Processes

Contents of Lecture:

- ❖ Process Concept
- ❖ Process Scheduling
- ❖ Operations on Processes

References for This Lecture:

- ✓ *Abraham Silberschatz, Peter Bear Galvin and Greg Gagne, Operating System Concepts, 9th Edition, **Chapter 3***

Process Concept

- ❖ An operating system executes a variety of programs:
  - ✓ Batch system executes jobs
  - ✓ Time-shared systems has user programs or tasks

Textbook uses the terms job and process almost interchangeably

- ❖ **Process** is a program in execution; process execution must progress in sequential fashion
- ❖ A process is more than the **program code**, which is sometimes 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**.
  - ✓ **Stack** containing temporary data
    - ▪ Function parameters, return addresses, local variables
  - ✓ **Data section** containing global variables
  - ✓ **Heap** containing memory dynamically allocated during run time

- ❖ The structure of a process in memory is shown in next figure (Figure 3.1).

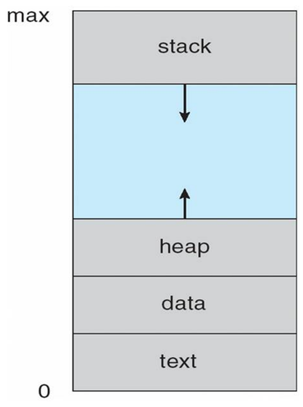


Figure 3.1 Process in memory.

- A **program** is a **passive entity**, such as a file containing a list of instructions stored on disk (often called an **executable file**).
    - ✓ Program becomes process when executable file loaded into memory (**process** is an **active entity**).

- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
    - ✓ Consider multiple users executing the same program

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.

- A process may be in one of the following states:
    - ✓ **New:** The process is being created.
    - ✓ **Ready:** The process is waiting to be assigned to a processor.
    - ✓ **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).
    - ✓ **Terminated:** The process has finished execution.

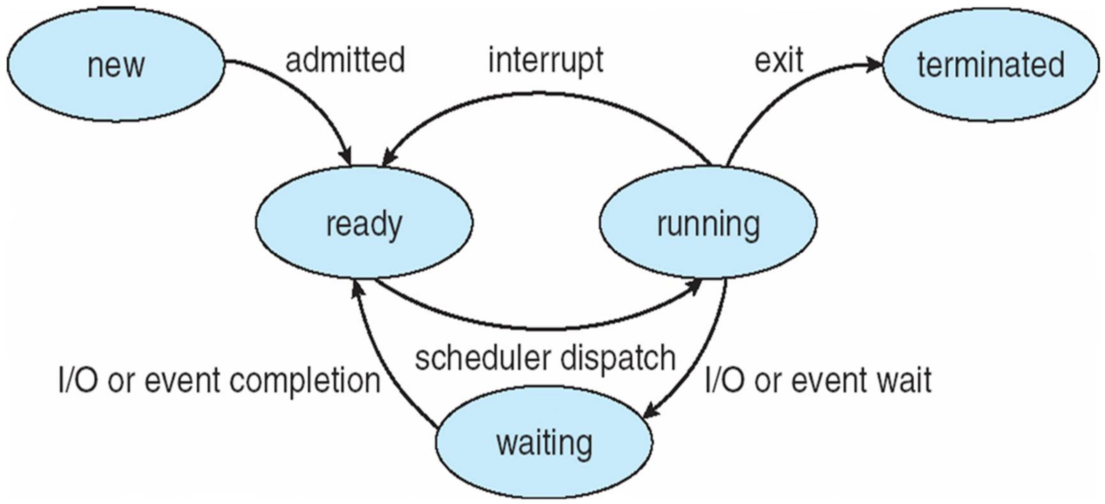


Figure 3.2 Diagram of process state.

Process Control Block (PCB)

❖ Each process is represented in the operating system by a **process control block** (PCB)—also called a **task control block**. A PCB is shown in next figure (Figure 3.3).

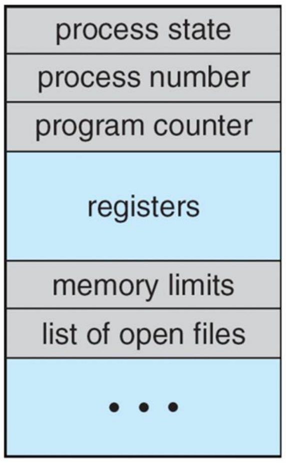


Figure 3.3 Process control block (PCB).

❖ It contains many pieces of information associated with a specific process, including these:

- ✓ **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.
- ✓ **CPU scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- ✓ **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.
- ✓ **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.

