Process Management

Process vs Program

- An operating system executes a variety of programs that run as a process.
- **Process** is a program in execution.
- Process is considered as an instance of a program, replete with registers, variables, and a program counter
- A process is an activity of some kind. It has a program, input, output and a state.
- A **program** is a set of instructions, written to complete some task
- Program is **passive** entity stored on disk (**executable file**); process is **active**
  - Program becomes process when an executable file is loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc are examples of processes
- If a program is running twice, it counts as two processes.

The difference between a process and a program is subtle, but absolutely crucial. An analogy may help you here. Consider a culinary-minded computer scientist who is **baking a birthday cake** for his young daughter. He has a birthday cake recipe and a kitchen well stocked with all the input: flour, eggs, sugar, extract of vanilla, and so on. In this analogy, the **recipe is the program**, that is, an algorithm expressed in some suitable notation, the **computer scientist is the processor** (CPU), and the **cake ingredients are the input data**. The **process is the activity consisting of our baker reading the recipe, fetching the ingredients, and baking the cake**. Now imagine that the computer scientist’s son comes running in screaming his head off, saying that he has been stung by a bee. The computer scientist **records where he was in the recipe (the state of the current process is saved),** gets out a first aid book, and begins following the directions in it. Here we see **the processor being switched from one process (baking) to a higher-priority process (administering medical care),** each having a **different program (recipe versus first aid book).** When the bee sting has been taken care of, the computer scientist **goes back to his cake**, continuing at the point where he left off.

NOTE: A single processor may be shared among several processes, with some scheduling algorithm being accustomed to determine when to stop work on one process and service a different one. In contrast, a program is something that may be stored on disk, not doing anything.

A program and a process are related terms but are not same. The major difference between program and process is that program is a group of instructions to carry out a specified task whereas the process is a program in execution. While a process is an active entity, a program is *considered to be a passive one.*

Multiple parts in a process

- The program code, also called **text section**
- Current activity including **program counter**, processor registers
- **Stack** containing temporary data
  - Function parameters, return addresses, local variables
- **Data section** containing global variables
- **Heap** containing memory dynamically allocated during run time

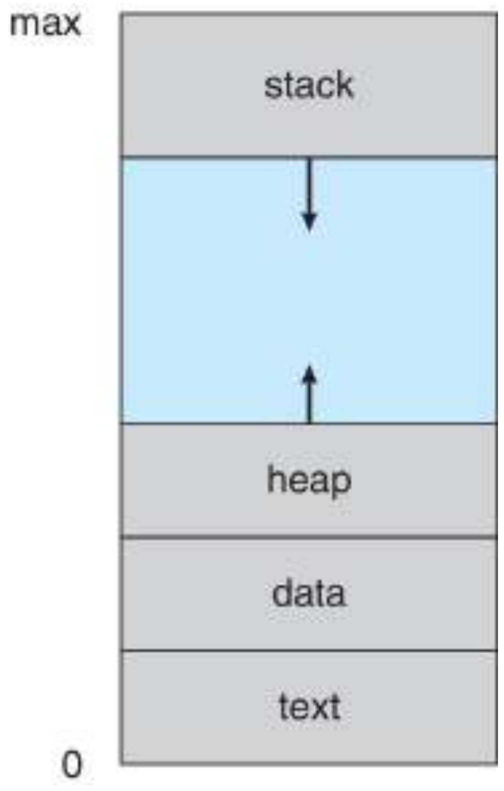
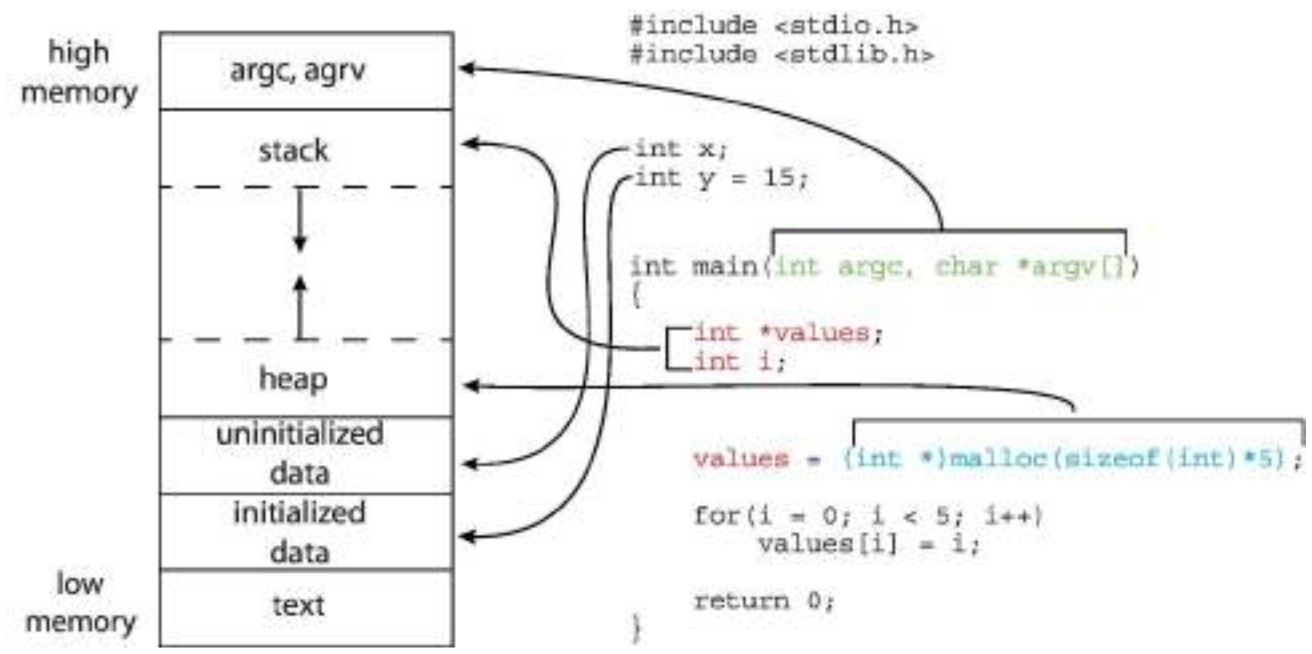


