Lecture 3

Processes in the OS

Introduction

- A ***process*** is a program in execution. A process will need certain resources—such as CPU time, memory, files, and I/O devices— to accomplish its task. These resources are typically allocated to the process while it is executing.
- A process is the unit of work in most systems. Systems consist of a collection of processes: operating-system processes execute system code, and user processes execute user code. All these processes may execute concurrently.

The process

- The status of the current activity of a process is represented by the value of the program counter and the contents of the processor's registers. The memory layout of a process is typically divided into multiple sections, and is shown in Figure 3.1.

These sections include:

- Text section— the executable code
- Data section—global variables
- Heap section—memory that is dynamically allocated during program run time
- Stack section— temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)

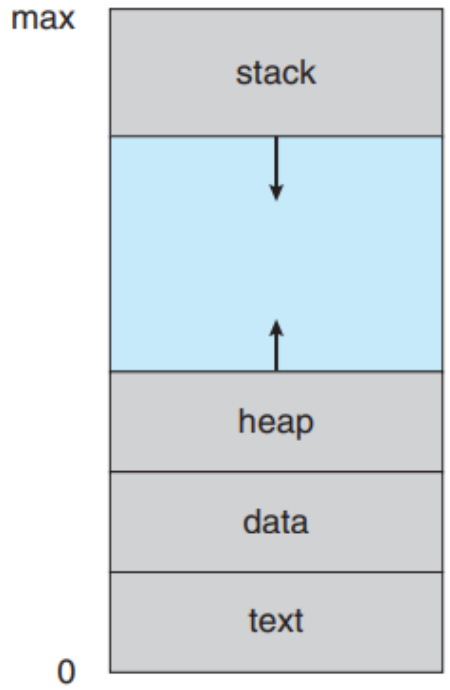

Figure 3.1 Layout of a process in memory.

- We emphasize that a program by itself is not a process. A program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable fil). In contrast, a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources.

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.
- **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.

- These names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems also more finely delineate process states. It is important to realize that only one process can be running on any processor core at any instant. Many processes may be ready and waiting, however. The state diagram corresponding to these states is presented in Figure 3.2.

