Operating System

Chapter 2: Processes and Threads

Prepared By:
Amit K. Shrivastava
Asst. Professor
Nepal College Of Information Technology

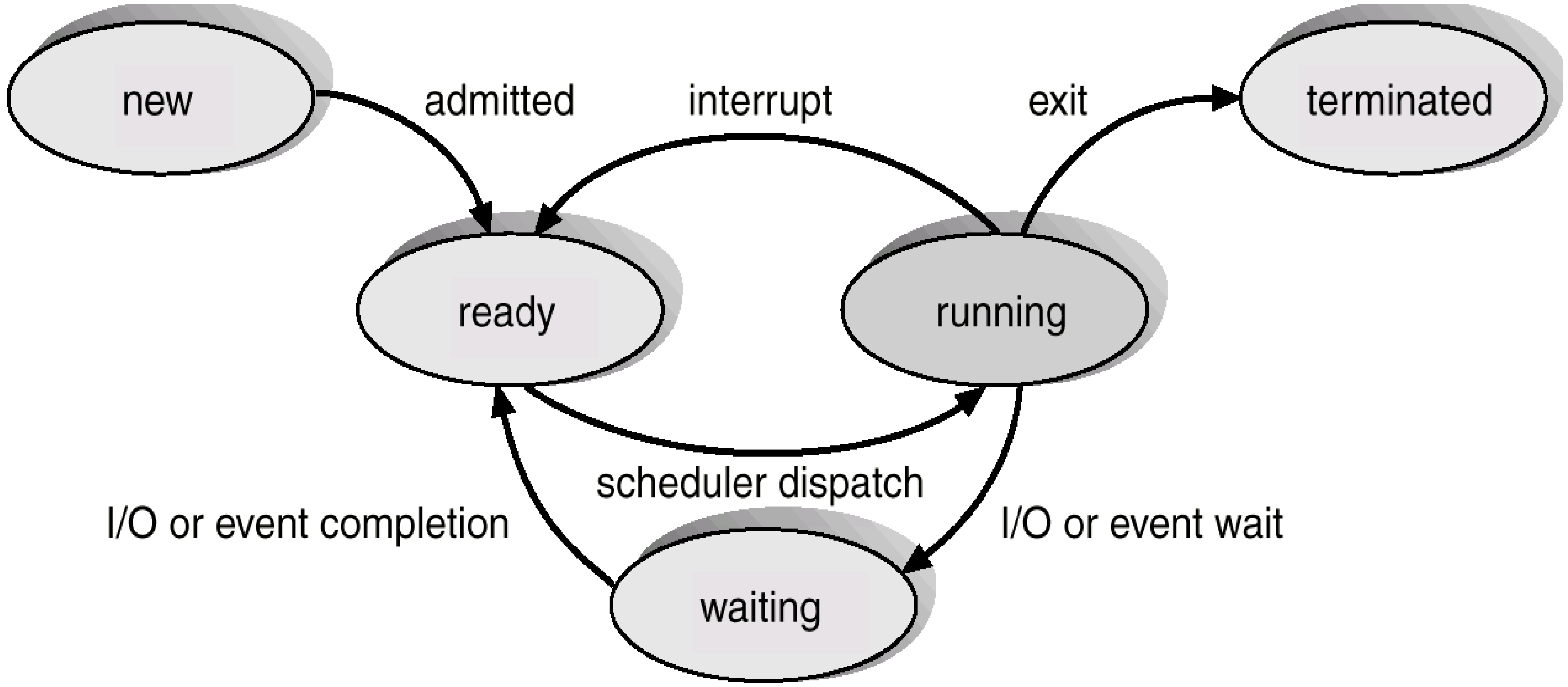
Process Concept

- An operating system executes a variety of programs:
  - ✦ Batch system – jobs
  - ✦ Time-shared systems – user programs or tasks
- The terms job and process are used almost interchangeably.
- Process – a program in execution; process execution must progress in sequential fashion.
- A process includes:
  - ✦ program counter
  - ✦ stack
  - ✦ data section

Process State

- As a process executes, it changes state
  - ✦ new: The process is being created.
  - ✦ running: Instructions are being executed.
  - ✦ waiting: The process is waiting for some event to occur.
  - ✦ ready: The process is waiting to be assigned to a process.
  - ✦ terminated: The process has finished execution.

Diagram of Process State

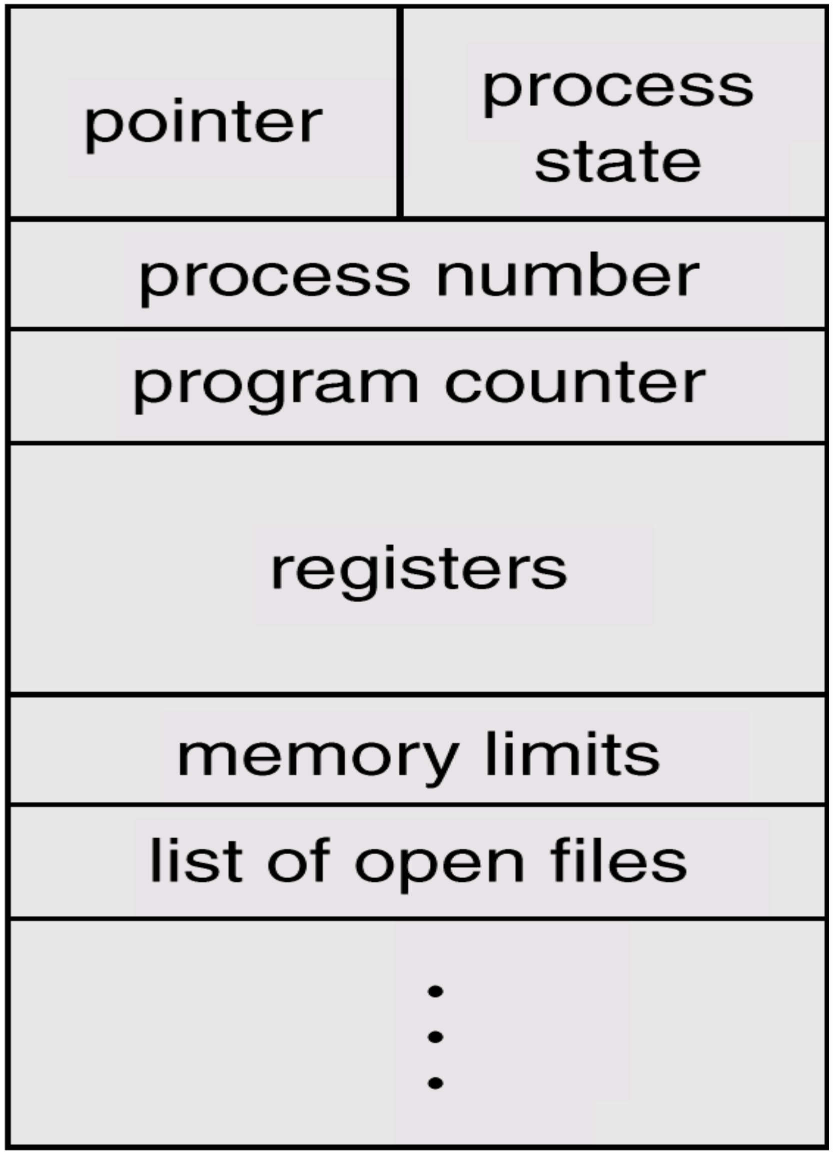


Process Control Block (PCB)

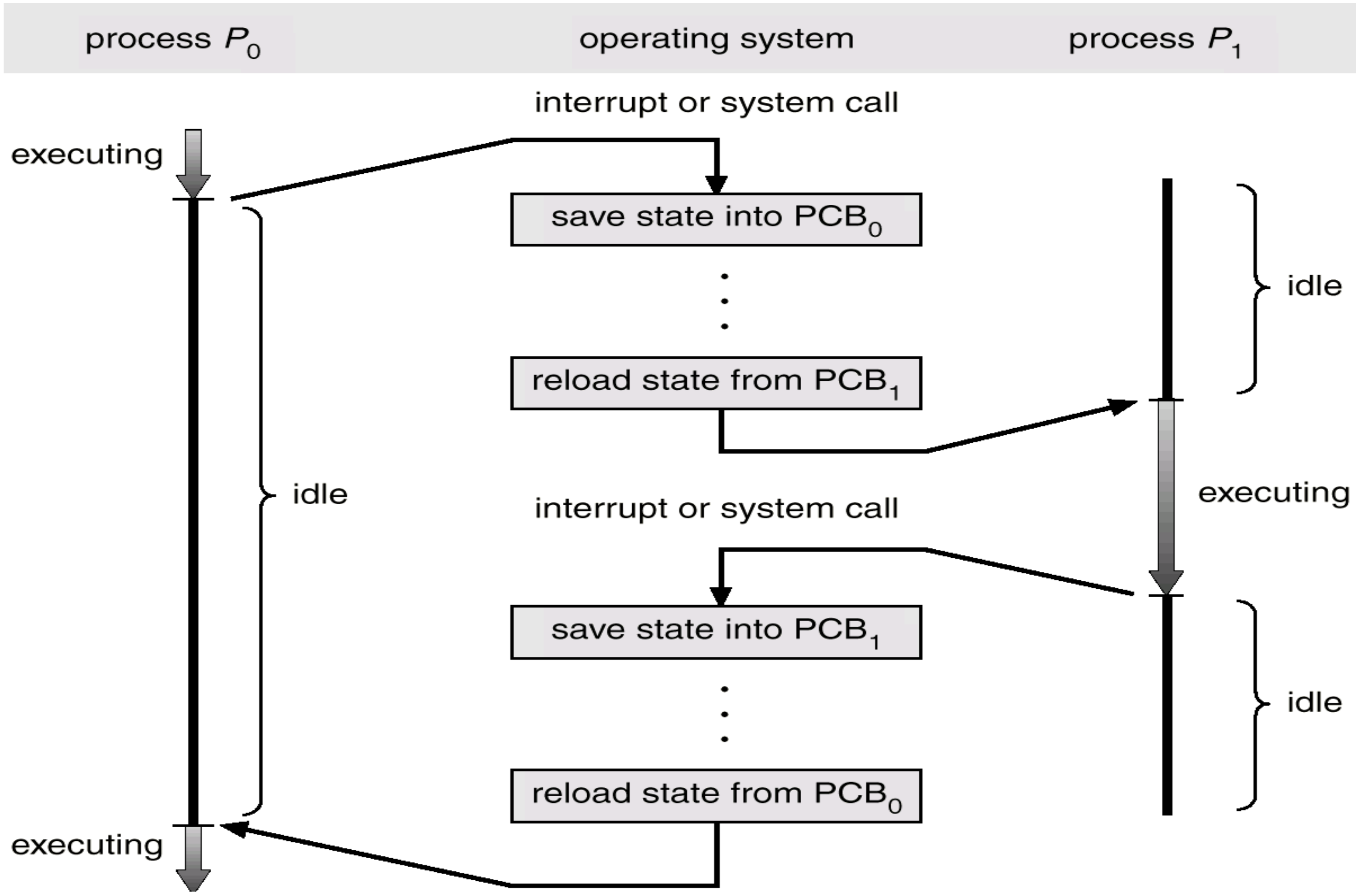
Information associated with each process.

- Process state- new, ready, …
- Program counter- indicates the address of the next instruction to be executed for this program.
- CPU registers- includes accumulators, stack pointers, …
- CPU scheduling information- includes process priority, pointers to scheduling queues.
- Memory-management information- includes the value of base and limit registers (protection) …
- Accounting information- includes amount of CPU and real time used, account numbers, process numbers, …
- I/O status information- includes list of I/O devices allocated to this process, a list of open files, …

Process Control Block (PCB)



CPU Switch From Process to Process



Cooperating Processes

- Independent process cannot affect or be affected by the execution of another process.
- Cooperating process can affect or be affected by the execution of another process

Advantages of process cooperation:

- 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 these types of resources.
- 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. Such a speedup can be achieved only if the computer has multiple processing elements (such as CPUSor I/O channels).

-Modularity: We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads,

-Convenience: Even an individual user may have many tasks on which to work at one time. For instance, a user may be editing, printing, and compiling in parallel.