Figure: A process in memory

<u>Memory Layout of a C Program</u>



```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

Figure: Memory Layout od a C Program

Multiprogramming

- In Multiprogramming, to execute the processes, only one CPU is used.
- In a multiprogramming system there are one or more programs loaded in main memory which are ready to execute. Only one program at a time is able to get the CPU for executing its instructions (i.e., there is at most one process running on the system) while all the others are waiting their turn.
- Multiprogramming is based on context switching which doesn't allow CPU to sit idle thereby maximizing CPU utilization.

The Process Model

- In this model, all the runnable software on the computer, sometimes including the operating system, is organized into a number of **sequential processes**, or just processes for short. A process is just an instance of an executing program, including the current values of the program counter, registers, and variables.
- Conceptually, each process has its own virtual CPU. In reality, of course, the real CPU switches back and forth from process to process, but to understand the system, it is much

easier to think about a collection of processes running in (pseudo) parallel than to try to keep track of how the CPU switches from program to program. This rapid switching back and forth is called multiprogramming.

- In any **multiprogramming system**, the CPU switches from process to process quickly, running each for tens or hundreds of milliseconds. While, strictly speaking, at any one instant the CPU is running only one process, in the course of 1 second it may work on several of them, giving the illusion of parallelism. Sometimes people speak of *pseudoparallelism* in this context, to contrast it with the true hardware parallelism of *multiprocessor* systems (which have two or more CPUs sharing the same physical memory).

<u>Illustration of the multiprogramming concept</u>

- In following figure (a) we see a computer multiprogramming four programs in memory. In figure (b) we see four processes, each with its own flow of control (i.e., its own logical program counter), and each one running independently of the other ones. Of course, there is only one physical program counter, so when each process runs, its logical program counter is loaded into the real program counter. When it is finished (for the time being), the physical program counter is saved in the process' stored logical program counter in memory. In figure(c) we see that, viewed over a long enough time interval, all the processes have made progress, but at any given instant only one process is actually running.

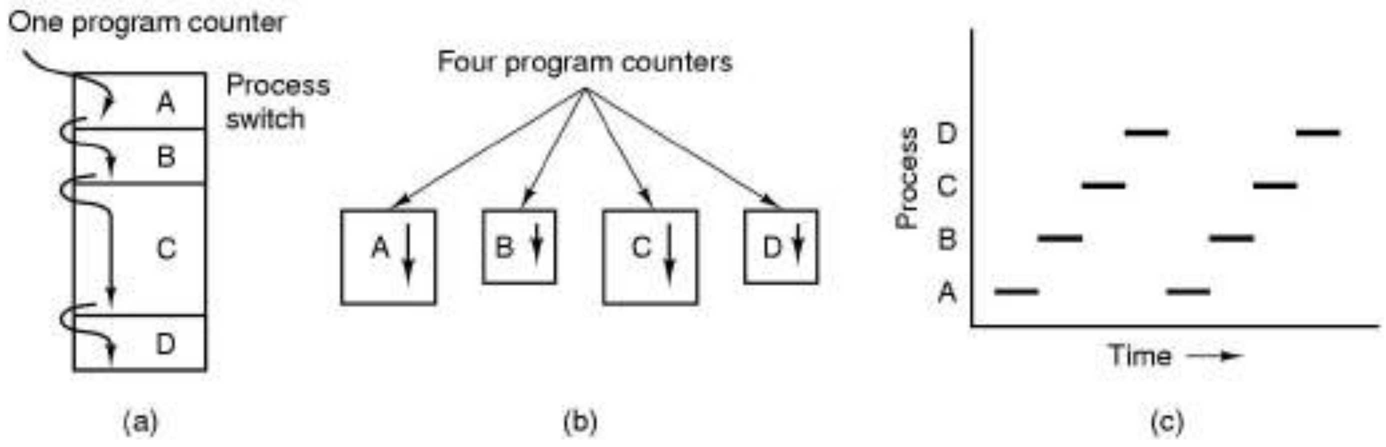


Figure : (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

Process Creation

Operating systems need some way to create processes. Events which can cause process creation

- System initialization.
- Execution of a process creation system call by a running process.
- A user request to create a new process.
- Initiation of a batch job.

Process Termination

After a process has been created, it starts running and does whatever its job is. However, nothing lasts forever, not even processes.

Events which cause process termination:

- Normal exit (voluntary).
- Error exit (voluntary).
- Fatal error (involuntary).
- Killed by another process (involuntary).

Process States

As a process executes, it changes **state**. The state of a process is defined in part by the current activity of that process. A process may be in one of the following states:

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

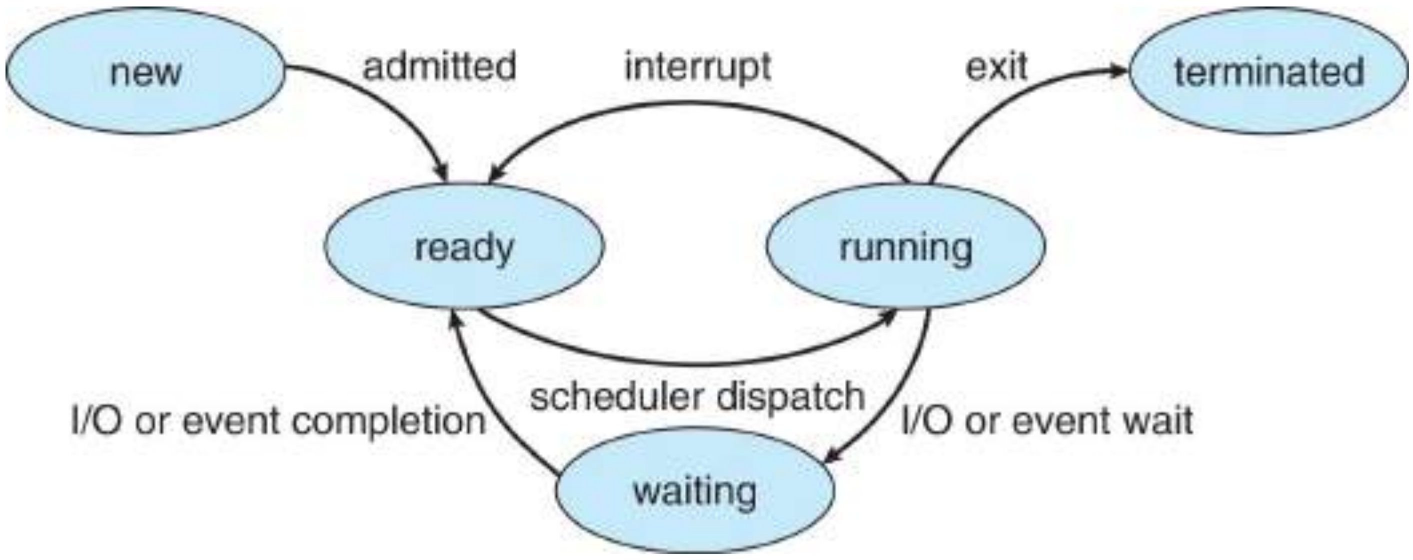


Figure: Process states

Process Control Block

- To implement the process model, the operating system maintains a table (an array of structures), called the process table, with one entry per process. This is also called as process control block (PCB) or task control block.
- Each process is represented in the operating system by a process control block (PCB.
- A PCB is shown in following figure. It contains many pieces of information associated with a specific process, including these:
    - Process state – running, waiting, etc.
    - Program counter – location of instruction to next execute
    - CPU registers – contents of all process-centric registers
    - CPU scheduling information- priorities, scheduling queue pointers
    - Memory-management information – memory allocated to the process
    - Accounting information – CPU used, clock time elapsed since start, time limits
    - I/O status information – I/O devices allocated to process, list of open files

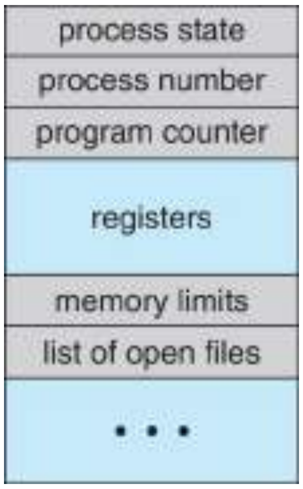
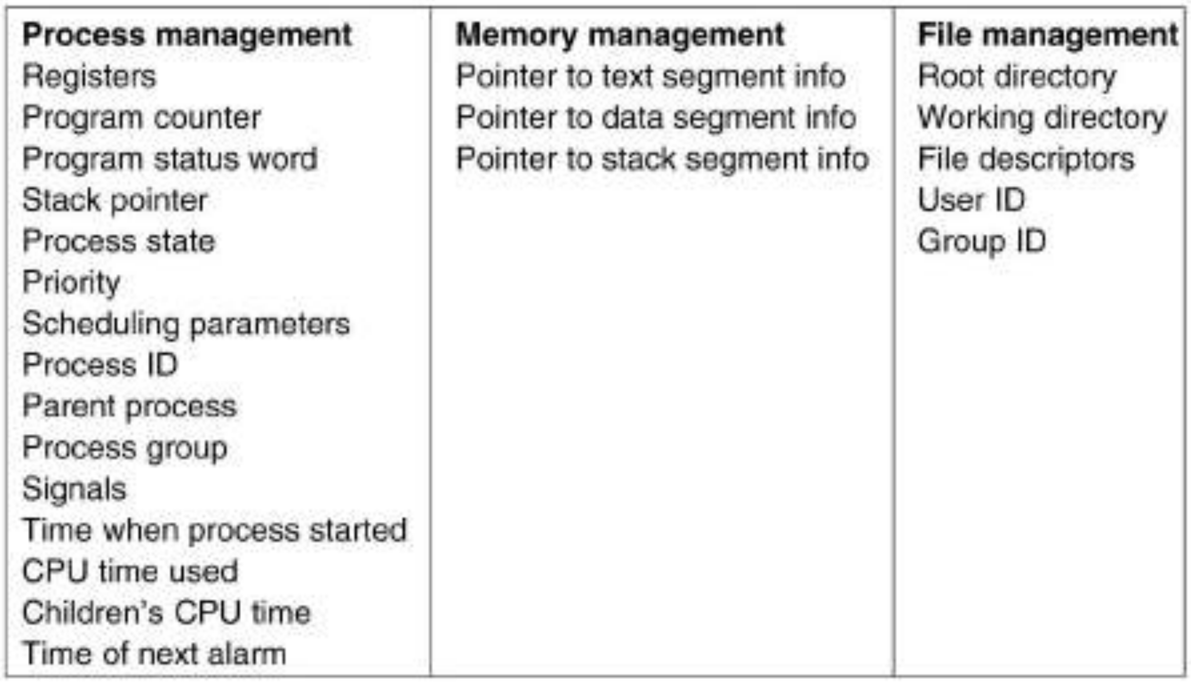


Figure: A process control block (PCB).

Following figure shows some of the key fields in a typical system. The fields in the first column relate to process management. The other two relate to memory management and file management, respectively. It should be noted that precisely which fields the process table has is highly system dependent, but this figure gives a general idea of the kinds of information needed.



Process management	**Memory management**	**File management**
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Figure: Some of the fields of a typical process table / PCB entry.

Threads

- A **thread** is a light-weight smallest part of a process that can run concurrently with the other parts (other threads) of the same process. It has a separate path of execution
- A process can have multiple threads.
- Threads are independent. If a thread gets an exception or an error at the time of its execution, it doesn't affect the execution of the other threads.
- All the threads share a common memory and have their own stack, local variables and program counter.
- Because threads have some of the properties of processes, they are sometimes called lightweight processes.
- When multiple threads are executed in parallel at the same time, this process is known as **multithreading**.
- Each thread belongs to exactly one process and no thre.ad can exist outside a process. Each thread represents a separate flow of control.
- A **thread has** :
  - A **program counter** that keeps track of which instruction to execute next.
  - **Registers**, which hold its current working variables.
  - A **stack**, which contains the execution history, with one frame for each procedure called but not yet returned from

Thread vs Process

- Although a thread must execute in some process, the thread and its process are different concepts and can be treated separately.
- Processes are used to group resources together; threads are the entities scheduled for execution on the CPU.
- What threads add to the process model is to allow multiple executions to take place in the same process environment, to a large degree independent of one another
- Having *multiple threads running in parallel in one process is analogous to having multiple processes running in parallel in one computer*.
  - In the former case, the *threads share an address space and other resources*.
  - In the latter case, *processes share physical memory, disks, printers, and other resources.*

- Because threads have some of the properties of processes, they are sometimes called **lightweight processes**.
- The term **multithreading** is also used to describe the situation of allowing multiple threads in the same process
- In following figure (a) we see three traditional processes. Each process has its own address space and a single thread of control. In contrast, in (b) we see a single process with three threads of control. Although in both cases we have three threads, in (a) each of them operates in a different address space, whereas in (b) all three of them share the same address space

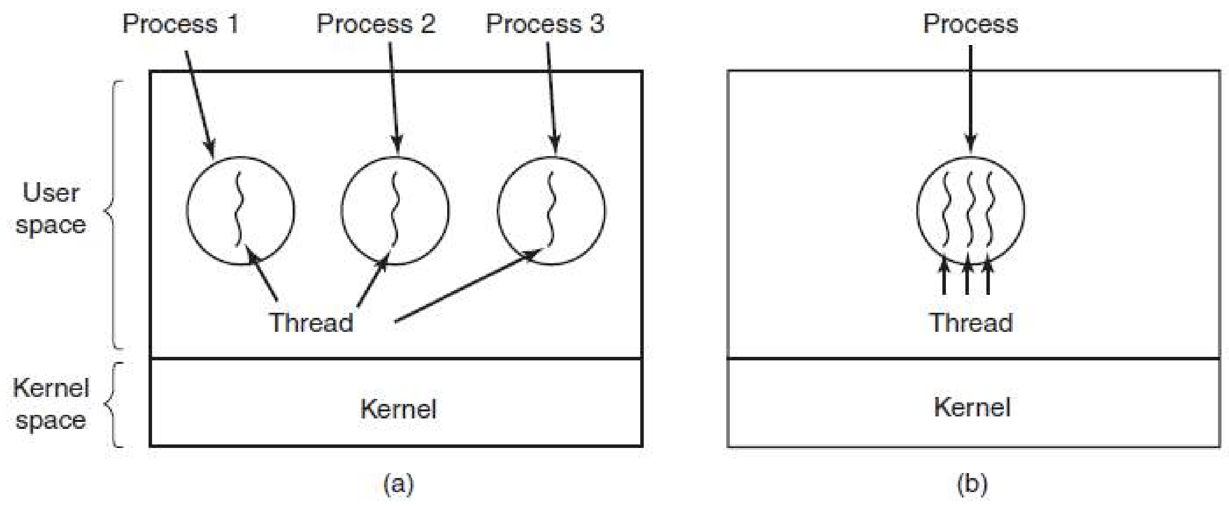


Figure: (a) Three processes each with one thread. (b) One process with

- Different threads in a process are not as independent as different processes.
- Like a traditional process (i.e., a process with only one thread), a thread can be in any one of several states: running, blocked, ready, or terminate.

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
- Process creation is heavy-weight while thread creation is light-weight

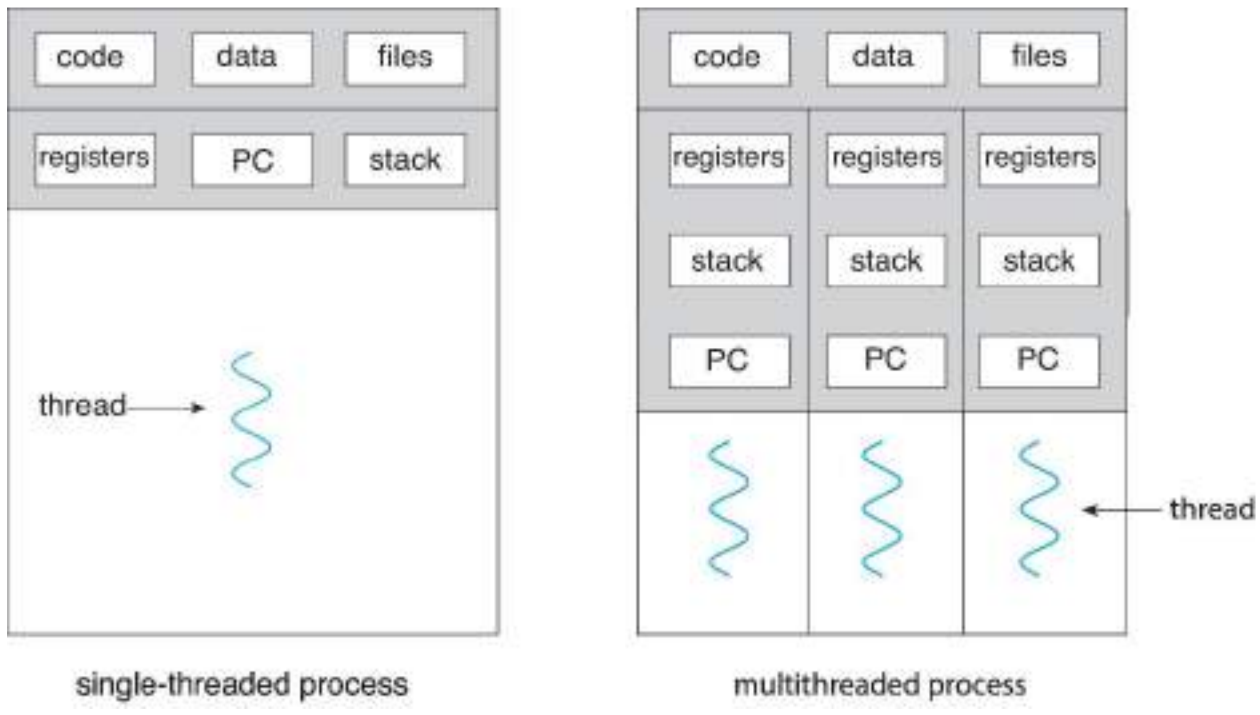


Figure: Single-threaded and multithreaded processes.

Benefits of threads

- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multicore architectures

User and Kernel Space Threads

- There are two main places to implement threads: **user space** and the **kernel**.

User level threads

- In **user level threads**, the threads package is put entirely in user space. The kernel knows nothing about them. User level threads are managed by users.
- The user-level threads are implemented and managed in user space.

- As far as the kernel is concerned, it is managing ordinary, single-threaded processes. The first, and most obvious, advantage is that a user-level threads package can be implemented on an operating system that does not support threads.
- With this approach, threads are implemented by a library (thread library).
- The threads run on top of a ***run-time system***, which is a collection of procedures that manage threads.
- When threads are managed in user space, each process needs its own private **thread table** to keep track of the threads in that process (This table is analogous to the kernel's process table, except that it keeps track only of the per-thread properties, such as each thread's program counter, stack pointer, registers, state, and so forth. The thread table is managed by the run-time system.)
- User-level threads are small and much faster than kernel level threads.

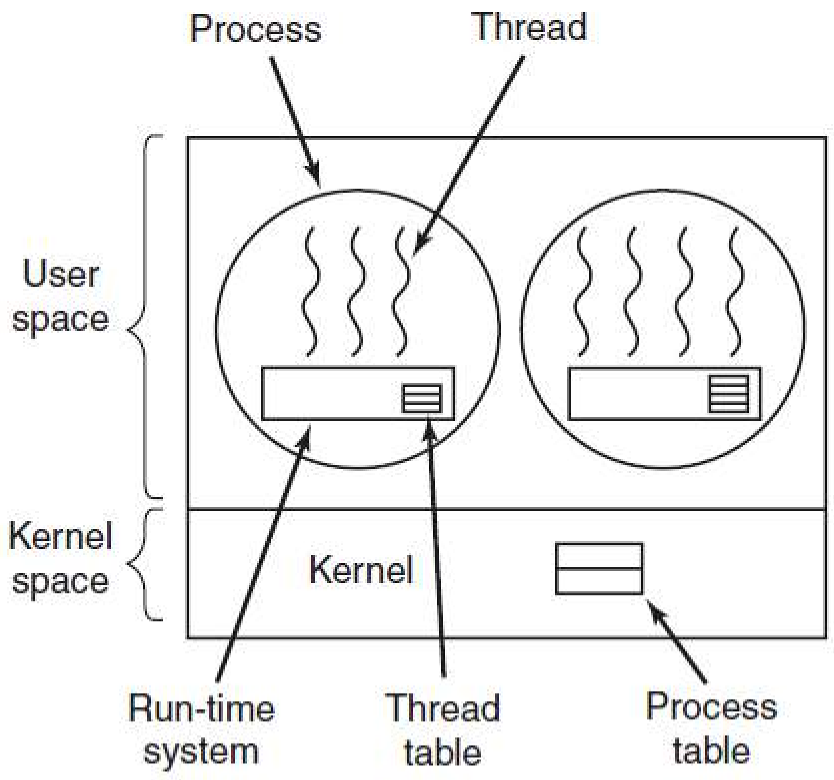


Figure: A user-level threads package

Advantages:

- Thread table contains info about threads (program counter, stack pointer...) so that run time system can manage them
- If thread blocks, run time system stores thread info in table and finds new thread to run.
- They allow each process to have its own customized scheduling algorithm.
- The procedure that saves the thread’s state and the scheduler are just local procedures, so invoking them is much more efficient than making a kernel call.
- ☞ User-level threads are easier and faster to create than kernel-level threads. They can also be more easily managed.
- ☞ User-level threads can be run on any operating system.
- ☞ There are no kernel mode privileges required for thread switching in user-level threads.

Limitations:

- ☞ The entire process is blocked if one user-level thread performs blocking operation.
- ☞ There is a lack of coordination between threads and operating system kernel.

Kernel Level Threads

- **Kernel-level threads** are handled by the operating system directly and the thread management is done by the kernel.
- The kernel knows about and manages the threads. No run-time system is needed.
- No thread table in each process.
- The **kernel has a thread table** that keeps track of all the threads in the system. The kernel’s thread table holds each thread’s registers, state, and other information.
- The information is the same as with user-level threads, but now kept in the kernel instead of in user space (inside the run-time system). This information is a subset of the information that traditional kernels maintain about their single threaded processes, that is, the process state. In addition, the **kernel also maintains the traditional process table to keep track of processes.**
- When a thread wants to create a new thread or destroy an existing thread, it makes a kernel call, which then does the creation or destruction by updating the kernel thread table

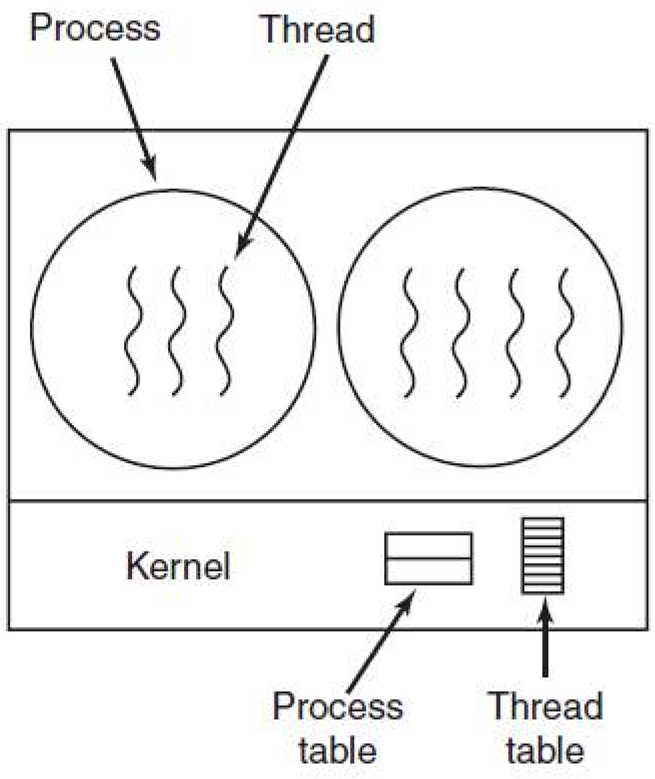


Figure: A threads package managed by the kernel.

Advantages:

- Kernel keeps same thread table as user table
- If thread blocks, kernel just picks another one
    - Not necessarily from same process!
- When one thread blocks, other threads from process can be scheduled

Limitations:

- Kernel-level threads are slower to create as well as manage as compared to user-level threads.
- Expensive to manage the threads in the kernel and takes valuable kernel space

Hybrid approach of implementing threads

- Multiplex user-level threads onto kernel level threads
  - Various ways have been investigated to try to combine the advantages of user level threads with kernel-level threads. One way is use kernel-level threads and then multiplex user-level threads onto some or all of them

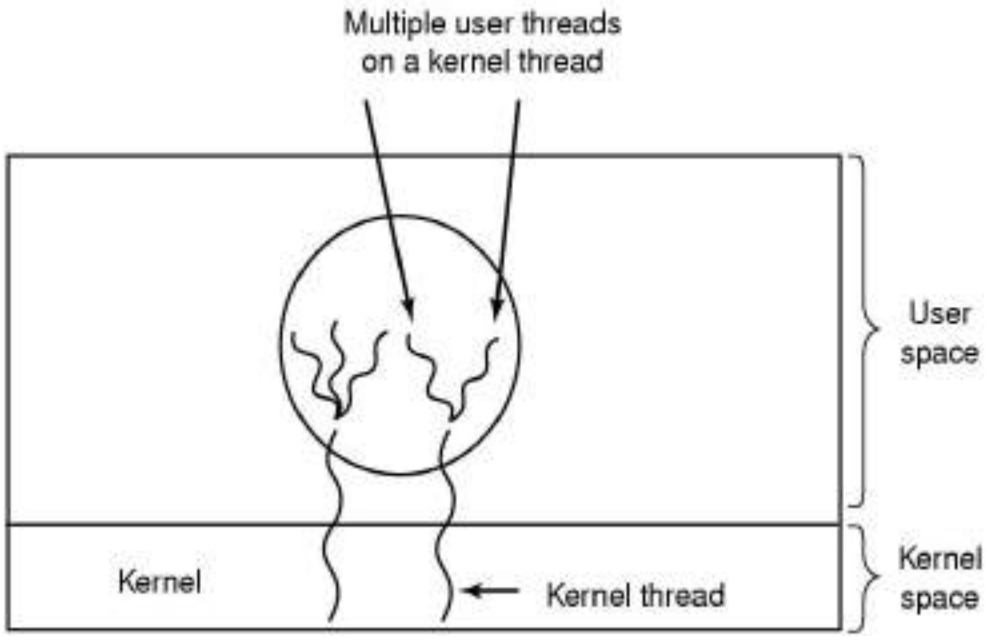


Figure : Multiplexing user-level threads onto kernel-level threads.

  - When this approach is used, the programmer can determine how many kernel threads to use and how many user-level threads to multiplex on each one. This model gives the ultimate in flexibility
  - With this approach, the kernel is aware of only the kernel-level threads and schedules those. Some of those threads may have multiple user-level threads multiplexed on top of them. These user-level threads are created, destroyed, and scheduled just like user-level threads in a process that runs on an operating system without multithreading capability. In this model, each kernel-level thread has some set of user-level threads that take turns using it.

Hybrid Approach

- ✓ Kernel is aware of kernel threads only
- ✓ User level threads are scheduled, created destroyed independently of kernel thread
- ✓ Programmer determines how many user level and how many kernel level threads to use

Inter Process Communication

Inter Process Communication (IPC) is the mechanism provided by the operating system that allows processes to communicate with each other. This communication could involve a process letting another process know that some event has occurred or the transferring of data from one process to another.

- A process **is independent** if it does not share data with any other processes executing in the system.
- A process is **cooperating** if it can affect or be affected by the other processes executing in the system.
- An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes.
- Cooperating process can affect or be affected by other processes, including sharing data
- Cooperating processes need **interprocess communication** (**IPC**)
- There are several reasons for providing an environment that allows **process cooperation**:
  - **Information sharing**: Since several applications may be interested in the same piece of information (for instance, copying and pasting), we must provide an environment to allow concurrent access to such information.
  - **Computation speedup**: If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.
  - **Modularity**: We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.

Three Issues to deal with

1. **How to do IPC**
   - How one process can pass information to another?
2. **How to deal with process conflicts**
   - Making sure two or more processes do not get in each other's way.
   - For example, **two processes** in an **airline reservation system** each trying to **grab the last seat on a plane** for a **different customer**.
3. **How to do correct sequencing when dependencies are present**
   - If process **A produces data** and process **B prints them**, B **has to wait** until A has produced some data before starting to print

NOTE:

SAME ISSUES FOR THREADS AS FOR PROCESSES-SAME SOLUTIONS AS WELL!!

Two Models of IPC

Cooperating processes require an inter process communication (IPC) mechanism that will allow them to exchange data— that is, send data to and receive data from each other. There are two fundamental models of inter process communication:

- **Shared memory**
- **Message passing**

- In the **shared-memory model**, a region of memory that is shared by the cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.
- In the **message-passing model**, communication takes place by means of messages exchanged between the cooperating processes

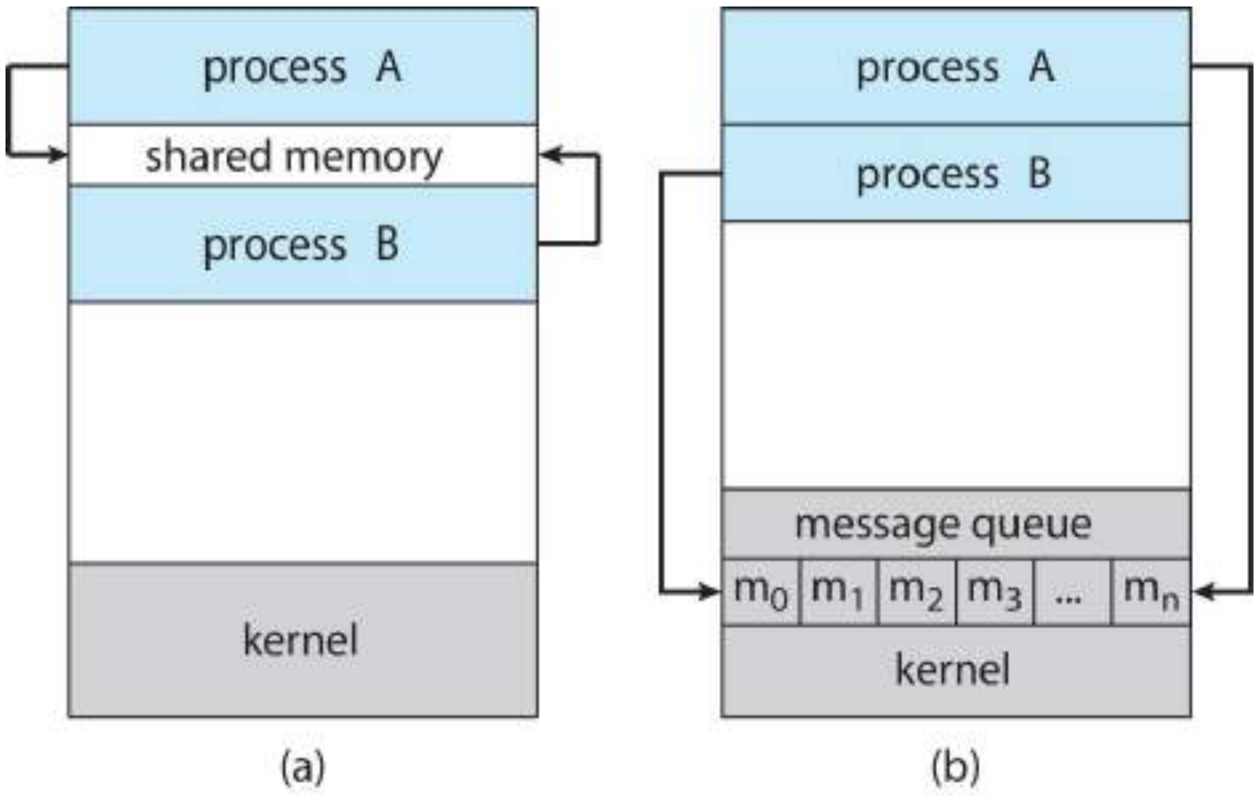


Figure : Communications models. (a) Shared memory. (b) Message passing.

Race Conditions

- The Situations, 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**.
- A race condition is a condition when there are many processes and every process shares the data (resource) with each other and accessing the data (resource) concurrently, and the output of execution depends on a particular sequence in which they share the data (resource) and access
- The inconsistent output produced by race conditions may cause bugs that are difficult to detect.
- Unfortunately, with increasing parallelism due to increasing numbers of cores, race condition are becoming more common.

Example to illustrate race condition

- To see how interprocess communication works in practice, let us now consider a simple but common example**: a print spooler**.
- When a process wants to print a file, it enters the file name in a special **spooler directory**.
- Another process, the **printer daemon**, periodically checks to see if there are any files to be printed, and if there are, it prints them and then removes their names from the directory.
- Imagine that our spooler directory has a very large number of slots, numbered 0, 1, 2, ..., each one capable of holding a file name.
- Also imagine that there are **two shared variables**, ***out***, which points to the next file to be printed, and ***in***, which points to the next free slot in the directory.
- At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files queued for printing). More or less simultaneously, processes A and B decide they want to queue a file for printing. (This situation is shown in figure below)

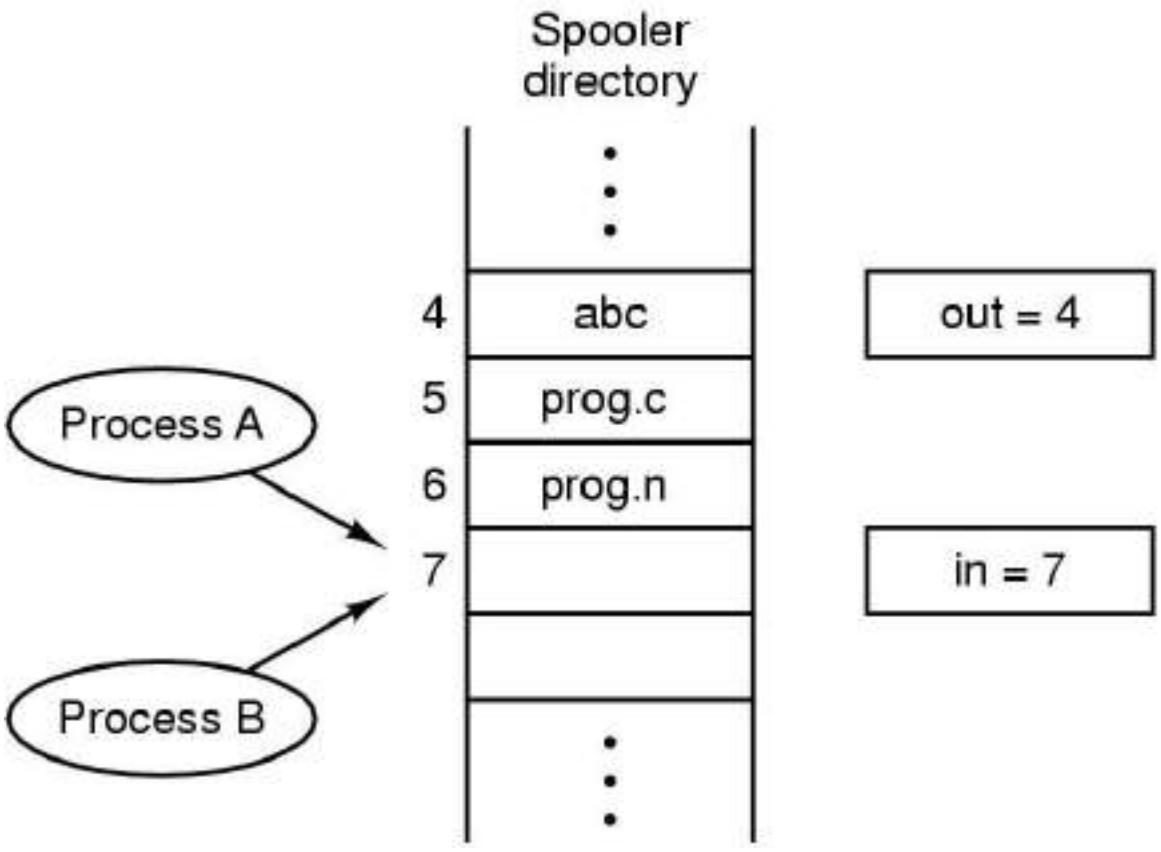


Figure: Tw o processes want to access shared memory at the same time

.

- Let's assume the following situation
  - **Process *A* reads** *in* and stores the value, 7, in a local variable called *next_free_slot*
  - Just then a clock interrupt occurs and the CPU decides that process *A* has run long enough, so **it switches to process *B*.**
  - **Process *B* also reads** *in* and also gets a 7. It, too, stores it in *it's* local variable *next_free_slot*
  - At this instant both processes think that the next available slot is 7.
  - **Process *B* now continues to run**. It stores the name of its file in slot 7 and updates *in* to be an 8. Then it goes off and does other things.
  - Eventually, **process *A* runs again**, starting from the place it left off. It looks at *next_free_slot*, finds a 7 there, and writes its file name in slot 7, erasing the name that process *B* just put there. Then it computes *next free slot* + 1, which is 8, and sets *in* to 8.
  - The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process ***B* will never receive any output**.
  - User *B* will hang around the printer for years, wistfully hoping for output that never comes
  - This situation is **race condition**

How to avoid races

- ✓ **Mutual exclusion**–only one process at a time can use a shared variable/file
- ✓ **Critical regions-**shared memory which leads to races
- ✓ **Solution-** Ensure that two processes can't be in the critical region at the same time

Critical Regions (Critical Sections)

- 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 races.
- Although this requirement avoids race conditions, *it is not sufficient* for having parallel processes cooperate correctly and efficiently using shared data. We need following four conditions to hold to have a good solution: **(Properties of a good solution)**
    1. No two processes may be simultaneously inside their critical regions.*(Mutual exclusion)*
    2. No assumptions may be made about speeds or the number of CPUs.
    3. No process running outside its critical region may block any process.
    4. No process should have to wait forever to enter its critical region. *(No starvation)*
- In an abstract sense, the behavior that we want is shown in following figure.
    - Here process A enters its critical region at time T_1.
    - 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.

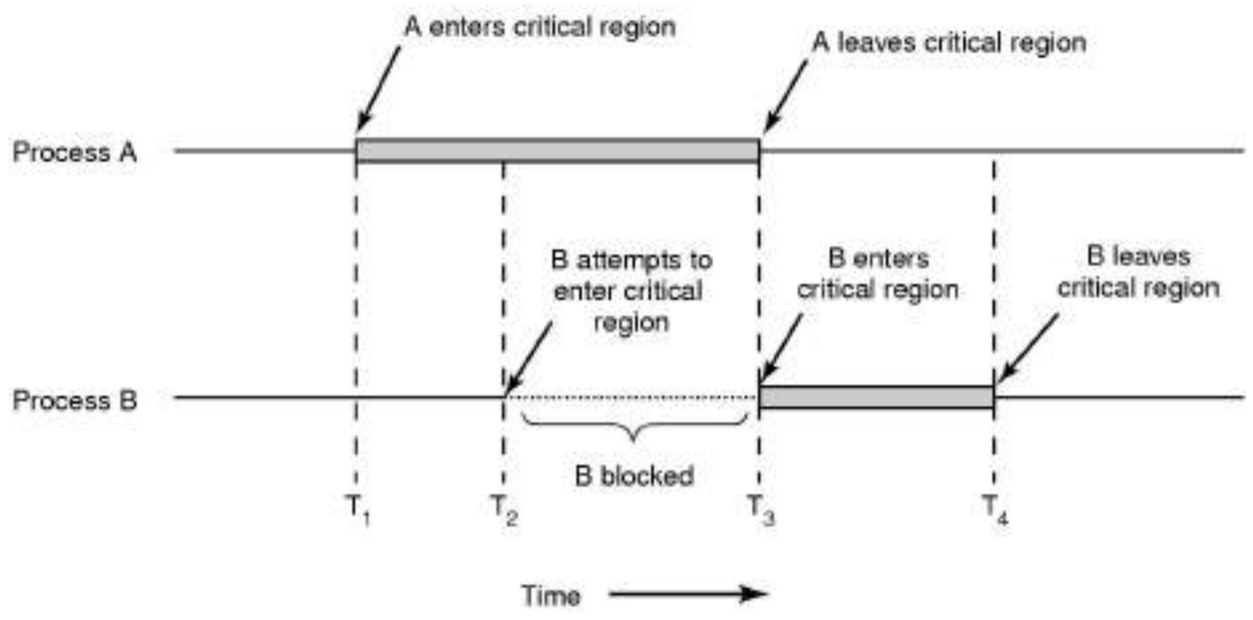


Figure: Mutual exclusion using critical regions

Implementing Mutual Exclusion

- The key to avoid race conditions in 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. Put in other words, what we need is **mutual exclusion**, that is, 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.
- There are different ways to implement mutual exclusion

Mutual Exclusion with Busy Waiting

- Techniques for achieving mutual exclusion, so that while one process is busy updating shared memory in its critical region, no other process will enter its critical region and cause trouble.
    - Disabling interrupts
    - Lock variables
    - Strict alternation
    - Peterson's solution
    - Test and Set Lock (The TSL Instruction)

Disabling interrupts

- On a single-processor system, the simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it.
- **Idea:** process disables interrupts, enters critical region, enables interrupts when it leaves critical region.
- **Problems:**
    - It is unwise to give user processes the power to turn off interrupts. Process might never enable interrupts, crashing system
    - Won't work on multi-core chips (or multiprocessor system) as disabling interrupts only effects one CPU at a time.
        - Disabling interrupts affects only the CPU that executed the disable instruction. The other ones will continue running and can access the shared memory.

Lock variables

- It is a software solution.
- Consider having 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,

    - ☞ lock = 0 => no process is in its critical section
    - ☞ lock = 1 => some process is in its critical section.

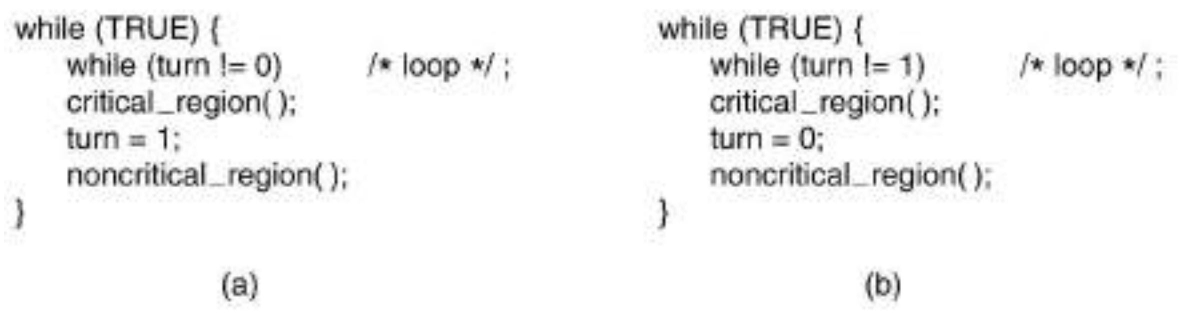
Problem:

- ❖ Race condition
    - If process P_1 sees the value of lock variable 0 and before it can set it to 1 context switch occurs.
    - Now process P_2 runs and finds value of lock variable 0, so it sets value to 1, enters critical region.
    - At some point of time P_1 resumes, sets the value of lock variable to 1, enters critical region.

- Now two processes are in their critical regions accessing the same shared memory, which violates the mutual exclusion condition.

Strict alternation

- This approach is the software mechanism implemented at user mode.
- It is a busy waiting solution which can be implemented only for two processes.
- In this approach, the integer variable **turn**, initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory
    - ✓ Initially, **turn** value is set to 0.
    - ✓ **turn = 0** means it is the turn of process P_0 to enter the critical section.
    - ✓ **turn = 1** means it is the turn of process P_1 to enter the critical section.
- IDEA: First me, then you!



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

(a)

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

(b)

Figure: A proposed solution (Strict alternation) to the critical-region problem. (a) Process 0. (b) Process 1.

This mechanism works as explained in the following scenes-

Scenario 1	Scenario 2	Scenario 3
▪ Process 0 (P_0) arrives. ▪ It executes the *turn! =0* instruction. ▪ Since ***turn*** value is set to 0, so it returns 0 (i.e. FALSE) to the while loop. ▪ The while loop condition breaks. ▪ Process P_0 enters the critical section and executes. ▪ Now, even if process P_0 gets preempted in the middle, process P1 cannot enter the critical section. ▪ Process 1(P_1) cannot enter unless process P_0 completes and sets the turn value to 1.	▪ Process P1 arrives. ▪ It executes the *turn! =1* instruction. ▪ Since ***turn*** value is set to 0, so it returns 1 (i.e. TRUE) to the while loop. ▪ The returned value 1 does not break the while loop condition. ▪ The process P_1 is trapped inside an infinite while loop. (busy waiting) ▪ The while loop keeps the process P_1 busy until the ***turn*** value becomes 1 and its condition breaks.	▪ Process P_0 comes out of the critical section and sets the ***turn*** value to 1. ▪ The while loop condition of process P_1 breaks. ▪ Now, the process P_1 waiting for the critical section enters the critical section and execute. ▪ Now, even if process P_1 gets preempted in the middle, process P_0 cannot enter the critical section.

Problems with strict alternation

Consider following scenario:

- Process 0 (P_0) exits the critical region and sets ***turn*** to 1. Process 1 (P_1) is also executing in non-critical region At this point turn is 1 and both processes are executing in their noncritical regions.
- Suddenly, P_0 finishes its noncritical region and goes back to the top of its loop. Unfortunately, it is not permitted to enter its critical region now, because ***turn*** is 1 and P_1 is busy with its noncritical region. It hangs in its while loop until process P_1 sets ***turn*** to 0.

This situation violates condition 3 set out above ("No process running outside its critical region may block any process."): process 0 is being blocked by a process not in its critical region.

☞ Taking turns is not a good idea when one of the processes is much slower than the other.

☞ Problems:

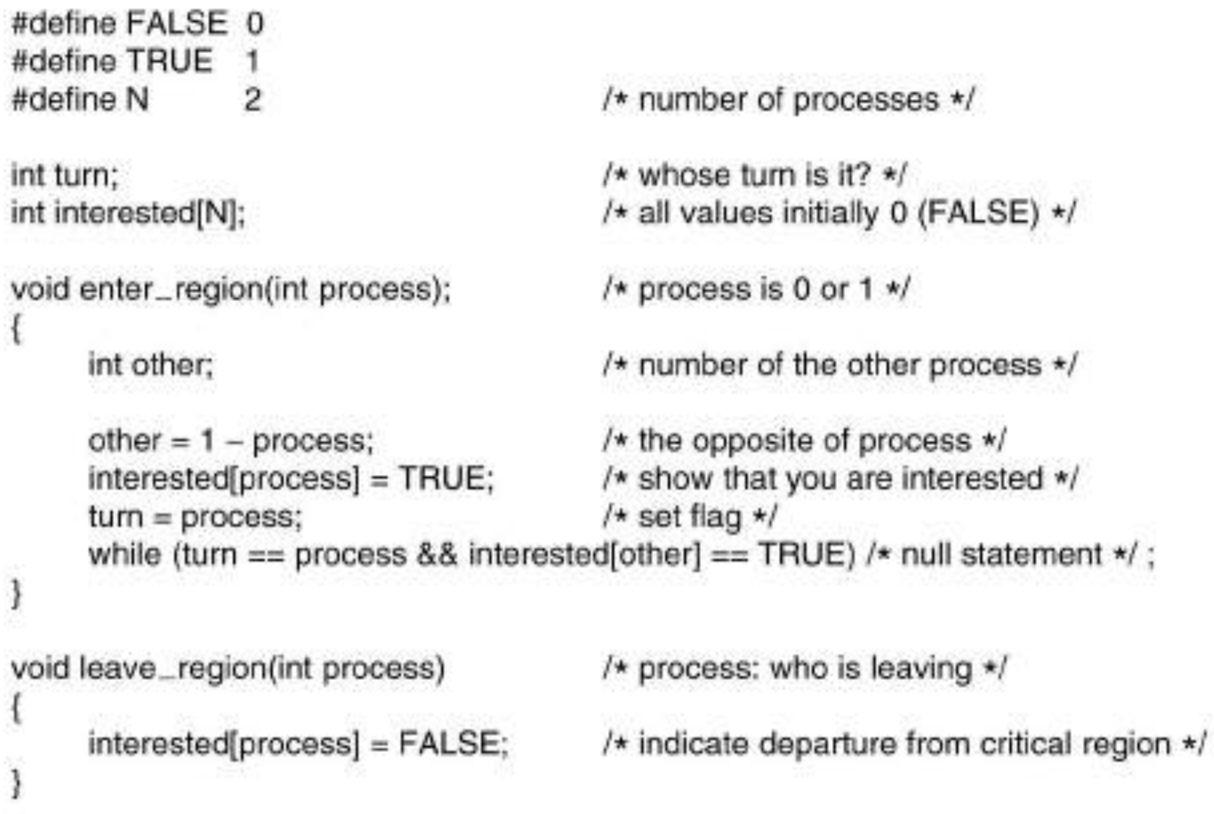
- Employs busy waiting-while waiting for the cr, a process spins
- If one process is outside the cr and it is its turn, then other process has to wait until outside guy finishes both outside AND inside (cr) work

NOTES:

- ❖ Continuously testing a variable until some value appears is called ***busy waiting*.** It should usually be avoided, since it wastes CPU time. Only when there is a reasonable expectation that the wait will be short is busy waiting used. A lock that uses busy waiting is called a ***spin lock***.
- ❖ In fact, this solution (strict alternation) requires that the two processes **strictly alternate in entering their critical regions,** for example, in spooling files. Neither one would be permitted to spool two in a row. While this algorithm does avoid all races, it is not really a serious candidate as a solution because it violates condition 3.

Peterson's Solution

- ❖ It is a classical software-based solution to the critical section
- ❖ This solution is restricted to two processes that alternate execution between their critical sections and remainder sections.



