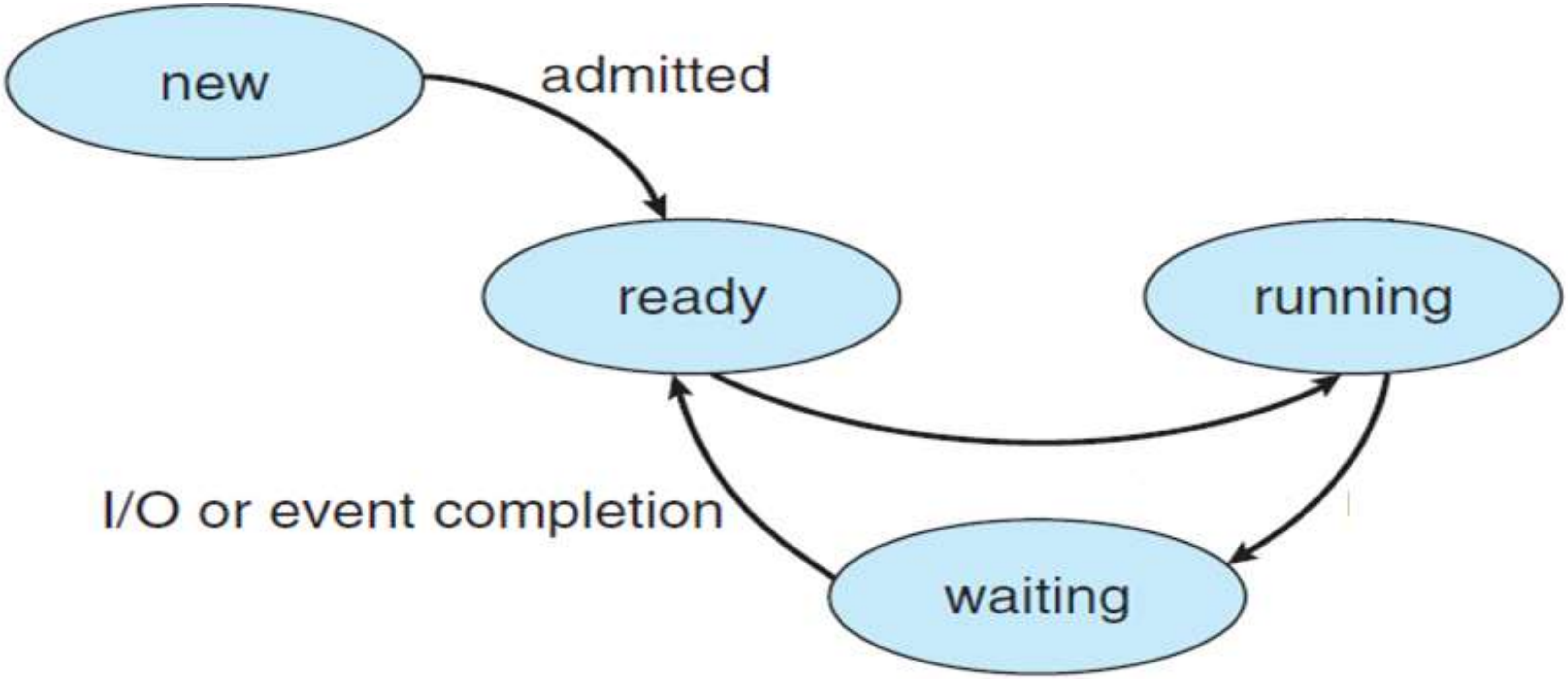
Running state – Process scheduled on CPU and executes



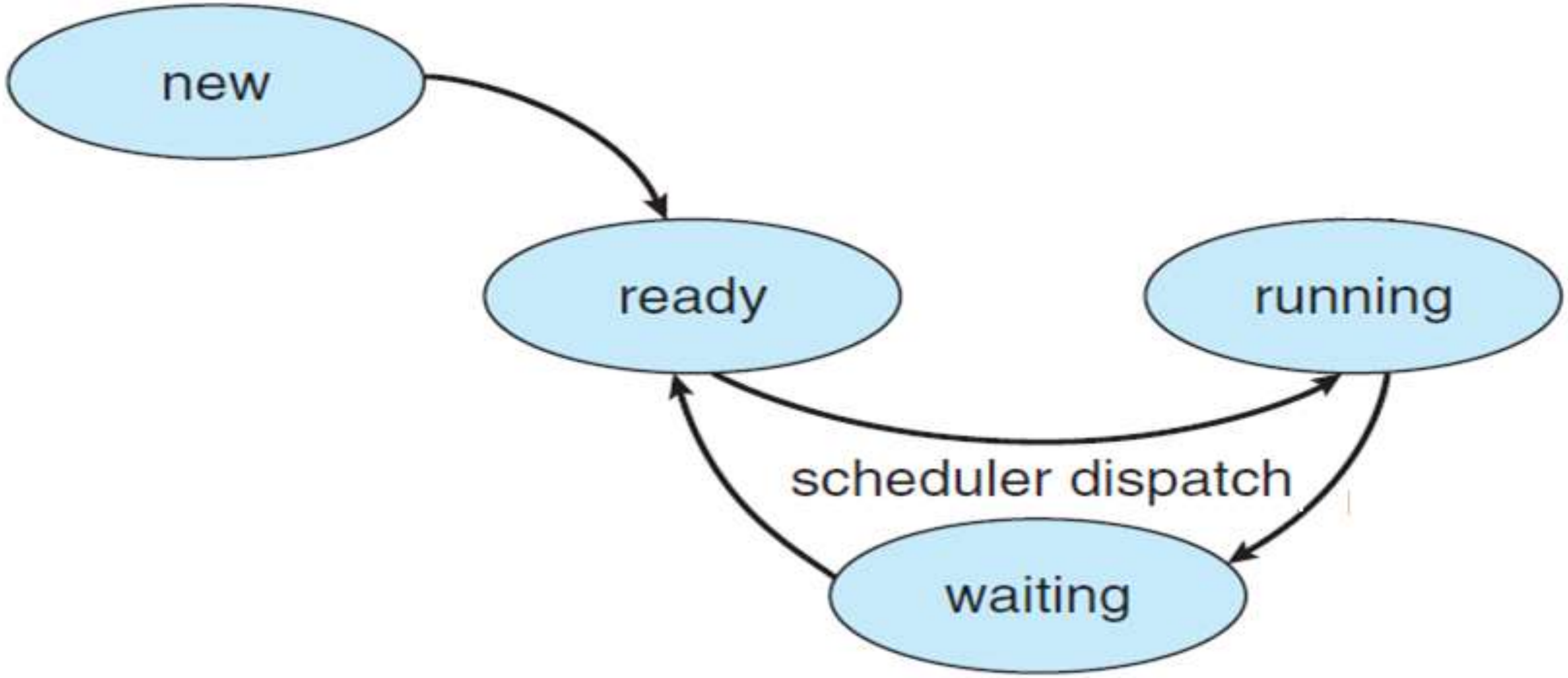
Waiting state – Process reaches an I/O operation and begins to wait for its completion



