**UNIT II** **PROCESS MANAGEMENT**

Processes - Process Concept - Process Scheduling - Operations on Processes - Inter-process Communication; CPU Scheduling - Scheduling criteria - Scheduling algorithms: Threads - Multithread Models – Threading issues; Process Synchronization - The Critical-Section problem - Synchronization hardware – Semaphores – Mutex - Classical problems of synchronization - Monitors; Deadlock - Methods for handling deadlocks, Deadlock prevention, Deadlock avoidance, Deadlock detection, Recovery from deadlock.

## I The Process

Early computers were batch systems that executed **jobs**, followed by the emergence of time-shared systems that ran **user programs**, or **tasks**. Even on a single-user system, a user may be able to run several programs at one time: a word processor, a web browser, and an e-mail package. And even if a computer can execute only one program at a time, such as on an embedded device that does not support multitasking, the operating system may need to support its own internal programmed activities, such as memory management. In many respects, all these activities are similar, so we call all of them **processes**.

### Process Concept

Informally, as mentioned earlier, a process is a program in execution. 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

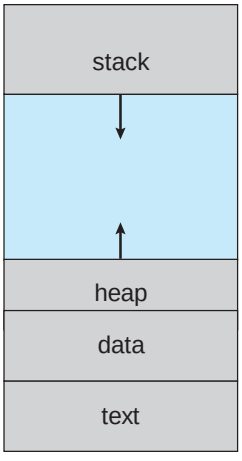


**Figure** Layout of a process in memory.

**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)

Each time a function is called, an **activation record** containing function parameters, local variables, and the return address is pushed onto the stack; when control is returned fromthe function, the activation record is popped from the stack.

A program is a ***passive*** entity, such as a file containing a list of instructions stored on disk (often called an **executable fil**

## (i) Process State

As a process executes, it changes **state**. The state of a process is defined in partby 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.

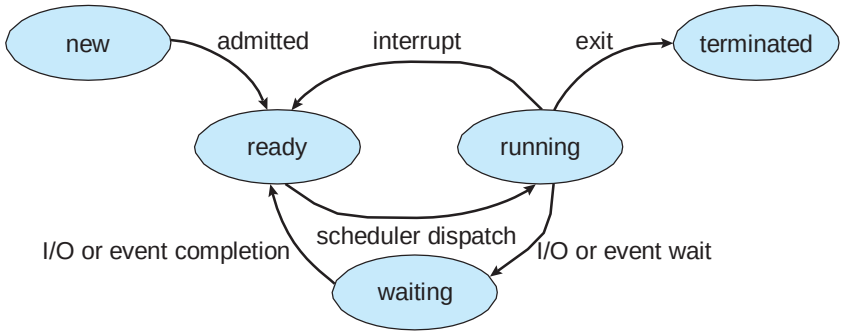



**Figure** Diagram of process state.

- **Terminated**. The process has finished execution.

## (ii) Process Control Block

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 Figure 3.3. 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.

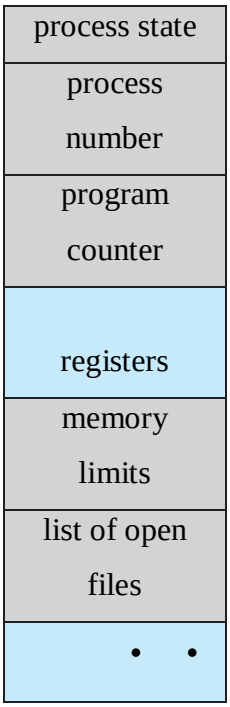



**Figure** Process control block (PCB).

- **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 informa-tion. 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 prior- ity, 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 (Chapter 9).
- **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.

### (iii)Threads

The process model discussed so far has implied that 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. Thus, the user cannot simultaneously type in characters and run the spell checker.

### (iv) 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 switcha 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.

For a system with a single CPU core, there will never be more than one processrunning at a time, whereas a multicore system can run multiple processes at one time.

If there are more processes than cores, excess processes will have to wait until a core is free and can be rescheduled. The number of processes currently in memory is known as the **degree of multiprogramming**.

Balancing the objectives of multiprogramming and time sharing alsorequires taking the general behavior of a process into account. In general, mostprocesses 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.