```
#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 */
}
```

Figure : Peterson's solution for achieving mutual exclusion.

- 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 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. (no busy waiting)
        - If process 1 now makes a call to ***enter_region***, it will hang there until *interested*[0] goes to *FALSE*, an event that happens only 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 overwritten and 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 no enter its critical region until process 0 exits its critical region.

Disadvantages:

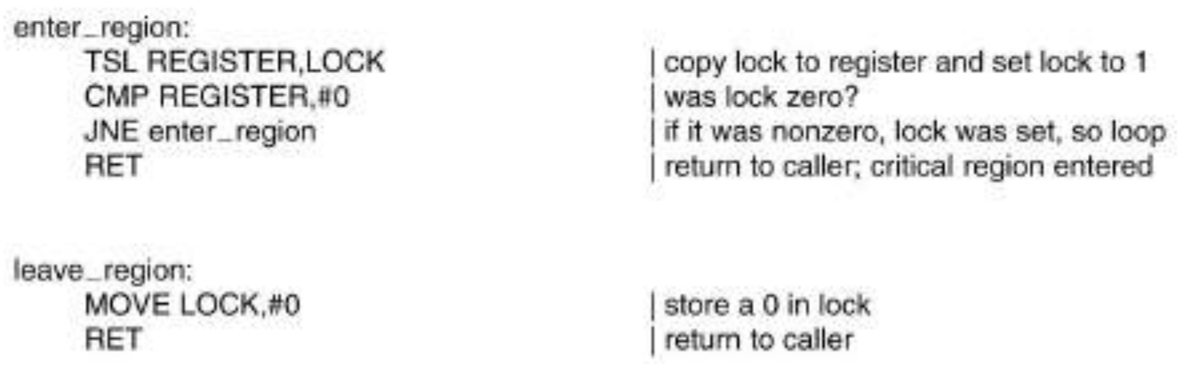
- Peterson's solution is limited to two processes.
- It involves Busy Waiting.

Test and Set Lock

- Hardware based solution.
- It uses a test and set instruction to provide the synchronization among the processes executing concurrently.
- Some computers, especially those designed with multiple processors in mind, have an instruction like `TSL RX,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*.
    - ☞ *TSL reads lock into register and stores NON ZERO VALUE in lock (e.g. process number)*
  - 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.
    - ☞ *Instruction is atomic: done by freezing access to memory bus line (bus is disabled)*

> ✍ TSL instruction is an instruction that **returns the old value of a memory location and sets the memory location value to nonzero** as a **single atomic operation**.
> ✍ If one process is currently executing a test-and-set, no other process is allowed to begin another test-and-set until the first process test-and-set is finished.
> ✍ TSL is atomic. Memory bus is locked until it is finished executing.

- To use the TSL instruction, a shared variable, ***lock***, is used 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.
    - ☑ ***lock = 0*** means the critical section is currently vacant and no process is present inside it.
    - ☑ ***lock = 1*** means the critical section is currently occupied and a process is present inside it.