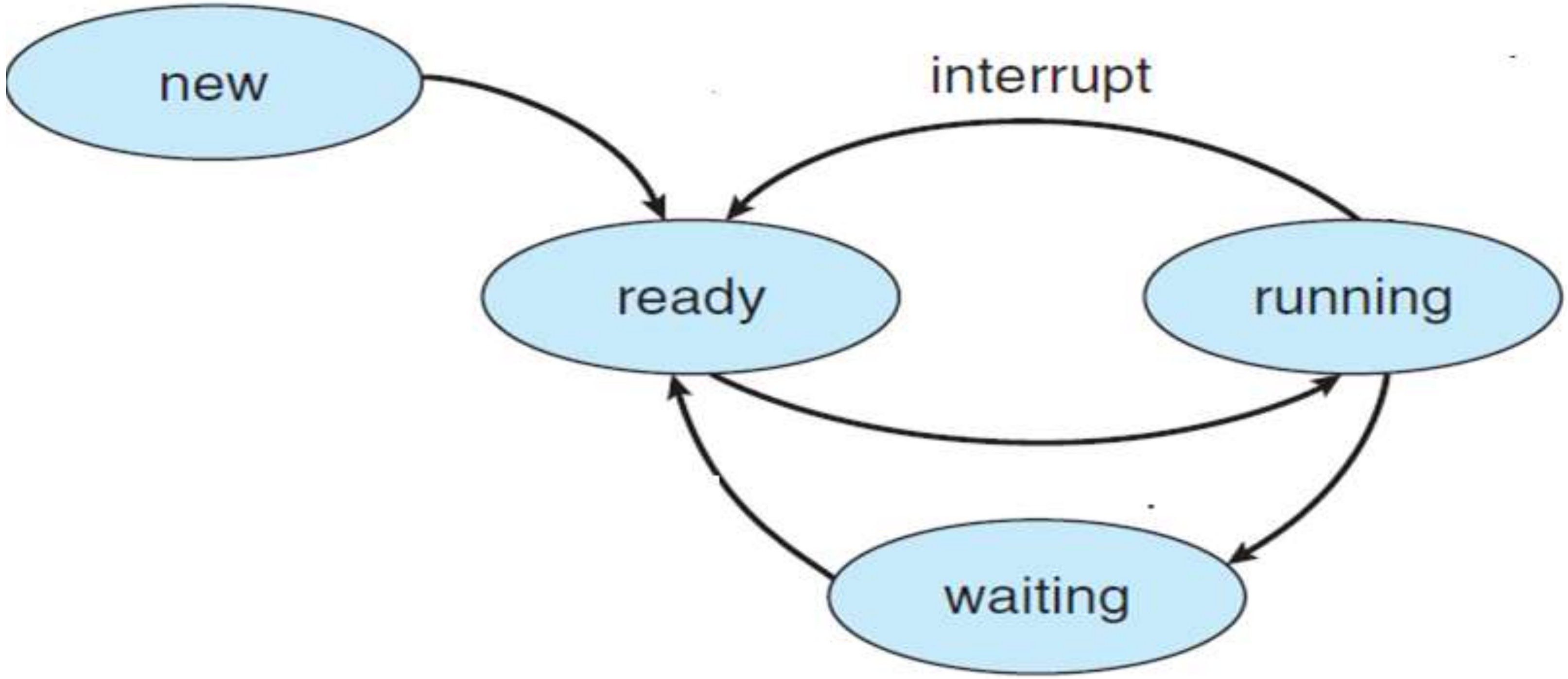
Ready state – Process completed I/O and rejoins ready queue



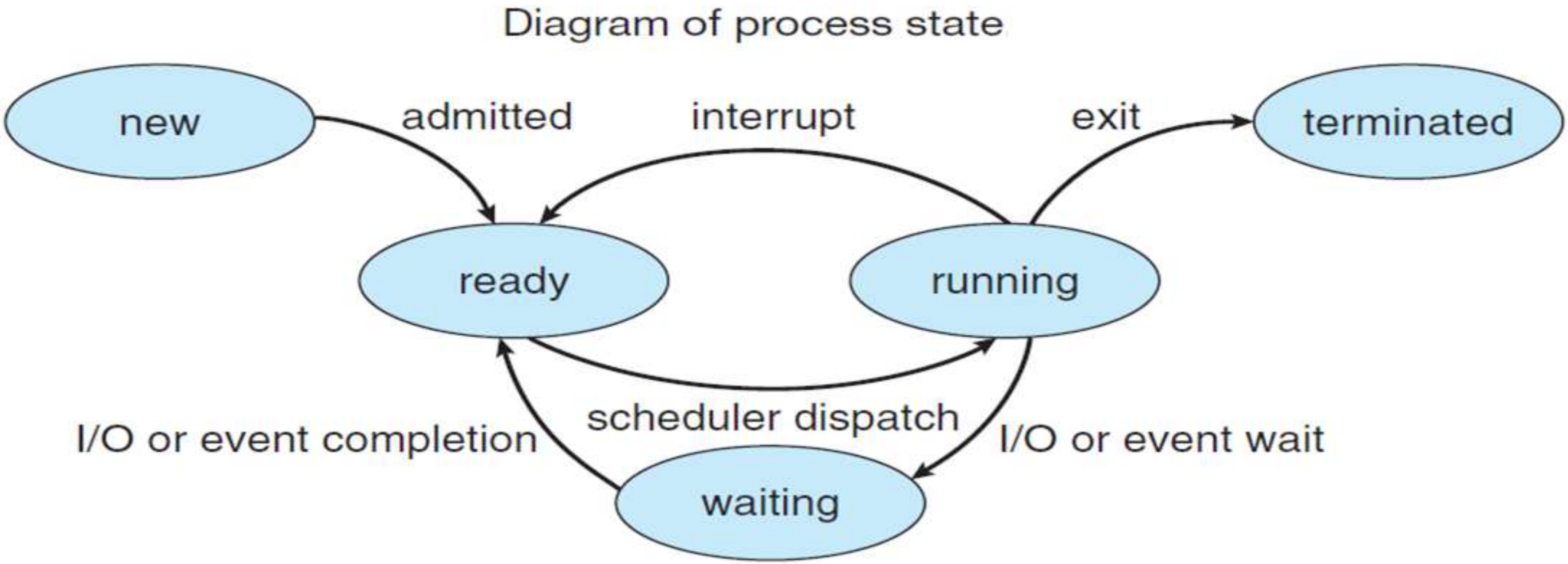
Running state – Process again scheduled on CPU and executes

