4)The Critical-Section Problem :

We begin our consideration of process synchronization by discussing the so called critical-section problem. Consider a system consisting of n processes {P0, P1, ..., Pn−1}. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section. The general structure of a typical process Pi is shown in Figure 5.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code. A solution to the critical-section problem must satisfy the following three requirements:

1. Mutual exclusion. If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.

2. Progress. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. Bounded waiting. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted. We assume that each process is executing at a nonzero speed. However

Peterson’s Solution

```
do { flag[i] = true;
 turn = j;
while (flag[j] && turn == j);
critical section
 flag[i] = false;
 remainder section
 } while (true);
```

Peterson’s solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P0 and P1. For convenience, when presenting Pi , we use Pj to denote the other process; that is, j equals 1 − i. Peterson’s solution requires the two processes to share two data items: int turn; boolean flag[2]; The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process Pi is allowed to execute in its critical section. The flag array is used to indicate if a process is ready to enter its critical section. For example, if flag[i] is true, this value indicates that Pi is ready to enter its critical section. With an explanation of these data structures complete, we are now ready to describe the algorithm shown in Figure 5.2. To enter the critical section, process Pi first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately. The eventual value of turn determines which of the two processes is allowed to enter its critical section first. We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

7)semaphore

semaphore is simply **an integer variable that is shared between threads**. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment. Semaphores are of two types:
Binary Semaphore – This is also known as mutex lock.
A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal(). The wait() operation was originally termed P (from the Dutch proberen, “to test”); signal() was originally called V
definition of wait() is as follows:(reduces semaphore value)

```
 wait(S)
 { while (S <= 0) ;
 // busy wait
S--; }
```

 The definition of signal() is as follows: (increases semaphore value)

```
signal(S)
 {
S++;
 }
```

The Dining-Philosophers Problem

```
semaphore chopstick[5];
do {
wait(chopstick[i]);
wait(chopstick[(i+1) % 5]);
... /* eat for awhile */ ..
signal(chopstick[i]);
signal(chopstick[(i+1) % 5]);
... /* think for awhile */ ...
}
while (true);
```
(along with dinning philosophers matter)

1)Interprocess Communication :

A process is independent if it cannot affect or be affected by the 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.

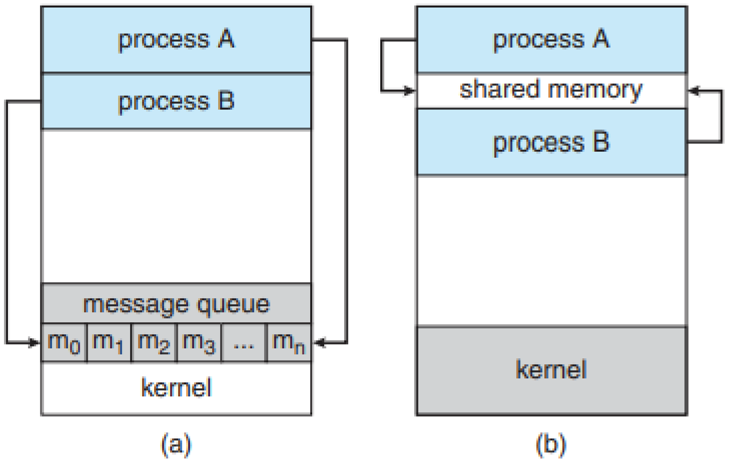
• **Information sharing.** Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.

• **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, as we discussed in Chapter 2.

• **Convenience.** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication: shared memory and message passing



Shared memory:

Inter process communication through shared memory is a concept where two or more process can access the common memory. And communication is done via this shared memory where changes made by one process can be viewed by another process.The problem with pipes, fifo and message queue – is that for two process to exchange information. The information has to go through the kernel.

Message passing:

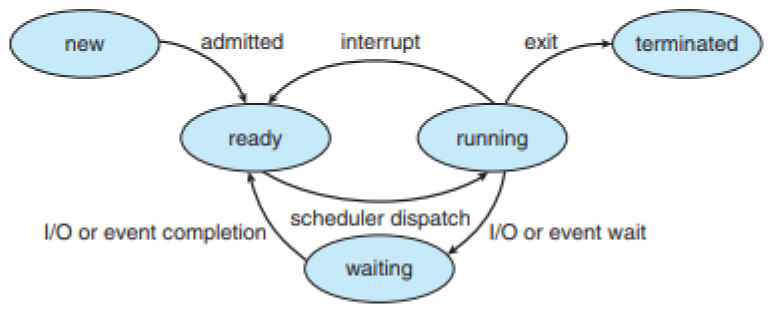
Message passing model allows multiple processes to read and write data to the message queue without being connected to each other. Messages are stored on the queue until their recipient retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems.

6.i)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:

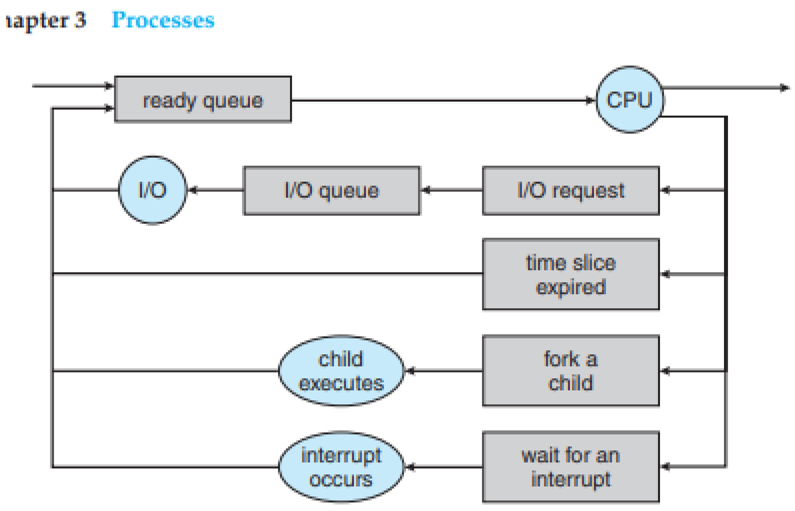
- **New.** The process is being created.
- **Running**. Instructions are being executed.
- **Waiting**. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready.** The process is waiting to be assigned to a processor.
- **Terminated.** The process has finished execution.

108 Chapter 3 Processes



6.ii)Scheduling Queues

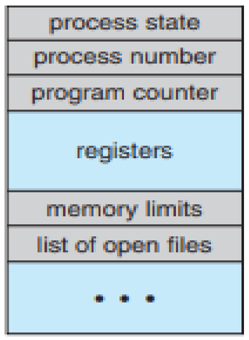
As processes enter the system, they are put into a job queue, which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue. This queue is generally stored as a linked list. A ready-queue header 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. 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. 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 queueing diagram, such as that in 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. 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. In the first two cases, the process eventually switches from the waiting state to 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

5.i)



• **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
• **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