```
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 critica:region using the TSL instruction.

- The TSL instruction can be used to prevent two processes from simultaneously entering their critical regions. The solution is given in above figure (algorithm).
- There a four-instruction subroutine (***enter_region***) in a fictitious (but typical) assembly language is shown.
    - The first instruction (***TSL REGISTER,LOCK***) copies the old value of lock to the register and then sets lock to 1.
    - Then the old value is compared with 0 (***CMP REGISTER,#0***) .
    - If it is nonzero, the lock was already set, so the program just goes back to the beginning and tests it again.(***REGISTER,#0***)
    - Sooner or later it will become 0 (when the process currently in its critical region is done with its critical region), and the subroutine returns, with the lock set. (***RET)***
- Another subroutine, ***Leave_region*** is used for clearing the lock which is very simple. The program just stores a 0 in lock. No special synchronization instructions are needed.

Disadvantages:

- It is a busy waiting solution which keeps the CPU busy when the process is actually waiting.
- TSL doesn't provide Architectural Neutrality. It depends on the hardware platform. The TSL instruction is provided by the operating system. Some platforms might not provide that. Hence it is not Architectural natural.

Main Problem in Peterson's solution and TSL instruction

- Both Peterson’s solution and the solutions using TSL are correct, but both have the defect of requiring busy waiting.
- In essence, what these solutions do is this: ***when a process wants to enter its critical region, it checks to see if the entry is allowed. If it is not, the process just sits in a tight loop waiting until it is.***
- Not only does this approach waste CPU time, but it can also have unexpected effects (Priority Inversion Problem)

Problems:

- Busy waiting – waste of CPU time
- Priority Inversion Problem

.

Priority Inversion Problem

- Priority inversion is the situation when high priority process is waiting for low priority process.
- Consider a computer with two processes, H, with high priority, and L, with low priority.
    - The scheduling rules are such that H is run whenever it is in ready state.
    - At a certain moment, with L in its critical region, H becomes ready to run (e.g., an I/O operation completes).
    - H now begins busy waiting, but since L is never scheduled while H is running, L never gets the chance to leave its critical region, so H loops forever.
- This situation is sometimes referred to as the **priority inversion problem.**

Solutions:

There are some interposes communication primitives that block instead of wasting CPU time when they are not allowed to enter their critical regions.

Sleep and Wakeup

- There are some interposes communication primitives that block instead of wasting CPU time when they are not allowed to enter their critical regions. One of the simplest is the **sleep and wakeup.**
- As its name suggests, it uses two system calls: **sleep** and **wakeup**. If a process is unable to enter a critical section, it is put to sleep (i.e. temporarily suspended). Afterwards, when the critical section is no longer occupied, the process is woken up.
    - **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.
- 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.

Example to show how the sleep and wakeup primitives are used

❖ **Producer Consumer Problem**

- Also known as the bounded-buffer problem
- **Two processes share a common, fixed-size buffer. One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out.**

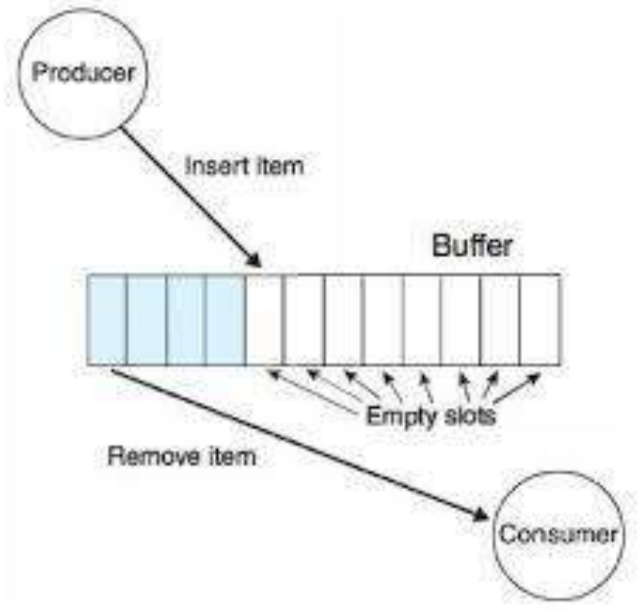
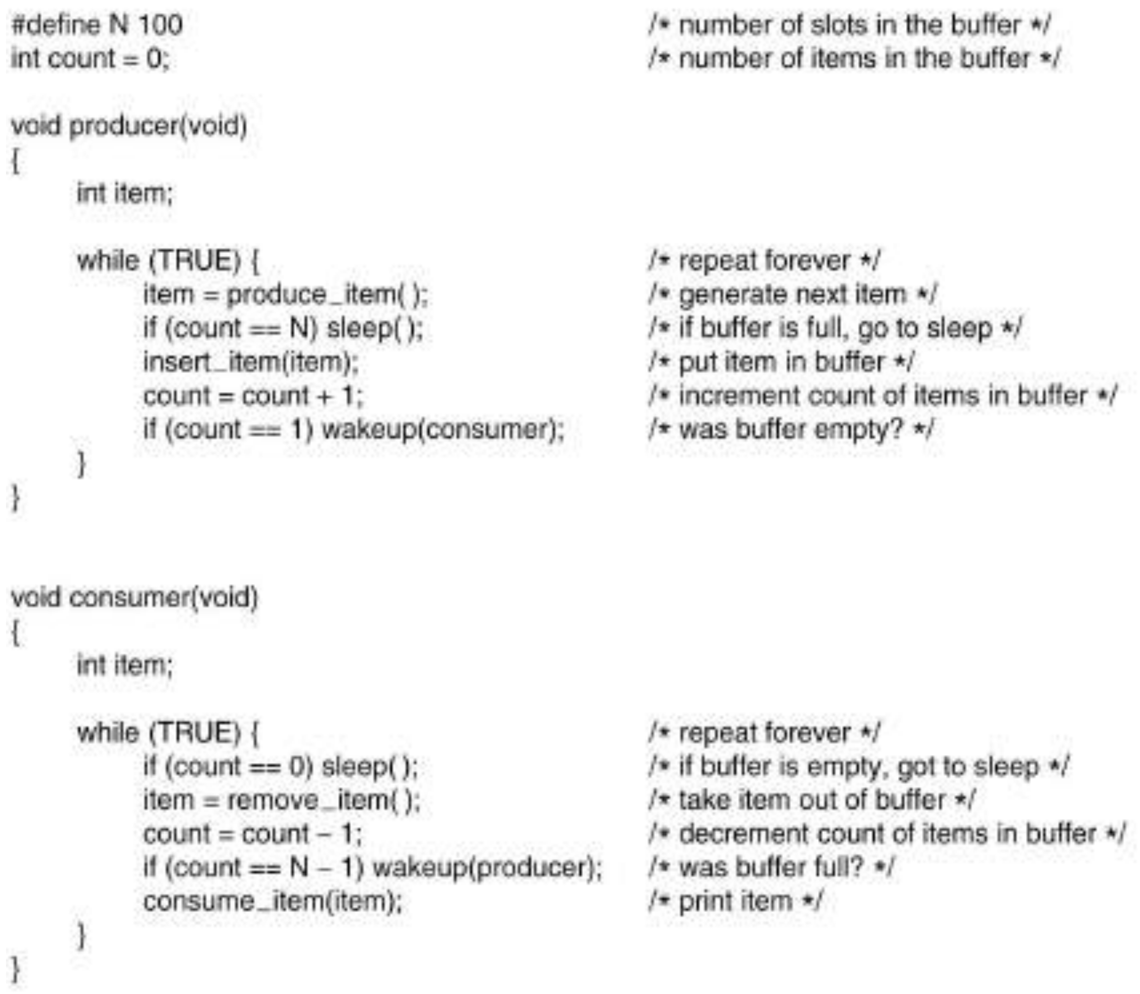


Figure: Producer-Consumer Problem

(It is also possible to generalize the problem to have **m** producers and **n** consumers, but we will consider only the case of one producer and one consumer because this assumption simplifies the solutions.)