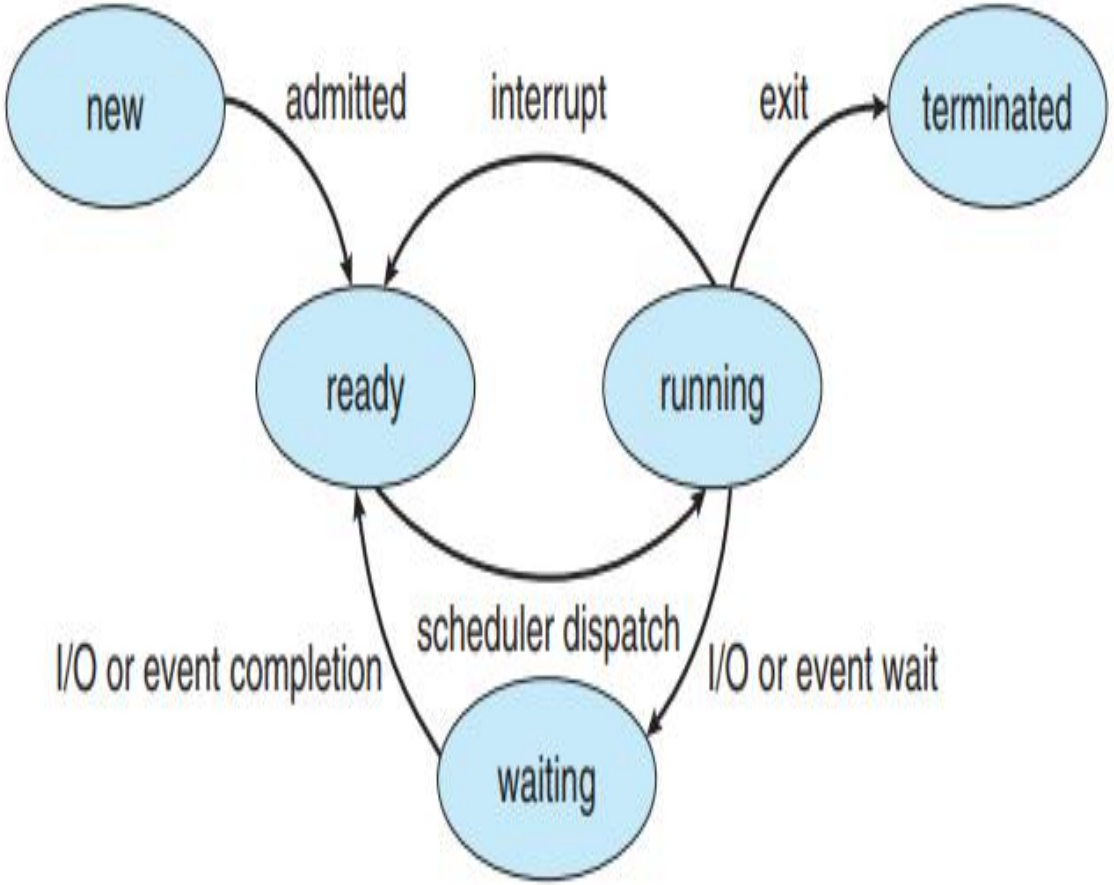

Figure 3.2 Diagram of process state.

Process Control Block

• **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 when it is rescheduled to run.
- **CPU-scheduling information**. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters. (Chapter 5 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
- **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.

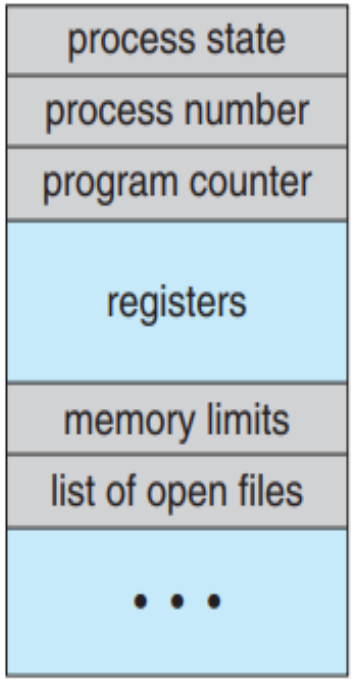

Figure 3.3 Process control block (PCB).

Process Scheduling

- The objective of multiprogramming is to have some process running at all times so as to maximize CPU utilization. The objective of time sharing is to switch a CPU core 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 a core. Each CPU core can run one process at a time.

Scheduling Queues

- As processes enter the system, they are put into a ready queue, where they are ready and waiting to execute on a CPU's core This queue is generally stored as a linked list; a ready-queue header contains pointers to the first PCB in the list, and each PCB includes a pointer field that points to the next PCB in the ready queue.

Queue

- Once the process is allocated a CPU core 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 wait queue.
- The process could create a new child process and then be placed in a wait queue while it awaits the child's termination.
- The process could be removed forcibly from the core, as a result of an interrupt or having its time slice expire and be put back in the ready queue.

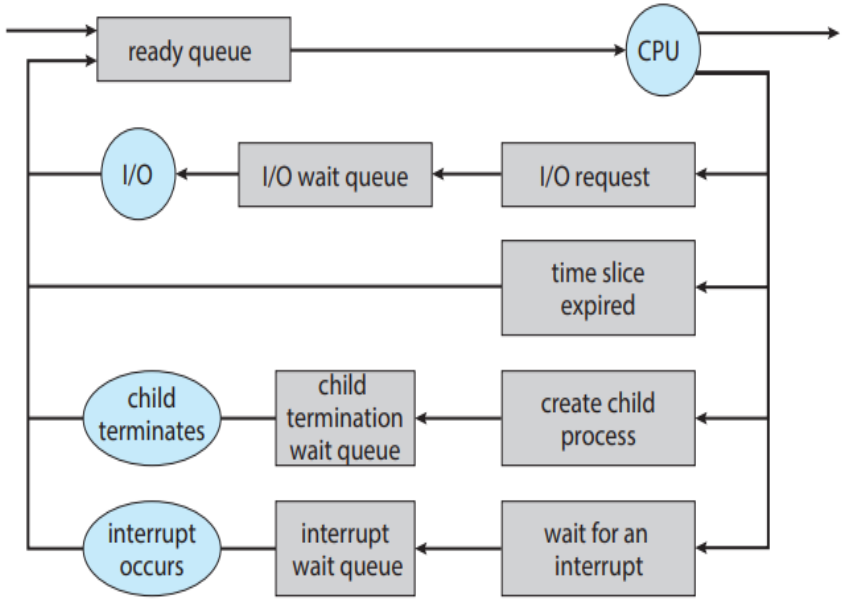

Figure 3.5 Queueing-diagram representation of process scheduling.

