Operating System

Bachelors in Computer Engineering

Chapter two

Processes and Threads

OUTLINES:

2.1. Process

- Definition of Process, States and Transition diagram
- PCB (Process Control Block)
- Concurrent and Parallel Processing

2.2. IPC (Inter-Process Communication) and Synchronization

- Introduction, Race Condition, Critical Regions and condition, Avoiding critical region: Mutual Exclusion and Serializability,
- Mutual exclusion conditions. mutual exclusion:
  disabling interrupts, lock variable, strict alteration (Dekker's algorithms, Peterson's algorithms), The TSL instruction, sleep and wakeup, producer and consumer problem
- Types of mutual exclusion (Semaphore, Monitors, Bounded buffer, Message passing),
- Classical IPC Problems (The Dining Philosophers problem, The Readers and Writers problem, The Sleeping Barber Problem)
- Serializability: Locking Protocols and Time Stamp Protocols

2.3 Process Scheduling

- Basic Concept, Type of scheduling (Preemptive scheduling, Non-preemptive scheduling, Batch, Interactive, Real time scheduling),
- Scheduling criteria and performance analysis, scheduling algorithm with examples (First come first served, Shortest-job-first, Round- robin, Shortest process next, Shortest remaining time next, Real time, Priority fair share, guaranteed, Lottery scheduling)

2.4 Threads

- Definitions of Threads, Types of thread process (Single and multithreaded process), Benefits of Multithreading
- Threads Models (Many-to-one model, One-to- one Model, Many-to many model)
- User Space and Kernel Space Threads,
- Difference between Processes and Threads

Introduction

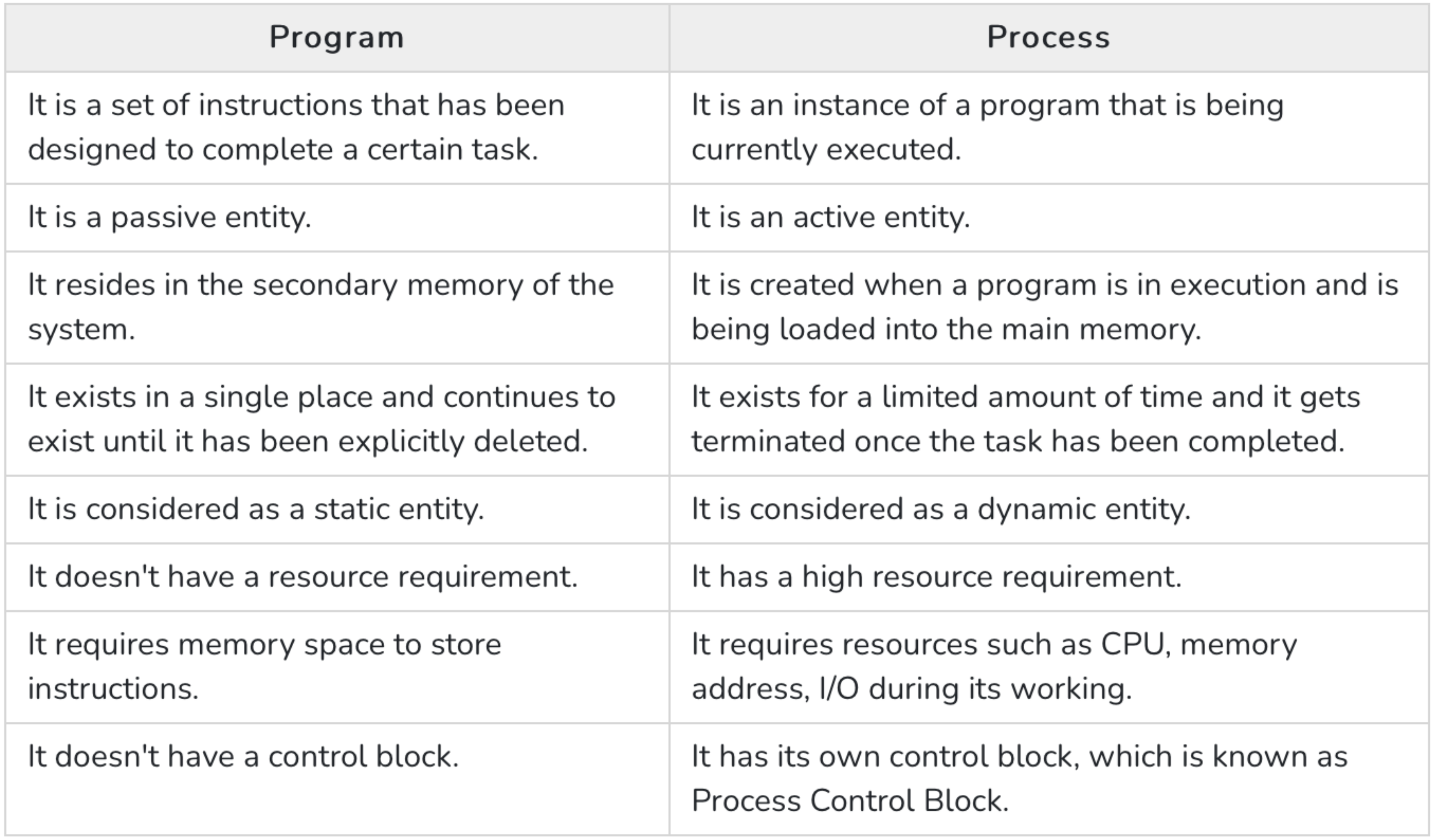
Program:

- A program is a piece of code which may be a single line or millions of lines.
- written by a computer programmer in a programming language.
- A collection of instructions that performs a specific task when executed by a computer.
- When we compare a program with a process, a process is a dynamic instance of a computer program.

Process

- A process is an instance of a program in execution.
- 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 disks. (often called an executable file),
  - whereas a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources.
- A program becomes process when executable file is loaded in the main memory.
- A process is defined as an entity which represents the basic unit of work to be implemented in the system.
- Each process has its own address space and process control block (PCB)

Program vs Process



Program	Process
It is a set of instructions that has been designed to complete a certain task.	It is an instance of a program that is being currently executed.
It is a passive entity.	It is an active entity.
It resides in the secondary memory of the system.	It is created when a program is in execution and is being loaded into the main memory.
It exists in a single place and continues to exist until it has been explicitly deleted.	It exists for a limited amount of time and it gets terminated once the task has been completed.
It is considered as a static entity.	It is considered as a dynamic entity.
It doesn't have a resource requirement.	It has a high resource requirement.
It requires memory space to store instructions.	It requires resources such as CPU, memory address, I/O during its working.
It doesn't have a control block.	It has its own control block, which is known as Process Control Block.

Process states and its transition

- When a process executes, it passes through different states.
- These stages may differ in different operating systems, and the names of these states are also not standardized.
- In general, a process can have one of the following five states at a time:
  - **New**: The process is being created.
  - **Running:**Instructions are being executed.
  - **Waiting:** The process is waiting for some event to occur (such as 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.

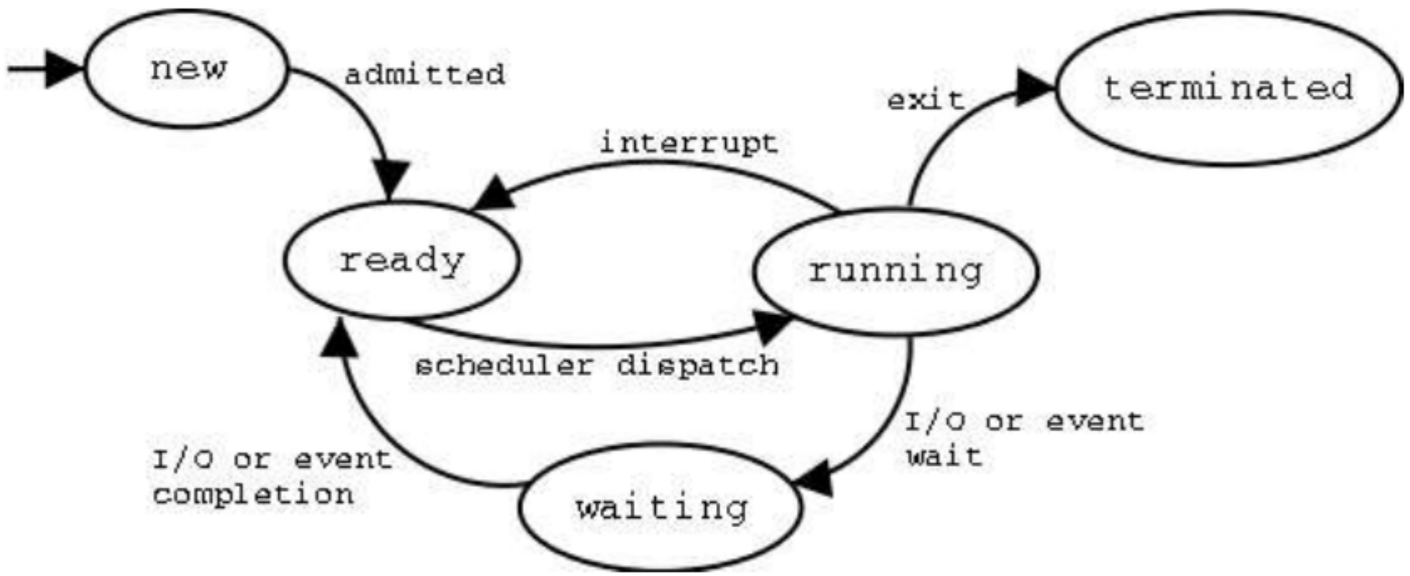
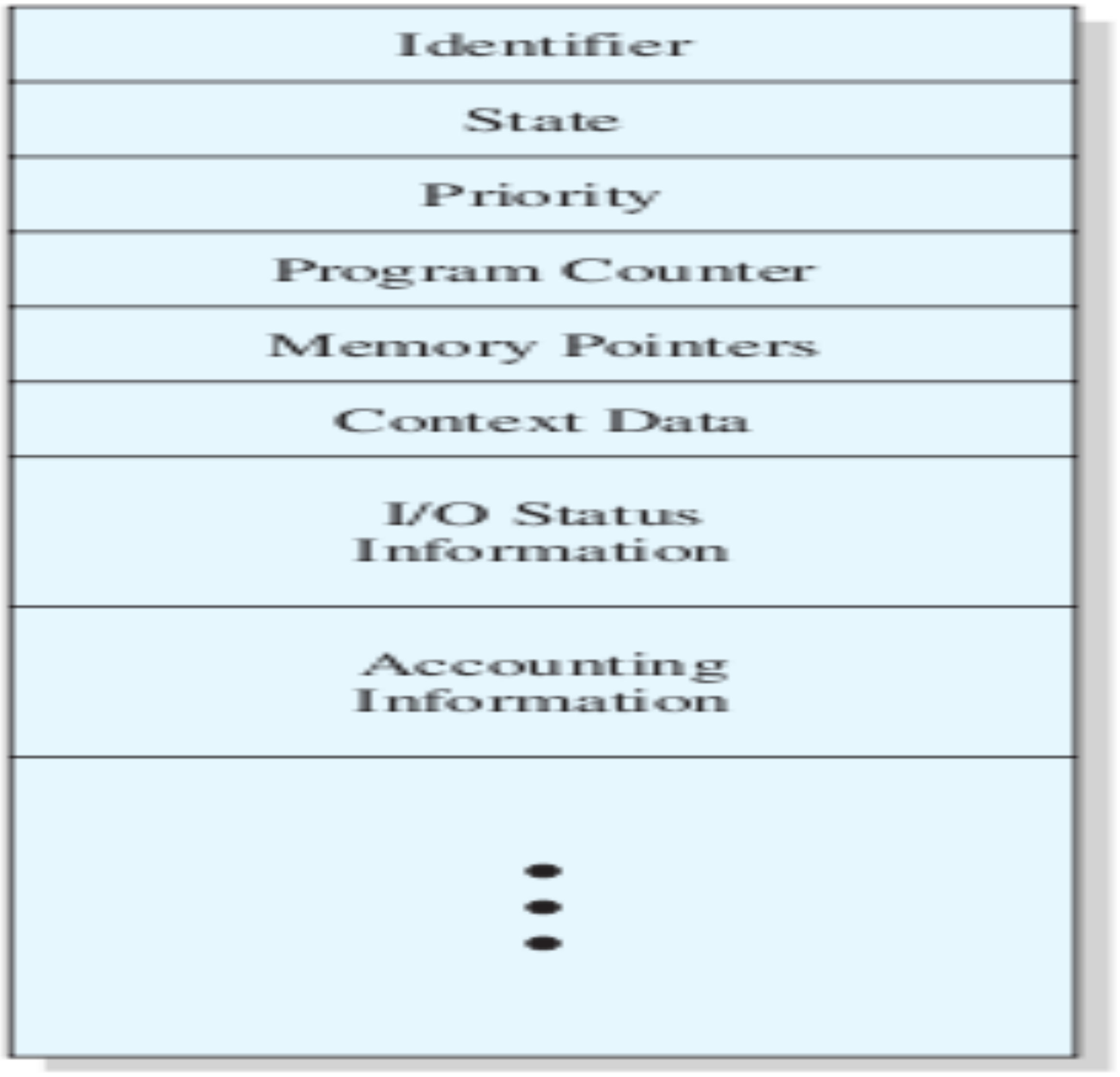