- ☞ 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.
- ☞ To keep track of the number of items in the buffer, we will need a variable, *count*. If the maximum number of items the buffer can hold is *N*,
  - o The **producer's code** will 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.
  - o The **consumer's code** is similar: 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 processes also tests to see if the other should be awakened, and if so, wakes it up. The code for both producer and consumer is shown in following figure.



```
#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);                /* 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: The producer-consumer problem with a fatal race condition

Problem:

This approach sounds simple enough, but it leads to the same kinds of race conditions we saw earlier with the spooler directory. It can occur because access to *count* is unconstrained. As a consequence, the following situation could possibly occur.

- The buffer is empty and the consumer has just read *count* to see if it is 0.
- At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer.
- The producer inserts an item in the buffer, increments *count*, and notices that it is now 1.
- Reasoning that *count* was just 0, and thus the consumer must be sleeping, the producer calls *wakeup* to wake the consumer up.
- Unfortunately, the consumer is not yet logically asleep, so the **wakeup signal is lost**.

- When the consumer next runs, it will test the value of *count* it previously read, find it to be 0, and **go to sleep**.
- Sooner or later the producer will fill up the buffer and also **go to sleep**. Both will **sleep forever.**

NOTE: The essence of the problem here is that a wakeup sent to a process that is not (yet) sleeping is lost. If it were not lost, everything would work.

Semaphore

- A semaphore is a variable that provides an abstraction for controlling access of a shared resource by multiple processes in a parallel programming environment.
- There are 2 types of semaphores:

1. **Binary semaphores**
    - Binary semaphores can take only 2 values (0/1).
    - Binary semaphores have 2 methods associated with it (up, down / lock, unlock).
    - They are used to acquire locks.
2. **Counting semaphores**
    - Counting semaphore can have possible values more man two.

NOTE: Dijkstra (1965) suggested using an integer variable to count the number of wakeups saved for future use. In his proposal, a new variable type, which he called a **semaphore**, was introduced. A semaphore could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending.

- Two operations on semaphores : **down** and **up**
    - Down checks semaphore. If not zero, decrements semaphore. If zero, process goes to sleep
    - Up increments semaphore. If more than one process asleep, one is chosen randomly and enters critical region (first does a down)
- Checking the value, changing it, and possibly going to sleep, are all done as a single, indivisible **atomic action**. It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked. This atomicity is absolutely essential to solving synchronization problems and avoiding race conditions.

- The operation of incrementing the semaphore and waking up one process is also indivisible. No process ever blocks doing an up, just as no process ever blocks doing a wakeup in the earlier model.

Solving Producer-Consumer Problem using Semaphores

```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                     /* TRUE is the constant 1 */
          item = produce_item( );       /* generate something to put in buffer */
          down(&empty);                 /* decrement empty count */
          down(&mutex);                 /* enter critical region */
          insert_item(item);            /* put new item in buffer */
          up(&mutex);                   /* leave critical region */
          up(&full);                    /* increment count of full slots */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                     /* infinite loop */
          down(&full);                  /* decrement full count */
          down(&mutex);                 /* enter critical region */
          item = remove_item( );        /* take item from buffer */
          up(&mutex);                   /* leave critical region */
          up(&empty);                   /* increment count of empty slots */
          consume_item(item);           /* do something with the item */
     }
}
```

Figure: The producer-consumer problem using semaphores

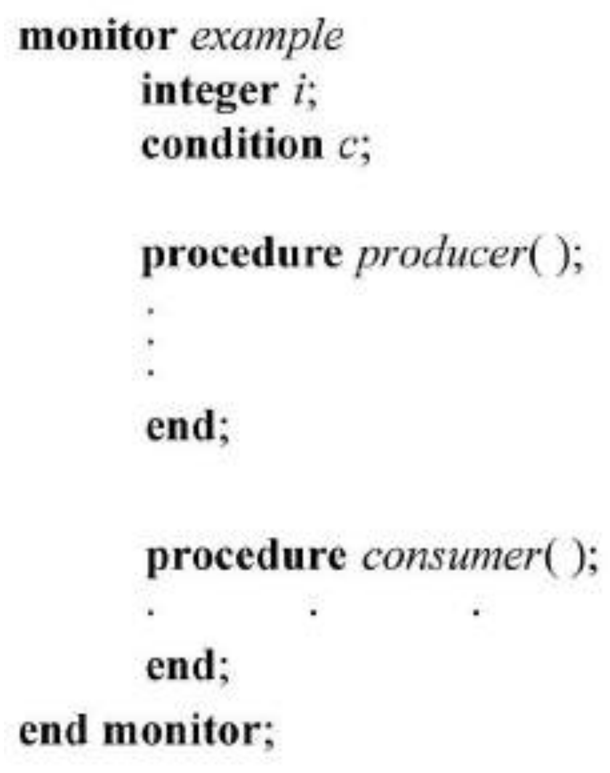
- Three semaphores: *full, empty* and *mutex* are used in this solution.
    - Full counts full slots (initially 0)
    - Empty counts empty slots (initially N (number of slots in the buffer))
    - Mutex makes sure that the producer and consumer do not access at the same time. (initially 1)
- If each process does a down just before entering its critical region and an up just after leaving it, mutual exclusion is guaranteed
- We want functions *insert_item* and *remove_item* such that the following hold:
    - *Mutually exclusive access to buffer:* At any time only one process should be executing (either insert_item or remove_item).
    - *No buffer overflow:* A process executes insert_item only when the buffer is not full (i.e., the process is blocked if the buffer is full).
    - *No buffer underflow:* A process executes remove_item only when the buffer is not empty (i.e., the process is blocked if the buffer is empty).
    - *No busy waiting.*
    - *No producer starvation:* A process does not wait forever at insert_item() provided the buffer repeatedly becomes full.
    - *No consumer starvation:* A process does not wait forever at remove_item() provided the buffer repeatedly becomes empty.
- In this example semaphores are used in two different ways:
    1. **For Mutual Exclusion:** The *mutex* semaphore is used for mutual exclusion. It is designed to guarantee that only one process at a time will be reading or writing the buffer and the associated variables.
    2. **For Synchronization:** The *full* and *empty* semaphores are needed to guarantee that certain event sequences do or do not occur. In this case, they ensure that the producer stops running when the buffer is full, and that the consumer stops running when it is empty.

Limitations

- ***Little errors cause disasters.***
  - *In Producer consumer with semaphores, interchange of two downs in producer code causes deadlock*
    - Suppose that the two downs in the producer's code were reversed in order, so *mutex* was decremented before *empty* instead of after it.
    - If the buffer were completely full, the producer would block, with *mutex* set to 0.
    - Consequently, the next time the consumer tried to access the buffer, it would do a down on *mutex*, now 0, and block too.
    - Both processes would stay blocked forever and no more work would ever be done.
    - This unfortunate situation is a deadlock situation.

Monitors

- A higher-level synchronization primitive
- 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
- In a monitor it is the job of the compiler, not the programmer to enforce mutual exclusion.
- Monitors have an important property that makes them useful to achieve mutual exclusion: *only one process at a time can be in the monitor*.
- Monitor is a **language construct** which enforces mutual exclusion and blocking mechanism
- C does not have monitor



```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .         .          .
        end;
end monitor;
```

Figure: A monitor - a picture

- Only one process at a time can be in the monitor
  - When a process calls a monitor, the first thing done is to check if another process is in the monitor. If so, calling process is suspended.
- Need to enforce blocking as well –
  - ☞ Use condition variables
  - ☞ Use wait , signal operations on condition variables
- When a monitor discovers that it can’t continue (e.g. buffer is full), issues a signal on a condition variable (e.g. full) causing process (e.g. producer) to block
- Another process is allowed to enter the monitor (e.g. consumer).This process can issue a signal, causing blocked process (producer) to wake up
- Process issuing signal leaves monitor

A skeleton of the **producer-consumer problem with monitors** is given in following figure in an imaginary language, Pidgin Pascal

```
monitor ProducerConsumer
        condition full, empty;
        integer count;

        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;

        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count − 1;
                if count = N − 1 then signal(full)
        end;

        count := 0;
end monitor;

procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;

procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Figure: An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has N *slots*

Limitations

- You need a language which supports monitors (eg: Java).
- OS's are written in C

NOTE:

- Monitors and semaphores only work for shared memory
- They don't work for multiple CPU's which have their own private memory (Eg: A distributed system consisting of multiple CPUs, each with its own private memory and connected by a local area network.)

Conclusion: The conclusion is that **semaphores are too low level** and **monitors are not usable except in a few programming languages**. Also, **none of the primitives allow information exchange between machines**. Something else (Message Passing) is needed.

Message Passing

- Message Passing is information exchange between machines
- This method of interprocess communication **uses two primitives**: ***send*** and ***receive,*** which, like semaphores and unlike monitors, are system calls rather than language constructs
- Two primitives in Message Passing
    1. Send: It is used to send the message.

       ```
       send (destination, &message)
       ```

       Here, *destination* is the process to which sender want to send *message* and message is what the sender wants to send.

    2. Receive: It is used to receive the message.

       ```
       receive (source,&message)
       ```

       Here, *source* is the process that has sent message and *message* is what the sender has sent.

- Message passing is commonly used in parallel programming systems. One well-known message-passing system, for example, is MPI (Message-Passing Interface).

The Producer-Consumer Problem with Message Passing

```
#define N 100                                /* number of slots in the buffer */

void producer(void)
{
        int item;
        message m;                           /* message buffer */

        while (TRUE) {
                item = produce_item( );      /* generate something to put in buffer */
                receive(consumer, &m);       /* wait for an empty to arrive */
                build_message(&m, item);     /* construct a message to send */
                send(consumer, &m);          /* send item to consumer */
        }
}

void consumer(void)
{
        int item, i;
        message m;

        for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
        while (TRUE) {
                receive(producer, &m);       /* get message containing item */
                item = extract_item(&m);     /* extract item from message */
                send(producer, &m);          /* send back empty reply */
                consume_item(item);          /* do something with the item */
        }
}
```

Classical IPC problems

Dining Philosophers Problem

- In 1965, Dijkstra posed and then solved a synchronization problem he called the **dining philosophers problem.** Since that time, everyone inventing yet another synchronization primitive has felt obligated to demonstrate how wonderful the new primitive is by showing how elegantly it solves the dining philosopher's problem.
- The problem can be stated quite simply as follows

  Five philosophers are seated 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. Between each pair of plates is one fork. The life of a philosopher consists of alternating periods of eating and thinking. When a philosopher gets sufficiently hungry, she tries to acquire her left and right forks, one at a time, in either order. If successful in acquiring two forks, she eats for a while, then puts down the forks, 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?

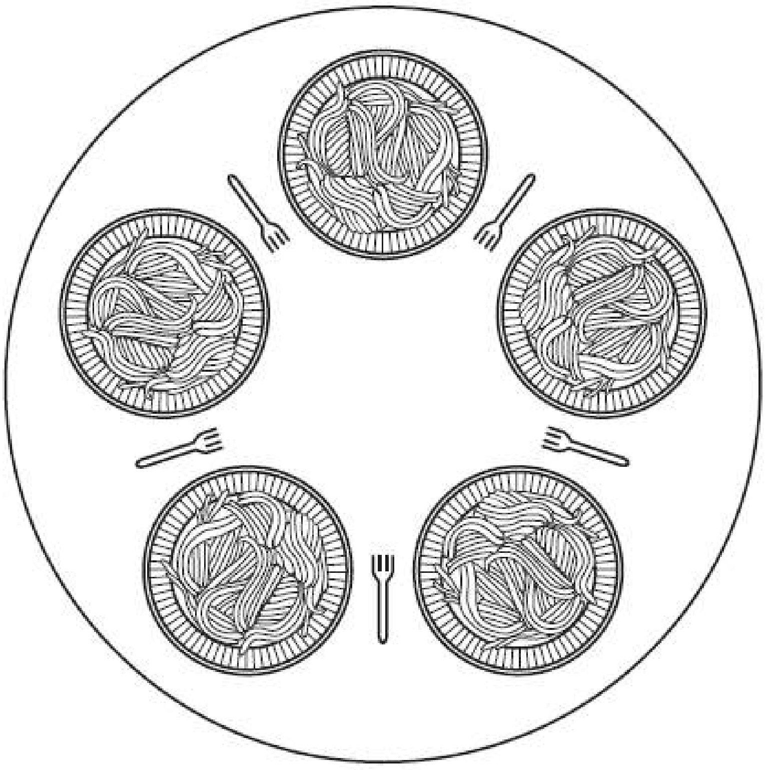
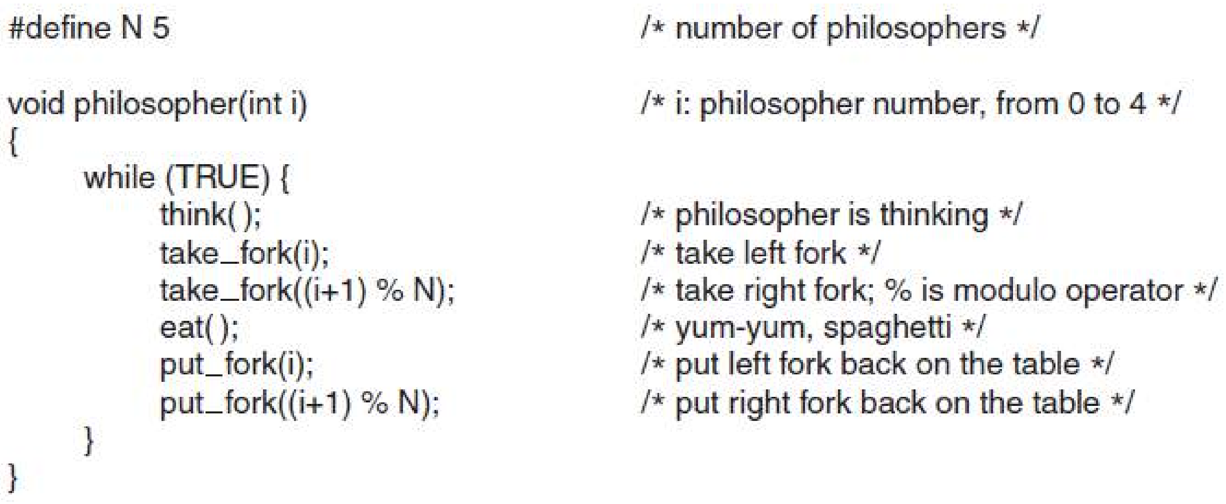


Figure: Lunch time in the Philosophy Department (Layout of Table in Dining Philosophers Problem).

- ☑ ***A dining table with N plates and N forks***
- ☑ ***N Philosophers eat or think***
- ☑ ***Eating needs two forks***
- ☑ ***Pick one fork at a time***
- ☑ ***How to prevent deadlock?***

- ➢ An obvious solution is illustrated in following figure. The procedure ***take_fork*** waits until the specified fork is available and then seizes it.



```
#define N 5                          /* number of philosophers */

void philosopher(int i)              /* i: philosopher number, from 0 to 4 */
{
      while (TRUE) {
            think( );                /* philosopher is thinking */
            take_fork(i);            /* take left fork */
            take_fork((i+1) % N);    /* take right fork; % is modulo operator */
            eat( );                  /* yum-yum, spaghetti */
            put_fork(i);             /* put left fork back on the table */
            put_fork((i+1) % N);     /* put right fork back on the table */
      }
}
```

Figure : A nonsolution to the dining philosophers problem

☞ *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 itis, pick it up
- Eat
- Put the left fork down
- Put the right fork down
- Repeat from the start

- Unfortunately, this solution is wrong.
    - 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 easily 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.**

  This proposal too, **fails**, although for a different reason:

    - Suppose all the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available, putting down their left forks, waiting, and picking 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.**

A solution presented below is **deadlock-free and allows the maximum parallelism for an arbitrary number of philosophers.**

- *It uses an array, state, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks).*
- *A philosopher may move into eating state only if neither neighbor is eating.*
- *Philosopher i's neighbors are defined by the macros LEFT and RIGHT. In other words, 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.
- ***A philosopher can only move to eating state if neither neighbor is eating.***

```
#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( );                    /* yum-yum, 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]);
     }
}
```

Figure : A solution to the dining philosophers problem

Sleeping Barber Problem

Figur:Barber Shop and Sleeping Barber

- This problem is based on a hypothetical barbershop with one barber.
- There is a barber shop which has one barber, one barber chair, and n chairs for waiting customers.
- The following rules apply:
    - *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.*
    - *When the barber finishes a haircut, he inspects the waiting room to see if there are any waiting customers and falls asleep if there are none.*
- Based on a naïve analysis, the above description should ensure that the shop functions correctly, with the barber cutting the hair of anyone who arrives until there are no more customers, and then sleeping until the next customer arrives.