CPU Scheduling

- A process migrates among the ready queue and various wait queues throughout its lifetime. The role of the CPU scheduler is to select from among the processes that are in the ready queue and allocate a CPU core to one of them. The CPU scheduler must select a new process for the CPU frequently. An I/O-bound process may execute for only a few milliseconds before waiting for an I/O request. Although a CPU-bound process will require a CPU core for longer durations, the scheduler is unlikely to grant the core to a process for an extended period. Instead, it is likely designed to forcibly remove the CPU from a process and schedule another process to run. Therefore, the CPU scheduler executes at least once every 100 milliseconds, although typically much more frequently

Swapping

- Some operating systems have an intermediate form of scheduling, known as **swapping**, whose key idea is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) 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 known as **swapping** because a process can be “swapped out” from memory to disk, where its current status is saved, and later “swapped in” from disk back to memory, where its status is restored. Swapping is typically only necessary when memory has been overcommitted and must be freed up.

Context Switch

- interrupts cause the operating system to change a CPU core 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 core so that it can restore that context when its processing is done, essentially suspending the process and then resuming it.

- Switching the CPU core 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** and is illustrated in Figure 3.6.

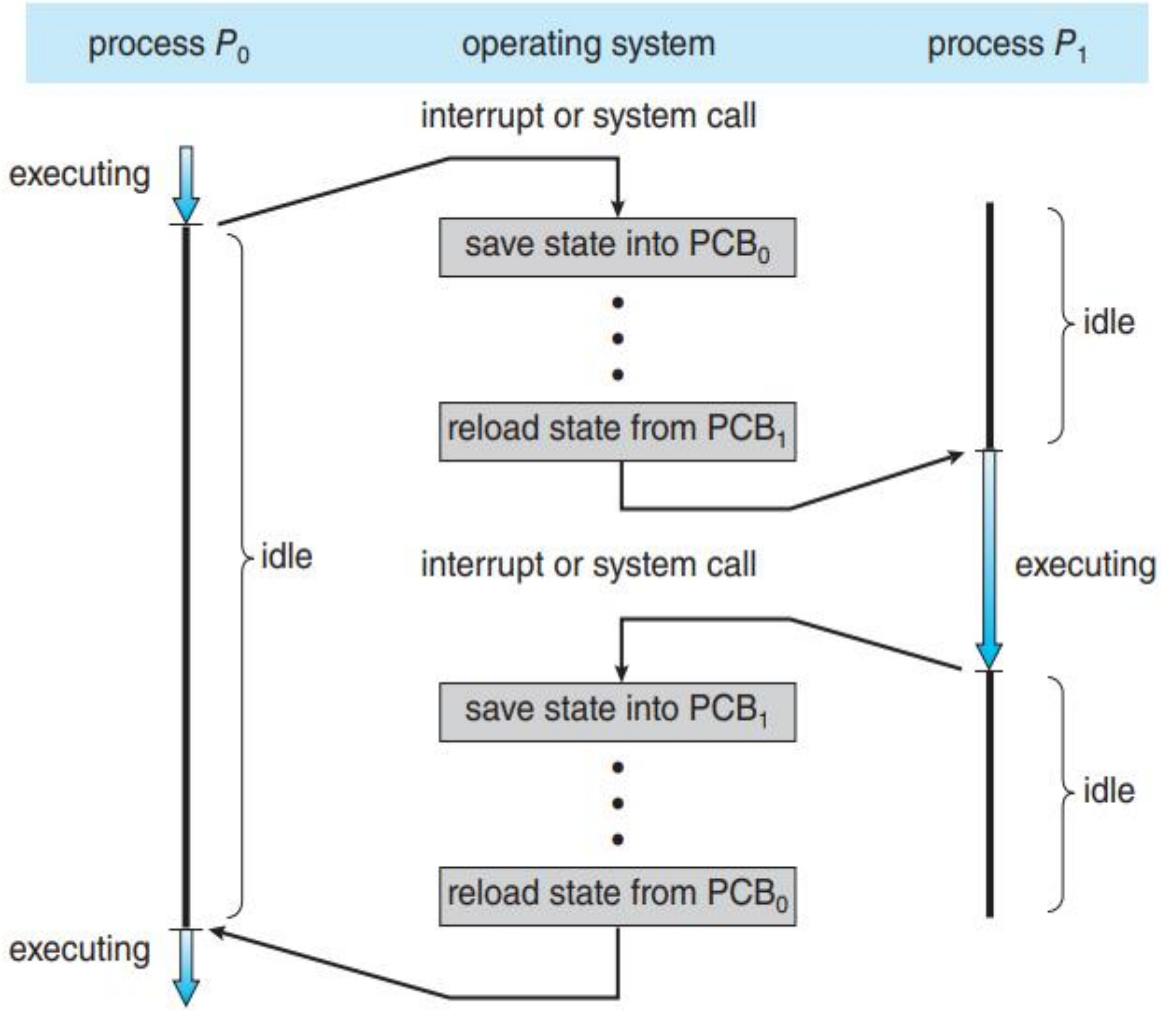

Figure 3.6 Diagram showing context switch from process to process.

Operations on Processes

- 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

Process Creation

- During the course of execution, a process may create several new processes. As mentioned earlier, 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.

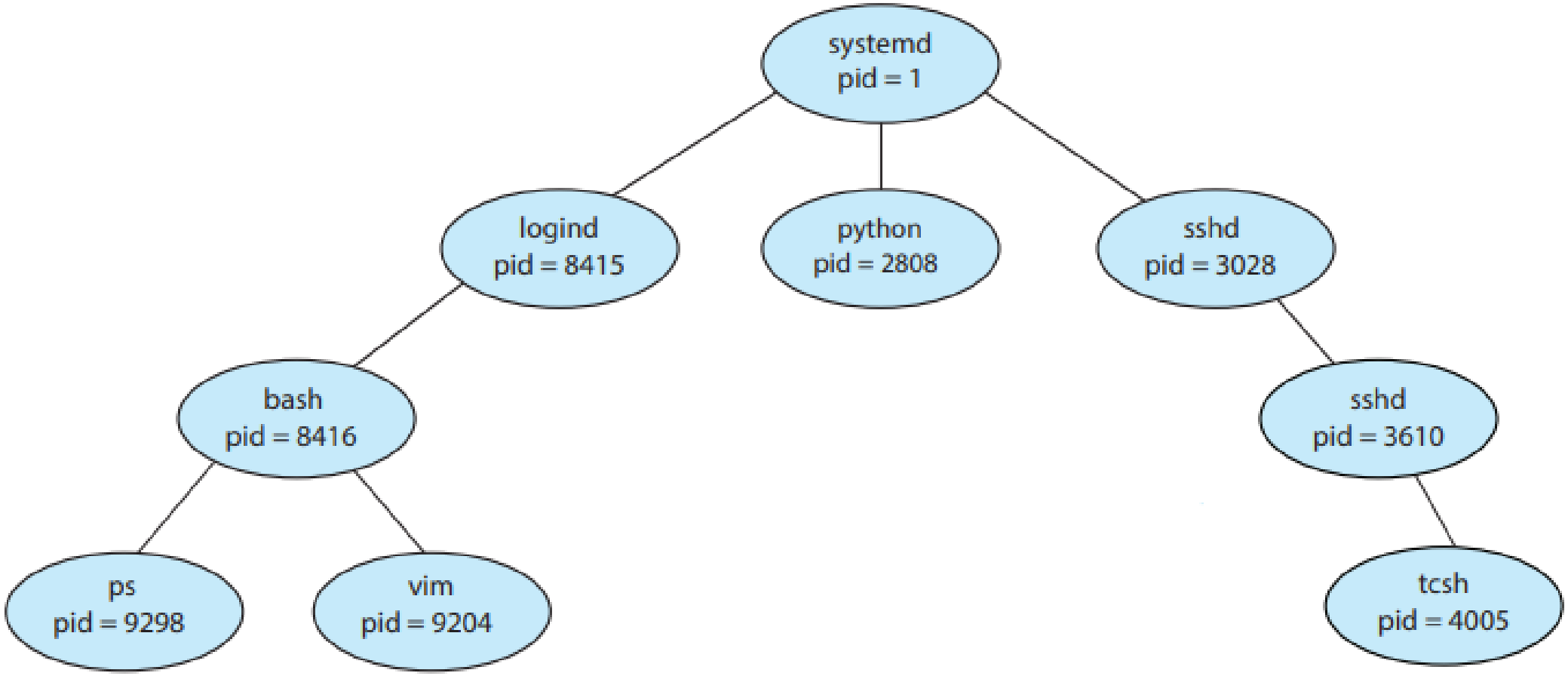

Figure 3.7 A tree of processes on a typical Linux system.