1. **Scheduling Queues**
2. **CPU Scheduling**
3. **Context Switch**

### 1. Scheduling Queues

- As processes enter the system, they are put into a **ready queue**, where they areready and waiting to execute on a CPU's core
- This queue is generally stored asa 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.
- The system also includes other queues. When a process is allocated a CPU core, it executes for a while and eventually terminates, is interrupted, or waitsfor 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 device such as a disk. Since devices run significantly slower than processors, the process will have to wait for the I/O to become available.
- A common representation of process scheduling is a **queueing diagram**, such as that in Figure. Two types of queues are present: the ready queue anda set of wait 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**. Once the process is allocated a CPU coreand is executing, one of several events could occur:

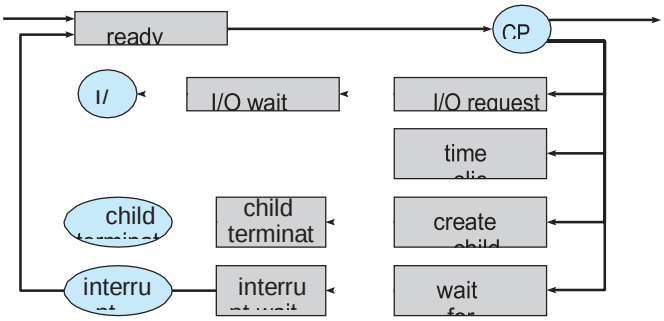



**Figure** Queueing-diagram representation of process scheduling.

- The process could issue an I/O request and then be placed in an I/O waitqueue.
- The process could create a new child process and then be placed in a waitqueue while it awaits the child's termination.
- The process could be removed forcibly from the core, as a result of aninterrupt or having its time slice expire, and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting stateto the ready state and is then 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.

## 2. CPU Scheduling

A process migrates among the ready queue and various wait queues through- out 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. TheCPU scheduler must select a new process for the CPU frequently. An I/O-boundprocess

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 dura-tions, 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 processand schedule another process to run. Therefore, the CPU scheduler executes atleast once every 100 milliseconds, although typically much more frequently.

Some operating systems have an intermediate form of scheduling, knownas **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 typicallyonly necessary when memory has been overcommitted and must be freed up.

### 3. Context Switch

As mentioned in Section 1.2.1, interrupts cause the operating system to changea CPU core from its current task and to run a kernel routine. Such operations happen frequently on general-purpose systems. When an interrupt occurs, thesystem 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, essentiallysuspending 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 (see Figure 3.2), and memory-management information. Generically, we perform a **state save** of the current state of the CPU core, be it in kernel or usermode, and then a **state restore** to resume operations.

Switching the CPU core to another process requires performing a state save of the current process and a state restore of a different process. Thistask is known as a **context switch** and is illustrated in Figure 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 executing

## b. Operations on Processes

The processes in most systems can execute concurrently, and they may be cre- ated and deleted dynamically. Thus, these systems must provide a mechanismfor process creation and termination.

### 1. Process Creation

## 2. Process Termination

## 1. 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 identifie** (or **pid**), which is typically an integer number.

Figure illustrates a typical process tree for the Linux operating system, showing the name of each process and its pid. (We use the term ***process*** rather loosely in this situation, as Linux prefers the term ***task*** instead.) The systemd process (which always has a pid of 1) serves as the root parent process for all user processes, and is the first user process created when the system boots. Once the system has booted, the systemd process creates processes which provide additional services such as a web or print server, an ssh server, and the like. In Figure 3.7, we see two children of systemd — logind and sshd. The logind process is responsible for managing clients that directly log onto the system.