- In practice, there are a number of problems that can occur that are illustrative of general scheduling problems.
- Some cases: (problems)
    - A customer may arrive and observe that the barber is cutting hair, so he goes to the waiting room. While he is on his way, the barber finishes the haircut he is doing and goes to check the waiting room. Since there is no one there (the customer not having arrived yet), he goes back to his chair and sleeps. The barber is now waiting for a customer and the customer is waiting for the barber.
    - Two customers may arrive at the same time when there happens to be a single seat in the waiting room. They observe that the barber is cutting hair, go to the waiting room, and both attempt to occupy the single chair.
- The solution of this problem is to include three Semaphores.
    - The first one, **customers**,to count the number of customers present in the waiting room.
    - The second one, **barbers,** for the barber. 0 and 1 are used to signify if the barber is idle or not.
    - The third, **mutex ,** is for mutual exclusion. It is needed for the program to run.
- 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
- 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. 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 not 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.
- 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.

```
#define CHAIRS 5  /*number of chairs for waiting customers */
typedef int semaphore;
semaphore customers = 0;             /* number of waiting customers */
semaphore barbers = 0; /* number of barbers waiting for customers */
semaphore mutex = 1;          /* for mutual exclusion */
int waiting = 0;        /* customers are waiting not being haircut */
void barber(void)
{
      while (TRUE)
     {
      down(&customers); /* go to sleep if number 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, non-CR */
       }
}
void customer(void)
{
       down(&mutex); /* enter CR */
       if (waiting < CHAIRS)
       {
       waiting = waiting + 1; /* increment count of waiting customers */
       up(&customers); /* wake up barber if necessary */
       up(&mutex); /* release access to ‘waiting’ */
       down(&barbers); /* wait if no free barbers */
       get_haircut(); /* non-CR */
       }
       else
       {
       up(mutex); /* shop is full, do not wait */
```

```
        }
}
```

Assignment: Discuss about Readers - Writers problem in IPC with solution.

Process Scheduling

- When a computer is multiprogrammed, it frequently has multiple processes or threads competing for the CPU at the same time. This situation occurs whenever two or more of them are simultaneously in the ready state.
- If only one CPU is 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**.

*The **process scheduling** is the activity of the process scheduler that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.*

☑ **Process scheduler** selects among available processes for next execution on CPU core

Who cares about scheduling algorithms?

- ☞ Batch servers
- ☞ Time sharing machines
- ☞ Networked servers
- ☞ You care if you have a bunch of users and/or if the demands of the jobs differ

Who doesn't care about scheduling algorithms?

- ☞ PC's
  - — One user who only competes with himself for the CPU

Process Behavior

Nearly all ***processes alternate bursts of computing with (disk or network) I/O requests***. (Process execution consists of cycle of CPU execution and I/O wait).

Some processes, such as the one in following figure (a), *spend most of their time computing*, while other processes, such as the one shown in (b), *spend most of their time waiting for I/O.*

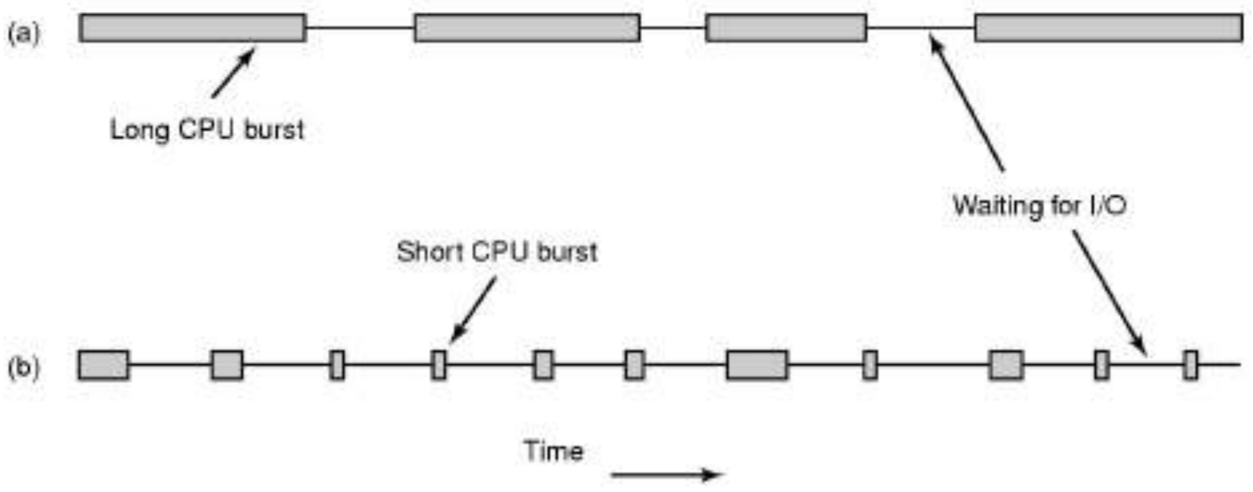


Figure: Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process

- ☑ The processes which spends most of their time computing are called **compute-bound or CPU bound process**
- ☑ The processes which spends most of their time waiting for I/O are called **I/O bound process**
- ☑ **CPU Burst** is the time that a process spends on the CPU executing some code. CPU Burst deals with the **running state** of the process.
- ☑ **I/O Burst** is the time that a process spends in waiting for the completion of the I/O request. I/O Burst deals with the **waiting state** of the process.
- ☑ *Compute-bound processes typically have long CPU bursts* and thus *infrequent I/O waits,* whereas *I/O-bound processes have short CPU bursts and thus frequent I/O waits*.
- ☑ Note that the **key factor is the length of the CPU burst**, **not the length of the I/O burst**.
  - o I/O-bound processes are I/O bound because they do not compute much between I/O requests, not because they hav e especially long I/O requests.
  - o It takes the same time to issue the hardware request to read a disk block no matter how much or how little time it takes to process the data after they arrive.

The basic idea*: If an I/O-bound process wants to run, it should get a chance quickly so that it can issue its disk request and keep the disk busy*

CPU Scheduler and Dispatcher

- ☑ Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the **sort-term scheduler** (or **CPU scheduler**). The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that processes.
- ☑ The **dispatcher** is the module that gives control of the CPU to the processes selected by the short-term scheduler. The time it's takes for the dispatcher to stop one process and start another running is known as the **dispatch latency.**

When to Schedule?

A key issue related to scheduling is when to make scheduling decisions. It turns out that there are a variety of situations in which scheduling is neede d.

- ***When a new process is created***, a decision needs to be made whether to run the parent process or the child process.
- ***When a process exits***. The process can no longer run (since it no longer exists), so some other process must be chosen from the set of ready processes.
- ***When a process blocks on I/O***, on a semaphore, or for some other reason, another process has to be selected to run.
- ***When an I/O interrupt occur***. If the interrupt came from an I/O device that has now completed its work, some process that was blocked waiting for the I/O may now be ready to run. It is up to the scheduler to decide whether to run the newly ready process, the process that was running at the time of the interrupt, or some third process.

Preemptive and Nonpreemptive Scheduling

Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts.

Nonpreemptive Scheduling

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

Preemptive Scheduling

- A **preemptive** scheduling algorithm 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 the time interval to give control of the CPU back to the scheduler.

In preemptive scheduling, the CPU is allocated to the processes for a limited time whereas, in Non-preemptive scheduling, the CPU is allocated to the process till it terminates or switches to the waiting state.

Scheduling Criteria

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms. Many criteria have been suggested for comparing CPU-scheduling algorithms. Some of them are as follows:

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – number of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced

Turnaround Time = Completion Time – Arrival Time

Waiting Time = Turnaround Time – Burst Time

Response Time = First CPU Entry Time – Arrival Time

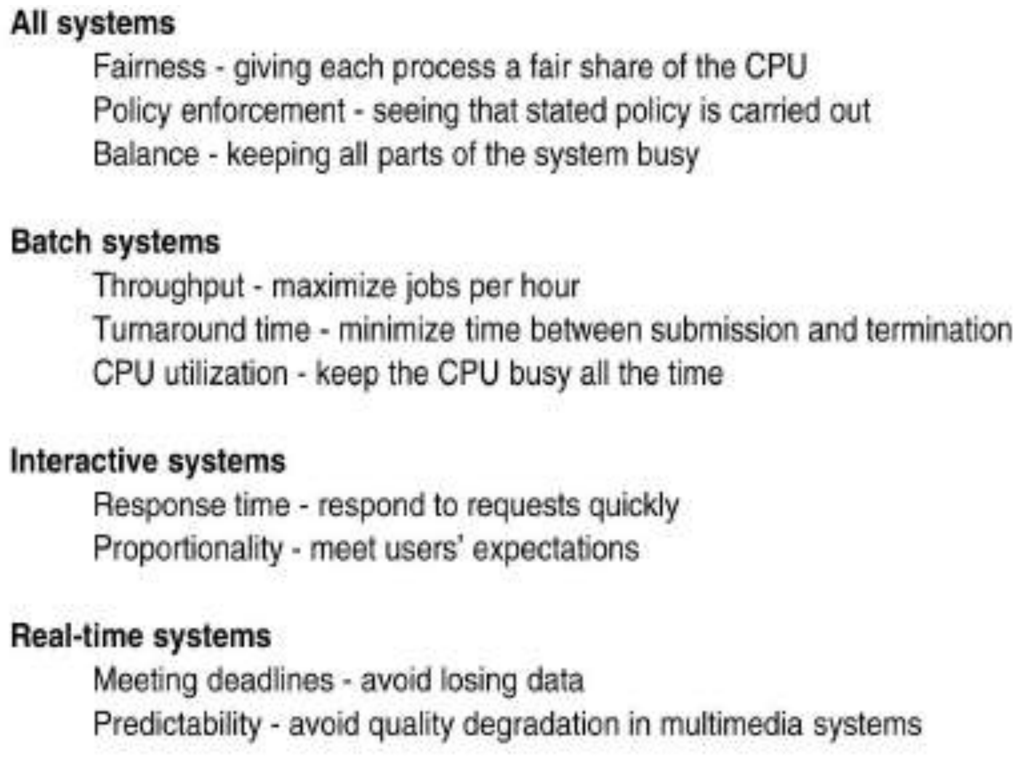
Categories of Scheduling Algorithms

In different environments different scheduling algorithms are needed. This situation arises because different application areas (and different kinds of operating systems) have different goals. In other words, what the scheduler should optimize for is not the same in all systems. Three environments worth distinguishing are

1. Batch systems.
2. Interactive systems.
3. Real time systems.

Scheduling Algorithm Goals

In order to design a scheduling algorithm, it is necessary to have some idea of what a good algorithm should do. Some goals depend on the environment (batch, interactive, or real time), but some are desirable in all cases. Some goals are listed below:



All systems
- Fairness - giving each process a fair share of the CPU
- Policy enforcement - seeing that stated policy is carried out
- Balance - keeping all parts of the system busy

Batch systems
- Throughput - maximize jobs per hour
- Turnaround time - minimize time between submission and termination
- CPU utilization - keep the CPU busy all the time

Interactive systems
- Response time - respond to requests quickly
- Proportionality - meet users' expectations

Real-time systems
- Meeting deadlines - avoid losing data
- Predictability - avoid quality degradation in multimedia systems

Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time

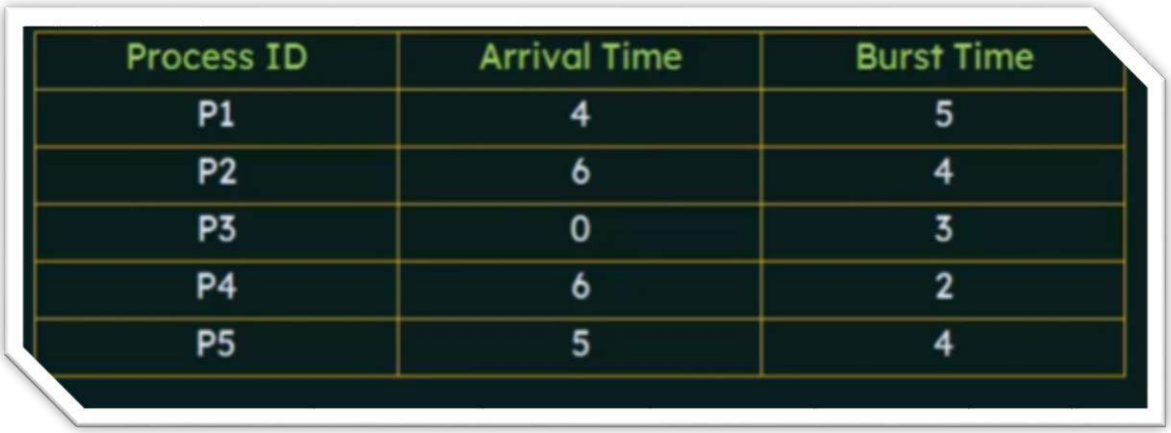
- Min waiting time
- Min response time

Batch System Scheduling

- First-come first-served
- Shortest job first
- Shortest remaining time next

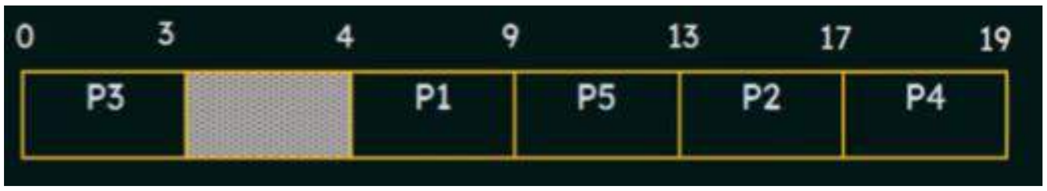
First- Come, First-Served (FCFS) Scheduling

- By far the simplest CPU-scheduling algorithm is the **first-come first-serve (FCFS)** scheduling algorithm.
- With this scheme, ***the process that requests the CPU first is allocated the CPU first.***
- Processes are scheduled in the order they arrived.
- The implementation of the FCFS policy is easily managed with a FIFO queue.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.
- FCFS scheduling algorithm is nonpreemptive
- Example:



Process ID	Arrival Time	Burst Time
P1	4	5
P2	6	4
P3	0	3
P4	6	2
P5	5	4

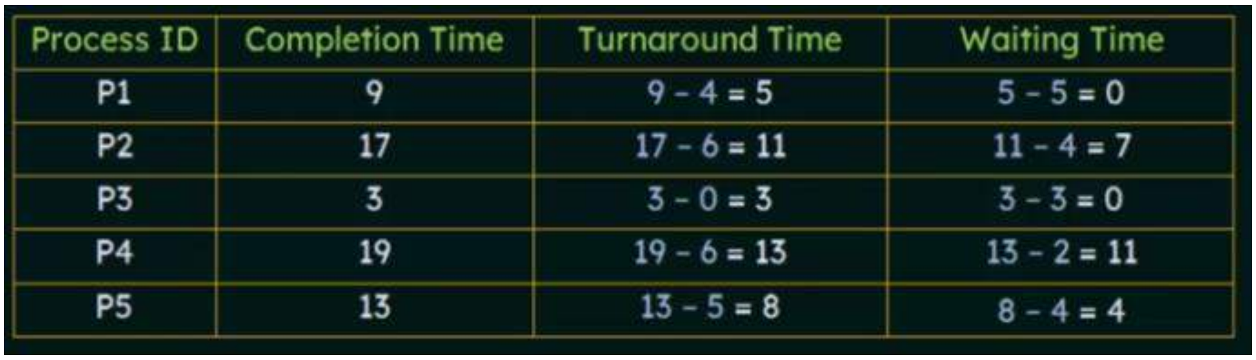
Gantt Chart for FCFS scheduling:



We know,

Turnaround Time = Completion Time – Arrival Time

Waiting Time = Turnaround Time – Burst Time



Process ID	Completion Time	Turnaround Time	Waiting Time
P1	9	9 - 4 = 5	5 - 5 = 0
P2	17	17 - 6 = 11	11 - 4 = 7
P3	3	3 - 0 = 3	3 - 3 = 0
P4	19	19 - 6 = 13	13 - 2 = 11
P5	13	13 - 5 = 8	8 - 4 = 4

- Average Turnaround Time = (5+11+3+13+8)/5 = 8 unit
- Average Turnaround Time = (0+7+0+11+4)/5 = 4.4 unit

Advantage:

- It is simple and fair.
- It can be easily implemented using queue data structure.
- It does not lead to starvation since every process gets chance to execute

Disadvantages:

- The average waiting time under the FCFS policy is often quite long.
- It does not consider the priority or burst time of the processes.
- Convoy effect is possible.
    - All small I/O bound processes wait for one big CPU bound process to acquire CPU.
    - Short process behind long process

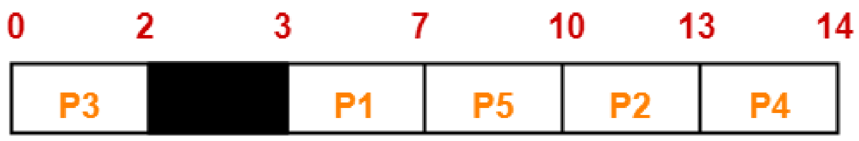
<u>Problem-01:</u>

Consider the set of 5 processes whose arrival time and burst time are given below. If the CPU scheduling policy is FCFS, calculate the average waiting time and average turn-around time.