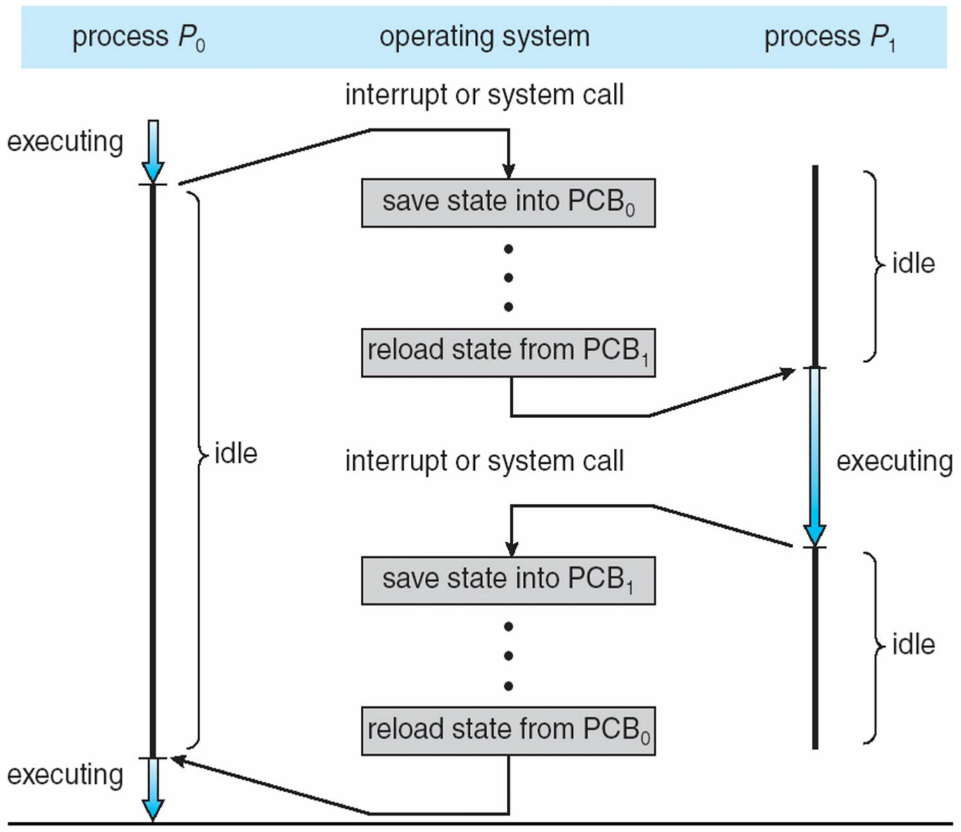


Figure 3.4 Diagram showing CPU switch from process to process

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.
  - ✓ Maximize CPU use, quickly switch processes onto CPU for time sharing

❖ **Process scheduler** selects an available process (possibly from a set of several available processes) for program execution on the CPU.

Scheduling queues

❖ **Job queue:** As processes enter the system, they are put into a job queue, which consists of all processes in the system.

❖ **Ready queue:** 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

❖ **Device queue:** The list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue next figure (Figure 3.5).
  - ✓ When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request.
  - ✓ 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.