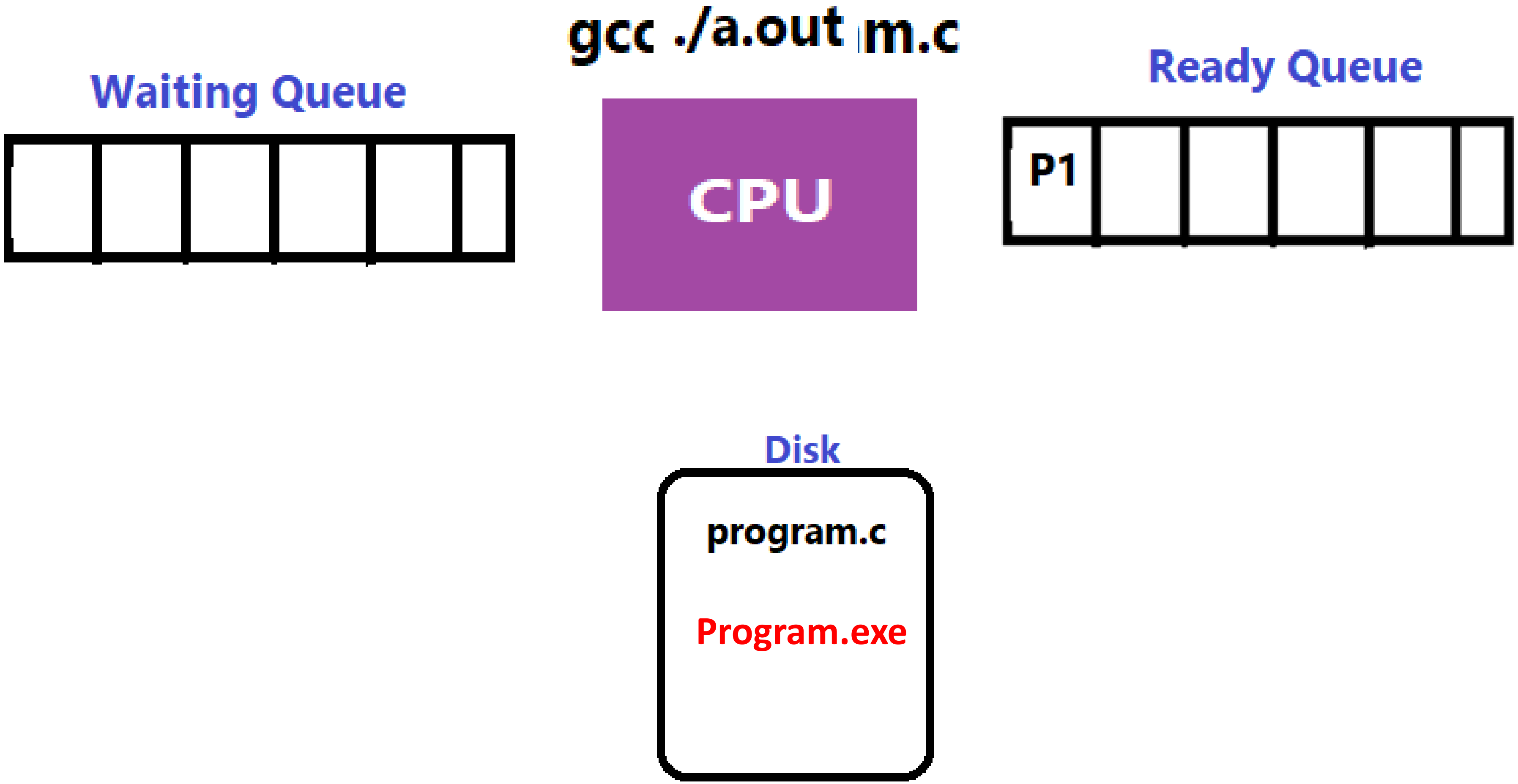


Ready state – Process pre-empted due to an interrupt and waits in ready queue

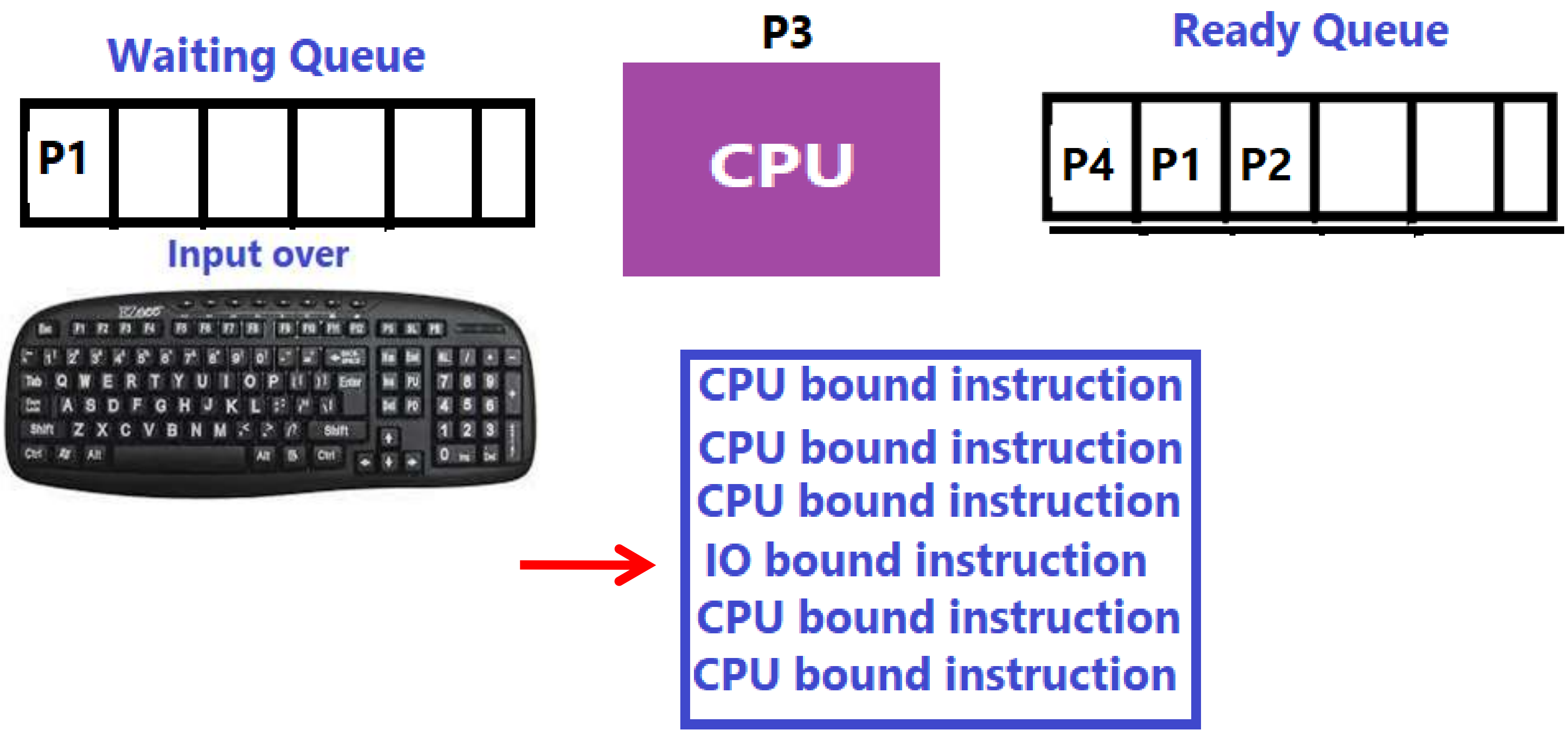


- **Terminated state** – Process finishes its execution or terminated for some illegal action or exeception




gcc ./a.out m.c
Waiting Queue
CPU
Ready Queue
P1
Disk
program.c
Program.exe

Time Quantum Over



Process Control Block

- Each process is represented by a **process control block (PCB)**—also called a **task control block.**
- It is a data structure that contains many pieces of information associated with a specific process, including the following:

- **Process state**. The state may be new, ready, running, waiting, halted, and so on.