Process	Arrival time	Burst time
P1	3	4
P2	5	3
P3	0	2
P4	5	1
P5	4	3

Solution:

Gantt Chart-



Here, black box represents the idle time of CPU.

Now, we know-

- Turnaround Time = Completion Time– Arrival Time
- Waiting time = Turnaround Time – Burst time

Process	**Completion time**	**Turnaround Time**	**Waiting Time**
P1	7	7 – 3 = 4	4 – 4 = 0
P2	13	13 – 5 = 8	8 – 3 = 5
P3	2	2 – 0 = 2	2 – 2 = 0
P4	14	14 – 5 = 9	9 – 1 = 8
P5	10	10 – 4 = 6	6 – 3 = 3

Average Turnaround Time = (4 + 8 + 2 + 9 + 6) / 5 = 29 / 5 = 5.8 unit

Average Waiting Time = (0 + 5 + 0 + 8 + 3) / 5 = 16 / 5 = 3.2 unit

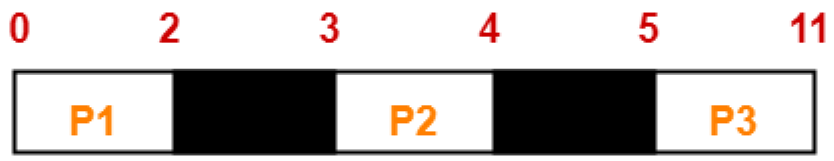
Problem-02:

Consider the set of 3 processes whose arrival time and burst time are given below. If the CPU scheduling policy is FCFS, calculate the average waiting time and average turn -around time.

Process	Arrival Time	Burst Time
P1	0	2
P2	3	1
P3	5	6

Solution:

Gantt Chart-



Here, black box represents the idle time of CPU.

Now, we know-

- Turnaround Time = Completion Time– Arrival Time
- Waiting time = Turnaround Time – Burst time

Process	Completion Time	Turnaround time	Waiting time
P1	2	2 – 0 = 2	2 – 2 = 0
P2	4	4 – 3 = 1	1 – 1 = 0
P3	11	11- 5 = 6	6 – 6 = 0

Now,

Average Turnaround time = (2 + 1 + 6) / 3 = 9 / 3 = 3 unit

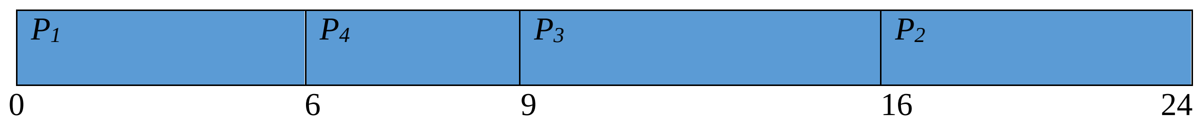
Average waiting time = (0 + 0 + 0) / 3 = 0 / 3 = 0 unit

Shortest-Job-First (SJF) Scheduling

- Need to know run times in advance
- The process, that requires shortest time to complete execution is served first
- Associate with each process the length of its next CPU burst
    - Use these lengths to schedule the process with the shortest time
- The implementation of the SJF strategy is easily managed by using a FIFO queue.
    - All processes in a queue are sorted in ascending order by their required CPU bursts
    - When CPU becomes free, a process from first position in a queue is selected to run
- SJF may be optimal – gives minimum average waiting time for a given set of processes. (Optimal only when the jobs are available simultaneously)
- Types: Preemptive and Nonpreemptive
- Example:

Processs	*Arrival Time*	*Burst Time*
P_1	0.0	6
P_2	2.0	8
P_3	4.0	7
P_4	5.0	3

- SJF (preemptive) scheduling chart:



- Turnaround Time = Completion Time– Arrival Time
- Waiting time = Turnaround Time – Burst time

Process	**Completion Time**	**Burst Time**	**Turnaround Time**	**Waiting Time**
P_1	6	6	6 - 0 = 6	6 - 6 = 0
P_2	24	8	24 - 2 = 22	22 - 8 = 14
P_3	16	7	16 - 4= 12	12 - 7 = 5
P_4	9	3	9 - 5 = 4	4 - 3 = 1

Average Turnaround time = (6 + 22 + 12 +4) / 4 = 44 / 4 = 11 unit

Average waiting time = (0 + 14 + 5 + 1) / 3 = 20 / 4 = 5 unit

Advantages:

- Reduced average waiting time compared to FCFS
- Good responses for short processes

Disadvantages:

- It is difficult to estimate the time required for a process to execute
- Starvation is possible for long processes. They may wait forever.

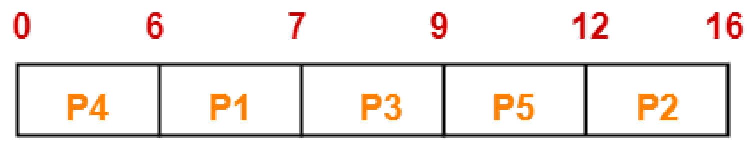
Problem-01:

Consider the set of 5 processes whose arrival time and burst time are given below. If the CPU scheduling policy is SJF non-preemptive, calculate the average waiting time and average turn - around time.

Process	**Arrival Time**	**Burst time**
P1	3	1
P2	1	4
P3	4	2
P4	0	6
P5	2	3

Solution-

Gantt Chart-



Now, we know-

- Turnaround time = Completion Time– Arrival time
- Waiting time = Turnaround time – Burst time

Process Id	**Completion Time**	**Turnaround time**	**Waiting time**
P1	7	7 – 3 = 4	4 – 1 = 3
P2	16	16 – 1 = 15	15 – 4 = 11
P3	9	9 – 4 = 5	5 – 2 = 3
P4	6	6 – 0 = 6	6 – 6 = 0
P5	12	12 – 2 = 10	10 – 3 = 7

Now,

- Average Turnaround time = (4 + 15 + 5 + 6 + 10) / 5 = 40 / 5 = 8 unit
- Average waiting time = (3 + 11 + 3 + 0 + 7) / 5 = 24 / 5 = 4.8 unit

Exercise: Schedule the given processes using SJF preemptive and find average turnaround time and average waiting time

Process ID	Burst Time	Arrival Time
P0	8	5
P1	5	0
P2	9	4
P3	2	1

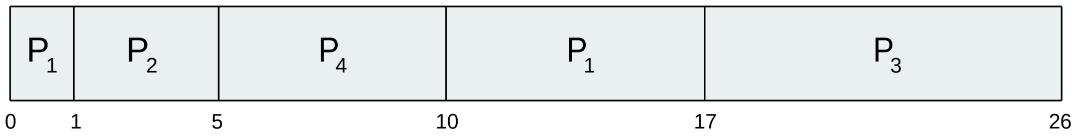
Shortest Remaining Time Next (SRTN) Scheduling

- Also called Shortest-Remaining-Time-First (SRTF) scheduling
- ***The process whose remaining running time is shortest, is served first.***
- Preemptive version of SJF scheduling**.**
- 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 (i.e. current process preempted)
- This strategy can also be implemented by using sorted FIFO queue.
  - All the processes in a queue are sorted in ascending order on their remaining run time
  - When CPU becomes free, a process from the first position of queue is selected to run.

  Example:

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- Preemptive SJF (or SRTN or SRTF) Gantt Chart



Process	**Completion Time**	**Burst Time**	**Turnaround Time**	**Waiting Time**
P_1	17	8	17 – 0 = 17	17 – 8 = 9
P_2	5	4	5 – 1 = 4	4 – 4 = 0
P_3	26	9	26 – 2 = 24	24 – 9 = 15
P_4	10	5	10 – 3= 7	7– 5 = 2

Average waiting time = (17+4+24+7)/4 = 52/4 = 13

Average waiting time = (9+0+15+2)/4 = 26/4 = 6.5

Advantages:

- Less average waiting time than SJF
- This scheme allows new short jobs to get good service.

Disadvantages:

- Requires additional computation for calculating remaining time.
- Favors short processes. Starvation is possible for long processes. Long process may wait forever.
- Context switch overhead is there.

Problem-01:

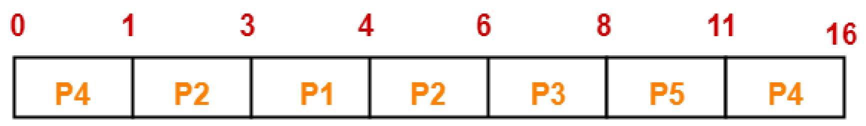
Consider the set of 5 processes whose arrival time and burst time are given below:

Process	**Arrival time**	**Burst time**
P1	3	1
P2	1	4
P3	4	2
P4	0	6
P5	2	3

If the CPU scheduling policy is SJF preemptive, calculate the average waiting time and average turn-around time.

Solution:

Gantt Chart:



Now, we know-

- Turnaround Time = Completion Time– Arrival Time
- Waiting time = Turnaround Time – Burst time

Process	**Completion Time**	**Turnaround time**	**Waiting time**
P1	4	4 – 3 = 1	1 – 1 = 0
P2	6	6 – 1 = 5	5 – 4 = 1
P3	8	8 – 4 = 4	4 – 2 = 2
P4	16	16 – 0 = 16	16 – 6 = 10
P5	11	11 – 2 = 9	9 – 3 = 6

Now,

- Average Turn Around time = (1 + 5 + 4 + 16 + 9) / 5 = 35 / 5 = 7 unit
- Average waiting time = (0 + 1 + 2 + 10 + 6) / 5 = 19 / 5 = 3.8 unit

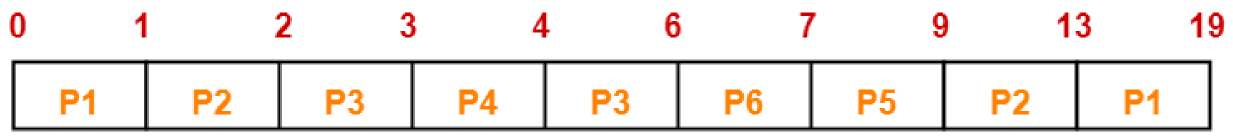
<u>Problem-02:</u>

- Consider the set of 6 processes whose arrival time and burst time are given below. If the CPU scheduling policy is shortest remaining time first, calculate the average waiting time and average turn -around time.

Process	**Arrival time**	**Burst time**
P1	0	7
P2	1	5
P3	2	3
P4	3	1
P5	4	2
P6	5	1

<u>Solution:</u>

<u>Gantt chart:</u>



Process	**Completion Time**	**Turn Around time**	**Waiting time**
P1	19	19 – 0 = 19	19 – 7 = 12
P2	13	13 – 1 = 12	12 – 5 = 7
P3	6	6 – 2 = 4	4 – 3 = 1
P4	4	4 – 3 = 1	1 – 1 = 0
P5	9	9 – 4 = 5	5 – 2 = 3
P6	7	7 – 5 = 2	2 – 1 = 1

Average Turn Around time = (19 + 12 + 4 + 1 + 5 + 2) / 6 = 43 / 6 = 7.17 unit

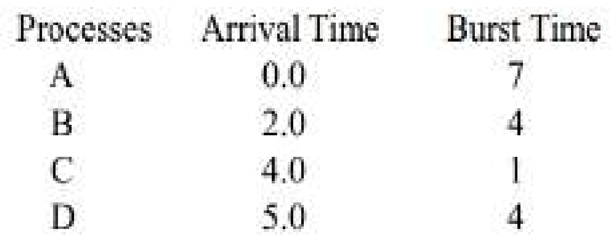
Average waiting time = (12 + 7 + 1 + 0 + 3 + 1) / 6 = 24 / 6 = 4 unit

Exercises:

1. Consider the set of 3 processes whose arrival time and burst time are given below. If the CPU scheduling policy is SRTF, calculate the average waiting time and average turn-around time. (Answer : 5, 12.33)

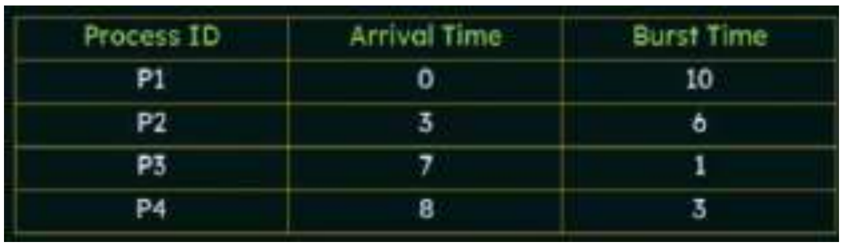
Process	**Arrival time**	**Burst time**
P1	0	9
P2	1	4
P3	2	9

2. Consider the set of following four processes whose arrival time and burst time are given in milliseconds. Now schedule the processes using SJF and SRTF and also compare the average waiting time. (Answer: SJF = 4, SRTF = 3)

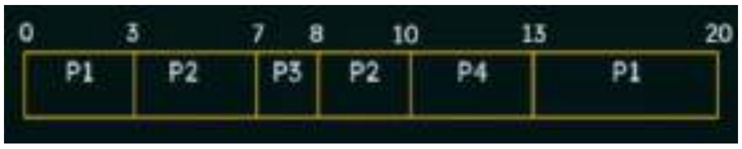


Processes	Arrival Time	Burst Time
A	0.0	7
B	2.0	4
C	4.0	1
D	5.0	4

3. Consider the set of 4 processes whose arrival time and burst time are given below. If the CPU scheduling policy is SRTF, calculate the average waiting time and average turn-around time.



Process ID	Arrival Time	Burst Time
P1	0	10
P2	3	6
P3	7	1
P4	8	3

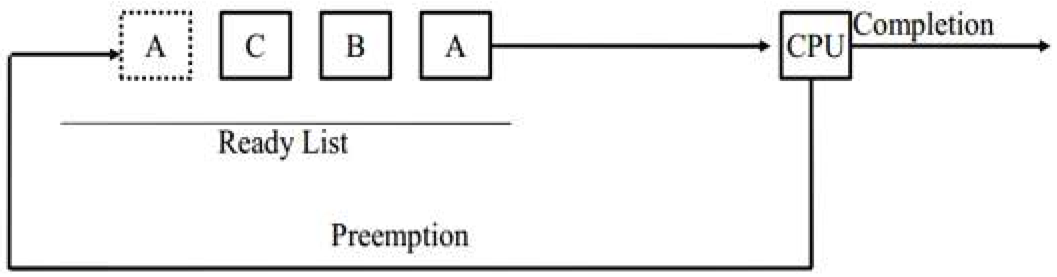
⇒ 

Scheduling in Interactive Systems

- ❖ Round robin
- ❖ Priority
- ❖ Multiple Queues

Round-Robin (RR) Scheduling

- ☑ One of the oldest, simplest, fairest, and most widely used algorithms
- ☑ Each process is assigned a time interval, called **time quantum or time slice**.
- ☑ The process is allowed to run only for this time interval (time quantum).
- ☑ Here, two scenarios are possible:
  - ☞ A process is either blocked or terminated before the quantum has elapsed. In this case CPU switching is done and another process (if any) is scheduled to run.
  - ☞ The process needs CPU burst longer than the time quantum. In this case process runs till the end of time quantum. Then it is preempted and another process runs. The preempted process is added to the end of ready queue.