IPC(Inter-Process Communication):

•Processes frequently need to communicate with other processes i.e they need to exchange data and information. Thus there is need a need for communication between processes, preferably in a well-structured way not using interrupts. This communication mechanism between processes are known as Inter-Process Communication.

• 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 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.

IPC(Inter-Process Communication) contd…:

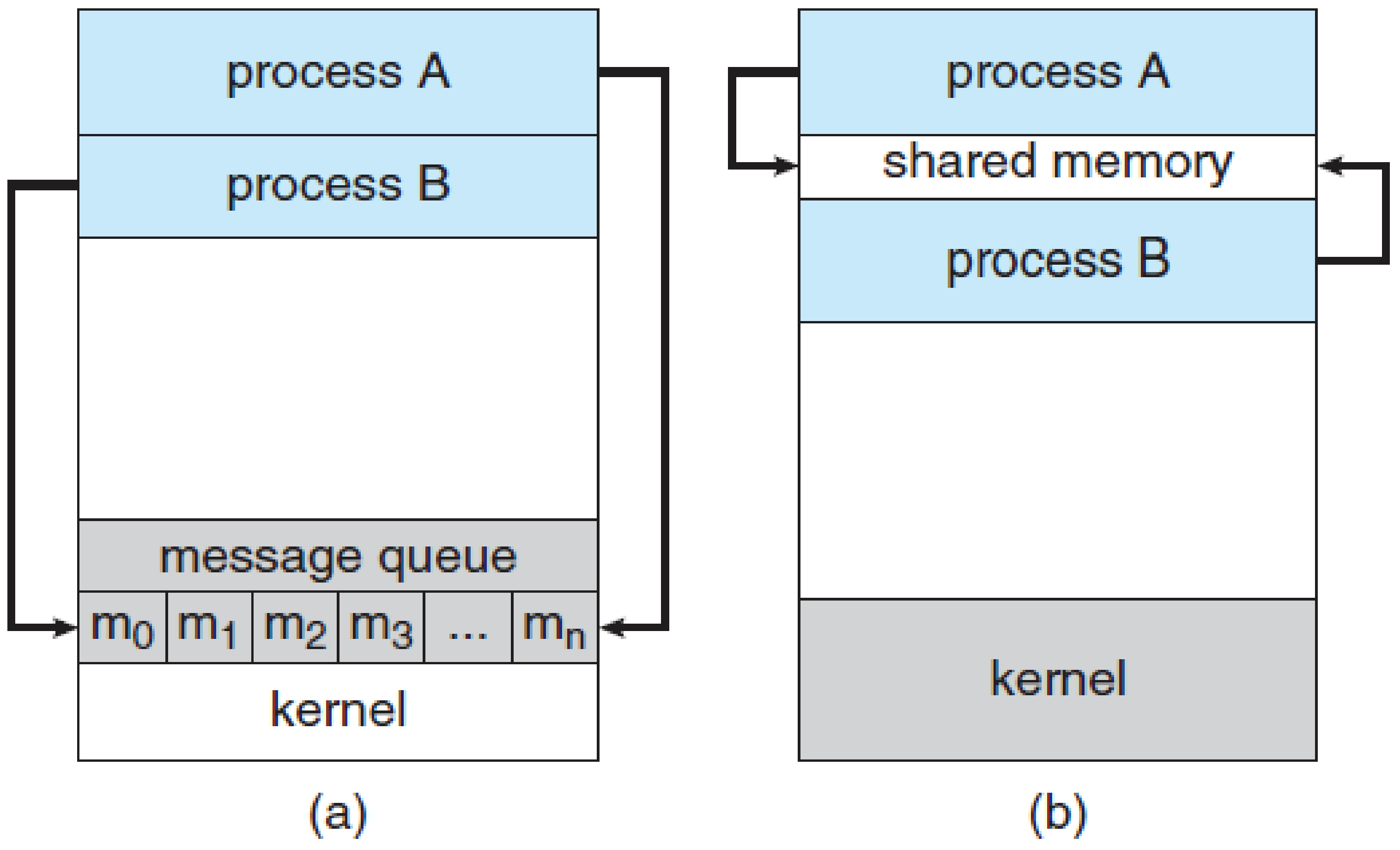


Figure 3.12 Communications models. (a) Message passing. (b) Shared memory.

Race Condition:

•In some operating systems, processes that are working together may share some common storage that each one can read and write.
•Situations like this, where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called **race conditions**.

Critical Regions:

•Sometimes a process has to access shared memory or files, or do other critical things that can lead to races. That part of the program where the shared memory is accessed is called the **critical region** or **critical section.**
• If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid race conditions.

Critical Regions(contd..):

• Although this requirement avoids race conditions, this is not sufficient for having parallel processes cooperate correctly and efficiently using shared data. We need four conditions to hold to have a good solution:

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.

- The behavior that we want is shown in Fig. of next slide.

[View full size image]

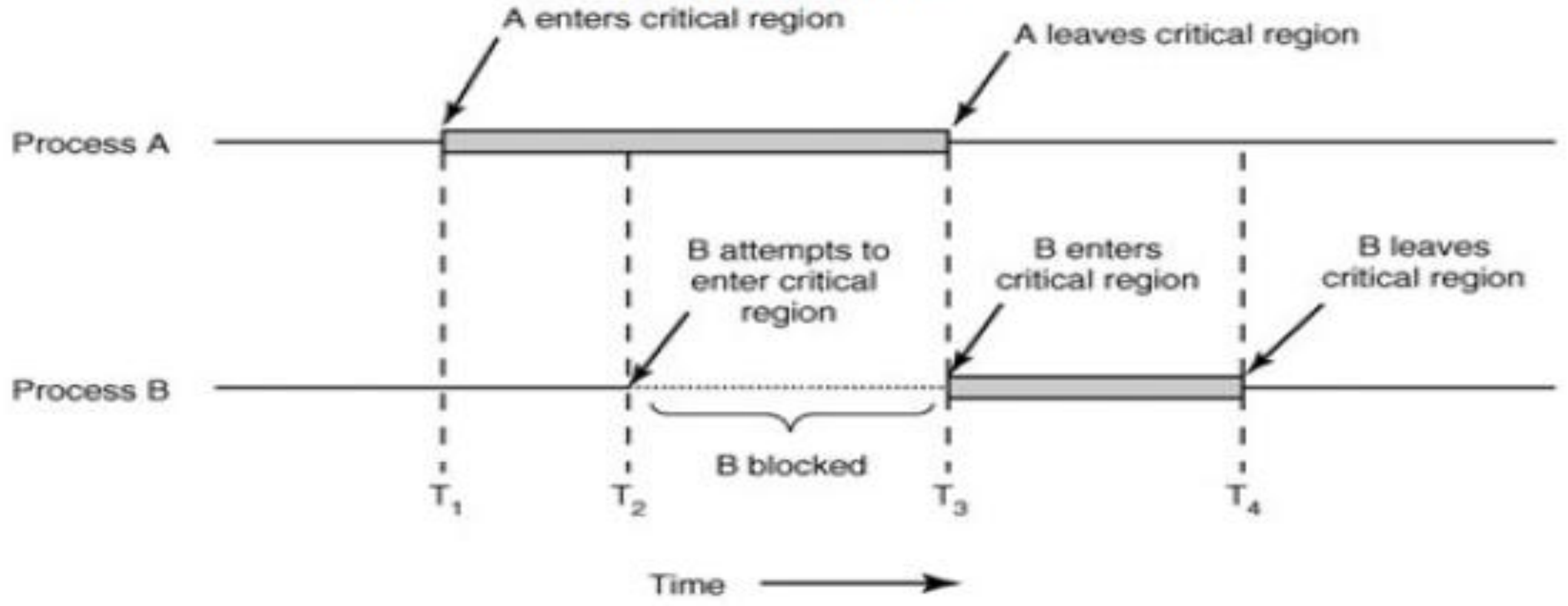
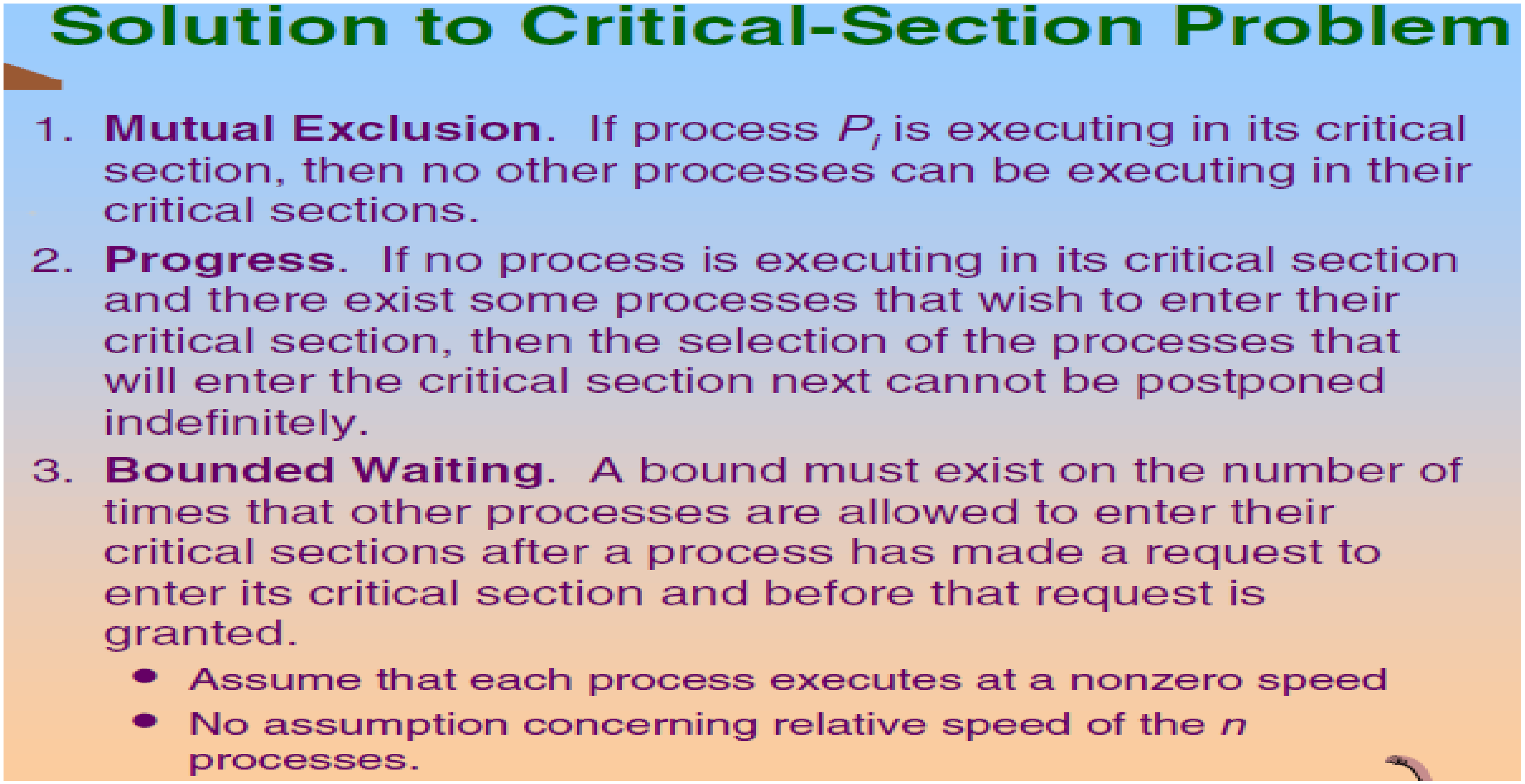


Fig: Mutual Exclusion using Critical Regions

Here process A enters its critical region at time T1 . A little later, at time T 2 process B attempts to enter its critical region but fails because another process is already in its critical region and we allow only one at a time. Consequently, B is temporarily suspended until time T 3 when A leaves its critical region, allowing B to enter immediately. Eventually B leaves (at T 4) and we are back to the original situation with no processes in their critical regions.

The Critical-Section Problem

- n processes all competing to use some shared data .
- Each process has a code segment, called critical section, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.



Solution to Critical-Section Problem

1. **Mutual Exclusion**. If process P_i 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 there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. **Bounded Waiting**. A bound must exist 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.
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the *n* processes.