- **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, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward (Figure 3.4).
- **CPU-scheduling information**. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters. (Chapter 6 describes process scheduling.)
- **Memory-management information**. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system (Chapter 8).
- **Accounting information**. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **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.

• In brief, the PCB serves as the **repository** for any information that may vary from process to process.
• The OS uses PCB as the means to manage processes.

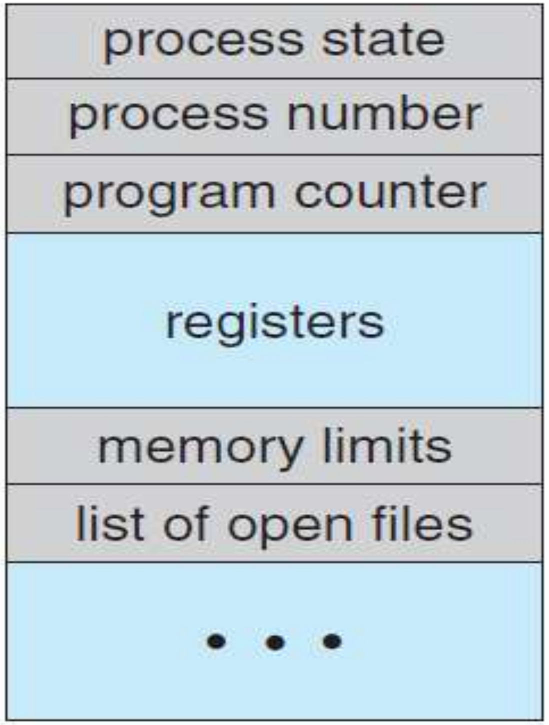


Figure 3.3 Process control block (PCB).