Figure: Process states and its transition

PCB (Process Control Block)



- Each process is represented by a process control block(PCB) or a task control block.
- It is a data structure that physically represent a process in the memory of a computer system.
- It contains many pieces of information associated with a specific process that includes the following.
    - **Identifier**: A unique identifier associated with this process, to distinguish it from all other processes.
    - **State**: If the process is currently executing, it is in the running state.
    - **Priority**: Priority level relative to other processes.
    - **Program counter**: The address of the next instruction in the program to be executed.
    - **Memory pointers:** Includes pointers to the program code and data associated with this process, plus any memory blocks shared with other processes.
    - **Context data**: These are data that are present in registers in the processor while the process is executing
    - **I/O status information:** Includes outstanding I/O requests, I/O devices (e.g., tape drives) assigned to this process, a list of files in use by the process, and so on.
    - **Accounting information:** May include the amount of processor time and clock time used, time limits, account numbers and so on.

Concurrent Process, Parallel Processing

Concurrent processing

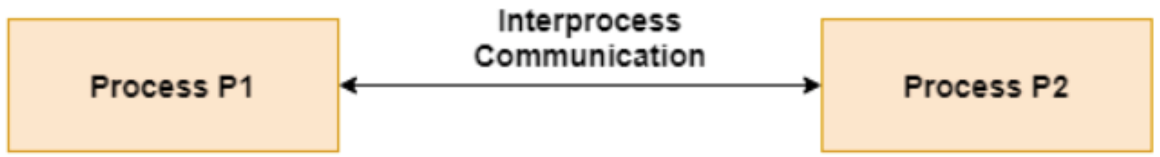
- Concurrent processing is a computing model in which multiple processors execute instructions simultaneously for better performance.
- Concurrent processing involves the execution of multiple tasks or processes in overlapping time intervals.
- Unlike parallel processing, concurrent tasks may not be executing simultaneously but share the CPU in a way that gives the appearance of simultaneous execution.
- Tasks are broken down into subtasks that are then assigned to separate processors to perform simultaneously, instead of sequentially as they would have to be carried out by a single processor.
- Concurrent processing is sometimes said to be synonymous with parallel processing.

Parallel Processing

- Parallel processing, one form of multiprocessing, is a situation in which two or more processors operate in unison.
- Two or more CPUs are executing instructions simultaneously.
- The Processor Manager has to coordinate the activity of each processor as well as synchronize cooperative interaction among the CPUs.
- There are two primary benefits to parallel processing systems:
  - Increased reliability
    - The availability of more than one CPU
    - If one processor fails, then the others can continue to operate and absorb the load.
  - Faster processing
- The increased processing speed is often achieved because sometimes Instructions can be processed in parallel, two or more at a time in one of several ways:
  - Some systems allocate a CPU to each program or job
  - Others allocate a CPU to each working set or parts of it

IPC(Inter-process Communication)

- IPC refers to the mechanisms and techniques used by processes to communicate with each other.
- Processes running concurrently in an operating system may need to exchange data, coordinate their activities, or synchronize their execution.
- IPC provides a way for these processes to communicate and share information.
- IPC mechanisms can include message passing, shared memory, and synchronization primitives like semaphores and mutexes.



Shared Memory:

– Here a region of memory that is shared by co-operating process is established.

– Process can exchange the information by reading and writing data to the shared region.

– Shared memory allows maximum speed and convenience of communication as it can be done at the speed of memory within the computer.

– System calls are required only to establish shared memory regions. Once shared memory is established no assistance from the kernel is required all access are treated as routine memory access.

Message Passing:

– communication takes place by means of messages exchanged between the co-operating process

– Message passing is useful for exchanging the smaller amount of data since no conflict need to be avoided.

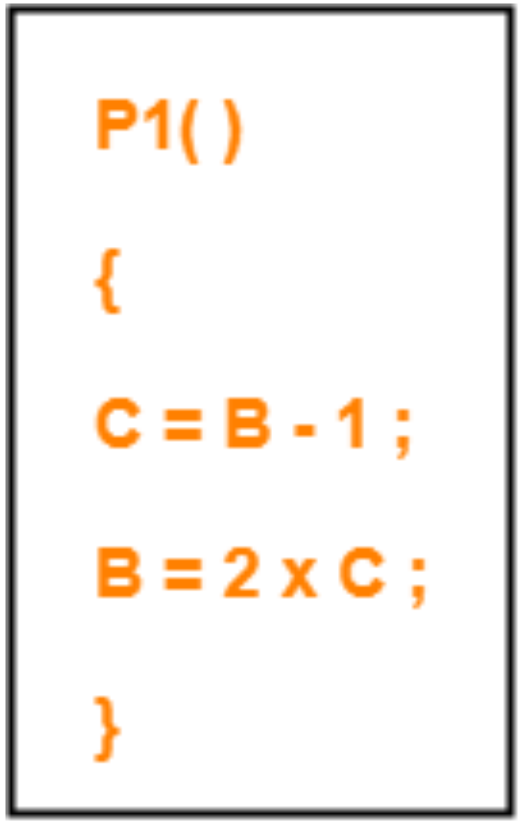
– Easier to implement than shared memory.

– Slower than that of Shared memory as message passing system are typically implemented using system call which requires more time consuming task of Kernel intervention.

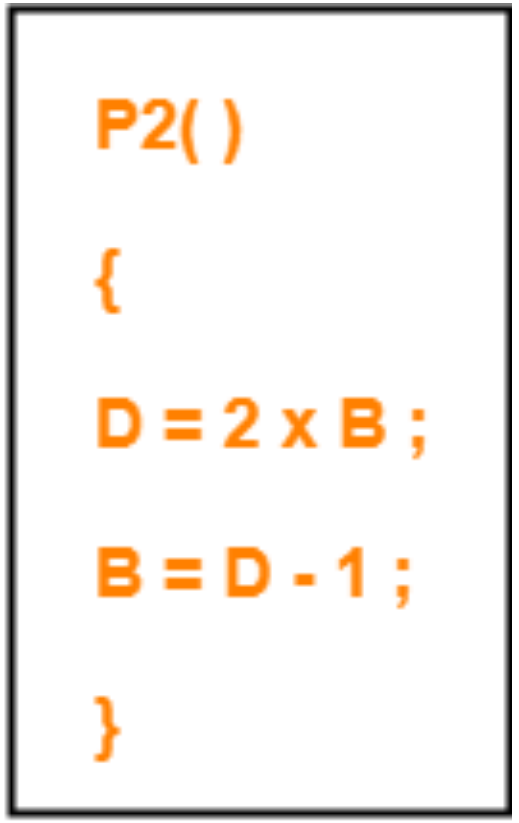
Context Switching

- Context switching is a process performed by the operating system scheduler to switch the CPU's attention from one process to another.
- In a multitasking environment, multiple processes are competing for the CPU's time. The operating system scheduler divides the CPU time among these processes by rapidly switching between them.
- During a context switch, the operating system saves the state of the currently running process (its context) and restores the saved state of the next process to be executed. This allows the CPU to seamlessly transition from one process to another.