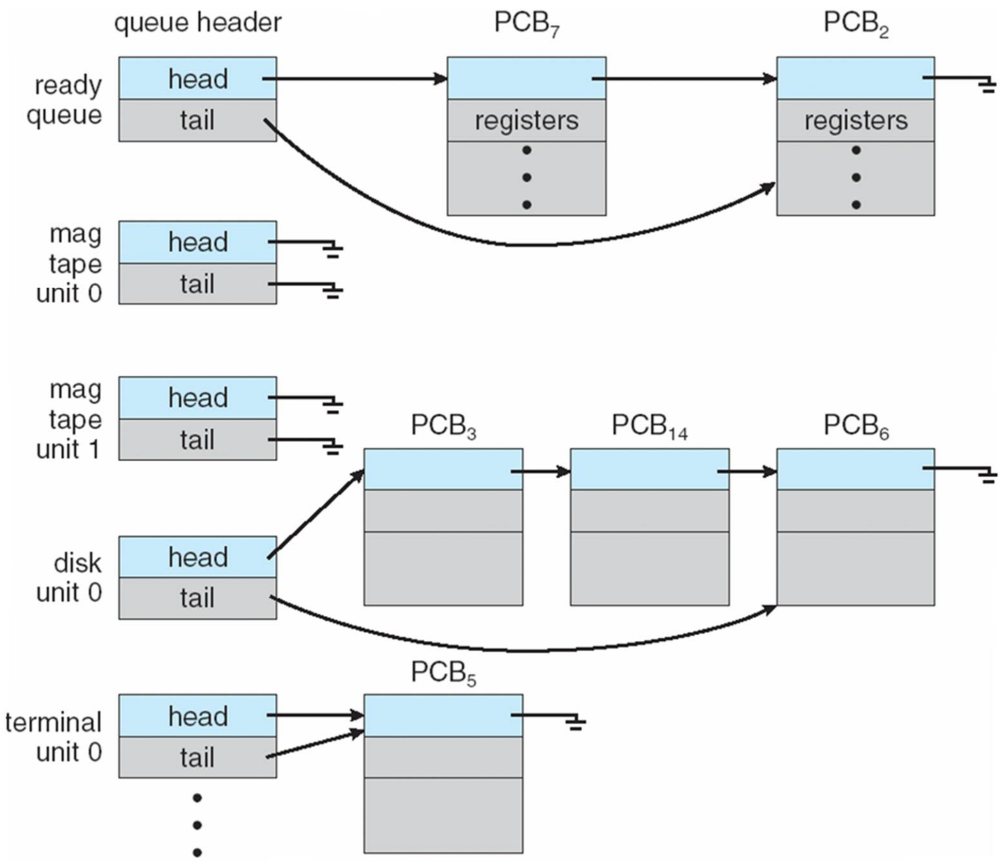


Figure 3.5 The ready queue and various I/O device queues.

Queueing diagram

❖ A common representation of process scheduling is a queueing diagram, such as that in next figure (Figure 3.6).
  - ✓ 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.

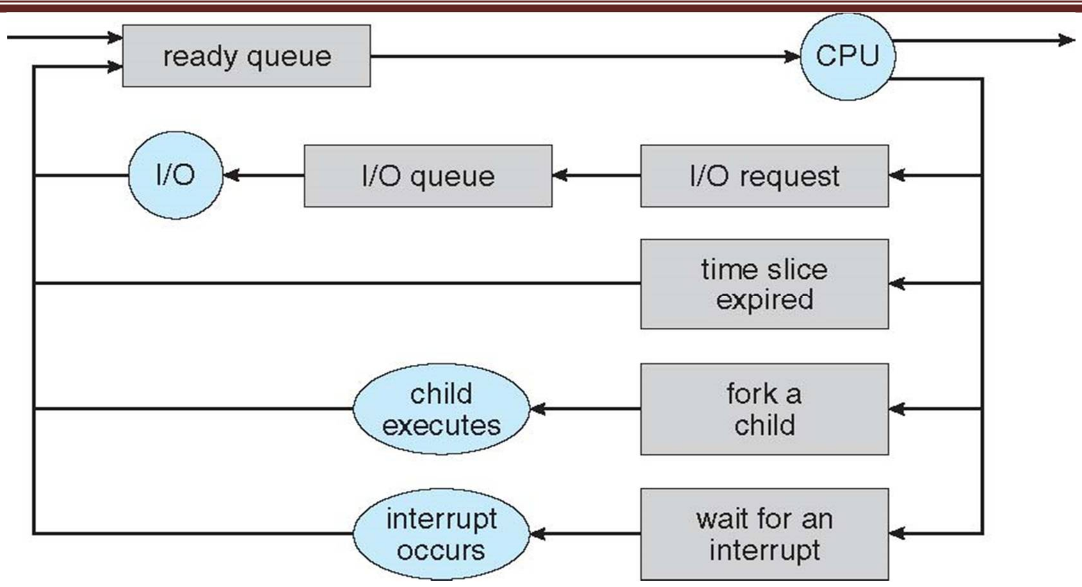


Figure 3.6 Queueing-diagram representation of process scheduling.

❖ A new process is initially put in the ready queue. It waits there until it is selected for execution, or dispatched. 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.

<u>Schedulers</u>

❖ 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.

❖ **Long-term scheduler** (or **job scheduler**):
  - ✓ 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.
    - ▪ selects which processes should be brought into the ready queue
  - ✓ Long-term scheduler is invoked infrequently (seconds, minutes) → (may be slow)
  - ✓ The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory).

- **Short-term scheduler** (or **CPU scheduler**):
  - ✓ 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.
    - selects which process should be executed next and allocates CPU
  - ✓ Sometimes the only scheduler in a system
  - ✓ Short-term scheduler is invoked frequently (milliseconds) → (must be fast)

- Processes can be described as either:
  - ✓ **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - ✓ **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good process mix

Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch.
- Context of a process represented in the PCB.
- Context-switch time is overhead; the system does no useful work while switching
  - ✓ The more complex the OS and the PCB → the longer the context switch
- Time dependent on hardware support
  - ✓ Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once

Operations on Processes

- The processes in most systems can execute concurrently, these systems must provide a mechanism for **process creation** and **termination**.