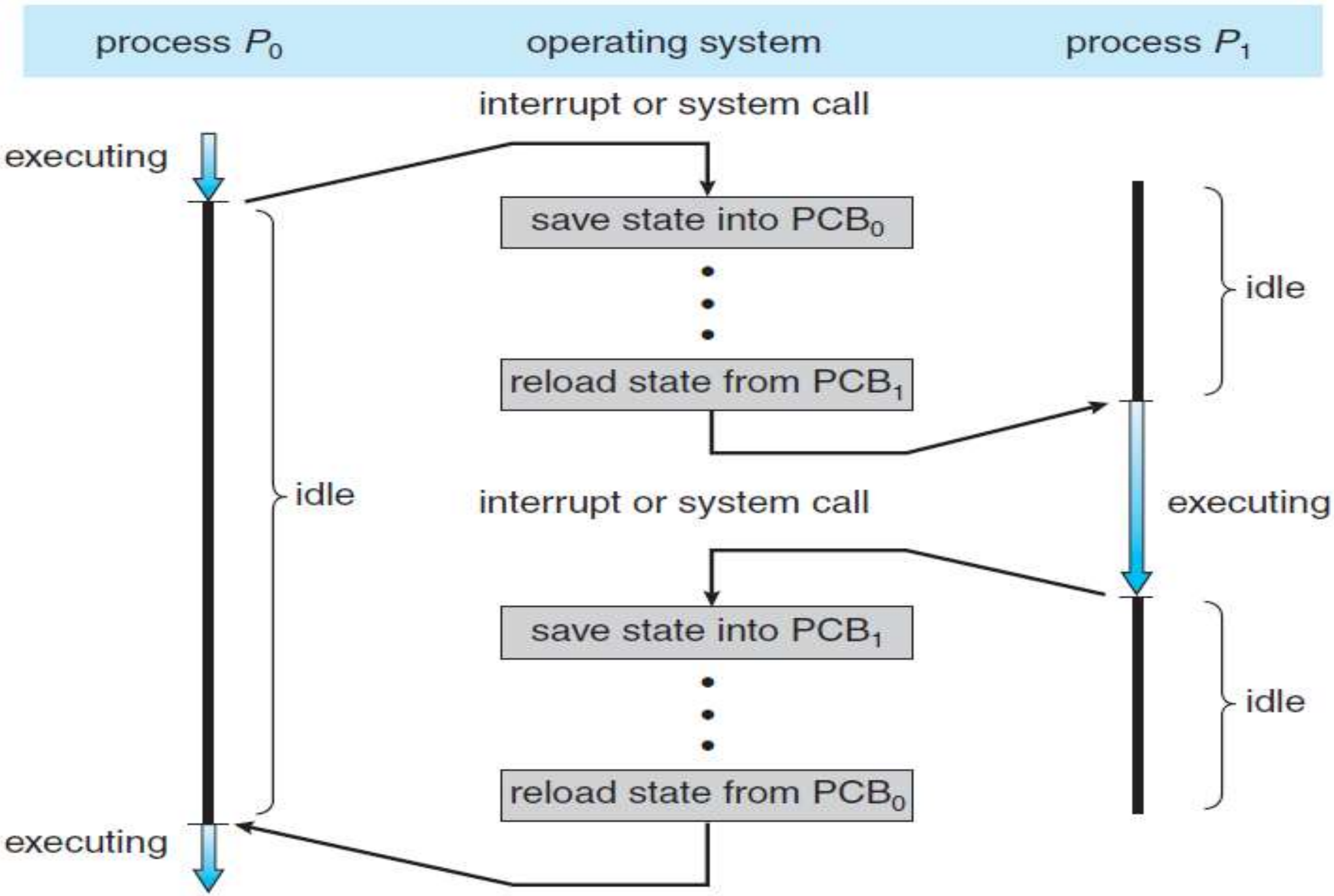


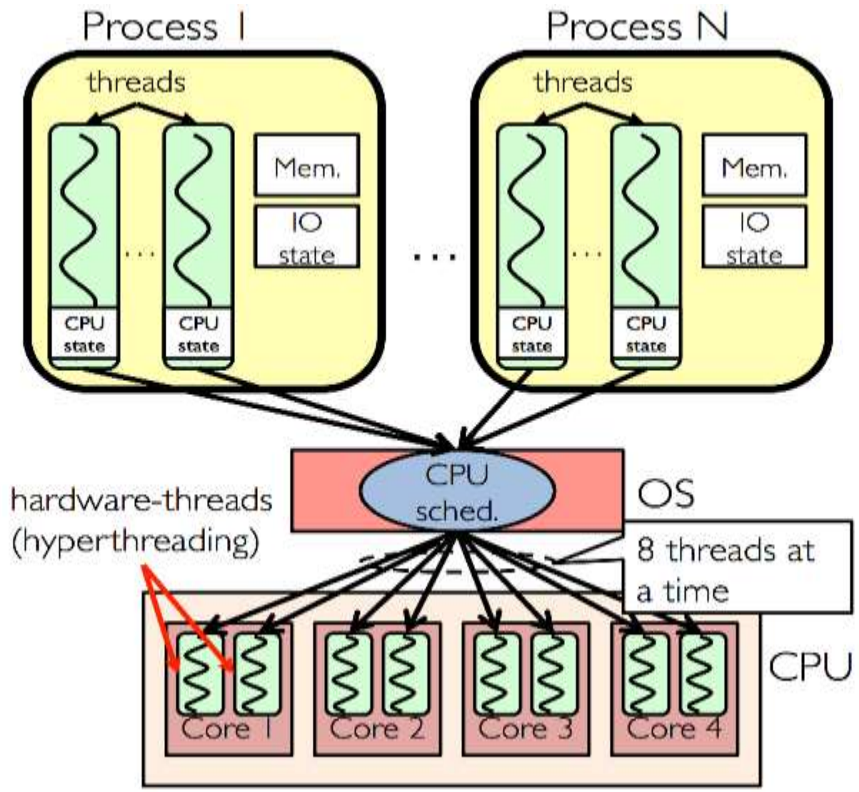
Figure 3.4 Diagram showing CPU switch from process to process.

Threads

- A process is a program that performs a single **thread of execution.**
- For example, when a process is running a **word-processor program**, a single thread of instructions is being executed.
- This single thread of control allows the process to perform **only one task** at a time.
- The user cannot simultaneously **type in characters** and run the **spell checker** within the same process, for example.

- Most modern operating systems 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.
- This feature is especially beneficial on **multi-core** systems, where **multiple threads** can run in **parallel**.

•On a system that supports threads, the **PCB** is **expanded** to include **information for each thread**.



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** execution on the CPU.
- For a **single-processor system**, there will not 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 assigned.

Scheduling Queues

- As processes enter the system, they are put into a **job queue, which consists** of all processes in the system.
- Processes that are residing in main memory and are ready to execute are kept on a queue 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**.
- When a process is allocated the CPU, it executes for a while and **eventually quits** or **interrupted** or **waits** for the occurrence of a particular event, such as the **completion of an I/O** request.
- It then enters into **waiting queue**
- Suppose the process makes an I/O request to a shared device, such as a **disk**
- Since there are many processes in the system, the **disk may be busy** with the I/O request of some other process.
- The process therefore may have to wait for the disk.
- The list of processes waiting for a particular I/O device is called a **device queue.**
- Each device has its **own device queue**

- A common representation of process scheduling is a **queuing diagram**
- Each **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.
- A **new process** is initially put in the **ready queue**.
- It waits there until it is selected for execution, or **dispatched.**