Race Condition
A race condition is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly.



```
P1( )
{
C = B - 1 ;
B = 2 x C ;
}
```



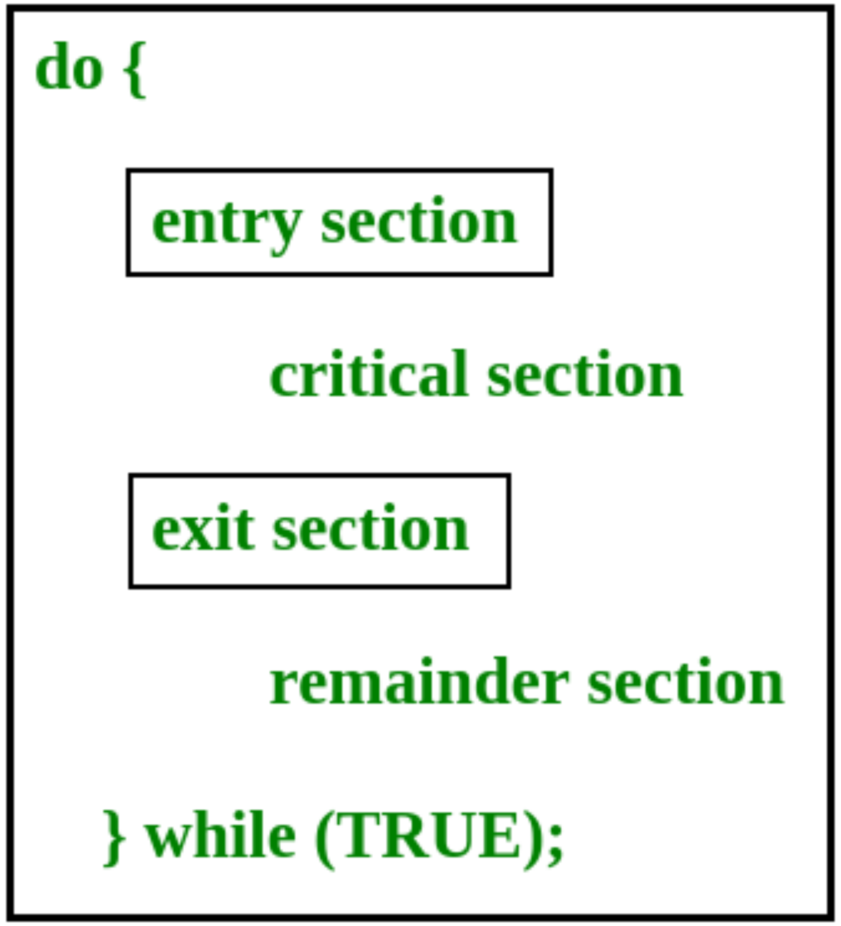
```
P2( )
{
D = 2 x B ;
B = D - 1 ;
}
```

When race conditions occur

- A race condition occurs when two threads access a shared variable at the same time.
- The first thread reads the variable, and the second thread reads the same value from the variable.
- Then the first thread and second thread perform their operations on the value, and they race to see which thread can write the value last to the shared variable.
- The value of the thread that writes its value last is preserved, because the thread is writing over the value that the previous thread wrote.

Critical Section

- A critical region is a section of code in a program that accesses shared resources, such as variables or data structures, memory location, CPU or any IO device which may be modified by multiple concurrent processes or threads.
- It is crucial to ensure that only one process or thread can execute the critical region at a time to avoid race conditions and data inconsistencies.
- When a process is accessing shared data, the process is said to be in critical section.
- When one process is in its critical section all other processes are excluded from entering in its critical section.
- Each process must request permission to enter its critical section.



```
do {

    entry section

        critical section

    exit section

        remainder section

  } while (TRUE);
```

We need four conditions that hold a good solution for avoiding race condition:

i. No two processes may be simultaneously inside their critical section.

ii. No process running outside its critical section may block other processes.

iii. When no process is in the critical section, any process requesting for entry in critical section must be permitted without delay.

iv. A process is granted entry in critical section for finite time only

Mutual Exclusion

- Mutual exclusion is a fundamental concept in concurrent programming, ensuring that only one process or thread accesses a shared resource at any given time.
- Prevents data corruption and maintains consistency when multiple processes manipulate shared data concurrently.
- When number of processes executes concurrently, the critical section of one process should not be executed concurrently with the critical section of another process.
- When a process enters into its critical section, all other processes are restricted until the current running process is completed. Thus, only one process at a time is allowed to access the shared resource.

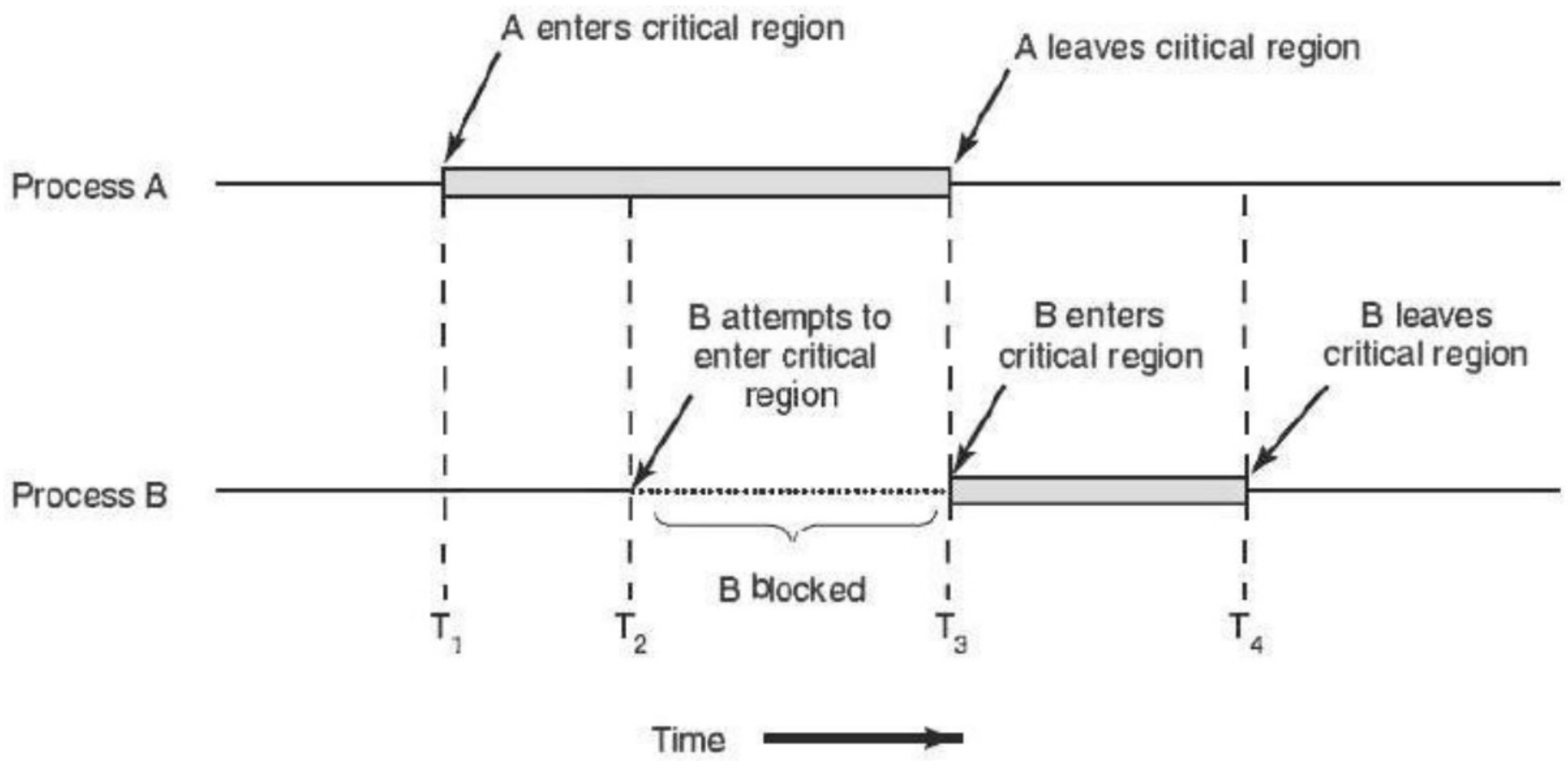


Figure 1. Mutual exclusion using critical regions.

- While one process is busy updating in its critical section, no other process will enter its critical section and cause trouble.
- The various proposals for achieving mutual exclusion are:
  - Disabling interrupts
  - Lock variable
  - Strict alteration
  - The TSL(Test and Set Lock) instruction

Disabling interrupts

- The CPU is only switched from process to process due to clock or other interrupts and with interrupts turned off the CPU will not be switched to another process.
- So, once a process has disabled interrupts, it can examine and update the share memory without fear that any other process will intervene(disturb).
- The simplest solution is to have each process disable all interrupts just after entering its critical section and re-enable them before leaving it.
- This approach is generally unattractive because it gives user process to turn off interrupts.
    - If the system is multiprocessor, disabling interrupts affects only the CPU that executed the disable instruction and the other ones will continue running and can access the shared memory.

Lock variable

- Basic synchronization mechanism to provide mutual exclusion.
- The Lock variable mechanism is a synchronization mechanism that is implemented in a user mode. It is a software procedure.
- Consider having a single shared variable called lock which is initially 0. When a process wants to enter its critical region, it first tests the value of lock variable.
- 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.
- ‘0’ means that no process is in its critical region and ‘1’ means that some process is in its critical region.

Drawbacks:

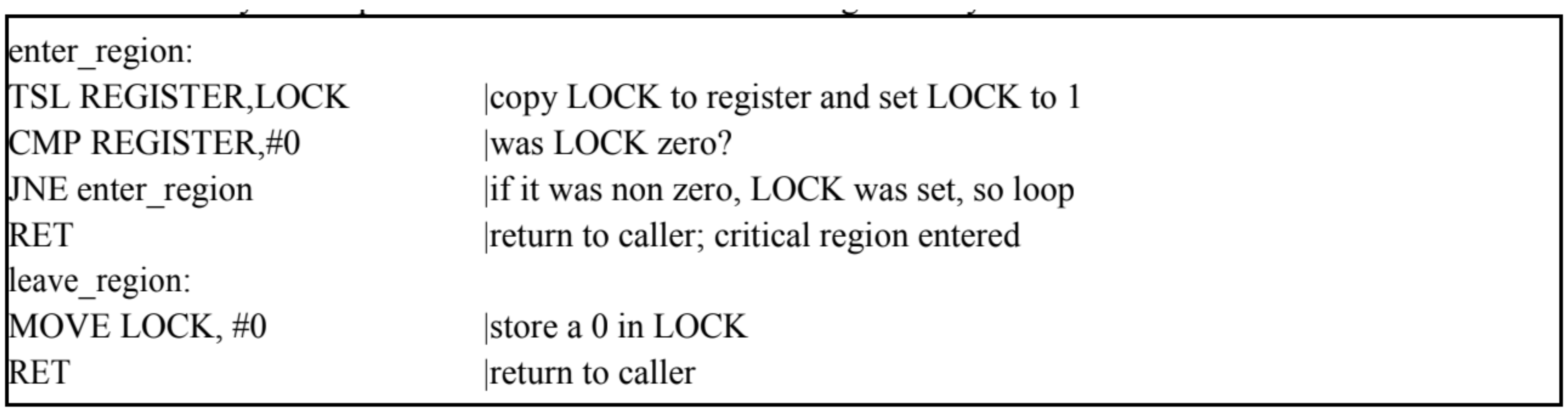
- Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time.