- On UNIX and Linux systems, we can obtain a listing of processes by using the ps command. For example, the command
  - `ps -el`

- 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. 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.

- 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.

- To illustrate these differences, let's first consider the UNIX operating system. In UNIX, as we've seen, each process is identified by its process identifier, which is a unique integer. 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()`, with one difference: 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 call 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. In this manner, the two processes are able to communicate and then go their separate ways. 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. Because the call to `exec()` overlays the process's address space with a new program, exec() does not return control unless an error occurs.

process creation in Windows

- As an alternative example, we next consider process creation in Windows. 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. However, 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. Furthermore, whereas fork() is passed no parameters, `CreateProcess()` expects no fewer than ten parameters.

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 waiting 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 and reclaimed 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() in Windows). Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, a user— or a misbehaving application—could arbitrarily kill another user's processes. 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

Zombie process

- When a 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**

Processes in Android

- **Foreground process**—The current process visible on the screen, representing the application the user is currently interacting with
- **Visible process**—A process that is not directly visible on the foreground but that is performing an activity that the foreground process is referring to (that is, a process performing an activity whose status is displayed on the foreground process)
- Service process—A process that is similar to a background process but is performing an activity that is apparent to the user (such as streaming music)
- **Background process**—A process that may be performing an activity but is not apparent to the user.
- **Empty process**—A process that holds no active components associated with any application

- Processes executing concurrently in the operating system may be either **independent processes** or **cooperating processes**. A process is independent if it does not share data with any other processes executing in the system. A process is cooperating if it can affect or be affected by the other processes executing in the system.

There are several reasons for providing an environment that allows process cooperation:

- Information sharing. Since several applications may be interested in the same piece of information (for instance, copying and pasting), we must provide an environment to allow concurrent access to such information.
- Computation speedup. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.
- Modularity. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads

interprocess communication (IPC)

- Cooperating processes require an **interprocess communication (IPC)** mechanism that will allow them to exchange data— that is, send data to and receive data from each other. There are two fundamental models of interprocess communication: **shared memory and message passing**. In the shared-memory model, a region of memory that is shared by the cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the **message-passing model**, communication takes place by means of messages exchanged between the cooperating processes.

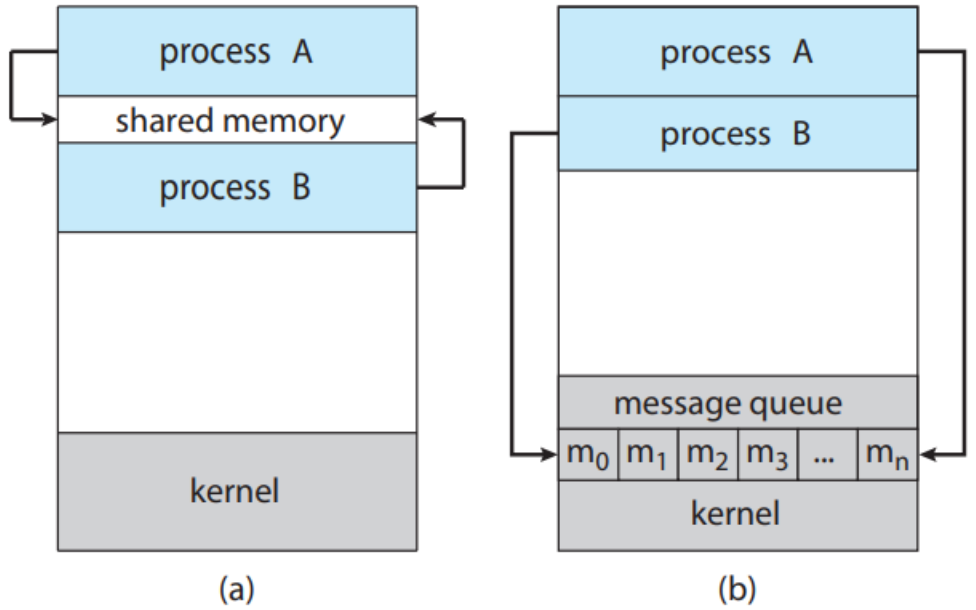

Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

shared memory model

- Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory.

- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network. For example, an Internet chat program could be designed so that chat participants communicate with one another by exchanging messages

Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send() and receive() primitives are defined as:

- send(P, message)—Send a message to process P.
- receive(Q, message)—Receive a message from process Q.

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.

Thank you for your attention!