1. Process Creation

- During the course of execution, a process may create 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 (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.

❖ Next figure (Figure 3.8) illustrates a typical process tree for the Linux operating system, showing the name of each process and its pid:

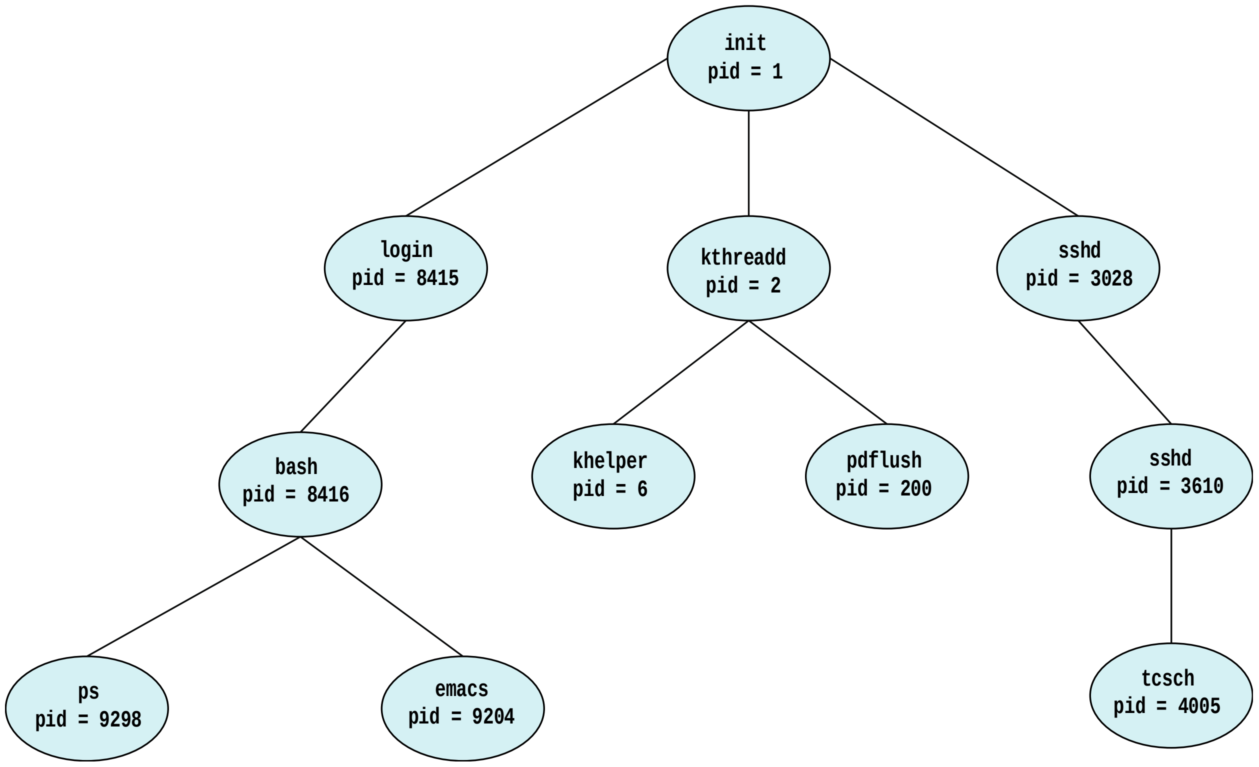


Figure 3.8 A tree of processes on a typical Linux system.

❖ 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, such as a web or print server.

❖ In a bove figure, we see the children of init—kthreadd and sshd.
  - ✓ The kthreadd process is responsible for creating additional processes that perform tasks on behalf of the kernel (in this situation, khelper and pdflush).
  - ✓ 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. 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 –el**).

Resource sharing options

❖ In general, 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.

- ✓ Parent and children share all resources
- ✓ Children share subset of parent's resources
- ✓ Parent and child share no resources

Execution options

❖ 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.