The TSL (Test and Set Lock) instruction

- TSL RX,LOCK

  The TSL 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.



```
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 non zero, 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
```

Fig: entering and leaving critical section using TSL instruction

Strict alteration

```
while (TRUE){
          while (turn!=0);
critical_region ();
turn =1;
noncritical_region ();
}
```

```
while (TRUE){
          while (turn!=1);
critical_region ();
turn =0;
noncritical_region ();
}
```

- Initially the integer variable “turn” is 0. The “turn” variable keeps the track of whose turn it is to enter the critical section and examine or update the shared memory. Initially, process 0 inspects turn, finds it to be 0 and enters its critical section. Process 1 also finds it to be 0 and therefore sits in a tight loop continually testing turn to see when it becomes 1.
- Continually testing a variable until some value appears is called busy waiting and it should be avoided since it wastes CPU time. When process 0 leaves the critical section, it sets “turn” to 1 to allow process 1 to enter its critical section. This way no two processes can enter critical section simultaneously.
- Taking “turn” is not good when one of the processes is much slower than other.

Peterson’s Solution

- Peterson’s solution provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.
- This solution is for 2 processes to enter into critical section. This solution works for only 2 processes.

Peterson's Solution

```
#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 we 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 */
}
```

Fig: Peterson's solution for achieving mutual exclusion

- Initially neither process is in 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 i.e. when process0 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 and 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.

Disadvantage of Peterson’s Solution:

- This solution works for 2 processes, but this solution is best scheme in user mode for critical section.
- This is also a busy waiting solution so CPU time is wasted. And because of that “SPIN LOCK” problem can come. And this problem can come in any of the busy waiting solution.

Dekker’s Algorithm

- Dekker’s Algorithm is a mutual exclusion algorithm for two processes or threads.
- It ensures that only one process is able to enter and execute the given critical section
- Dekker’s algorithm is a user-space algorithm (these are more efficient and don’t require the OS kernel to arbitrate).
- ‘‘A process P can enter the critical section if the other does not want to enter, otherwise it may enter only if it is its turn.’’

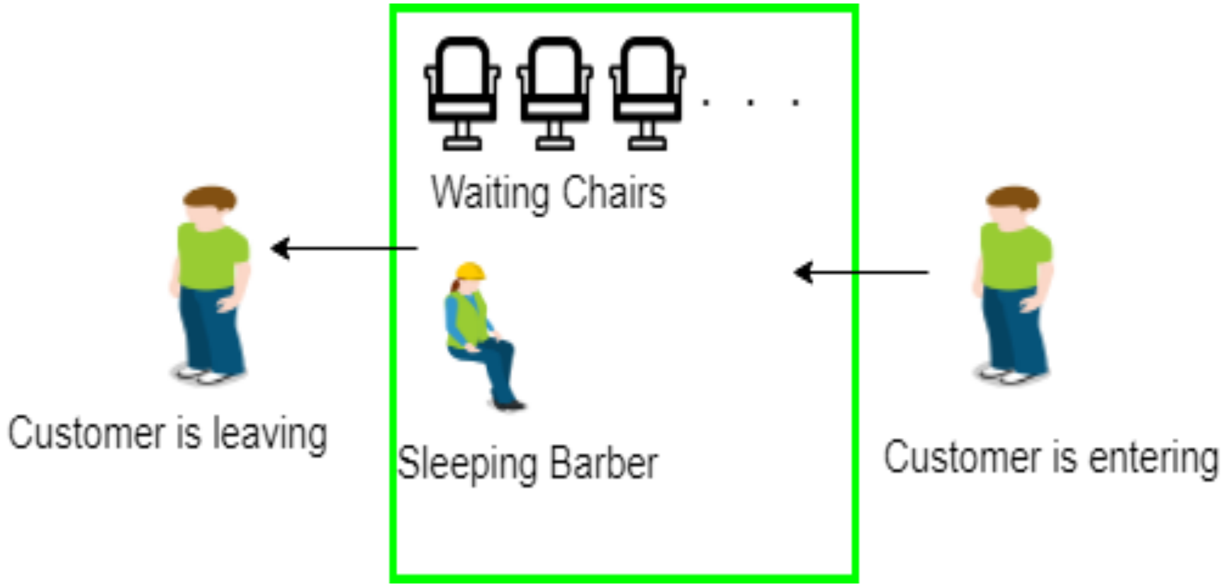
```
boolean flag [2];
int turn;
void P0()
{
     while (true) {
          flag [0] = true;
          while (flag [1]) {
                 if (turn == 1) {
                      flag [0] = false;
                      while (turn == 1) /* do nothing */;
                      flag [0] = true;
                 }
          }
          /* critical section */;
          turn = 1;
          flag [0] = false;
          /* remainder */;
     }
}
void P1( )
{
     while (true) {
          flag [1] = true;
          while (flag [0]) {
                 if (turn == 0) {
                      flag [1] = false;
                      while (turn == 0) /* do nothing */;
                      flag [1] = true;
                 }
          }
          /* critical section */;
          turn = 0;
          flag [1] = false;
          /* remainder */;
     }
}
void main ()
{
     flag [0] = false;
     flag [1] = false;
     turn = 1;
     parbegin (P0, P1);
}
```

Figure A.2 Dekker's Algorithm

Sleep and Wakeup

- The concept of sleep and wake is very simple.
- If the critical section is not empty then the process will go and sleep.
- It will be waked up by the other process which is currently executing inside the critical section so that the process can get inside the critical section.

- If there is no customer, then the barber sleeps in his own chair.
- When a customer arrives, he has to wake up the barber.
- If there are many customers and the barber is cutting a customer's hair, then the remaining customers either wait if there are empty chairs in the waiting room or they leave if no chairs are empty



Example to use sleep and wakeup primitives: Producer and Consumer’s Problem

- The producer-consumer problem is an example of a multi-process synchronization problem.
- Producers generate data items, while consumers consume these items.
- The challenge is to ensure that producers do not produce data items faster than consumers can consume them, and vice versa

Producers:

- Producers are responsible for generating data items that need to be consumed.
- Producers produce data items at some rate and place them into a shared buffer.
- Producers must be synchronized to avoid overpopulating the buffer when it's full.
- If the buffer is full, a producer should be temporarily blocked until there is space in the buffer.

Consumers:

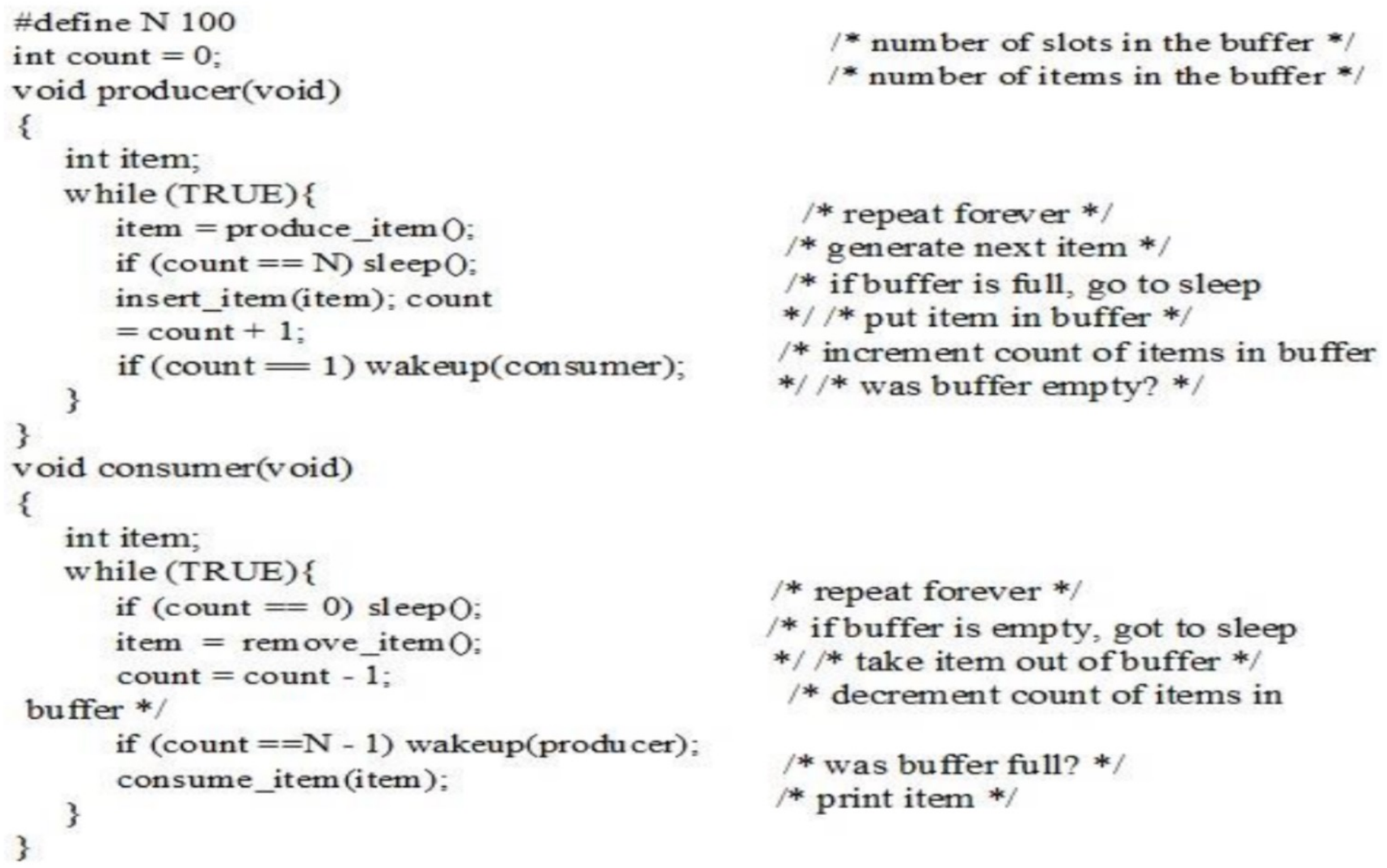
- Consumers are responsible for retrieving and consuming data items from the shared buffer.
- Consumers consume data items from the buffer at their own rate.
- Consumers must be synchronized to avoid attempting to consume from an empty buffer.
- If the buffer is empty, a consumer should be temporarily blocked until there is data available.

Shared Bounded Buffer:

- The shared buffer is a fixed-size data structure that acts as a communication channel between producers and consumers.
- It has a limited capacity (bounded) to prevent overuse of system resources.
- It supports operations like inserting a data item (producer) and removing a data item (consumer).
- Proper synchronization mechanisms are required to avoid race conditions and ensure mutual exclusion.

- **Problem:** Given the common fixed-size buffer, the task is to make sure that the producer can’t add data into the buffer when it is full and the consumer can’t remove data from an empty buffer.
- **Solution:** The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same manner, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.
- **Note:** An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

Let N be the maximum number of items a buffer can store and count is a variable to keep track of the number of items in the buffer. The methods for producer and consumer are as follows:



```
#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) sleep();               /* if buffer is full, go to sleep
        insert_item(item); count               */ /* put item in buffer */
        = 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 */
    }
}
```

Fig: The producer-consumer problem with a fatal race condition.