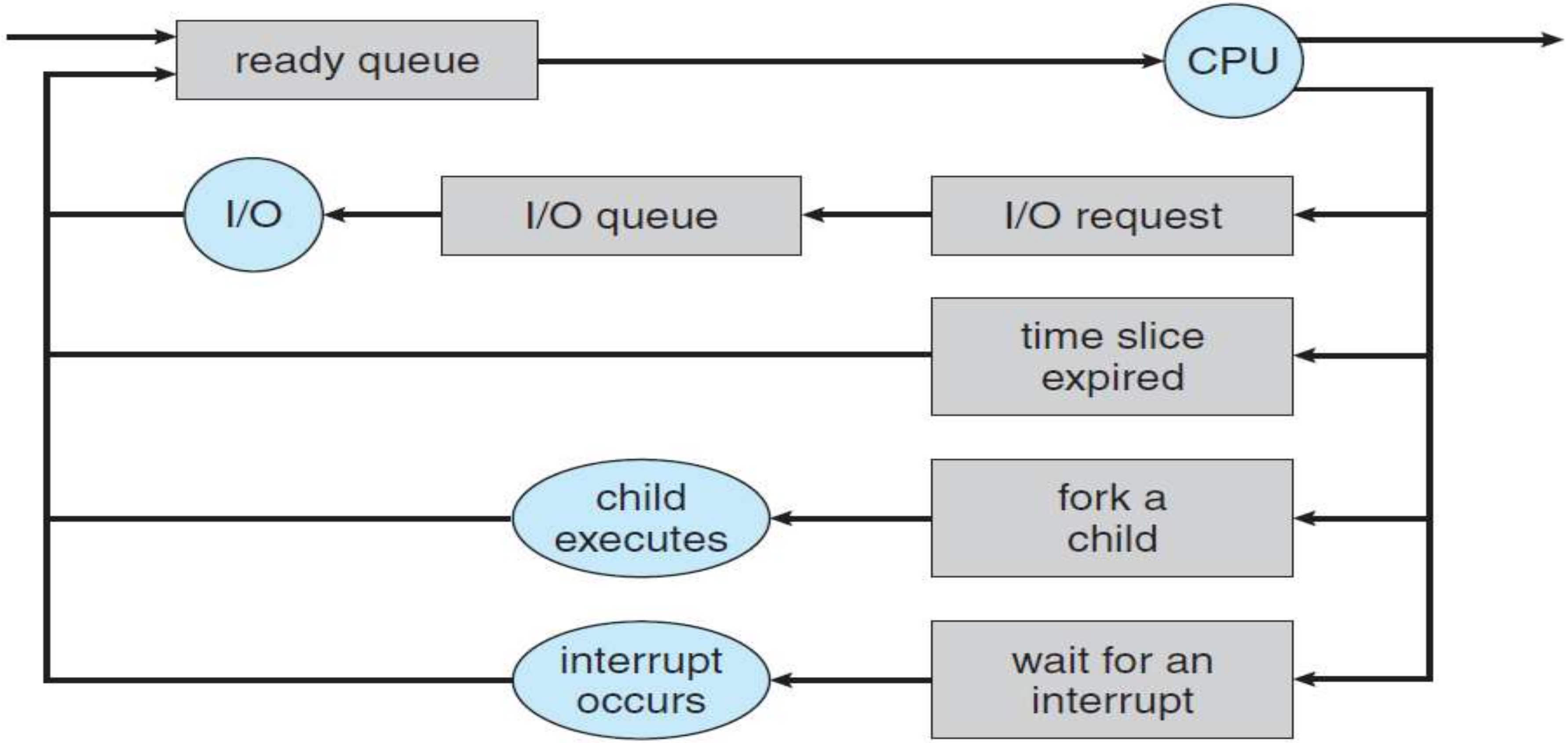


Figure 3.6 Queueing-diagram representation of process scheduling.

- Once the process is **allocated the CPU** and is **executing**, one of **several events** could occur:
  - The process could **issue an I/O** request and then be placed in an I/O queue.
  - The process could **create a new child process** and **wait** for the **child's termination**.
  - The process could be **removed forcibly** from the CPU, as a result of an **interrupt**, and be put back in the **ready queue**.
- 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

- A **process migrates** among the various scheduling **queues** throughout its lifetime.
- The operating system **must select**, for scheduling purposes, processes from these queues in some fashion.
- The **selection process** is carried out by the appropriate scheduler.
- 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

Types of Schedulers

- The **long-term scheduler**, or job scheduler, selects processes from the **disk** 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. Also known as **dispatcher**
- The **medium term scheduler** is responsible for s**wapping**.

- 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/O 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**.
- If it takes **10 milliseconds** to select a process to execute for **100 milliseconds**, then **9 percent** of the CPU is being used (wasted) simply for scheduling the work.

- 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.

I/O bound Vs CPU bound process

- It is important that the long-term scheduler make a careful selection.
- In general, most processes can be described as either I/O 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

- Some operating systems, such as **time-sharing systems**, may introduce an additional, **intermediate level of scheduling**.
- The key idea behind a **medium-term scheduler** is that sometimes it can be advantageous to **remove a process from memory** and thus reduce the degree of multiprogramming.
- 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.

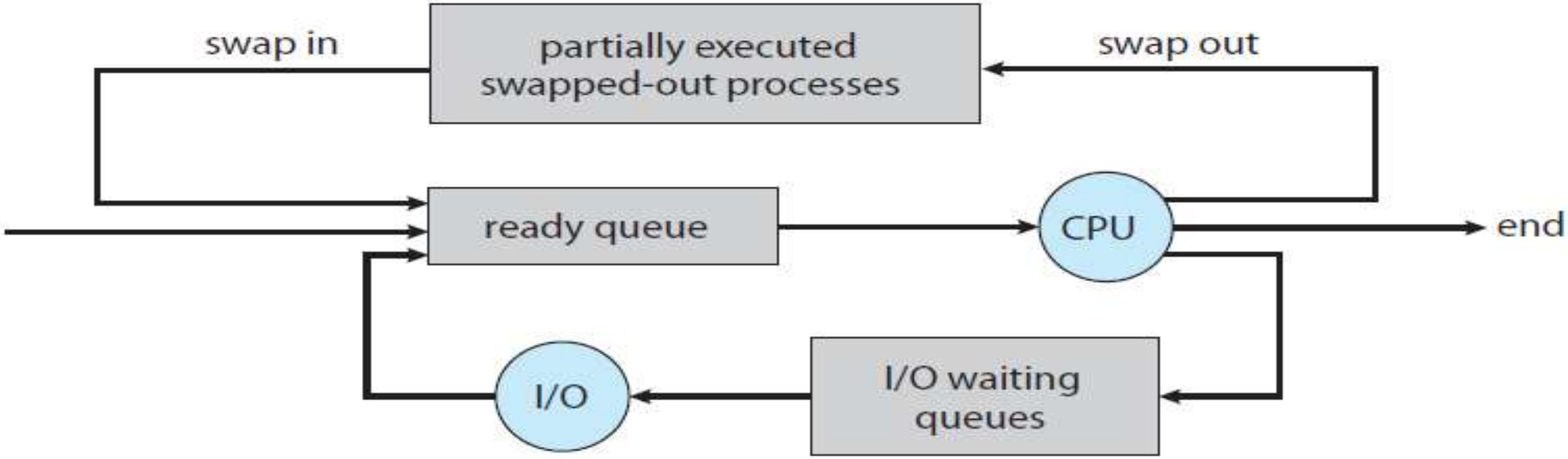


Figure 3.7 Addition of medium-term scheduling to the queueing diagram.