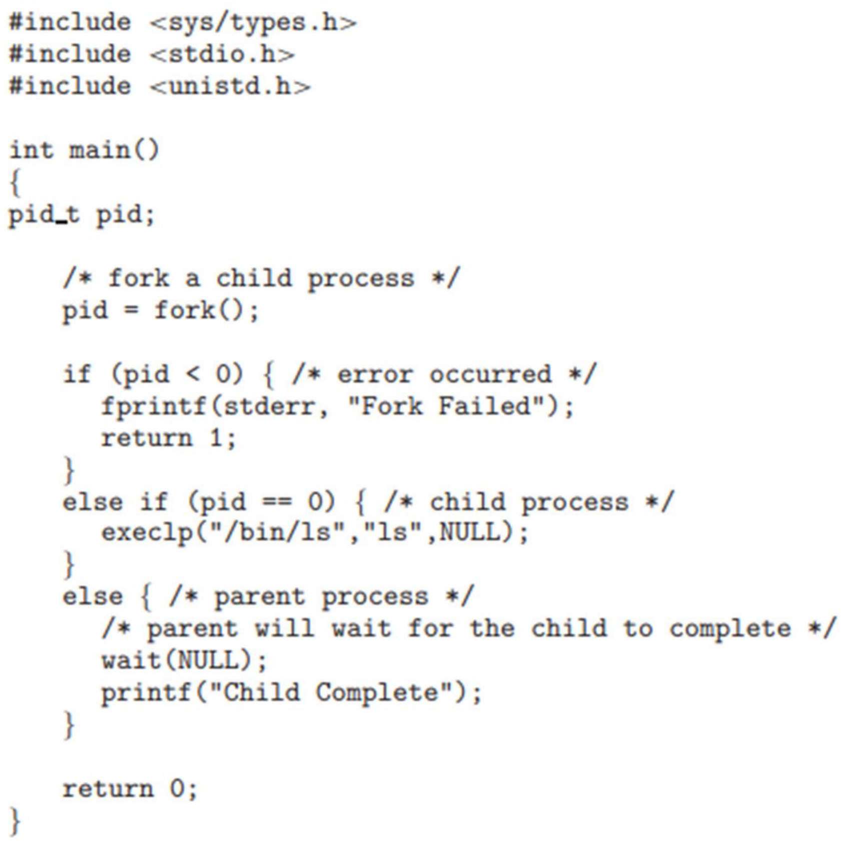
Address-space

❖ 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

UNIX examples

❖ The C program shown in next figure (Figure 3.9) illustrates the UNIX system calls. Have two different processes running copies of the same program.



```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

   /* fork a child process */
   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");
   }

   return 0;
}
```

Figure 3.9 Creating a separate process using the UNIX fork() system call.

- ❖ In UNIX, each process is identified by its process identifier, which is a unique integer.
  - ✓ A new process (child) is created by the fork() system call.
  - ✓ The child 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(), with one difference: the return code for the fork() is zero for the new (child) process, while that for the parent is an integer value greater than zero.

- ❖ After a fork() system call, one of the two processes typically uses the exec() system call to replace the process's memory space with a new program.

- ❖ The exec() system call loads a binary file into memory and starts its execution. In this manner:
  - ✓ The two processes are able to communicate and then go their separate ways.
  - ✓ The child process inherits privileges and scheduling attributes from the parent, as well certain resources, such as open files.

- ❖ The child process then overlays its address space with the UNIX command /bin/ls (used to get a directory listing) using the execlp() system call (execlp() is a version of the exec() system call). The parent waits for the child process to complete with the wait() system call.
  - ✓ Because the call to exec() overlays the process's address space with a new program, the call to exec() does not return control unless an error occurs.

- ❖ When the child process completes (by either implicitly or explicitly invoking exit()), the parent process resumes from the call to wait(), where it completes using the exit() system call.

- ❖ This is also illustrated in next figure (Figure 3.10).

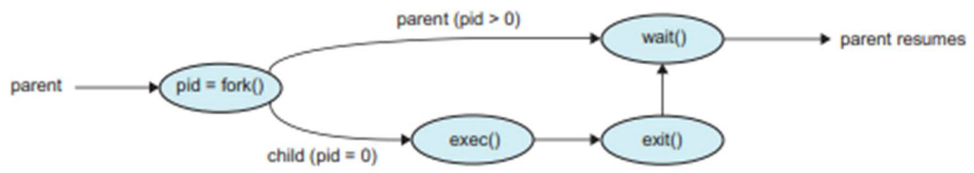


Figure 3.10 Process creation using the fork() system call.

2. <u>Process termination</u>

- ❖ 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 child process 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.

- ❖ 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. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
  - ✓ 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.

- ❖ Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - ✓ cascading termination. All children, grandchildren, etc. are terminated.
  - ✓ The termination is initiated by the operating system.

- ❖ The parent process may wait for termination of a child process by using the wait() system call. The call returns status information and the **pid** of the terminated process

```
pid t pid;
int status;
pid = wait(&status);
```

  - ✓ If no parent waiting (did not invoke wait()) process is a **zombie**
  - ✓ If parent terminated without invoking wait , process is an **orphan**