Producer code:

- The producers code is first test to see if count is N. If it is, the producer will go to sleep; if it is not the producer will add an item and increment count.

Consumer code:

- It is similar as of producer. First test count to see if it is 0. If it is, go to sleep; if it is nonzero remove an item and decrement the counter.
- Each of the process also tests to see if the other should be awakened and if so wakes it up.

Types of mutual exclusion :

- Semaphore,
- Monitors,
- Bounded buffer,
- Message passing

Monitors

- A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package.
- Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor.
- Only one process can be active in a monitor at any instant.
- Monitors are a programming language construct, so the compiler knows they are special and can handle calls to monitor procedures differently from other procedure calls.
- When a process calls a monitor procedure, the first few instructions of the procedure will check to see if any other process is currently active within the monitor.
- If so, the calling process will be suspended until the other process has left the monitor. If no other process is using the monitor, the calling process may enter.

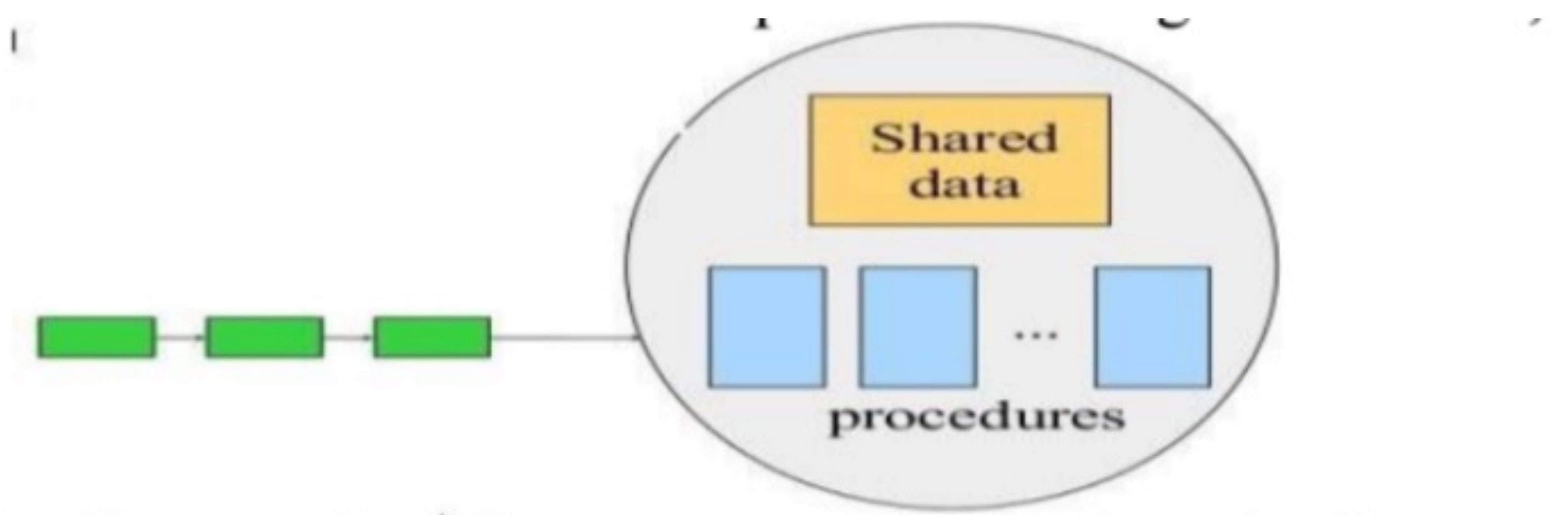


Fig: Queue of waiting processes trying to enter the monitor

Message Passing

- Message passing is a form of communication used in parallel computing, object-oriented programming, and interprocess communication.
- It is a method of communication where messages are sent from a sender to one or more recipients.
- Forms of messages include remote method invocation, signals, and data packets.
- Message passing uses two primitives send and receive. They can easily be put into library procedures as:

```
send(destination, &message);
and
receive(source, &message);
```

- Synchronous message passing systems requires the sender and receiver to wait for each other to transfer the message.
- Asynchronous message passing systems deliver a message from sender to receiver, without waiting for the receiver to be ready.

Semaphore

- Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.
- The definitions of signal and wait are given below:

Wait

It decrements the value of its A argument in case it is positive. In case A is zero or negative, then no operation would be performed.

```
wait(A)
{
while (A<=0);
A–;
}
```

Signal

This operation increments the actual value of its argument A.

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

Counting Semaphores

- These are integer value semaphores and have an unrestricted value domain.
- These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources.
- If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.

Binary Semaphores

The binary semaphores are like counting semaphores but their value is restricted to 0 and 1.

The wait operation on a binary semaphore p, can only proceed when the semaphore's value is 1.If the semaphore is 0 (indicating that a critical section is already locked), the wait operation will block or suspend the thread until the semaphore becomes 1.

The signal operation on a binary semaphore V, succeeds when the semaphore's value is 0. It sets the semaphore to 1.

It is sometimes easier to implement binary semaphores than counting semaphores.

Advantages of Semaphores

- Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
- There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.

- Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

Disadvantages of Semaphores

- Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.
- Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
- Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

Serializability

- When the several transaction execute concurrently in the database, the isolation properties of transaction may no longer be preserved. It is necessary for the system to control the interaction among the concurrent transaction.
- So, serializability ensures that the outcome of the database state and data value remain equal and consistent.
- To maintain the consistency in a database the concurrency control unit of the database system uses two types of protocol. They are:
    - Locking Protocols and
    - Time Stamp Protocols

Lock based protocols

- One way to ensure serializability is to require that access to database item be done in a mutually exclusive manner i.e. while one transaction is accessing a data item, no other transaction can modify that data item. The most common method used to implement this requirement is known as lock.
- There are various modes in which a data item may be locked. Some of them are;

a. Shared lock:

- If a transaction Ti has obtained a shared mode lock denoted by „S‟ on data item Q, then Ti can read but cannot write Q.

b. Exclusive lock:

- If a transaction Ti has obtained an exclusive mode lock denoted by X on data item Q then Ti can both read and write Q.

Time stamp protocols

- Another type of protocol which can maintain the conflict on a serializability of a database is known as time stamp based protocol. In this protocol the ordering of the transaction can be done with the help of advanced execution time i.e. the transaction which has less execution time will have the first priority to execute.

Classical IPC Problems

- The Dining Philosophers problem,
- The Readers and Writers problem,
- The Sleeping Barber Problem

The Dining Philosophers problem,

- Dijkstra posed and solved a synchronization problem called dining philosopher problem. This problem can be stated as follows:
- There are N philosophers sitting around a circular table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. There are only N fork and between each pair of plates is one fork. Now the problem remains that no philosopher should remain hungry or starve till the end, and it should also prevent deadlock.
- One idea is to instruct each philosopher to behave as follows:
  - think until the left fork is available; when it is, pick it up
  - think until the right fork is available; when it is, pick it up
  - eat
  - put the left fork down
  - put the right fork down
  - repeat from the start
- This solution is incorrect because it allows the system to reach deadlock. Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a deadlock.

- We could modify the program so that after taking the left fork, the program checks to see if the right fork is available. If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process. The problem with this approach is that if all the philosophers could start the algorithm simultaneously, they pick up their left forks, see that their right forks were not available, put down their left forks, wait, pick up their left forks again simultaneously, and so on forever. A situation like this, in which all the programs continue to run indefinitely but fail to make any progress is called starvation.
- The possible solution to this problem is as shown in the program below. It uses an array, state, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks). A philosopher may only move into eating state only if neither neighbor is eating. Philosopher i's neighbors are defined by the macros LEFT and RIGHT i.e. if i is 2, LEFT is 1 and RIGHT is 3.
- The program uses an array of semaphores, one per philosopher, so hungry philosophers can block if the needed forks are busy. Note that each process runs the procedure philosopher as its main code, but the other procedures, take_forks, put_forks, and test, are ordinary procedures and not separate processes.

Solution:

```
#define N           5           /* number of philosophers */
#define LEFT        (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher is trying to get forks */
#define EATING      2           /* philosopher is eating */
typedef int semaphore;          /* semaphores are a special kind of int */
int state[N];                   /* array to keep track of everyone's state */
semaphore mutex = 1;            /* mutual exclusion for critical regions */
semaphore s[N];                 /* one semaphore per philosopher */
void philosopher(int i)         /* i: philosopher number, from 0 to N-1 */
{
        while (TRUE){           /* repeat forever */
           think();             /* philosopher is thinking */
           take_forks(i);       /* acquire two forks or block */
           eat();               /* eat spaghetti */
           put_forks(i);        /* put both forks back on table */
        }
    }
```

```
void take_forks(int i)              /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                    /* enter critical region */
    state[i] = HUNGRY;                 /* record fact that philosopher i is hungry */
    test(i);                  /* try to acquire 2 forks */
    up(&mutex);                     /* exit critical region */
    down(&s[i]);                    /* block if forks were not acquired */
}
void put_forks(i)                  /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                    /* enter critical region */
    state[i] = THINKING;               /* philosopher has finished eating */
    test(LEFT);                    /* see if left neighbor can now eat */
    test(RIGHT);                    /* see if right neighbor can now eat */
    up(&mutex);                     /* exit critical region */
}
void test(i)                     /* i: philosopher number, from 0 to N-1* /
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
    state[i] = EATING;
         up(&s[i]);
    }
}
```

The readers and writers problem

- The dining philosopher‟s 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. 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.

Solution to Readers Writer problems is as given below;

```
typedef int semaphore;   /* use our 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 */
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 */
}
}
```

- 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.

Process Scheduling:

- The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.
- In a multiprogramming system, frequently multiple process competes for the CPU at the same time.
- When two or more process are simultaneously in the ready state a choice has to be made which process is to run next.
- This part of the operating system is called scheduler and the algorithm is called scheduling algorithm.
- Process execution consists of cycles of CPU execution and I/O wait. Processes alternate between these two states.
- Process execution begins with a CPU burst that is followed by I/O burst, which is followed by another CPU burst then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution.

Long term and short term scheduling:

- If we consider batch systems, there will often be more processes submitted than the number of processes that can be executed immediately. So, incoming processes are spooled to a disk.
- The **long-term scheduler** selects processes from this process pool and loads selected processes into memory for execution.
- The **short-term scheduler** selects the process to get the processor from the processes which are already in memory.
  - The short-time scheduler executes frequently So it has to be very fast in order to achieve better processor utilization.