- ☑ When a quantum time is over or process completes execution (whichever is earlier), it starts new process. Selection of new process is as per FCFS scheduling algorithm.
- ☑ This strategy can be implemented using FIFO queue.
  - ☞ If any process comes, or process releases CPU, or process is preempted. It is moved to the end of the queue.

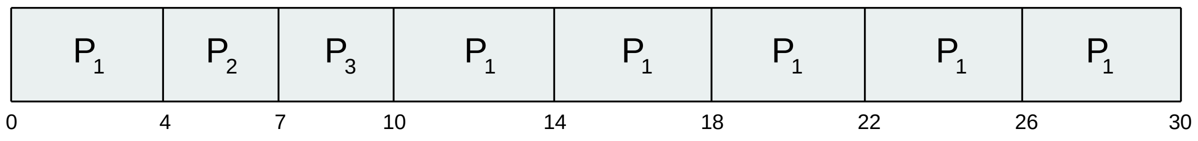
The ready queue is treated as circular queue.

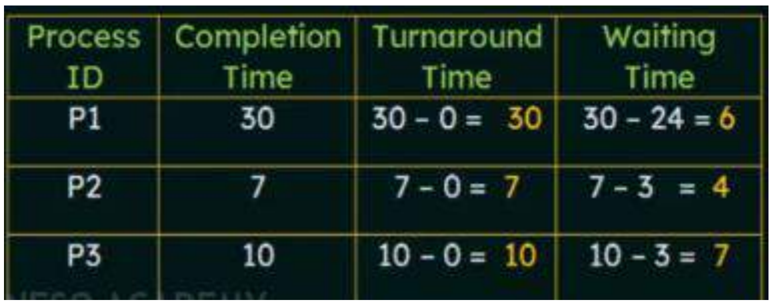
- ☑ When CPU becomes free, a process from the first position in a queue is selected to run.
- ☑ If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.

Example: RR Scheduling with quantum = 4 (Arrival time is 0 for all processes)

Process	Burst Time
P_1	24
P_2	3
P_3	3

The Gantt chart is:





Process ID	Completion Time	Turnaround Time	Waiting Time
P1	30	30 - 0 = 30	30 - 24 = 6
P2	7	7 - 0 = 7	7 - 3 = 4
P3	10	10 - 0 = 10	10 - 3 = 7

- Average Turnaround time = (30 + 7 + 10) / 3 = 47 / 3 = 15.66 unit
- Average waiting time = (6+4+7) / 3 = 17/ 3 = 5.66 unit

Advantages:

- ☞ It is simple, fare and most widely used algorithm.
- ☞ Don’t need to know run times in advance
- ☞ It gives the best performance in terms of average response time.
- ☞ It is best suited for time sharing system, client server architecture and interactive system.

Disadvantages:

- There is the overhead of context switch.
- It's performance heavily depends on time quantum. Selection of time quantum is crucial.
    - If time quantum is too short, it causes frequent context switches and lowers CPU efficiency.
    - If it is too long, it causes poor response for short interactive processes.
- Priorities cannot be set for the processes.

NOTES:

- ☑ 80% of CPU bursts should be shorter than quantum time
- ☑ A quantum around 20–50 msec is often a reasonable compromise

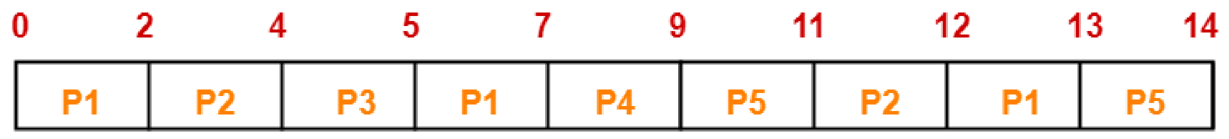
Problem-01:

Consider the set of 5 processes whose arrival time and burst time are given below. If the CPU scheduling policy is Round Robin with time quantum = 2 unit, calculate the average waiting time and average turn-around time.

Process Id	Arrival time	Burst time
P1	0	5
P2	1	3
P3	2	1
P4	3	2
P5	4	3

Solution-

Gantt Chart-



Process	**Completion Time**	**Turn Around time**	**Waiting time**
P1	13	13 – 0 = 13	13 – 5 = 8
P2	12	12 – 1 = 11	11 – 3 = 8
P3	5	5 – 2 = 3	3 – 1 = 2
P4	9	9 – 3 = 6	6 – 2 = 4
P5	14	14 – 4 = 10	10 – 3 = 7

- Average Turnaround time = (13 + 11 + 3 + 6 + 10) / 5 = 43 / 5 = 8.6 unit
- Average waiting time = (8 + 8 + 2 + 4 + 7) / 5 = 29 / 5 = 5.8 unit

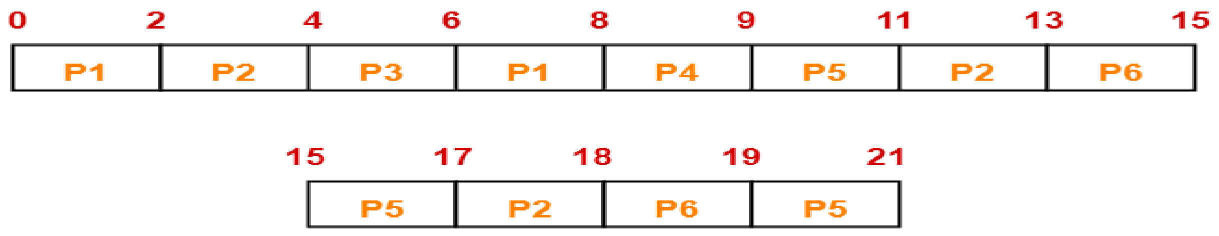
Problem-02:

Consider the set of 6 processes whose arrival time and burst time are given below. If the CPU scheduling policy is Round Robin with time quantum = 2, calculate the average waiting time and average turn-around time.

Process	Arrival time	Burst time
P1	0	4
P2	1	5
P3	2	2
P4	3	1
P5	4	6
P6	6	3

Solution-

Gantt chart-



Process	**Completion Time**	**Turnaround time**	**Waiting time**
P1	8	8 – 0 = 8	8 – 4 = 4
P2	18	18 – 1 = 17	17 – 5 = 12
P3	6	6 – 2 = 4	4 – 2 = 2
P4	9	9 – 3 = 6	6 – 1 = 5
P5	21	21 – 4 = 17	17 – 6 = 11
P6	19	19 – 6 = 13	13 – 3 = 10

- Average Turn Around time = (8 + 17 + 4 + 6 + 17 + 13) / 6 = 65 / 6 = 10.84 unit
- Average waiting time = (4 + 12 + 2 + 5 + 11 + 10) / 6 = 44 / 6 = 7.33 unit

Problem-03:

Consider the set of 4 processes whose arrival time and burst time are given below. If the CPU scheduling policy is Round Robin with time quantum = 4, calculate the average waiting time and average turn-around time.

Process	Arrival time	Burst time
P0	0	10
P1	1	6
P2	3	2
P3	5	4

Exercise:

Consider the set of 6 processes whose arrival time and burst time are given below. If the CPU scheduling policy is Round Robin with time quantum = 3, calculate the average waiting time and average turn-around time. (Answer: 16, 21.33)

Process	**Arrival time**	**Burst time**
P1	5	5
P2	4	6
P3	3	7
P4	1	9
P5	2	2
P6	6	3

.

Priority Scheduling

Even on a PC with a single owner, there may be multiple processes, some of them more important than others. For example, a daemon process sending electronic mail in the background should be assigned a lower priority than a process displaying a video film on the screen in real time.

IDEA: Each process is assigned a priority, and the runnable process with the highest priority is allowed to run.

- A priority is associated with each process, and the CPU is allocated to the process with the highest priority
- Priorities can be assigned to processes statically or dynamically
- Priorities are generally indicated by some fixed range of numbers, such as 0 to 10 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority. **Some systems use low numbers to represent low priority; others use low numbers for high priority.**
- Equal-priority processes are scheduled in FCFS order.
- Round-robin scheduling makes the implicit assumption that all processes are equally important
- An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.
- Two types:
    - Preemptive
        - A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
    - Nonpreemptive
        - A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

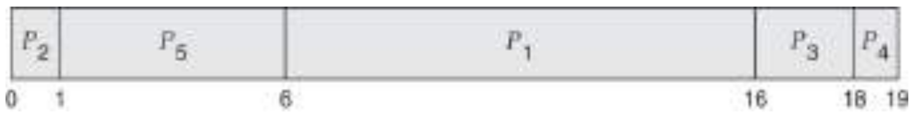
Nonpreemptive Priority Scheduling

- The process, that has highest priority, is served first.
- Once a process is selected, it runs until it blocks for I/O or some event or it terminates
- This strategy can be implemented by using sorted FIFO queue.
    - All process in a queue are sorted based on their priority with highest priority process at the front of the queue.
    - When CPU becomes free, a process from the first position in a queue is selected to run.

Example: (*lower integer represents higher priority and, arrival time of all process is 0 and order of arrival is* P_1, P_2, P_3, P_4, P_5)

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



- Average waiting time = 8.2

Advantages:

- Priority is considered, so critical processes can get better response

Disadvantages:

- Starvation – low priority processes may never execute

NOTE: This problem can be solve by using the technique, aging.

- ☑ Aging– gradually increase the priority of processes that wait in the system for a long time.
- ☑ For example, if priorities range from 127 (low) to 0 (high), we could periodically (say, every second) increase the priority of a waiting process by 1. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed. In fact, it would take a little over 2 minutes for a priority-127 process to age to a priority-0 process.

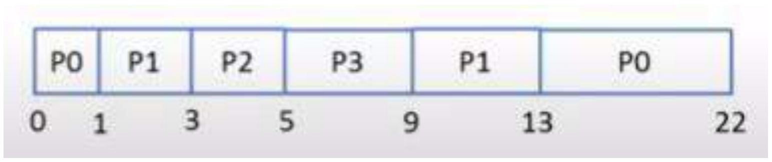
Preemptive Priority Scheduling

- The process, that has highest priority, is served first
- When a new process arrives, it's priority is compared with priority of currently running process.
    - If the new process has higher priority than the current process, the currently running process is suspended and new process is allowed to run.
- This strategy can be implemented by using sorted FIFO queue.
    - All process in a queue are sorted based on their priority with highest priority process at the front of the queue.
    - When CPU becomes free, a process from the first position in a queue is selected to run.

Example: (*lower integer represents higher priority*)

Process	Arrival Time	Burst Time	Priority
P0	0	10	5
P1	1	6	4
P2	3	2	2
P3	5	4	0

- Priority scheduling Gantt Chart



We know,

Turnaround Time = Exit – Arrival Time

Waiting Time = Trunaround Time – Burst Time

Process	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
P0	0	10	22	22 – 0 = 22	22– 10 = 12
P1	1	6	13	13 –1 = 12	12– 6 = 6
P2	3	2	5	5 – 3 = 2	2– 2 = 0
P3	5	4	9	9 –5 = 4	4– 4 = 0

- Average Turnaround time = (22+ 12 + 2 + 4) / 4 = 40 / 4 = 10 unit
- Average waiting time = (12 + 6 + 0 + 0) / 4 = 18 / 4 = 4.5 unit

Advantages:

- Priority is considered, so critical processes can get better response

Disadvantages:

- Starvation – low priority processes may never execute (solution : aging)
- Context switching overhead is there.

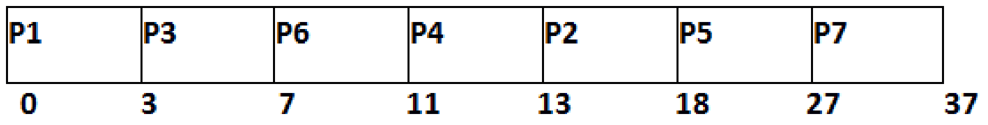
Problem 01:

Consider the following 7 processes P1, P2, P3, P4, P5, P6 and P7. Schedule the process using nonpreemptive priority scheduling algorithm and compute the average turn-around time and average waiting time. (Lower integer represents higher priority)

Process ID	Priority	Arrival Time	Burst Time
P1	2	0	3
P2	6	2	5
P3	3	1	4
P4	5	4	2
P5	7	6	9
P6	4	5	4
P7	10	7	10

Solution:

Gantt chart:



Process Id	Priority	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
P1	2	0	3	3	3 – 0 = 3	3 – 3 = 0
P2	6	2	5	18	18 – 2 = 16	16 – 5 = 11
P3	3	1	4	7	7 – 1 = 6	6 – 4 = 2
P4	5	4	2	13	13 – 4 = 9	9 – 2 = 7
P5	7	6	9	27	27– 6 = 21	21 – 9 = 12
P6	4	5	4	11	11 – 5 = 6	6– 4 = 2
P7	10	7	10	37	37 – 7 = 30	30 – 10 =20

- Average Turnaround time = (3+ 16 + 6 + 9 + 21+ 6 + 30) / 7 = 91 / 7 = 13 unit
- Average waiting time = (0 + 11 + 2 + 7 + 12+ 2 + 20) / 7 = 54 / 7 = 7.714 unit

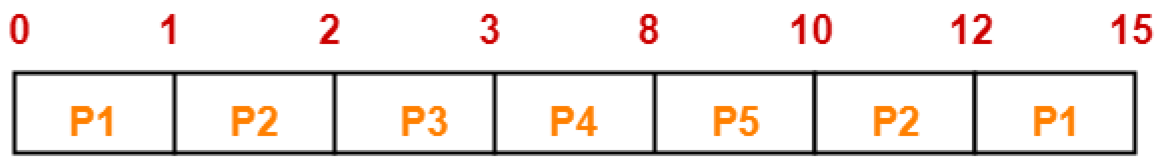
Problem-02:

Consider the set of 5 processes whose arrival time and burst time are given below. If the CPU scheduling policy is priority preemptive, calculate the average waiting time and average turn-around time. (Higher number represents higher priority)

Process Id	Arrival time	Burst time	Priority
P1	0	4	2
P2	1	3	3
P3	2	1	4
P4	3	5	5
P5	4	2	5

Solution:

<u>**Gantt Chart-**</u>



Process Id	**Completion Time**	**Turnaround time**	**Waiting time**
P1	15	15 – 0 = 15	15 – 4 = 11
P2	12	12 – 1 = 11	11 – 3 = 8
P3	3	3 – 2 = 1	1 – 1 = 0
P4	8	8 – 3 = 5	5 – 5 = 0
P5	10	10 – 4 = 6	6 – 2 = 4

Now,

- Average Turn Around time = (15 + 11 + 1 + 5 + 6) / 5 = 38 / 5 = 7.6 unit
- Average waiting time = (11 + 8 + 0 + 0 + 4) / 5 = 23 / 5 = 4.6 unit

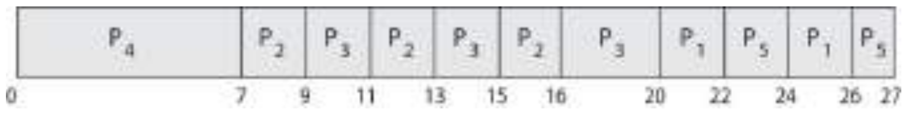
Another Example:

Priority Scheduling with Round-Robin

(Smaller value of priority represents higher priority arrival time of all process is 0)

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

- Run the process with the highest priority. Processes with the same priority run round-robin
- Gantt Chart with time quantum = 2



Multilevel Queue Scheduling

⇒ With both priority and round-robin scheduling, all processes may be placed in a single queue, and the scheduler then selects the process with the highest priority to run.

⇒ A multilevel Queue Scheduling algorithm partitions the ready queue into several separate queues.

- ☞ The processes are permanently assigned to one queue, generally based on some property of the process, such as memory space, process priority, or process type.
- ☞ Each queue has its own scheduling algorithm.

Example: Separate queues might be used for foreground and background processes

- ▪ The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm

Overview of Real Time System Scheduling

- A real-time system is one in which time plays an essential role.
- Real-time systems are generally categorized as **hard real time**, meaning there are absolute deadlines that must be met—or else!— and **soft real time**, meaning that missing an occasional deadline is undesirable, but nevertheless tolerable.
  - Hard Real Time System: robot control in a factory
  - Soft Real Time System: CD player
- The events that a real-time system may have to respond to can be further categorized as periodic (meaning they occur at regular intervals) or aperiodic (meaning they occur unpredictably).
- Real-time scheduling algorithms can be **static or dynamic**. The former (static) make their scheduling decisions before the system starts running. The latter (dynamic) make their scheduling decisions at run time, after execution has started.
  - — Static scheduling works only when there is perfect information available in advance about the work to be done and the deadlines that have to be met.
  - — Dynamic scheduling algorithms do not have these restrictions.