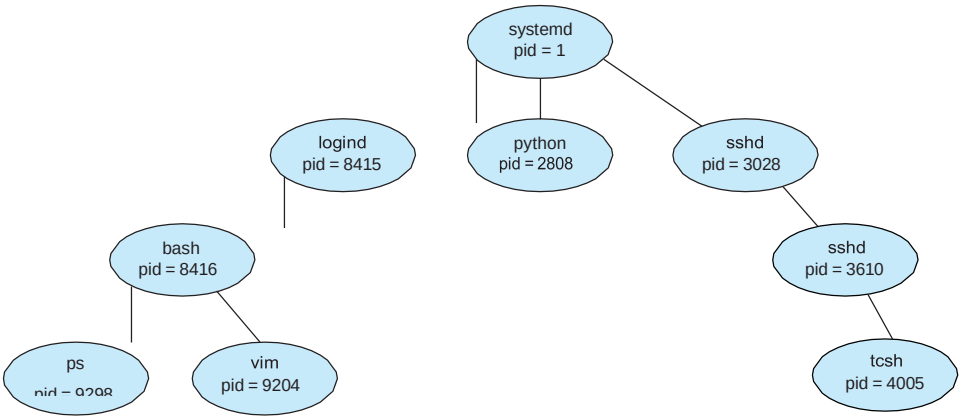



**Figure 3.7** A tree of processes on a typical Linux system.

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, whichis 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 ofthe 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 pro- gram. The exec() system call loads a binary file into memory (destroying the memory image of the program containing the exec() system call) and starts

```
#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** Creating a separate process using the UNIX fork() system call.

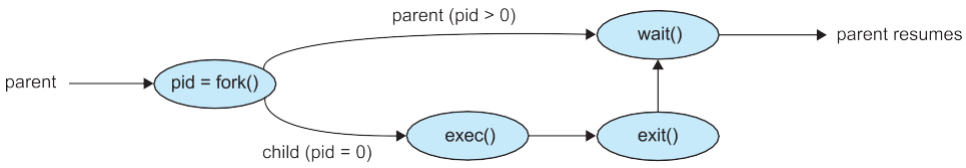



**Figure** Process creation using the fork() system call.

### ii. 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 waitingparent 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.

A parent may terminate the execution of one of its children for a variety ofreasons, such as these:

1. The child has exceeded its usage of some of the resources that it has beenallocated. (To determine whether this has occurred, the parent must havea mechanism to inspect the state of its children.)
2. The task assigned to the child is no longer required.
3. The parent is exiting, and the operating system 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), thenall its children must also be terminated. This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system.

To illustrate process execution and termination, consider that, in Linux and UNIX systems, we can terminate a process by using the `exit()` system call, providing an exit status as a parameter:

```
/* exit with status 1 */exit(1);
```

A parent process may wait for the termination of a child process by usingthe wait() system call. The wait() system call is passed a parameter thatallows the parent to obtain the exit status of the child. This system call alsoreturns 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);
```

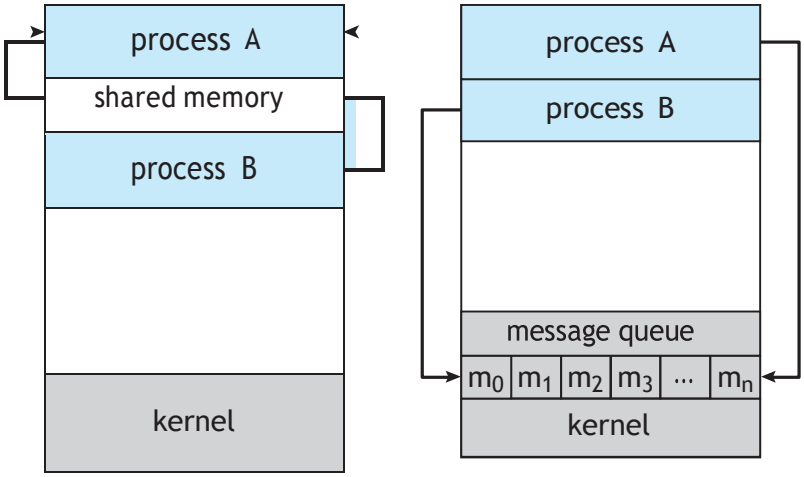
## Interprocess Communication

Processes executing concurrently in the operating system may be either inde- pendent 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 processcooperation:

- **Information sharing**. Since several applications may be interested in the same piece of information (for instance, copying and pasting).
- **Computation speedup**. If we want a particular task to run faster, we mustbreak it into subtasks, each of which will be executing in parallel with theothers.
- **Modularity**. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads,.

communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Figure




(a) (b)

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

## Synchronization

Communication between processes takes place through calls to send() andreceive() primitives. There are different design options for implementing each primitive. Message passing may be either **blocking**

or **nonblocking** — also known as **synchronous** and **asynchronous**. (Throughout this text, you will encounter the concepts of synchronous and asynchronous behavior in relation to various operating-system algorithms.)

- **Blocking send**. The sending process is blocked until the message is received by the receiving process or by the mailbox.
- **Nonblocking send**. The sending process sends the message and resumes operation.
- **Blocking receive**. The receiver blocks until a message is available.
- **Nonblocking receive**. The receiver retrieves either a valid message or a null.

```
message next produced;
while (true) {
/* produce an item in next produced */
send(next produced);
}
```

**Figure** The producer process using message passing.

Different combinations of send() and receive() are possible. When both send() and receive() are blocking, we have a **rendezvous** between the sender and the receiver. The solution to the producer – consumer problem becomes trivial when we use blocking send() and receive() statements. The producer merely invokes the blocking send() call and waits until the message is delivered to either the receiver or the mailbox. Likewise, when the consumer invokes receive(), it blocks until a message is available.

### Buffering

Whether communication is direct or indirect, messages exchanged by commu-nicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

- **Zero capacity**. The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- **Bounded capacity**. The queue has finite length *n;* thus, at most *n* messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without

```
message next consumed;
while (true) {
```

```
receive(next consumed);
/* consume the item in next consumed */
}
```

**Figure** The consumer process using message passing.

- **Unbounded capacity**. The queue's length is potentially infinite; thus, anynumber of messages can wait in it. The sender never blocks.

The zero-capacity case is sometimes referred to as a message system with no buffering. The other cases are referred to as systems with automatic buffering.