Types of Scheduling:

i. **Non-preemptive scheduling algorithm:** It picks a process to run and then just lets it run until it blocks (either on I/O or waiting for another process) or until it voluntarily releases the CPU. Even it runs for hours, it will not be forcibly suspended. In effect no scheduling decisions are made during clock interrupts.

ii. Preemptive scheduling algorithm: It picks a process and lets it run for a maximum of some fixed time. If it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run (if one is available). Doing preemptive scheduling requires having a clock interrupt occur at the end of time interval to give control of the CPU back to the scheduler. If no clock is available, non-preemptive scheduling is only the option.

iii. Batch scheduling: Batch processing is the execution of a series of programs ("jobs") on a computer without manual intervention. An operating environment is termed as “batch processing” because the input data are collected into batches or sets of records and each batch is processed as a unit. T

iv. Interactive scheduling: Interactive processes spend more time waiting for I/O and generally experience short CPU bursts. A text editor is an example of an interactive process with short CPU bursts. It works like short term scheduler. The short-term scheduler selects the process to get the processor from among the processes which are already in memory. They are executed frequently.

v. Real-time scheduling: They focuses on the meeting the time constraints. They prevent simultaneous access to shared resources and devices.

Scheduling criteria and performance analysis(**Process Scheduling Goals)**

1. **Fairness**: Each process gets fair share of the CPU.
2. **Efficiency**: When CPU is 100% busy then efficiency is increased.
3. **Response Time**: Minimize the response time for interactive user.
4. **Throughput**: Maximizes jobs per given time period.
5. **Waiting Time**: Minimizes total time spent waiting in the ready queue.
6. **Turn Around Time**: Minimizes the time between submission and termination.

Scheduling Algorithms:

- First come first served,
- Shortest-job-first,
- Round- robin,
- Shortest process next,
- Shortest remaining time next,
- Real time,
- Priority fair share,
- guaranteed,
- Lottery scheduling

First come First Serve:

- FCFS is the simplest non-preemptive algorithm.
- Processes are assigned the CPU in the order they request it i.e. the process that requests the CPU first is allocated the CPU first.
- The implementation of FCFS policy is managed with a FIFO (First in first out) queue.
- When the first job enters the system from the outside, it is started immediately and allowed to run as long as it wants to.
- As other jobs come in, they are put onto the end of the queue.
- When the running process blocks, the first process on the queue is run next. When a blocked process becomes ready, like a newly arrived job, it is put on the end of the queue.

Advantages:

- Easy to understand and program. With this algorithm a single linked list keeps track of all ready processes.
- Equally fair.
- Suitable especially for Batch Operating system.

Disadvantages:

- FCFS is not suitable for time-sharing systems where it is important that each user should get the CPU for an equal amount of arrival time.

Example: Consider the following set of processes having their burst time mentioned in milliseconds. CPU burst time indicates that for how much time the process needs the CPU.

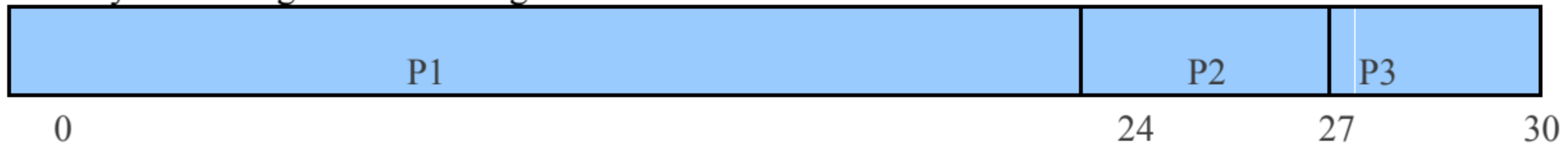
Process	Burst Time
P1	24
P2	3
P3	3

Calculate the average waiting time if the processes arrive in the order of:

a) P1, P2, P3 **b)** P2, P3, P1

Solution:

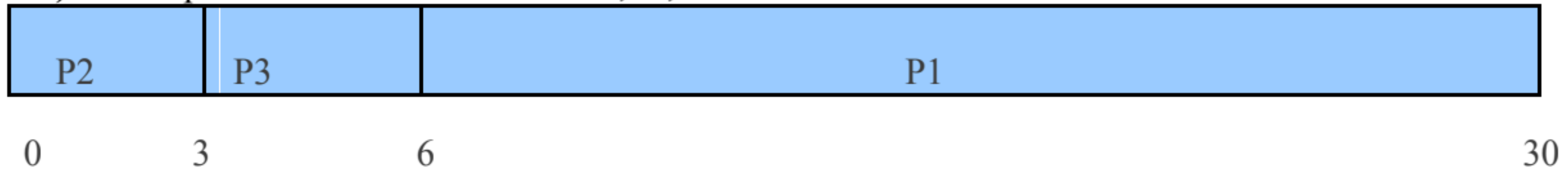
a) The process arrives at the order P1, P2, P3. Let us assume they arrive in the same time at 0 ms in the system. We get the following Gantt chart.



Waiting time for P1= 0ms , for P2 = 24 ms for P3 = 27ms

Average waiting time: (0+24+27)/3= 17

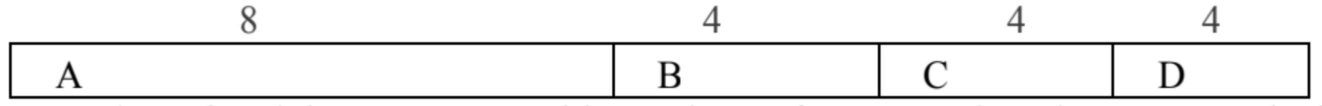
b) If the process arrive in the order P2,P3, P1



Average waiting time: (0+3+6)/3=3. Average waiting time vary substantially if the CPU burst time of the process vary greatly.

Shortest Job First (SJF) or Shortest Job Next (SJN) or Shortest Process Next (SPN):

- It is a non-preemptive algorithm. It is best approach to minimize waiting time.
- When several equally important jobs are sitting in an input queue waiting to be started, the scheduler picks the shortest jobs first.
- In shortest job first (SJF), waiting job (or process) with the smallest estimated run time to completion is run next.
- In other words, when CPU is available, it is assigned to the process that has smallest next CPU burst. SJF is optimal which gives minimum average waiting time for a given set of processes.
- The main disadvantage is that it has the potential for process starvation for the processes which will require a long time to complete if short processes are continually added.
- Another disadvantage of using shortest job next is that the total execution time of a job must be known before execution.



- Here we have four jobs A, B, C, D with run times of 8, 4, 4 and 4 minutes respectively. In the Shortest Job First process B, C, D and then A are executed. The turnaround times are, 4, 8, 12 and 20 minutes giving the average of 11.

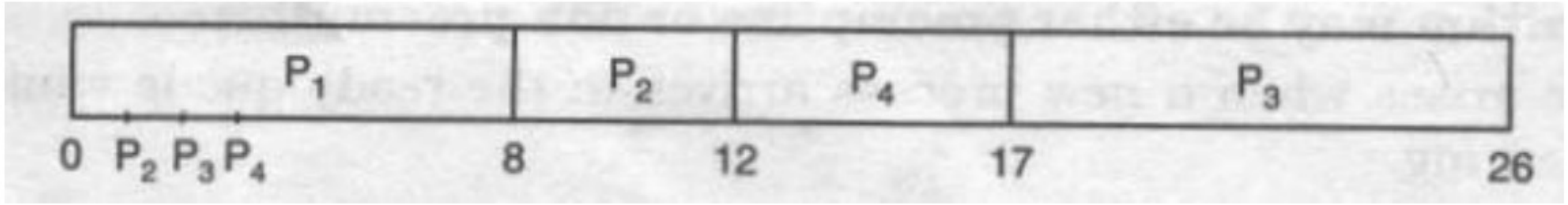
The SJF is either preemptive or non-preemptive.

- **Non pre-emptive SJF:** In this, once CPU is given to the process it cannot be preempted until it completes its CPU burst.
- **Preemptive SJF:** This scheme is known as the Shortest-Remaining-Time-First (SRTF) or Shortest Remaining Time Next (SRTN) scheduling. With this scheduling algorithm the scheduler always chooses the process whose remaining run time is shortest. When a new job arrives its total time is compared to the current process remaining time. If the new job needs less time to finish than the current process, the current process is suspended and the new job is started. This scheme allows new short jobs to get good service.

Example: Calculate the average waiting time in **Preemptive SJF** and **Non Preemptive SJF**

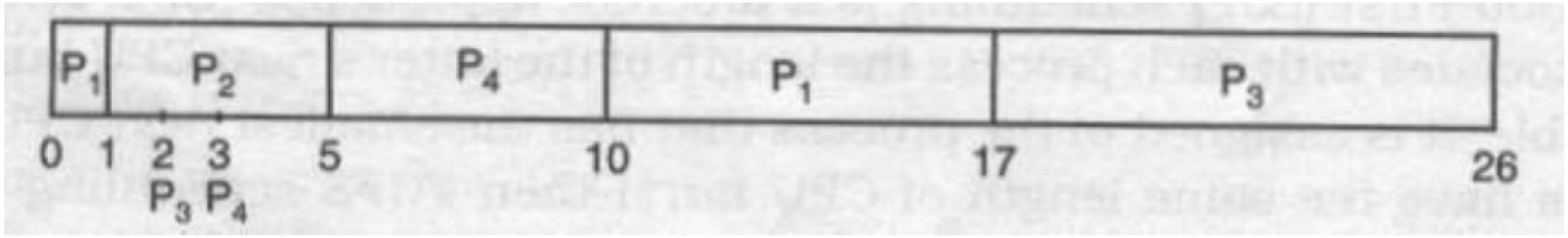
Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Solution:
<u>SJF Default: (Non Preemptive SJF)</u>



Since its non-preemptive, process is not preempted until it finishes its execution.
Waiting time for P1: 0ms
Waiting time for P2: (8-1)ms = 7ms
Waiting time for P3: (17 –2) ms = 15ms
Waiting time for P4: (12 –3)ms = 9ms
Average waiting time: (0+7+15+9)/4 = 7.75ms

Preemptive SJF (Shortest Remaining Time Next or Shortest Remaining Time First):



At t=0ms only one process P1 is in the system, whose burst time is 8ms and it starts its execution. After 1ms i.e., at t=1, new process P2 (Burst time= 4ms) arrives in the ready queue. Since its burst time is less than the remaining burst time of P1 (7ms), P1 is preempted and execution of P2 is started.

Again at t=2, a new process P3 arrive in the ready queue but its burst time (9ms) is larger than remaining burst time of currently running process (P2 3ms). So P2 is not preempted and continues its execution. Again at t=3, new process P4 (burst time 5ms) arrives . Again for same reason P2 is not preempted until its execution is completed.