Mutual Exclusion:

•How do we avoid race conditions? The key to preventing trouble here and in many other situations involving shared memory, shared files, and shared everything else is to find some way to prohibit more than one process from reading and writing the shared data at the same time. i.e, what we need is mutual exclusion some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing.

Mutual Exclusion

Disabling Interrupt:

- In a uniprocessor machine, concurrent processes cannot be overlapped; they can only be interleaved. Furthermore, a process will continue to run until it invokes an operating system service or until it is interrupted.
- Therefore, to guarantee mutual exclusion, it is sufficient to prevent a process from being interrupted. This capability can be provided in the form of primitives defined by the system kernel for disabling and enabling interrupts.

eg:

```
while (true) (
disable interrupts()
critical section enable interrupts()
) remainder
```

- *Because the critical section cannot be interrupted, mutual exclusion is guaranteed.*

Advantages Disabling Interrupt:

 It is frequently convenient for the kernel itself to disable interrupts for a few instructions while it is updating variables or lists. If an interrupt occurred while the list of ready processes, for example, was in an inconsistent state, race conditions could occur.

Disadvantages Disabling Interrupt:

1. It works only in a single processor environment.
2. Interrupts can be lost if not serviced promptly.
3. A process waiting to enter its critical section could suffer from starvation.

Lock Variables

- A single, shared, (lock) variable, initially 0.
- When a process wants to enter its critical region, it first tests the lock.
- If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.

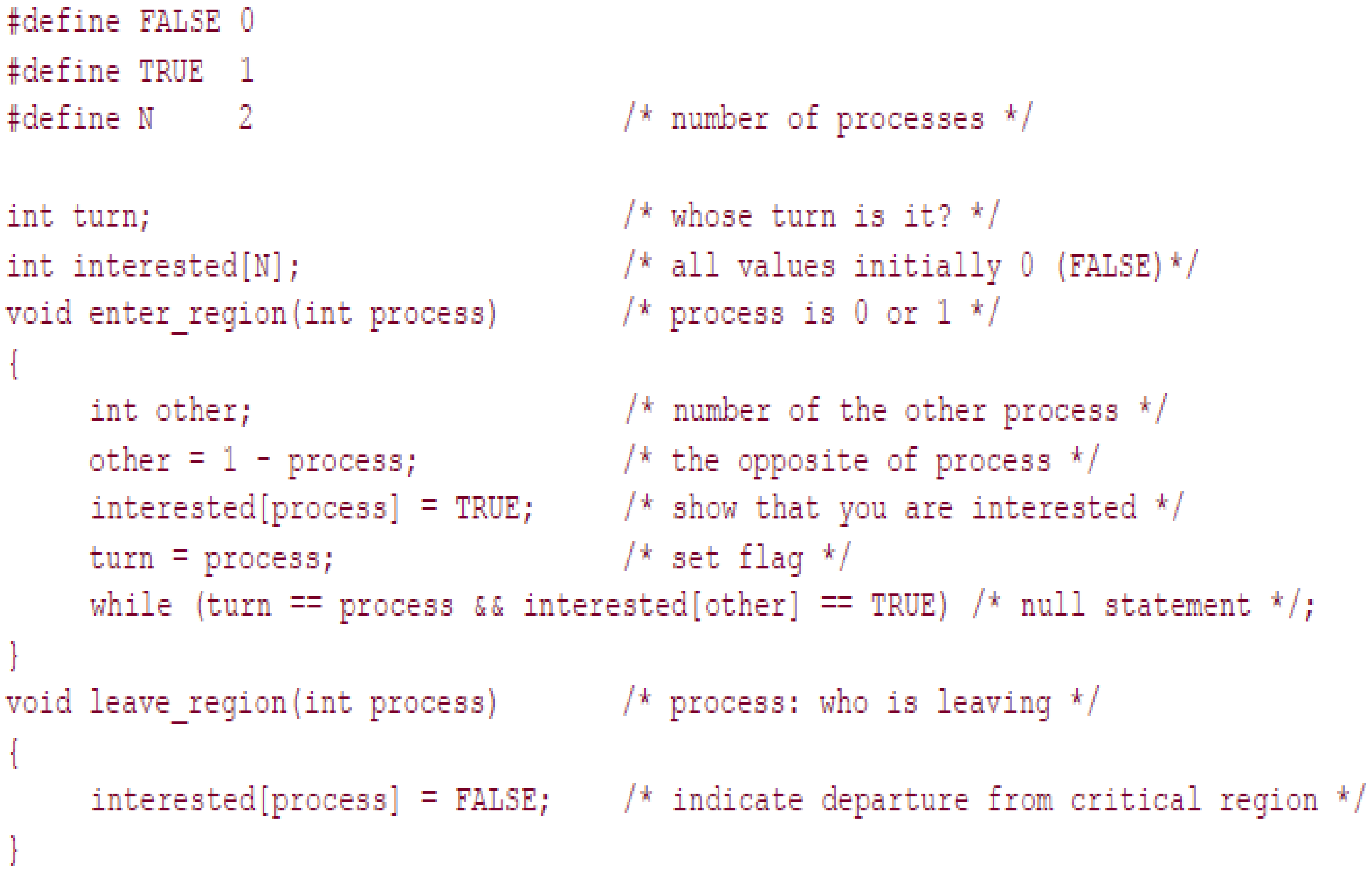
Dekker’s Algorithm:

- Dekker’s algorithm is designed to handle two processes (P0 and P1) that must access a shared resource (critical section) in a way that ensures mutual exclusion, i.e., only one process can be in the critical section at any given time. It is based on the principles of turn-taking and waiting flags to resolve conflicts between the two processes.

```
  Shared variables

  bool flag[2] = {false, false};  // Flags for each process (initially not interested)

 int turn;  // Turn variable (0 or 1)

// Process 0 (P0)

do {

   // Step 1: Indicate interest in critical section

   flag[0] = true;

   // Step 2: Wait until it's P0's turn and P1 isn't interested in the critical section

   while (flag[1]) {

      if (turn == 1) {

         // If it's P1's turn, release the critical section and wait

         flag[0] = false;

         while (turn == 1) {// Busy wait until it's P0's turn  }
```

Peterson’s Algorithm (Mutual Exclusion- Software Support) :

- In 1981, G.L P discovered a much simpler way to achieve mutual exclusion



```
#define FALSE 0
#define TRUE  1
#define N     2                     /* number of processes */

int turn;                           /* whose turn is it? */
int interested[N];                  /* all values initially 0 (FALSE)*/
void enter_region(int process)      /* process is 0 or 1 */
{
     int other;                     /* number of the other process */
     other = 1 - process;           /* the opposite of process */
     interested[process] = TRUE;    /* show that you are interested */
     turn = process;                /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */;
}
void leave_region(int process)      /* process: who is leaving */
{
     interested[process] = FALSE;   /* indicate departure from critical region */
}
```

Peterson’s Algorithm Explanation

•Before using the shared variables (i.e., before entering its critical region), each process calls enter_region with its own process number, 0 or 1, as the parameter. This call will cause it to wait, if need be, until it is safe to enter. After it has finished with the shared variables, the process calls leave_region to indicate that it is done and to allow the other process to enter, if it so desires.

• Let us see how this solution works. Initially, neither process is in its critical region. Now process 0 calls enter_region . It indicates its interest by setting its array element and sets turn to 0. Since process 1 is not interested, enter_region returns immediately. If process 1 now calls enter_region , it will hang there until interested [0] goes to FALSE , an event that only happens when process 0 calls leave_region to exit the critical region.

•Now consider the case that both processes call enter_region almost simultaneously. Both will store their process number in turn .Whichever store is done last is the one that counts; the first one is lost. Suppose that process 1 stores last, so turn is 1. When both processes come to the while statement, process 0 executes it zero times and enters its critical region. Process 1 loops and does not enter its critical region.

TSL(Test and Set Lock)-Mutual Exclusion Hardware Support

➢TSL is a hardware solution to the mutual exclusion problem. The key to success here is to have a single hardware instruction that reads a variable, stores its value in a save area, and sets the variable to a certain value. This instruction, often called test and set, once initiated will complete all of these functions without interruption. The instruction is like

TSL REGISTER, LOCK

➢(Test and Set Lock) that works as follows. It reads the contents of the memory word lock into register RX and then stores a nonzero value at the memory address lock. The operations of reading the word and storing into it are guaranteed to be indivisible—no other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

➢To use the TSL instruction, we will use a shared variable, lock, to coordinate access to shared memory. When lock is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory. When it is done, the process sets lock back to 0 using an ordinary move instruction.

TSL(contd..)

```
do {
   acquire lock
Critical section
   Release lock
remainder section
} while(TRUE);
```

fig: Solution to the critical-section problem using locks

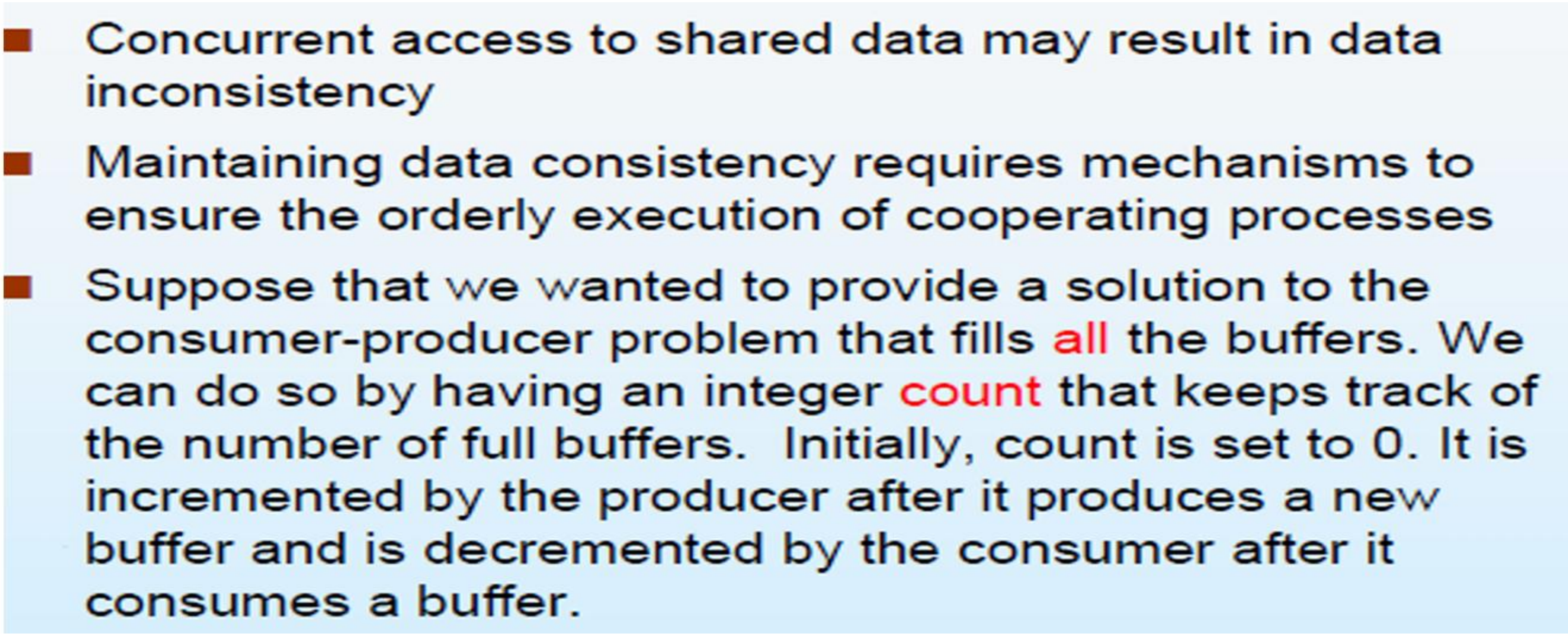
```
enter_region:
TSL REGiSTER,LOCK      / copy lock to register and set lock to 1
CMP REGiSTER,#0        / was lock zero?
JNE enter_region       / if it was nonzero, lock was set, so loop
RET                    / return to caller; critical region entered

leave_region:
MOVE LOCK,#0           / store a 0 in lock
RET                    / return to caller
```

Figure : Entering and leaving a critical region using the TSL instruction.