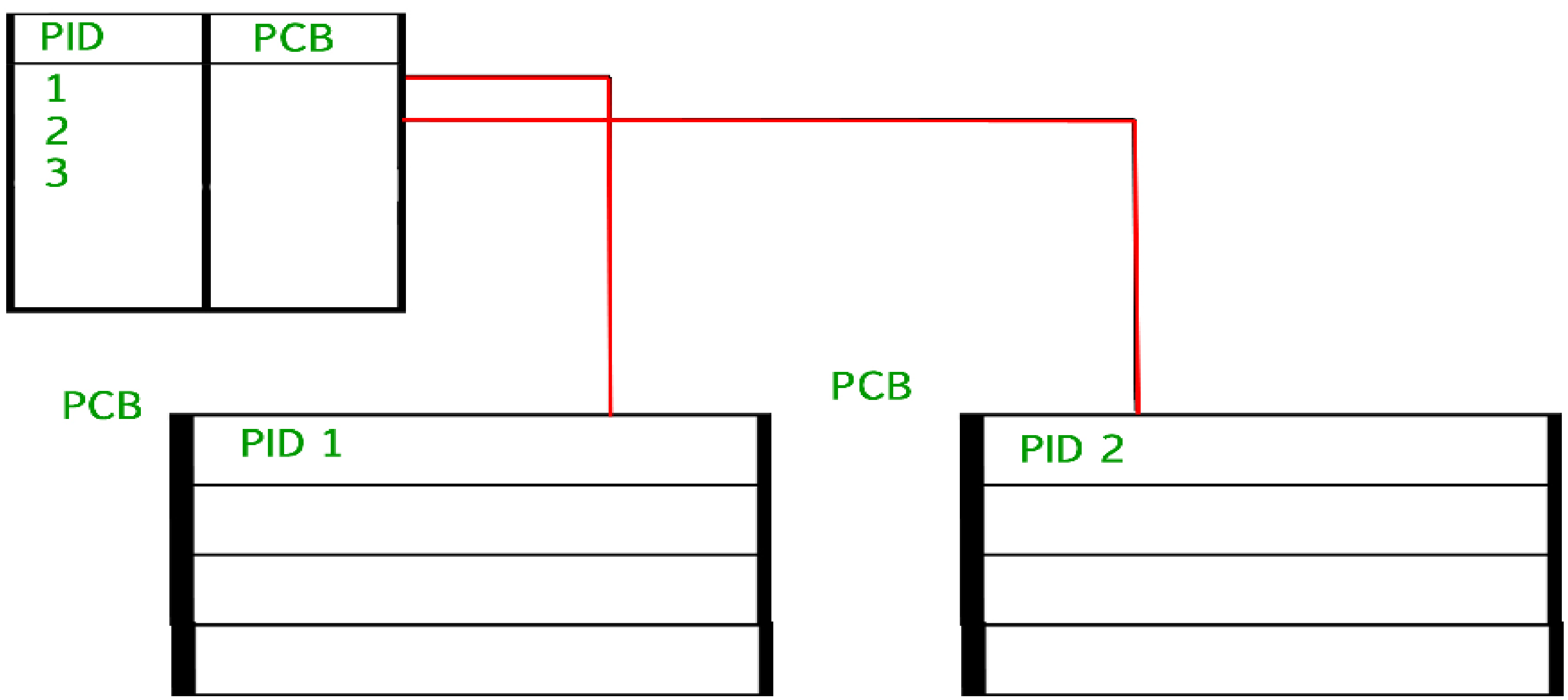
Context Switch

- **Interrupts** cause the operating system to change a CPU from its current task and to **run a kernel routine.**
- Such operations **happen frequently** on general-purpose systems.
- 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, essentially suspending the process and then resuming it.

- The context is represented in the **PCB** of the process.
- Generically, we perform a **state save** of the current state of the CPU, be it in kernel or user mode, 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.
- **Switching speed** 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).
- A typical speed is a **few milliseconds**.
- Context-switch times are highly dependent on h/w support.
- For instance, some processors (such as the Sun UltraSPARC) provide multiple sets of registers.
- A context switch here simply requires **changing the pointer to the current register set.**

- Also, the more complex the operating system, the **greater the amount of work** that must be done during a context switch.
- The **address space** of the current process must be preserved as the space of the next task is prepared for use.
- How the address space is preserved, and what amount of work is needed to preserve it, depend on the **memory-management scheme** of the operating system.



Process table and process control block

Process Creation and termination

- The processes in most systems can execute concurrently, and they may be **created** and **deleted** dynamically.
- Thus, these systems must provide a mechanism for **process creation and termination.**
- We explore the mechanisms involved in creating processes

Process Creation

- During the course of execution, a process may create (**spawn**) several new processes.
- 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 operating systems 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**.