Waiting time of P1: 0ms + (10 –1) ms = 9ms
Waiting time of P2: 1ms –1ms = 0ms
Waiting time of P3: 17ms –2ms = 15ms
Waiting time of P4: 5ms –3ms = 2ms
Average waiting time: (9+0+15+2)/4 = 6.5ms

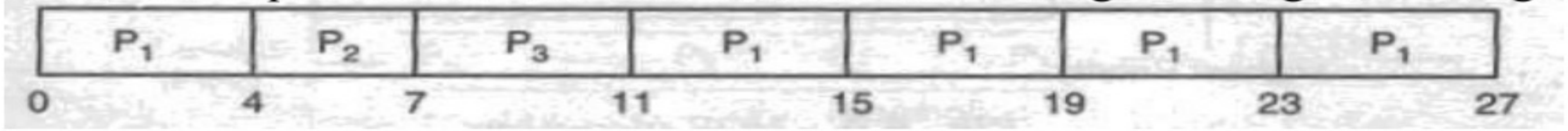
Round-Robin Scheduling Algorithms:

- One of the oldest, simplest, fairest and most widely used algorithms is round robin (RR).
- In the round robin scheduling, processes are dispatched in a FIFO manner but are given a limited amount of CPU time to execute called a time-slice or a quantum.
- If a process does not complete before its CPU-time expires, the CPU is preempted and given to the next process waiting in a queue. The preempted process is then placed at the back of the ready list.
- If the process has blocked or finished before the quantum has elapsed the CPU switching is done.
- Round Robin Scheduling is preemptive (at the end of time-slice) therefore it is effective in time-sharing environments in which the system needs to guarantee reasonable response times for interactive users.
- The only interesting issue with round robin scheme is the length of the quantum. Setting the quantum too short causes too many context switches and lower the CPU efficiency. On the other hand, setting the quantum too long may cause poor response time and approximates FCFS.

Example: Consider the following set of processes that arrives at time 0 ms.

Process	Burst Time
P1	20
P2	3
P3	4

If we use time quantum of 4ms then calculate the average waiting time using R-R scheduling.



- According to R-R scheduling processes are executed in FCFS order. So, firstly P1 (burst time=20ms) is executed but after 4ms it is preempted and new process P2 (Burst time = 3ms) starts its execution whose execution is completed before the time quantum. Then next process P3 (Burst time=4ms) starts its execution and finally remaining part of P1 gets executed with time quantum of 4ms.
- Waiting time of Process P1: 0 ms + (11 –4) ms = 7 ms
- Waiting time of Process P2: 4ms
- Waiting time of Process P3: 7ms
- Average Waiting time: (7+4+7)/3=6ms

Scheduling in Real Time System

- A real-time system is one in which time plays an essential role.
- Typically, one or more physical devices external to the computer generate activity, and the computer must react appropriately to them within a fixed amount of time.
- For example; the computer in a compact disc player gets the bits as they come off the drive and must convert them into music with a very tight time interval.
- If the calculation takes too long the music sounds strange.
- Other example includes;
    - Auto pilot in Aircraft
    - Robot control in automated factory.
    - Patient monitoring in Factory (ICU)
- Real time systems are of two types:

    i. Hard Real Time system: There are absolute deadline that must be met.

    ii. Soft Real Time system: Missing an occasional deadline is undesirable but is tolerable.

- In both cases real time behavior is achieved by dividing the program into a number of processes, each of whose behavior is predictable & known in advance. These processes are short lived and can run to completion. It's the job of schedulers to schedule the process in such a way that all deadlines are met.
- If there are m periodic events and event i occur with period Pi and require Ci seconds of CPU time to handle each event, then the load can only be handled if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

- A real-time system that meets these criteria is said to be schedulable.
- As an example, consider a soft real-time system with three periodic events, with periods of 100, 200, and 500 ms respectively. If these events require 50, 30, and 100 ms of CPU time per event, respectively, the system is schedulable because 0.5 + 0.15 + 0.2 < 1.

Priority Scheduling

- Priority scheduling is one of the most common scheduling algorithms in batch systems in which each process is assigned a priority.
- Process with highest priority is to be executed first and so on. Processes with same priority are executed on first come first served (FCFS) basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.
- Priorities can be assigned statically or dynamically.
- For UNIX system there is a command nice for assigning static priority.
- To prevent high priority process from running indefinitely the scheduler may decrease the priority of the currently running process at each clock interrupt.
- If this causes its priority to drop below that of the next highest process, a process switch occurs.
- Priority scheduling can be either preemptive or non-preemptive
  - A preemptive priority algorithm will preemptive the CPU if the priority of the newly arrival process is higher than the priority of the currently running process.
  - A non-preemptive priority algorithm will simply put the new process at the head of the ready queue.

A problem in Priority Scheduling is **Starvation or indefinite blocking** in which low priority process may never execute. The solution to this problem is **Aging** in which as time progress, the priorities of process increases.

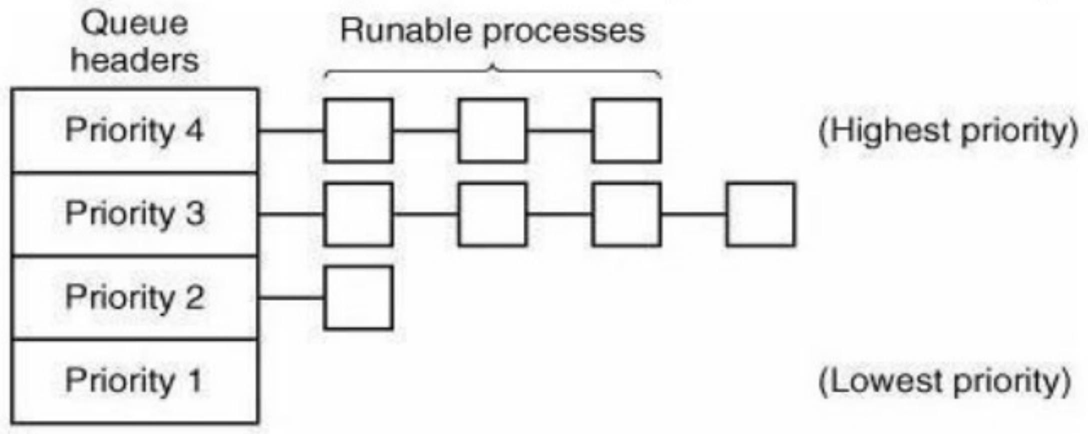


Fig: A scheduling algorithm with four Priority classes

Example: Consider a CPU and also consider a list in which the processes are listed as follows,

Arrival	Process	Burst Time	Priority
0	1	3	2
1	2	2	1
2	3	1	3

| Process-1	| Process-2	| Process-2	| Process-1	|Process-3	
0	1	2	3	5	6

A shortened view of the above time-line is as follows,

| Process-1	| Process-2	| Process-1	| Process-3	|
0	1	3	5	6

So total waiting time = (Waiting Time of Process-1)+ (Waiting Time of Process-2)
+ (Waiting Time of Process-3)
= {(0-0) + (3-1)} + {1-1} +{5-2}s
= (2+0+3) s
= 5s

So the average waiting time is = (Total waiting time / Number of Processes) s
= (5/3) s
= 1.66s

Fair-Share Scheduling:

- Fair-share scheduling is a scheduling algorithm for computer operating systems in which the CPU usage is equally distributed among system users or groups, as opposed to equal distribution among processes.
- Each fair-share scheduling assigns a fixed number of shares to each user or group. These shares represent a fraction of the resources that are available in the cluster. The most important users or groups are the ones with the most shares.
- Users who have no shares cannot run jobs in the queue or host partition.
- In this model, each user is allocated some fraction of the CPU and the scheduler picks processes in such a way as to enforce it.
- Thus if two users have each been promised 50% of the CPU, they will each get that, no matter how many processes they have in existence.
- As an example, consider a system with two users, each of which has been promised 50% of the CPU. User 1 has four processes, A, B, C, and D, and user 2 has only 1 process, E. If round-robin scheduling is used, a possible scheduling sequence that meets all the constraints is this one:

  A E B E C E D E A E B E C E D E ...

- On the other hand, if user 1 is entitled to twice as much CPU time as user 2, we might get

  A B E C D E A B E C D E...

Guaranteed Scheduling:

- A scheduling algorithm used in multitasking operating systems that guarantees fairness by monitoring the amount of CPU time spent by each user and allocating resources accordingly is guaranteed scheduling.
- It makes real promises to the users about performance.
- If there are n users logged in while we are working, we will 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.

Lottery Scheduling:

- Lottery Scheduling is a probabilistic scheduling algorithm for processes in an operating system. Processes are each assigned some number of lottery tickets for various system resources such as CPU time and the scheduler draws a random ticket to select the next process. The distribution of tickets need not be uniform; granting a process more tickets provides it a relative higher chance of selection. This technique can be used to approximate other scheduling algorithms, such as shortest job next and Fair-share scheduling.
- Lottery scheduling solves the problem of starvation. Giving each process at least one lottery ticket guarantees that it has non-zero probability of being selected at each scheduling operation.
- More important process can be given extra tickets to increase their odd of winning. If there are 100 tickets outstanding, & one process holds 20 of them it will have 20% chance of winning each lottery. In the long run it will get 20% of the CPU. A process holding a fraction f of the tickets will get about a fraction of the resource in questions.

Thread

- Thread is a flow of control within a process.
- It is a basic unit of CPU utilization which has a thread program counter, register and a stack.
- It shares its code section, data section and other operating system resources with other threads belonging to the same process.
- A process can consist of several threads, each of which execute separately.

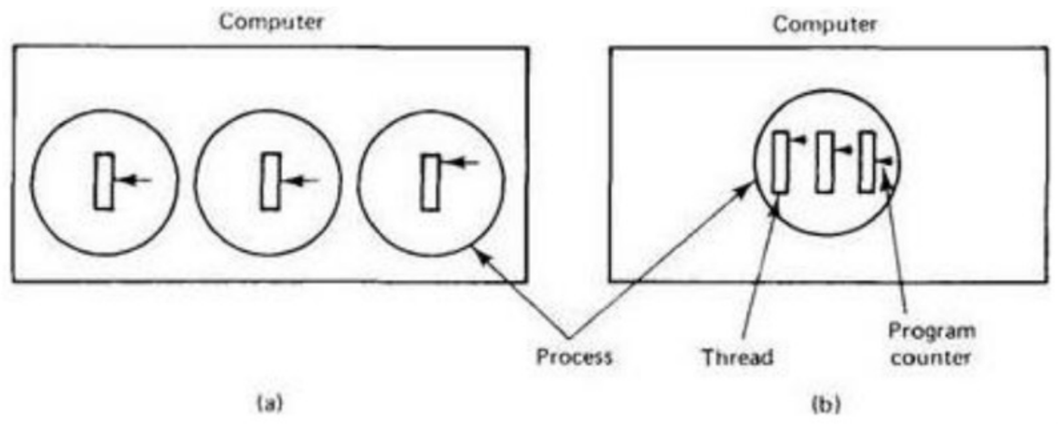


Fig: (a) Three processes with one thread each (b) A process with 3 threads

model of Thread Process

- Generally, there are two types of thread process.

i. Single threaded process

  - In single threaded process, each process has its own address space and a single thread of control.
  - It does not provide parallelism and hence gives up performance.

ii. Multithreaded process

  - In multithreaded process, a single process has multiple threads of control.
  - It provides parallelism and hence improves performance.