Sleep and Wakeup:

- Sleep and wakeup are system calls that blocks process instead of wasting CPU time when they are not allowed to enter their critical region.
- Sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up.
- The wakeup call has one parameter, the process to be awakened.

Process Synchronization:



- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer and Consumer Problem(Bounded Buffer Problem)

- To illustrate the concept of cooperating processes, let us consider the producer-consumer problem, which is a common paradigm for cooperating pro-cesses.
- A producer process produces information that is consumed by a consumer process. For example, a print program produces characters that are consumed by the printer driver. A compiler may produce assembly code, which is consumed by an assembler.
- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- A producer can produce one item while the consumer is consuming another item.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced. In this situation, the consumer must wait until an item is produced.

◆Unbounded-buffer places no practical limit on the size of the buffer- The consumer may have to wait for new items, but the producer can always produce new items.

◆Bounded-buffer assumes that there is a fixed buffer size.In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

◆The buffer may either be provided by the operating system through the use of an interprocess-communication (IPC) facility or by explicitly coded by the application programmer with the use of shared memory.

• Trouble arises when the producer wants to put a new item in the buffer, but it is already full. The solution is for the producer to go to sleep, to be awakened when the consumer has removed one or more items. Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up.

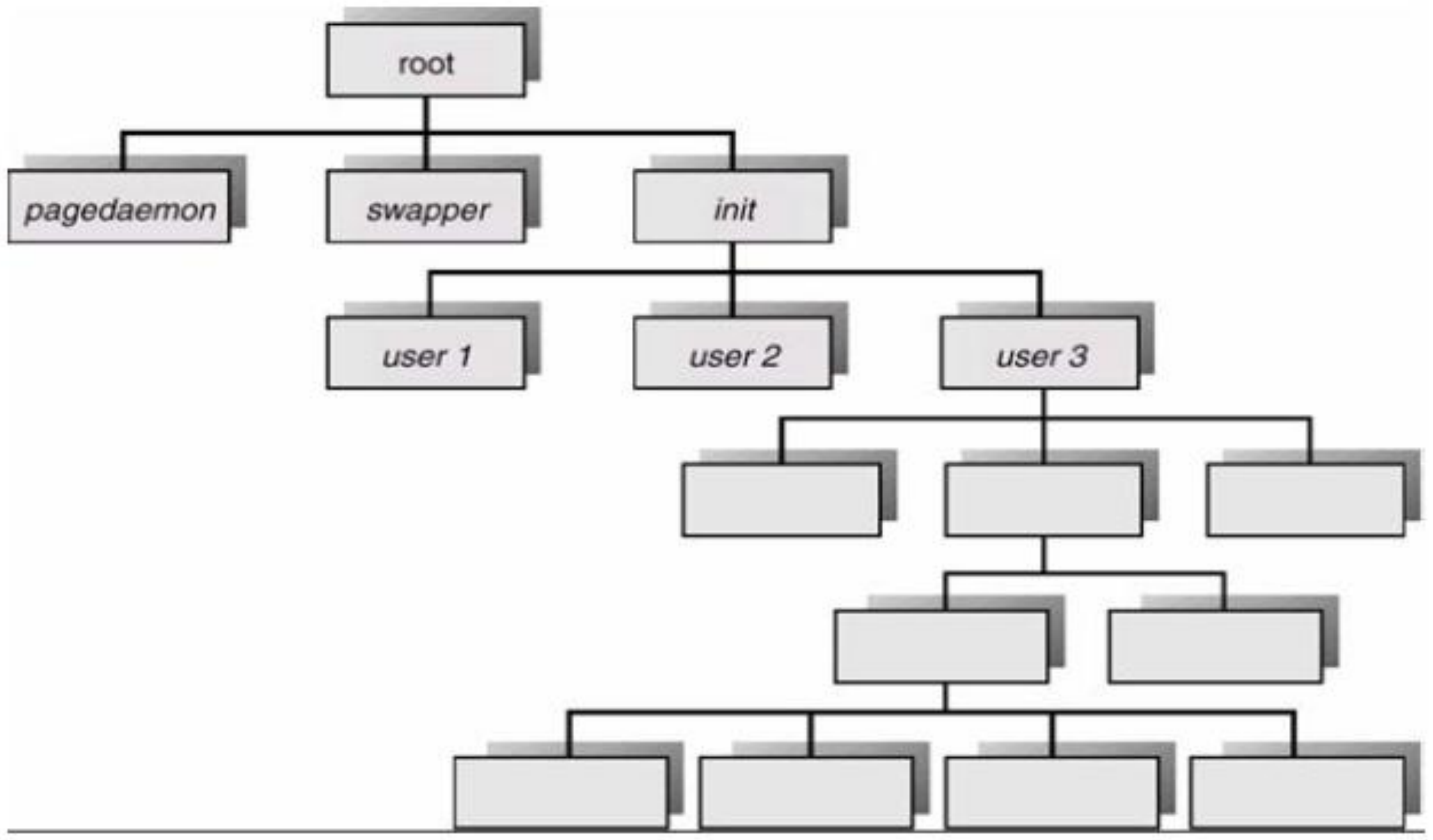
```
#define N 100                                      /* number of slots in the buffer */
int count = 0;                                     /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {                                 /* repeat forever */
        item = produce_item();                     /* generate next item */
        if (count = N) sleepQ;                     /* if buffer is full, go to sleep */
        insert_item(item);                         /* put item in buffer */
        count = count + 1;                         /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);          /* was buffer empty? */
    }
}


void consumer(void)
{
    int item;

    while (TRUE) {                                 /* repeat forever */
        if (count = 0) sleep();                    /* if buffer is empty, got to sleep */
        item = remove_item();                      /* take item out of buffer */
        count = count - 1;                         /* decrement count of items in buffer */
        if (count == N - 1 ) wakeup(producer);     /* was buffer full? */
        consume_item(item);                        /* print item */
    }
}
```

Figure 2-27. The producer-consumer problem with a fatal race condition.

Operations in Process:
<u>Process Creation</u>

- A process may create several new processes, via a create-process system call, during execution.
- Parent process creates children processes, which, in turn create other processes, forming a tree of processes.
- Resource sharing, such as CPU time, memory, files, I/O devices …
  - – Parent and children share all resources.
  - – Children share subset of parent’s resources.
  - – Parent and child share no resources.
- When a process creates a new process, two possibilities exist in terms of execution:
  - – Parent and children execute concurrently.
  - – Parent waits until children terminate.
- There are also two possibilities in terms of the address space of the new process:
  - – Child duplicate of parent.
  - – Child has a program loaded into it.
- UNIX examples
  - – **fork** system call creates new process

A Tree of Processes On A Typical UNIX System



Process Termination

- Process executes last statement and asks the operating system to delete it by using the **exit** system call.
  - Output data from child to parent via **wait** system call.
  - Process' resources are deallocated by operating system.
- Parent may terminate execution of children processes via **abort** system call for a variety of reasons, such as:
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Message Passing

When Process interact with one another,two fundamental requirements must be satisfied: synchronization and communication. One approach to providing both of these functions is message passing, i.e message passing allows information exchange between machines. This method of interprocess communication uses two primitives, send and receive, system calls. As such, they can easily be put into library procedures, such as

```
        Send(destination, & message);
    and
        Receive(source, & message);
```

The former call sends a message to a given destination and the latter one receives a message from a given source.

Design Issues for Message Passing Systems

- ➢Message passing systems have many challenging problems and design issues, especially if the communicating processes are on different machines connected by a network. For example, messages can be lost by the network. To guard against lost messages, the sender and receiver can agree that as soon as a message has been received, the receiver will send back a special <u>acknowledgement</u> message. If the sender has not received the acknowledgement within a certain time interval, it retransmits the message.
- ➢To avoid duplication, consecutive sequence numbers can be put in each original messages.

Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.
- The representation of a monitor type consists of declarations of variables whose values define the state of an instance of the type, as well as the bodies of procedures or functions that implement operations on the type.

```
        monitor monitor-name
{
       shared variable declarations
        procedure body P1 (...) {
        ...
        }
        procedure body P2 (...) {
        ...
        }
       procedure body Pn (...) {
        ...
       }
        {
       initialization code
        }
}
```

Monitors(contd..)

- To allow a process to wait within the monitor, a **condition** variable must be declared, as

  condition x, y;

- Condition variable can only be used with the operations **wait** and **signal.**

  The operation

  x.wait();

  means that the process invoking this operation is suspended until another process invokes

  x.signal();

  The **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect.

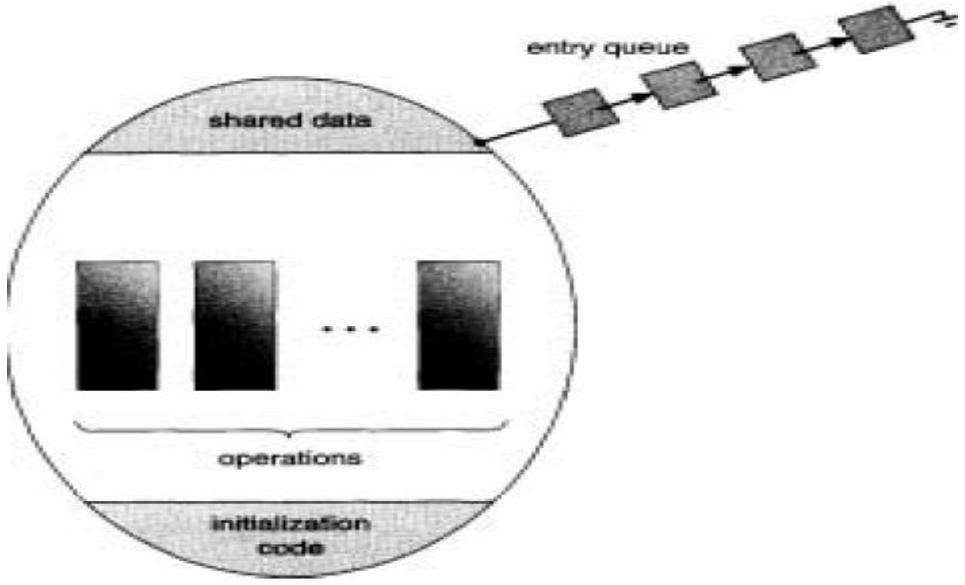


Figure 7.20 Schematic view of a monitor.

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

  -
    ```
    wait (S) {
         while S <= 0
              ; // no-op
            S--;
      }
    ```

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

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion
  - Semaphore S;   // initialized to 1
  - wait (S);

```
        Critical Section
 signal (S);
```

Producer Consumer Problem Using Semaphore

Producer Process

The code that defines the producer process is given below:

```
do {
      .
      . PRODUCE ITEM
      .
      wait(empty);
      wait(mutex);
      .
      . PUT ITEM IN BUFFER
      .
      signal(mutex);
      signal(full);

} while(1);
```

- In the above code, mutex, empty and full are semaphores. Here mutex is initialized to 1, empty is initialized to n (maximum size of the buffer) and full is initialized to 0.
- The mutex semaphore ensures mutual exclusion. The empty and full semaphores count the number of empty And full spaces in the buffer.
- After the item is produced, wait operation is carried out on empty. This indicates that the empty space in the buffer has decreased by 1. Then wait operation is carried out on mutex so that consumer process cannot interfere.
- After the item is put in the buffer, signal operation is carried out on mutex and full. The former indicates that consumer process can now act and the latter shows that the buffer is full by 1.

Consumer Process

The code that defines the consumer process is given below:

```
do {
    wait(full);
    wait(mutex);
    .      .
    .  REMOVE ITEM FROM BUFFER
    .
    signal(mutex);
    signal(empty);
    .
    . CONSUME ITEM
    .
} while(1);
```

- The wait operation is carried out on full. This indicates that items in the buffer have decreased by 1. Then wait operation is carried out on mutex so that producer process cannot interfere.
- Then the item is removed from buffer. After that, signal operation is carried out on mutex and empty. The former indicates that consumer process can now act and the latter shows that the empty space in the buffer has increased by 1.

Classical IPC Problems

There are number of different synchronization problem. Some of them are:
The Dining-Philosophers Problem:
<u>Statement</u>: Consider five philosophers who spend their lives thinking and eating. The philosophers share a common circular table surrounded by five chairs, each belonging to one philosopher. In the centre of the table there is a bowl of rice, and the table is laid with five single chopsticks. When a philosophers gets hungry she tries to pick up her left and right chopstick, one at a time, in either order. If successful in acquiring two chopsticks, she eats for a while, then puts down the chopsticks and continues to think. The key question is: Can you write a program for each philosopher that does what it is supposed to do and never gets stuck?

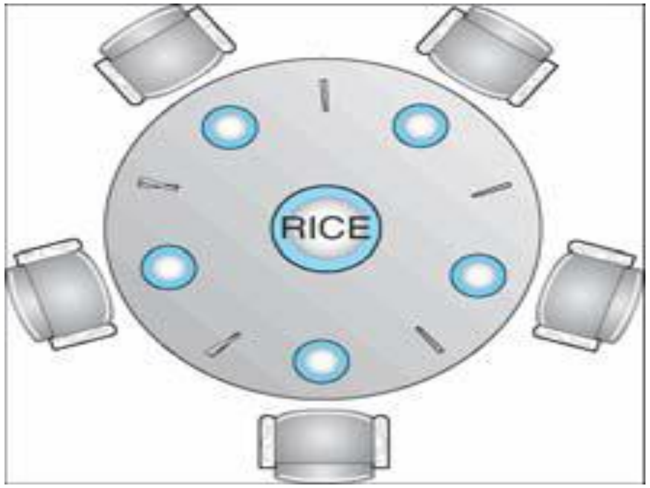


fig:The situation of the dining philosophers

Solution to the Dining-Philosophers Problem:

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore; she releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus , the shared data are

```
          semaphore chopstick  [5];
```

The structure of Philosopher i is shown below:

```
     do {
     wait ( chopstick[i] );
     wait ( chopStick[ (i + 1) % 5] );
       . . . .
     // eat
       . . . .
     signal ( chopstick[i] );
     signal (chopstick[ (i + 1) % 5] );
       . . .
     // think
       . . .
     } while (true) ;
```

fig: the structure of philosopher i.

But if this solution simply applied deadlock will occur.

Solution to the Dining-Philosophers Problem(contd…):

Several possible remedies to the deadlock problem are listed. Here is a solution to the dining-philosophers problem that ensures freedom from deadlocks.

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this she must pick them up in a critical section).
- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

Readers Writer Problems:

• The dining philosophers problem is useful for modeling processes that are competing for exclusive access to a limited number of resources, such as I/O devices. Another famous problem is the readers and writers problem which models access to a database (Courtois et al., 1971). Imagine, for example, an airline reservation system, with many competing processes wishing to read and write it. It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other process may have access to the database, not even a reader. The question is how do you program the readers and the writers? One solution is shown below.

```
typedef int semaphore; /* use your imagination */
semaphore mutex = 1; /* controls access to 'rc' */
semaphore db = 1; /* controls access to the database */

int rc = 0; /* # of processes reading or wanting to */
```

Readers Writer problems(contd..)

```
void reader(void)
{
    while (TRUE){ /* repeat forever */
        down(&mutex); /* get exclusive access to 'rc' */
        rc = rc + 1; /* one reader more now */
        if (rc == 1) down(&db); /* if this is the first reader ... */
        up(&mutex); /* release exclusive access to 'rc' */
        read_data_base(); /* access the data */
        down(&mutex); /* get exclusive access to 'rc' */
        rc = rc - 1; /* one reader fewer now */
        if (rc == 0) up(&db); /* if this is the last reader ... */
        up(&mutex); /* release exclusive access to 'rc' */
        use_data_read(); /* noncritical region */
    }
}
void writer(void)
{
    while (TRUE){ /* repeat forever */
        think_up_data(); /* noncritical region */
        down(&db); /* get exclusive access */
        write_data_base(); /* update the data */
        up(&db); /* release exclusive access */
    }
}
```

fig: A solution to the readers writer problem

Readers Writer problems(contd..)

• In this solution, the first reader to get access to the data base does a down on the semaphore *db .* Subsequent readers merely have to increment a counter, *rc . As readers leave, they decrement* the counter and the last one out does an up on the semaphore, allowing a blocked writer, if there is one, to get in.
• Suppose that while a reader is using the data base, another reader comes along. Since having two readers at the same time is not a problem, the second reader is admitted. A third and subsequent readers can also be admitted if they come along.
• Now suppose that a writer comes along. The writer cannot be admitted to the data base, since writers must have exclusive access, so the writer is suspended. Later, additional readers show up. As long as at least one reader is still active, subsequent readers are admitted. As a consequence of this strategy, as long as there is a steady supply of readers, they will all get in as soon as they arrive. The writer will be kept suspended until no reader is present. If a new reader arrives, say, every 2 seconds, and each reader takes 5 seconds to do its work, the writer will never get in.
•To prevent this situation, the program could be written slightly differently: When a reader arrives and a writer is waiting, the reader is suspended behind the writer instead of being admitted immediately. In this way, a writer has to wait for readers that were active when it arrived to finish but does not have to wait for readers that came along after it. The disadvantage of this solution is that it achieves less concurrency and thus lower performance.

The Sleeping Barber Problem

•Another classical IPC problem takes place in a barber shop. The barber shop has one barber, one barber chair, and n chairs for waiting customers, if any, to sit on. If there are no customers present, the barber sits down in the barber chair and falls asleep, as illustrated in [Figure 2-35]. When a customer arrives, he has to wake up the sleeping barber. If additional customers arrive while the barber is cutting a customer's hair, they either sit down (if there are empty chairs) or leave the shop (if all chairs are full). The problem is to program the barber and the customers without getting into race conditions. This problem is similar to various queueing situations, such as a multiperson helpdesk with a computerized call waiting system for holding a limited number of incoming calls.
•This solution uses three semaphores, customers, which counts waiting customers (excluding the customer in the barber chair, who is not waiting), barbers, the number of barbers (0 or 1) who are idle, waiting for customers, and mutex, which is used for mutual exclusion. We also need a variable, waiting, which also counts the waiting customers. The reason for having waiting is that there is no way to read the current value of a semaphore. In this solution, a customer entering the shop has to count the number of waiting customers. If it is less than the number of chairs, he stays; otherwise, he leaves.
•Our solution is shown [below]. When the barber shows up for work in the morning, he executes the procedure *barber, causing him to block on the semaphore customers* because it is initially 0. The barber then goes to sleep, as shown in [Figure 2.35]. He stays asleep until the first customer shows up.
•When a customer arrives, he executes *customer, starting by acquiring mutex to enter a* critical region. If another customer enters shortly thereafter, the second one will no be able to do anything until the first one has released *mutex. The customer then checks to* see if the number of waiting customers is less than the number of chairs. If not, he releases *mutex and leaves without a haircut.*

The sleeping Barber Preoblem(contd…)

•If there is an available chair, the customer increments the integer variable, *waiting.* Then he does an *Up on the semaphore customers, thus waking up the barber. At this* point, the customer and the barber are both awake. When the customer releases *mutex,* the barber grabs it, does some housekeeping, and begins the haircut.

•When the haircut is over, the customer exits the procedure and leaves the shop. Unlike our earlier examples, there is no loop for the customer because each one gets only one haircut. The barber loops, however, to try to get the next customer. If one is present, a haircut is given. If not, the barber goes to sleep.