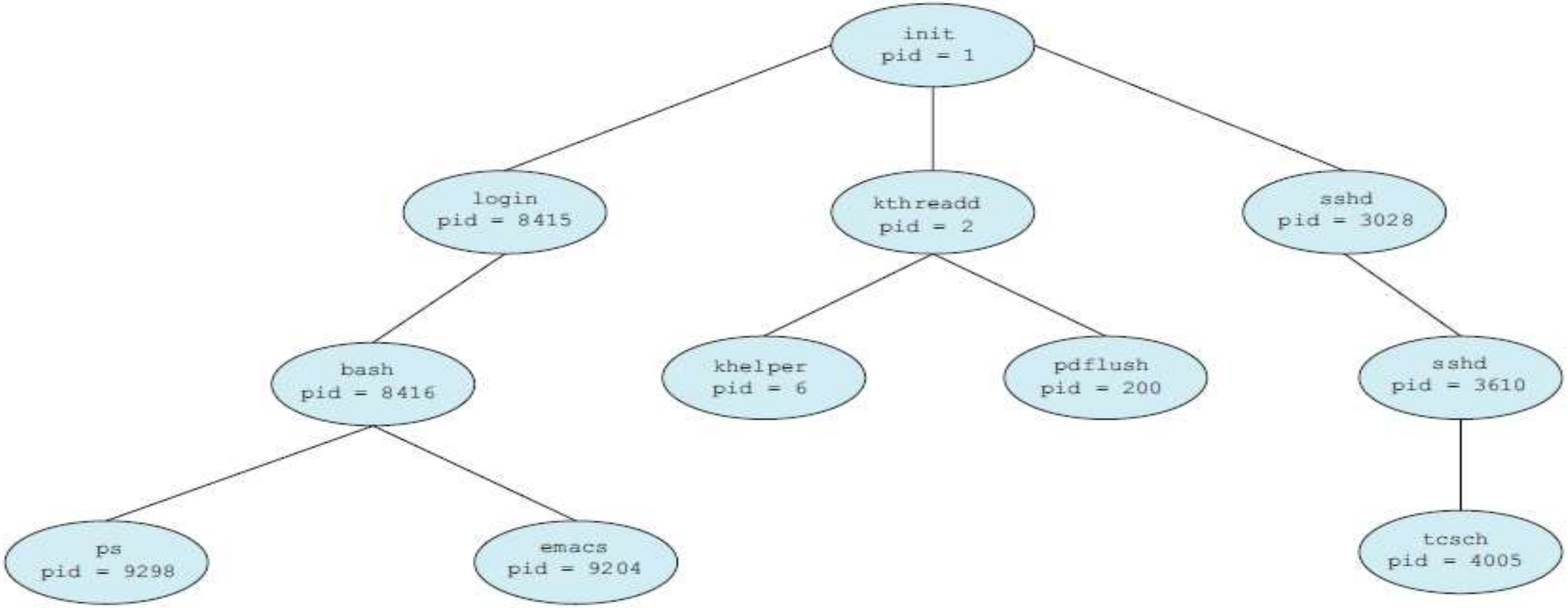


Figure 3.8 A tree of processes on a typical Linux system.

- 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.
- In Figure we see three **children of init : login, kthreadd** and **sshd**.
  - **kthreadd** process is responsible for creating additional processes that perform tasks on behalf of the kernel.
  - **sshd** process is responsible for managing clients that connect to the system by using ssh (which is short for ***secure shell).***
  - **login process** is responsible for managing clients that directly log onto the system.
- In this example, **a client** has logged on and is using the **bash shell**, which has been assigned **pid 8416**.
- Using the bash command-line interface, this user has created the process **ps** as well as the **emacs editor**

- On UNIX and Linux systems, we can obtain **a listing of processes** by using the ps command. For example, the command
  - ps -els
- will **list complete information for all processes** currently active in the system

Child Process

- 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 **physica**l and **logical resources**, the parent process may pass along initialization data (**input**) to the child process.
- When a process creates a new process, two possibilities for execution exist:
  - The parent continues to execute concurrently with its children
  - The parent waits until some or all of its children have terminated
- There are also two address-space possibilities for the new process:
  - The child process is a **duplicate of the parent** process (it has the same program and data as the parent).
  - The child process has a **new program loaded** into it.

Fork

- 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 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

- After a fork() system call, one of the two processes typically uses the **exec() system c**all to replace the process's memory space with a new program.
- The exec() system call **loads a binary file** into memory (destroying the memory image of the program containing the exec() system call) and starts its execution.
- 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.

```c
pid = fork();

if (pid < 0) { /* error occurred */
   fprintf(stderr, "Fork Failed");
   return 1;
}
else if (pid == 0) { /* child process */
   execlp("/bin/ls","ls",NULL);
}
else { /* parent process */
   /* parent will wait for the child to complete */
   wait(NULL);
   printf("Child Complete");
}
```

execlp() is a version of the exec() system call)

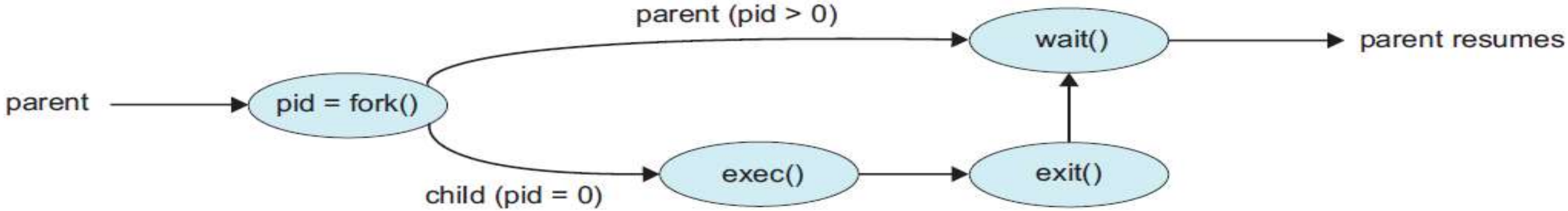


Figure 3.10 Process creation using the `fork()` system call.

CreateProcess() Vs fork()

- Processes are created in the Windows 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**, CreateProcess() requires **loading a specified program into the address space** of the child process at process creation.
- fork() is passed **no parameters**, CreateProcess() expects no **fewer than ten parameters**.

```
/* create child process */
if (!CreateProcess(NULL, /* use command line */
 "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
 NULL, /* don't inherit process handle */
 NULL, /* don't inherit thread handle */
 FALSE, /* disable handle inheritance */
 0, /* no creation flags */
 NULL, /* use parent's environment block */
 NULL, /* use parent's existing directory */
 &si,
 &pi))
{
```

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 cal**l.
- 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 **de-allocated** by the operating system.

- Termination can **occur in other circumstances** as well.
- A process can cause the **termination of another process** via an appropriate **system call** (for example, **TerminateProcess()** inWindows).
- 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 (**fork return value**)

Parent killing child

- 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. This phenomenon, referred to as <u>**cascading termination**</u>, is normally initiated by the operating system.

- We can terminate a process by using the **exit() system call.**
- A **parent process** may **wait for the termination** of a child process by using the **wait() system call**.
- The wait() system call is **passed a parameter** that allows the parent to obtain the **exit status** of the child.
- This system call also returns the **process identifier** of the terminated child so that the parent can tell which of its children has terminated:

```
pid_t pid;
int status;

pid = wait(&status);
```

- When a child process terminates, its **resources are deallocated** by the operating system.
- 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**
- Once the parent **calls wait(),** the process identifier of the **zombie process** and its entry in the **process table are released.**

- 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.