User Level Threads

- Thread management is done in user space of main memory by a user-level thread library.
- The user-level or programmer takes the information from thread library which contains the code about creation, destroy, schedule, execution and restoring of threads.
- Kernel don't know about these threads
- User Level Threads are faster to create and manage

Advantages:

1. The most obvious advantage of this technique is that a user-level threads package can be implemented on an Operating System that does not support threads.
2. User-level threads do not require modification to operating systems.
3. Simple Representation: Each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.
4. Simple Management: This simply means that creating a thread, switching between threads and synchronization between threads can all be done without intervention of the kernel.
5. Fast and Efficient: Thread switching is not much more expensive than a procedure call.

Disadvantages:

1. User-Level threads are not a perfect solution as with everything else, they are a tradeoff.

-Since, User-Level threads are invisible to the OS they are not well integrated with the OS. As a result, Os can make poor decisions like scheduling a process with idle threads, blocking a process whose thread initiated an I/O even though the process has other threads that can run and unscheduling a process with a thread holding a lock.

-Solving this requires communication between between kernel and user-level thread manager.

2. There is a lack of coordination between threads and operating system kernel.

-Therefore, process as whole gets one time slice irrespective of whether process has one thread or 1000 threads within. It is up to each thread to relinquish control to other threads.

3. User-level threads require non-blocking systems call i.e., a multithreaded kernel. Otherwise, entire process will block in the kernel, even if there are runnable threads left in the processes.

-For example, if one thread causes a page fault, the process blocks.

Kernel Level Threads

- Thread that are defined and managed by directly kernel of Operating system is kernel level threads
- Kernel perform thread creation, scheduling and management
- Kernel threads are used in internal working of operating system
- Kernel threads are slower to create and manage

Advantages:

1. Because kernel has full knowledge of all threads, Scheduler may decide to give more time to a process having large number of threads than process having small number of threads.
2. Kernel-level threads are especially good for applications that frequently block.

Disadvantages:

1. The kernel-level threads are slow and inefficient. For instance, threads operations are hundreds of times slower than that of user-level threads.
2. Since kernel must manage and schedule threads as well as processes. It requires a full thread control block (TCB) for each thread to maintain information about threads. As a result, there is significant overhead and increased in kernel complexity.

Thread usage

- Threads provide a convenient way of allowing an application to maximize its usage of CPU resources in a system, especially in a multiple processor configuration.
- In a Remote Desktop Services environment, however, multiple users can be running multithreaded applications, and all of the threads for all of the users compete for the central CPU resources of that system.
  - With this in mind, you should tune and balance application thread usage for a multiuser, multiprocessor Remote Desktop Services environment.
- While a poorly designed multithreaded application with idle or wasted threads might perform adequately on a client computer, the same application might not perform well on a multiuser Remote Desktop Services server.

Benefits of Multithread:

- Multithread programming provides following benefits:

i. Responsiveness

- Multithreading is an interactive application that allow a program to continue running even if part of it is blocked. Hence, multithreading increases responsiveness to the user.

ii. Resource sharing

- Threads share the memory and other resources of the process to which they belong. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

iii. Economy

- Allocating memory and resources for process creation is costly. Threads share resources of the process to which they belongs, so it is more economical to create threads. Thread are easier to create and destroy than process.

iv. Utilization of multiprocessor architecture

- In multithreading architecture, threads may be running in parallel on different processor. Multithreading on multiprocessor can increase concurrency.

Multithreading Model

- Threads may be provided either at the user level for user thread or at kernel level for kernel threads.
- User threads are managed without kernel support whereas kernel thread are supported and managed directly by the operating system.
- There exist a relationship between user threads and kernel threads.
- There are three common ways of establishing this relationship.
  - Many to one model
  - One to one model
  - Many to many model

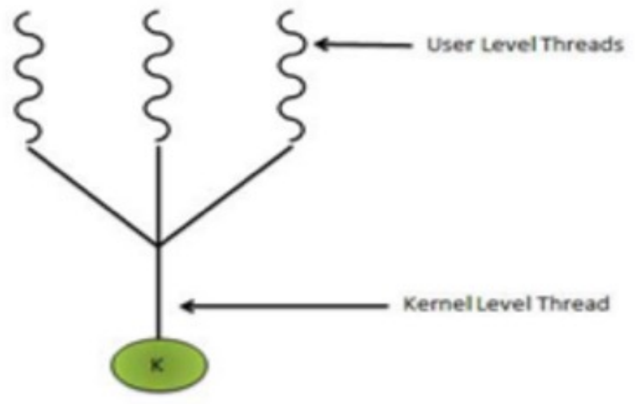


Fig: Many-to-one model

Many to one model:

- **many user-level threads** (the threads that a program uses) **are mapped to a single kernel-level thread** (managed by the operating system).
- Thread management is done by the thread library in user space.
  - So it is efficient but the entire process will block if a thread makes a blocking system call.
- Only one thread can access the kernel at a time. So, multiple threads are unable to run in parallel on multiprocessors.
- **Advantage:** It's simple and efficient in terms of overhead because there's only one kernel thread to manage

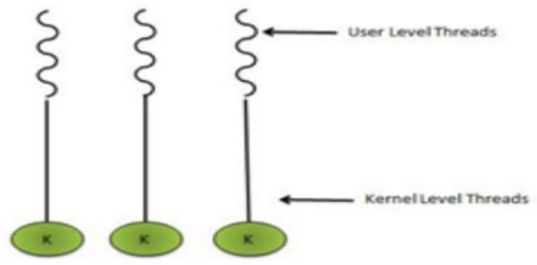


Fig: One-to-one model

One to one model:

- Each user-level thread corresponds to a separate kernel-level thread. So, if a program has three threads, the operating system manages three separate threads for them.
- It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call i.e. it also allows multiple threads to run in parallel on multiprocessor.
- **Advantage:** It allows for parallel execution and better utilization of multiple processor cores.
- The drawback is that creating a user thread requires creating the corresponding kernel thread and the overhead of creating the corresponding kernel thread can burden the performance of an application.

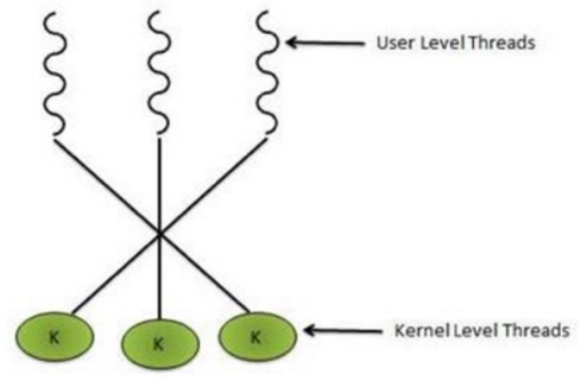
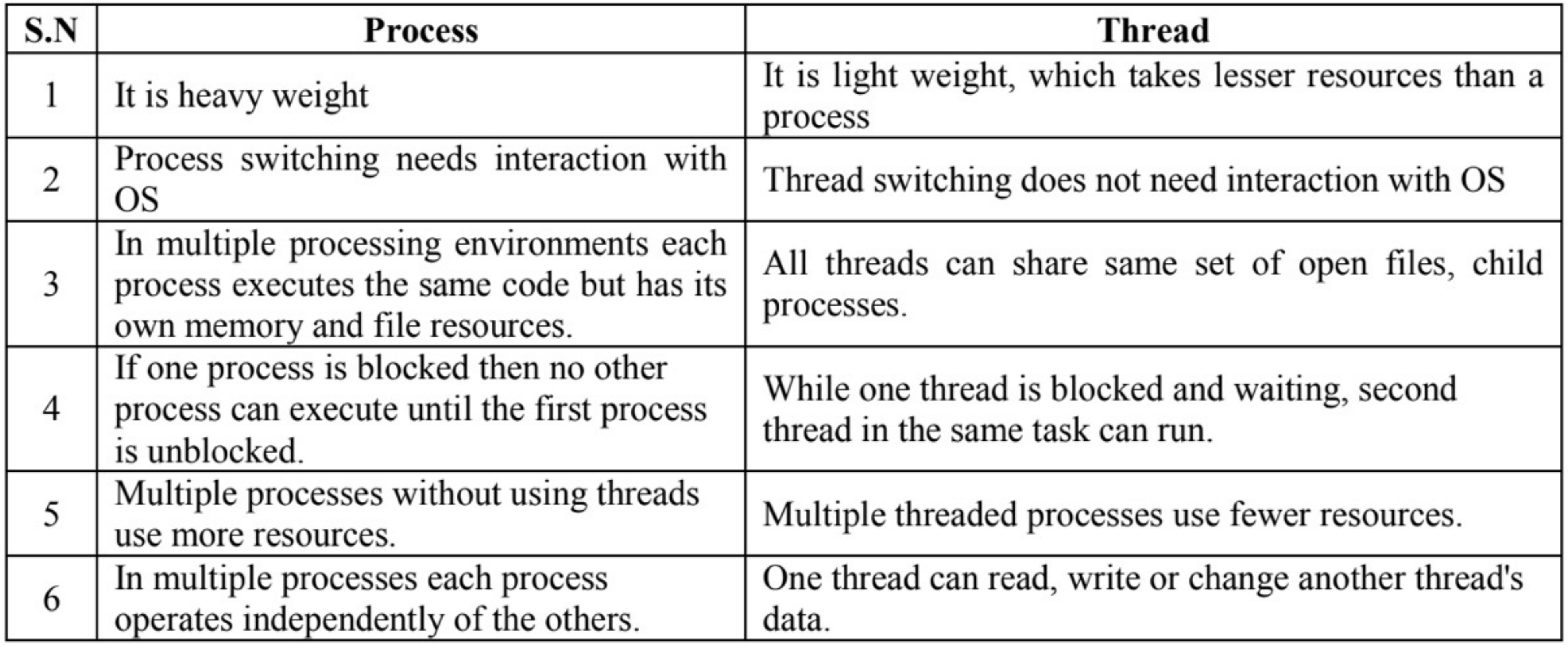


Fig: Many-to-many model

Many- to - many model:

- It multiplexes many user-level threads to a smaller or equal number of kernel threads.
- The mapping can be flexible, meaning not every user thread has a dedicated kernel thread.
- The number of kernel threads may be specific to either a particular application or a particular machine.
- In this model, developer can create as many user threads as necessary and the corresponding kernel threads can run in parallel on multiprocessor.



S.N	Process	Thread
1	It is heavy weight	It is light weight, which takes lesser resources than a process
2	Process switching needs interaction with OS	Thread switching does not need interaction with OS
3	In multiple processing environments each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Thank You