```
#define CHAIRS 5 /* # chairs for waiting customers */
typedef int semaphore; /* use your imagination */
semaphore customers = 0; /* # of customers waiting for service */
semaphore barbers = 0; /* # of barbers waiting for customers */
semaphore mutex = 1; /* for mutual exclusion */
int waiting = 0; /* customer are waiting (not being cut) */
void barber(void)
{
while (TRUE) {
down(&customers); /* go to sleep if # of customers is 0 */
down(&mutex); /* acquire access to "waiting' */
waiting = waiting - 1; /* decrement count of waiting customers */
up(&barbers); /* one barber is now ready to cut hair */
up(&mutex); /* release 'waiting' */
cut_hair(); /* cut hair (outside critical region */
}
}
void customer(void)
{
down(&mutex); /* enter critical region */
if (waiting < CHAIRS) { /* if there are no free chairs, leave */
waiting = waiting + 1; /* increment count of waiting customers */
up(&customers); /* wake up barber if necessary */
up(&mutex); /* release access to 'waiting' */
down(&barbers); /* go to sleep if # of free barbers is 0 */
get_haircut(); /* be seated and be served */
} else {
up(&mutex); /* shop is full; do not wait */
}
}
```

fig:A solution to the Sleeping Barber Problem

Seriazability:

It is a way to maintain database consistency for concurrent transactions. Two ways to achieve seriazability: Lock based protocols and Timestamp protocols.

Lock Based Protocols is a mechanism in which a transaction cannot Read or Write the data until it acquires an appropriate lock. Lock based protocols help to eliminate the concurrency problem in DBMS for simultaneous transactions by locking or isolating a particular transaction to a single user.

A lock is a data variable which is associated with a data item. This lock signifies that operations that can be performed on the data item. Locks in DBMS help synchronize access to the database items by concurrent transactions. All lock requests are made to the concurrency-control manager. Transactions proceed only once the lock request is granted.

Timestamp based Protocol is an algorithm which uses the System Time or Logical Counter as a timestamp to serialize the execution of concurrent transactions. The Timestamp-based protocol ensures that every conflicting read and write operations are executed in a timestamp order.

The older transaction is always given priority in this method. It uses system time to determine the time stamp of the transaction. This is the most commonly used concurrency protocol.

Lock-based protocols help you to manage the order between the conflicting transactions when they will execute. Timestamp-based protocols manage conflicts as soon as an operation is created.

Scheduling:

- When a computer is multiprogrammed, it frequently has multiple processes competing for the CPU at the same time. When more than one process is in the ready state and there is only one CPU available, a choice has to be made which process to run next. The part of the operating system that makes the choice is called the scheduler, and the algorithm it uses is called the scheduling algorithm and the mechanism is called scheduling.

<u>Preemptive Scheduling and Non Preemptive Scheduling</u>

▪ **Non preemptive:** In this case, once a process is in the running state, It continues to execute until (a) it terminates or (b) blocks itself to wait for I/O or to request some operating system service.

▪ **Preemptive:** The currently running process may be interrupted and moved to the ready state by the operating system. The decision to preempt may be performed when a new process arrives or periodically based on a clock interrupt OR when a new process switches from the waiting state to the ready state(for example, at completion of I/O)

<u>Scheduling Technique</u>

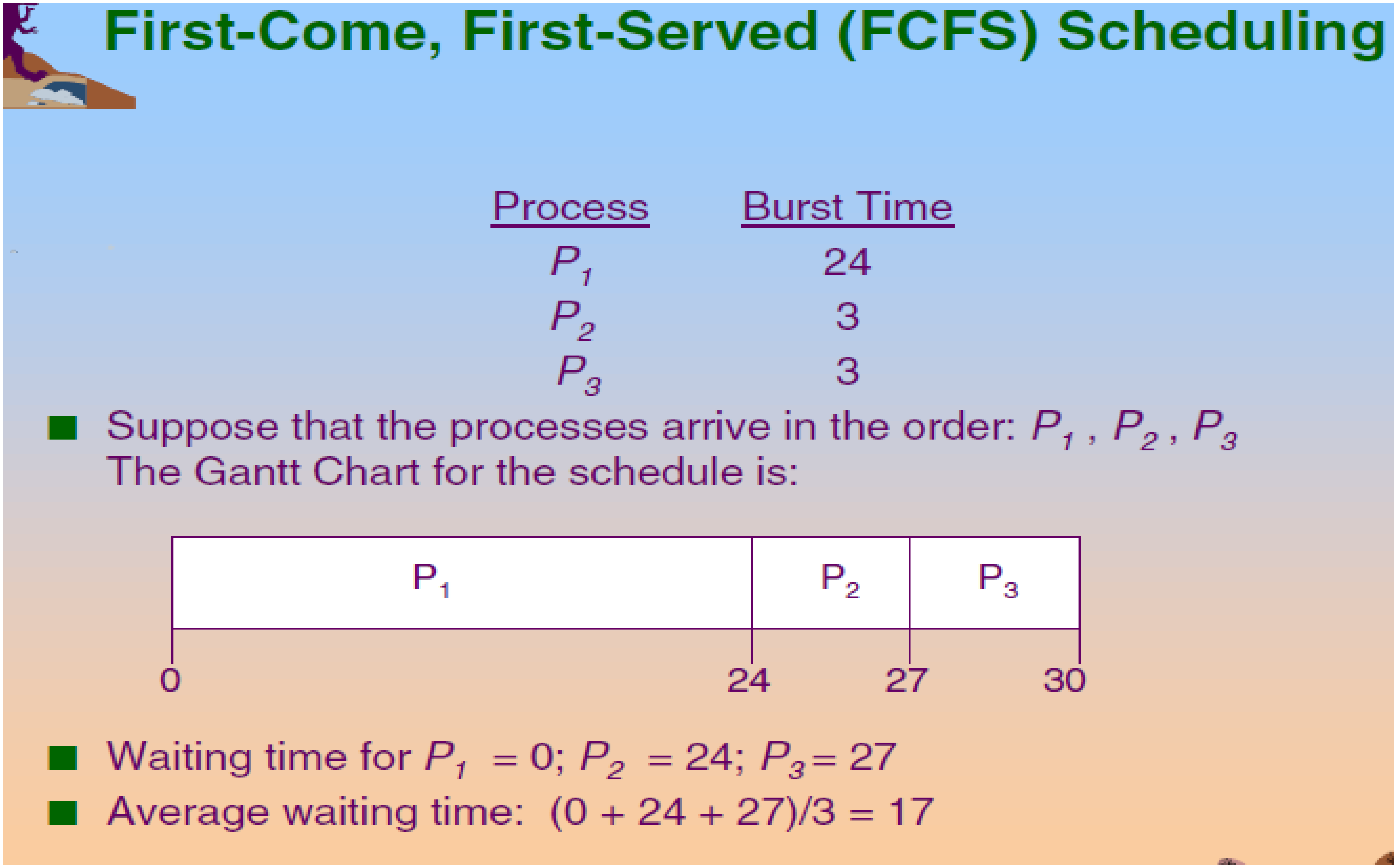
➢**First-Come-First-Served(First-In-First-Out) Scheduling:** With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue.

Scheduling Criteria :

The criteria include the following:

- CPU utilization: The CPU should keep as busy as possible. CPU utilization may range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
- Throughput: If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes completed per time unit, called throughput.
- Turnaround time: From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. IT is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- Waiting Time: Waiting time is the sum of the periods spent waiting in the ready queue.
- Response Time: Response time is the time it takes to start responding.

Consider the following set of processes that arrive at the time 0, with the length of the CPU burst given in milliseconds.



First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:

P1
P2
P3
0
24
27
30

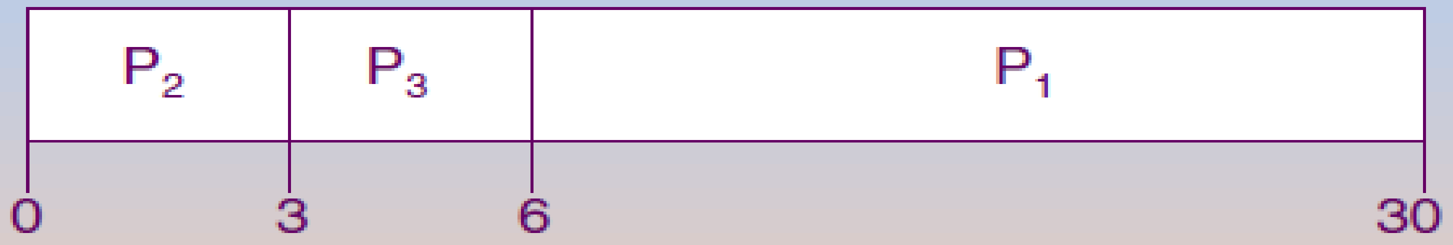
- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

P_2 , P_3 , P_1 .

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case.
- *Convoy effect* short process behind long process

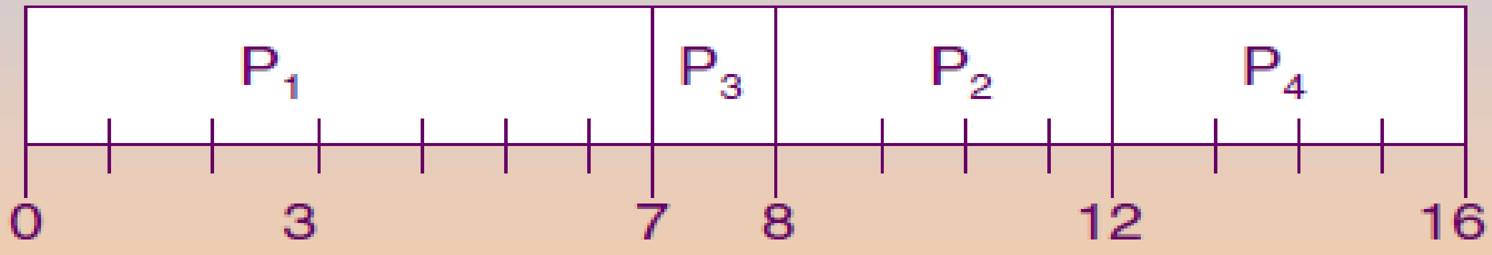
Shortest-Job-First (SJR) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst.
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF).
- SJF is optimal – gives minimum average waiting time for a given set of processes.

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

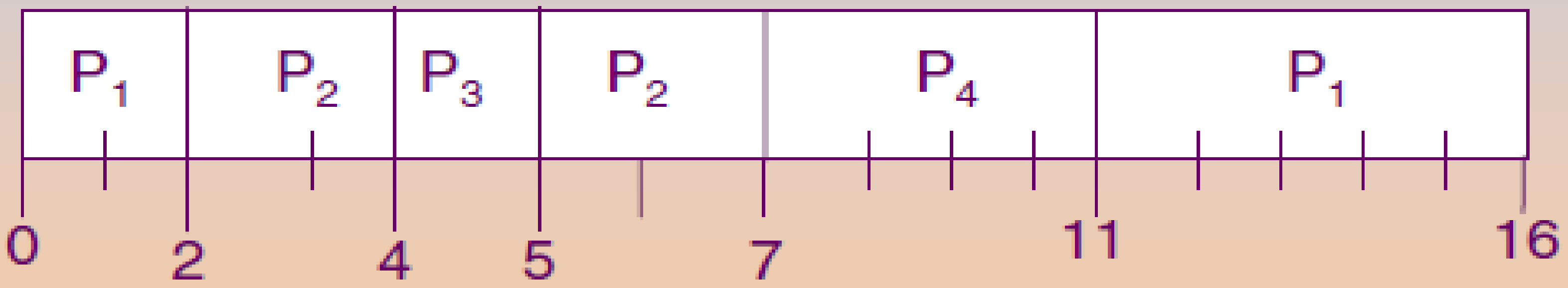


- Average waiting time = (0 + 6 + 3 + 7)/4 - 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



- Average waiting time = (9 + 1 + 0 +2)/4 - 3

Priority Scheduling

➢ A priority number (integer) is associated with each process
 The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority).

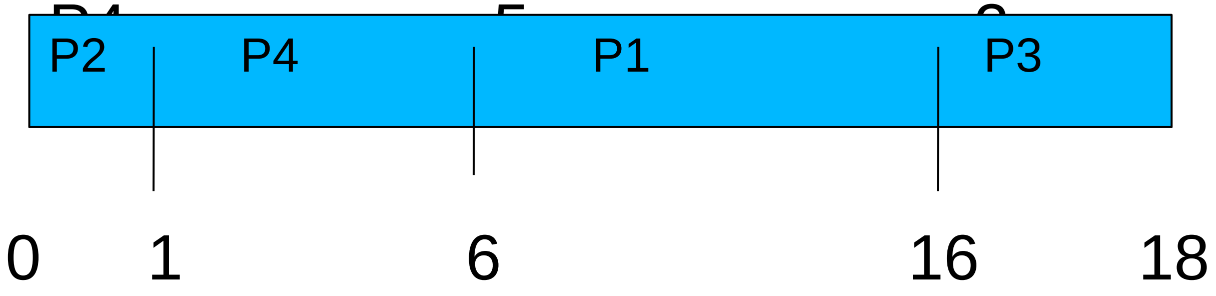
✦ Preemptive ✦ nonpreemptive

➢ SJF is a priority scheduling where priority is the predicted next CPU burst time.

 Problem ≡ Starvation – low priority processes may never execute.
 Solution ≡ Aging – as time progresses increase the priority of the process.

Consider following set of process arrived at time 0

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	5	2



Average Waiting Time=8.2 ms

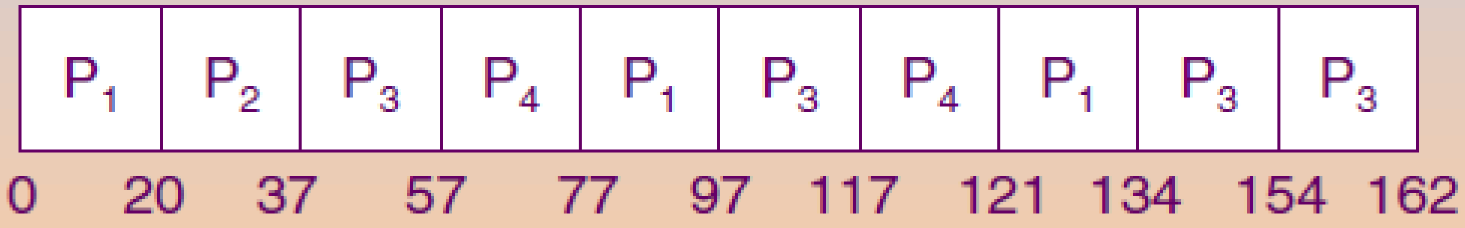
Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high.

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better *response.*

Highest-Response-Ratio-Next(HRN) Scheduling

- HRN is a nonpreemptive scheduling discipline in which the priority of each job is a function not only of the job’s service time but also of the amount of the time the job has been waiting for service. Once a job gets the CPU, it runs to completion. Dynamic priorities in HRN are calculated according to the formula

$$\text{priority} = \frac{\text{time waiting + Service}}{\text{Service Time}}$$

- Because the service time appears in the denominator, shorter job will get preference. But because time waiting appears in the numerator, longer jobs that have been waiting will also be given favorable sum.

Fair Share Scheduling (FSS)

- Fair-share CPU scheduling is a scheduling algorithm used in operating systems that aims to allocate CPU resources fairly among different user or process groups.
- The fair-share scheduler assigns a weight to each process or group based on its historical usage and allocates CPU resources based on these weights, ensuring that no group is starved of resources for an extended period of time.
- It allows for better resource utilization and provides equal opportunities for all groups to access the CPU.
- Fair-share scheduling is commonly used in multi-user systems and virtualized environments where multiple users or virtual machines share a single physical CPU.
- Time quantum is the processor time allowed for a process to run. But in a round-robin scheduling where the time slices or time quanta are allocated equally in a circular order, due to the distribution of time slices in a circular manner, any equal amount of time quanta will produce a similar output, therefore, an arbitrary distribution is required in this scenario.

Deadline Scheduling (Real Time Scheduling)

➢In deadline scheduling certain jobs are scheduled to be completed by a specific time or deadline. These jobs may have very high value if delivered on time and may be worthless if delivered later than the deadline. Deadline scheduling is complex for many reasons.

- The user must supply the precise resource requirements of the job in advance. Such information is rarely available.
- The system must run the deadline job without severely degrading service to other users.
- If many deadline jobs are to be active at once, scheduling could become so complex that sophisticated optimization methods might be needed to ensure that deadlines are met.

Lottery Scheduling:

Here, the scheduling of processes is done at random. Preemptive or non-preemptive lottery scheduling is possible. It also addresses the issue of hunger. Giving each process at least one lottery ticket ensures that it will be chosen at some point during the scheduling process.

Every process in this scheduling has several tickets, and the scheduler selects one at random. The process with that ticket is the winner, and it is executed for a time slice, before the scheduler selects another ticket.

Example

If we have two processes A and B, each with 60 and 40 tickets out of a total of 100, A has a 60% CPU share, while B has a 40% CPU share. These percentages are calculated in a probabilistic rather than deterministic manner.

Explanation

•A and B are the two processes we have. A has 60 tickets (from 1 to 60), while B has 40 tickets (ticket no. 61 to 100).

•The scheduler chooses a number between 1 and 100 at random. If the chosen number is between 1 and 60, A is executed; otherwise, B is executed.

•The table will look like

Ticket number - 73 82 23 45 32 87 49 39 12 09.
Resulting Schedule - B B A A A B A A A A.

Guaranteed Scheduling

- Make real promises to the users about performance.
- If there are n users logged in while working, it receive about 1/n of the CPU power.
- Similarly, on a single-user system with n processes running, all things being equal, each one should get 1/n of the CPU cycles.
- To make good on this promise, the system must keep track of how much CPU each process has had since its creation.
- CPU Time entitled= (Time Since Creation)/n
- Then compute the ratio of Actual CPU time consumed to the CPU time entitled.
- A ratio of 0.5 means that a process has only had half of what it should have had, and a ratio of 2.0 means that a process has had twice as much as it was entitled to.
- The algorithm is then to run the process with the lowest ratio until its ratio has moved above its closest competitor.

Threads

- A thread, sometimes called a lightweight process (LWP), is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It is defined as the unit of dispatching

- It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.

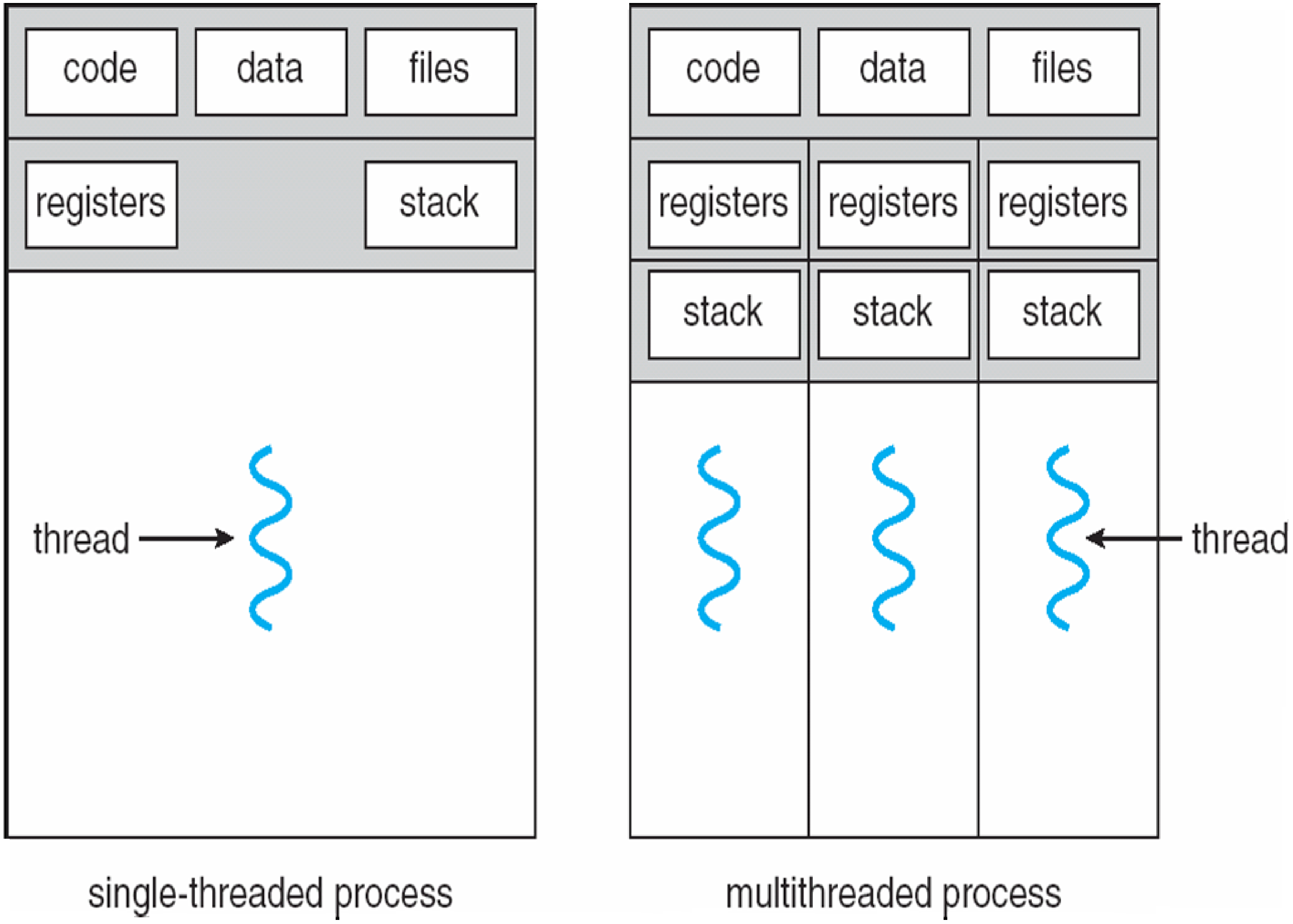
- A traditional (or heavyweight) process has a single thread of control. If the process has multiple threads of control, it can do more than one task at a time.

Multithreading

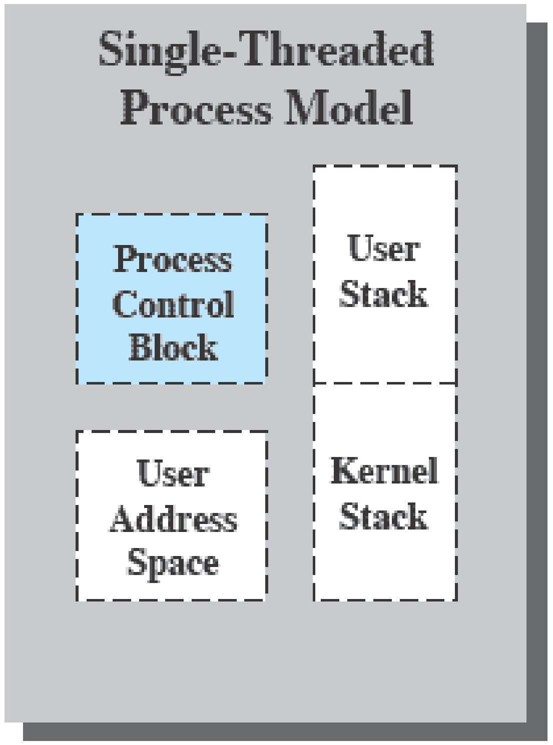
- Multithreading is the ability of an operating System to support multiple threads of execution within a single process

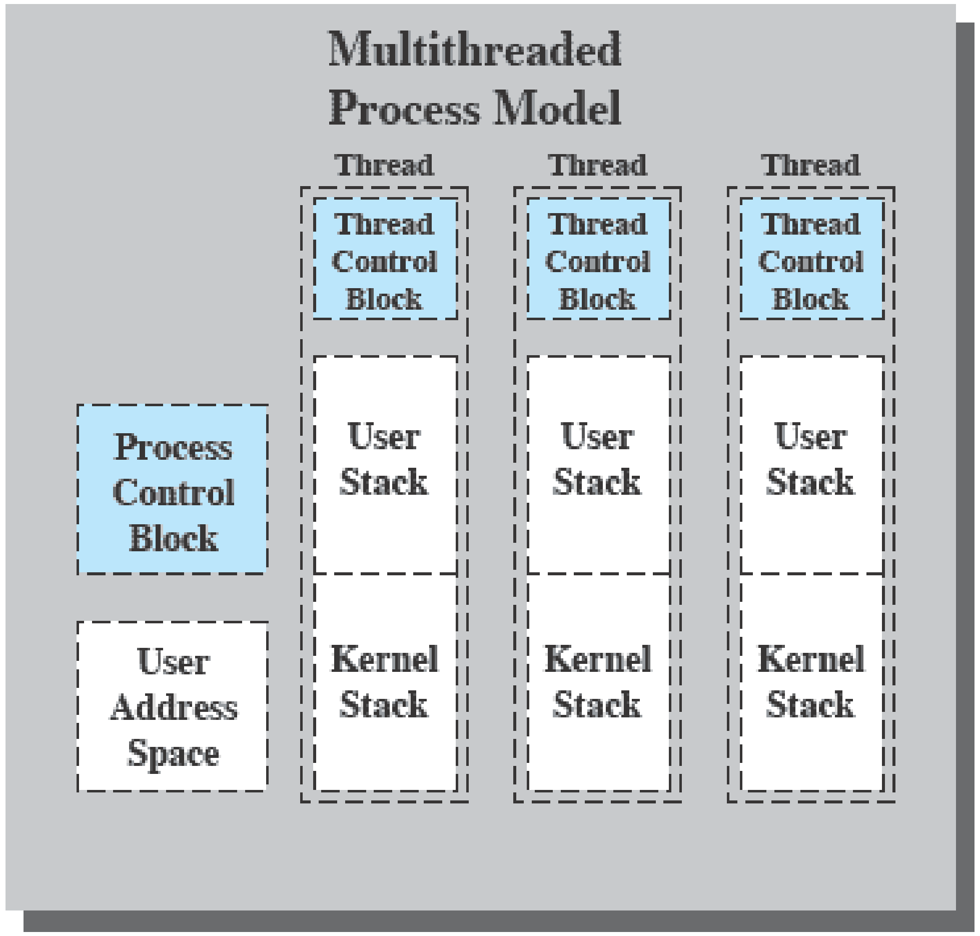
- Multiple threads run in the same address space, share the same memory areas
  - The creation of a thread only creates a new thread control structure, not a separate process image

Multithreaded Process Model



Single-threaded vs Multi-threaded





Thread Usage

- Many software packages that run on modern desktop PCs are multithreaded. An application typically is implemented as a separate process with several threads of control.

- A web browser might have one thread display images or text while another thread retrieves data from the network.

- A word processor may have a thread for displaying graphics, another thread for reading keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

Advantages of Threads

The benefits of multithreaded programming can be broken down into four major categories:

1. Responsiveness: Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. For instance, a multithreaded web browser could still allow user interaction in one thread while an image is being loaded in another thread.

2. Resource sharing: By default, threads share the memory and the resources of the process to which they belong. The benefit of code sharing is that it allows an application to have several different threads of activity all within the same address space.

Advantages of Threads(contd..)

3. Economy: Allocating memory and resources for process creation is costly. Alternatively, because threads share resources of the process to which they belong, it is more economical to create and context switch threads.

4. Utilization of multiprocessor architectures: The benefits of multithreading can be greatly increased in a multiprocessor architecture, where each thread may be running in parallel on a different processor. A single-threaded process can only run on one CPU, no matter how many are available.Multithreading on a multi-CPU machine increases concurrency.

Types of Threads:

Threads is implemented in two ways:

- User Level Threads
- Kernel Level Threads

User Space Threads

- ➢User threads are supported above the kernel and are implemented by a thread library at the user level.

- ➢The library provides support for thread creation, scheduling, and management with no support from the kernel.

- ➢Because the kernel is unaware of user-level threads, all thread creation and scheduling are done in user space without the need for kernel intervention.Therefore, user-level threads are generally fast to create and manage.

- ➢Its drawback is that if the kernel is single-threaded, then any user-level thread performing a blocking system call will cause the entire process to block, even if other threads are available to run within theapplication.

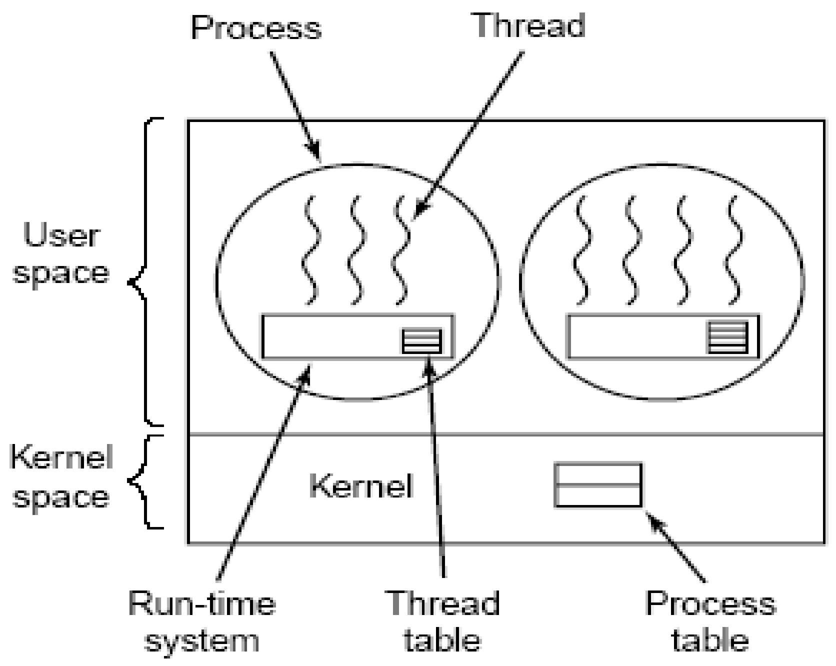
Kernel Space Threads

- ➢Kernel threads are supported directly by the operating system. The kernel performs thread creation, scheduling, and management in kernel space.

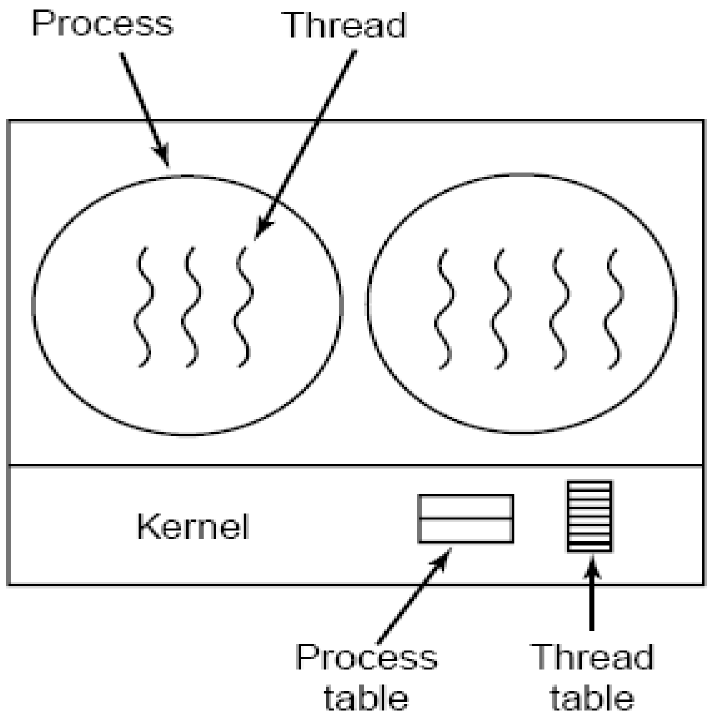
- ➢Because thread management is done by the operating system, kernel threads are generally slower to create and manage than are user threads.

- ➢However, since the kernel is managing the threads, if a thread performs a blocking system call, the kernel can schedule another thread in the application for execution.

- ➢Also, in a multiprocessor environment, the kernel can schedule threads on different processors.



User space threads



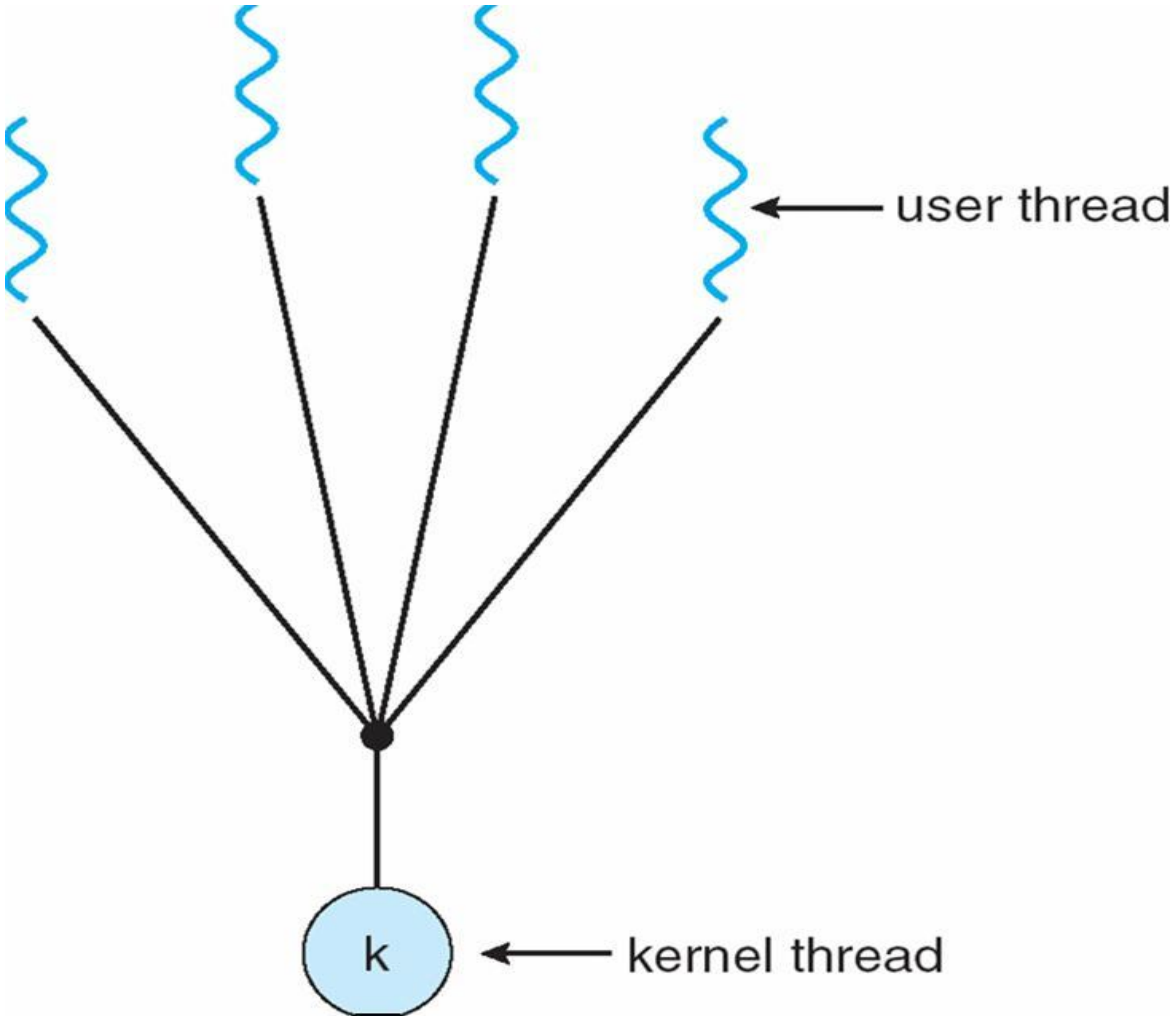
Kernel space threads

Multithreading Models

Many systems provide support for both user and kernel threads, resulting in different multithreading models.

Many-to-One Model:

- All user-level threads of one process mapped to a single kernel-level thread
- Thread management in user space
    - Efficient
    - Application can run its own scheduler implementation
- One thread can access the kernel at a time
    - Limited concurrency, limited parallelism
- Examples
    - “Green threads” (e.g. Solaris)
    - Gnu Portable Threads



Many-to-one model

Multithreading Models(contd…)

One-to-One Model:

- The one-to-one model maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- It also allows multiple threads to run in parallel on multiprocessors.
- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread. Because the overhead of c most implementations of this model restrict the number of threads supported by the system
- Examples:

  -Linux,along with the family of windows operating systems.

Multithreading Models(contd…)

Many-to-Many Model:

- The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine.
- The many-to-many model suffers from neither of these shortcomings of above two model. Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
- Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

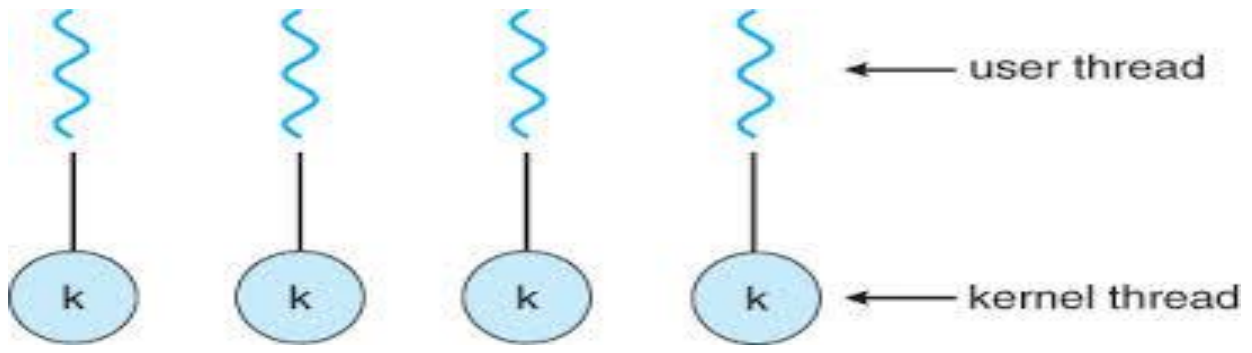


Fig: One-to One model

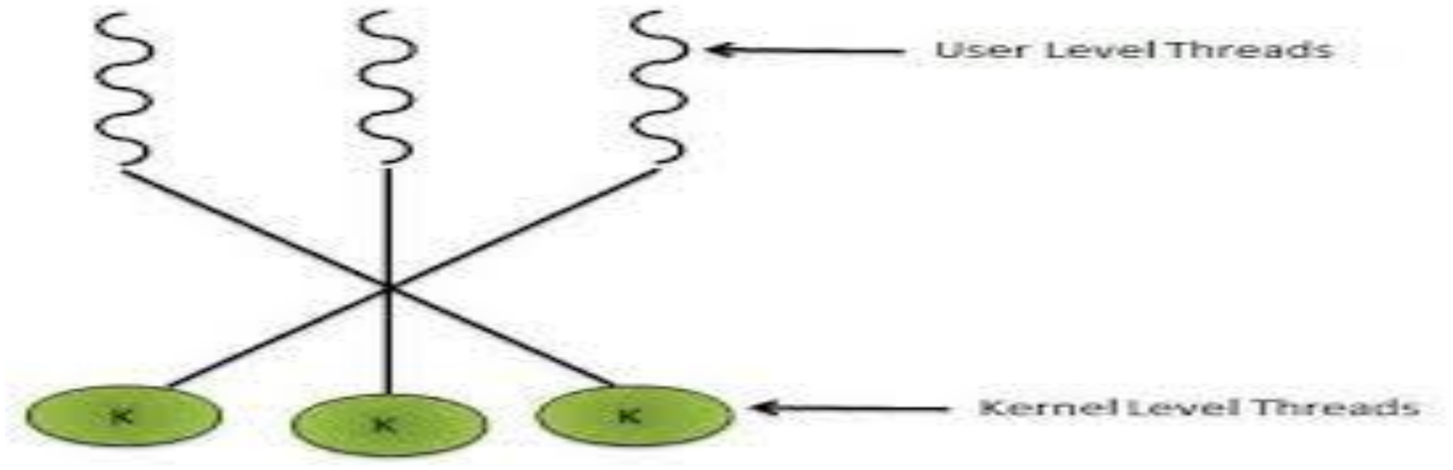


Fig: Many-to Many
model

Differences Between Processes and Threads:

- A process is a program in execution, whereas a thread is a path of execution within a process.
- Processes are generally used to execute large, ‘heavyweight’ jobs such as running different applications, while threads are used to carry out much smaller or ‘lightweight’ jobs such as auto saving a document in a program, downloading files, etc. Whenever we double-click an executable file such as Paint, for instance, the CPU starts a process and the process starts its primary thread.
- Each process runs in a separate address space in the CPU. But threads, on the other hand, may share address space with other threads within the same process. This sharing of space facilitates communication between them. Therefore, Switching between threads is much simpler and faster than switching between processes.
- Threads can directly communicate with other threads of its process; processes must use interprocess communication to communicate with sibling processes.
- Threads also have a great degree of control over other threads of the same process. Processes only have control over child processes.