MODULE-II

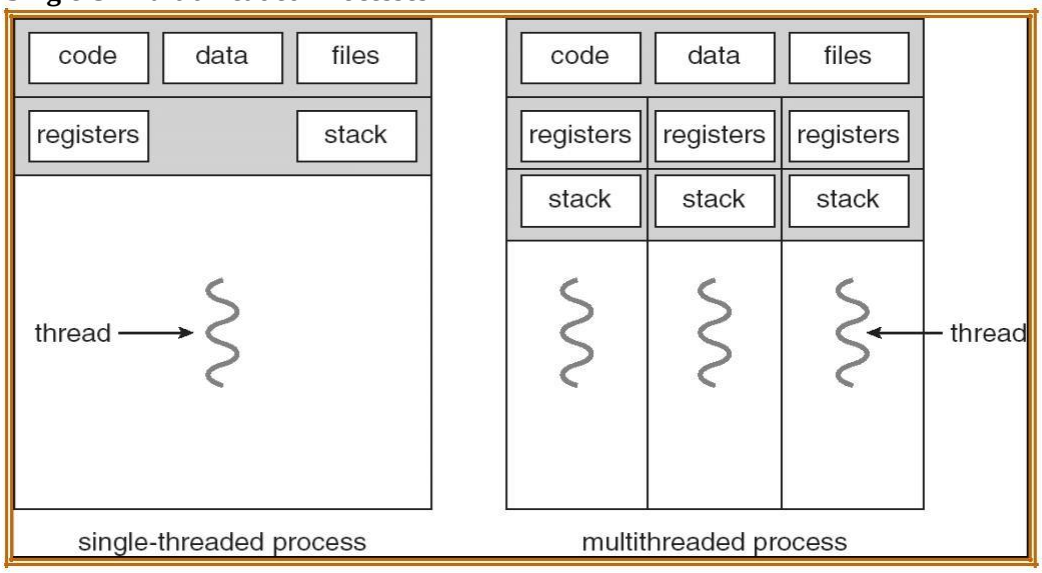
2.1 THREADS

2.1.1 Overview

2.1.2 Multicore Programming

2.1.3 Multithreading Models

Single & Multithreaded Processes



Benefits

- Responsiveness
- Resource Sharing
- Economy
- Utilization of MP Architectures

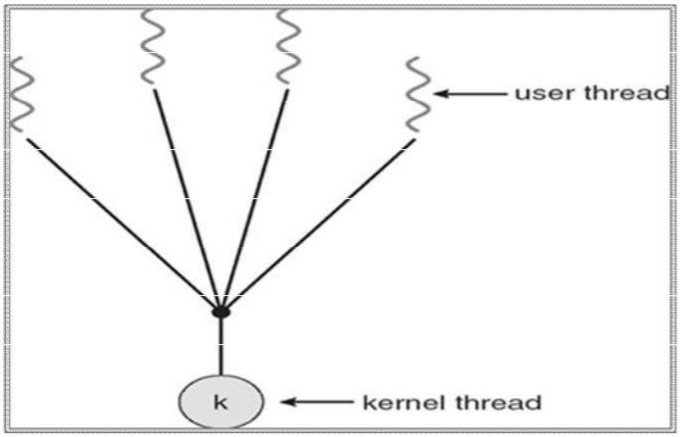
User Threads

- Thread management done by user-level threads library
- Three primary thread libraries:
    - POSIX Pthreads
    - Win32 threads
    - Java threads

Multithreading Models

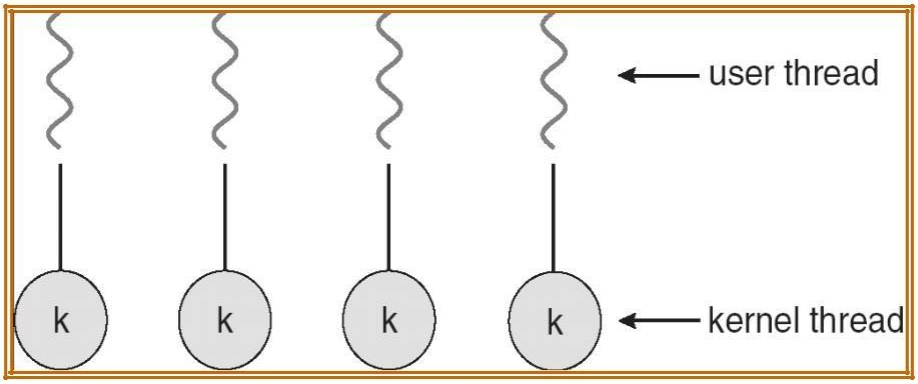
- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One



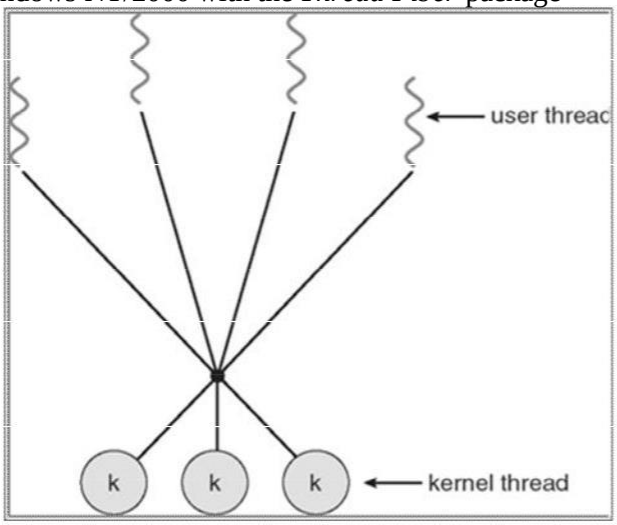
Many-to-One Model

One-to-One



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *Thread Fiber* package



Many-to-Many Model

2.1.4 Thread Libraries

2.1.5 Implicit Threading 2.1.6 Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data
- Scheduler activations

Thread Cancellation

- Terminating a thread before it has finished
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

Windows XP Threads

- Implements the one-to-one mapping
  - Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
  - The register set, stacks, and private storage area are known as the **context** of the threads
- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

Linux Threads

- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through clone() system call
- clone() allows a child task to share the address space of the parent task (process)

Java Threads

- Java threads are managed by the JVM
  - Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

2.2 Process Synchronization

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

Producer

```
while (true) {

        /* produce an item and put in next Produced */
                while (count == BUFFER_SIZE)
                        ; // do nothing
                buffer [in] = next Produced;
                in = (in + 1) % BUFFER_SIZE;
                count++;
}
```

Consumer

```
while (true) {
                while (count == 0)
                        ; // do nothing
                        next Consumed = buffer[out];
                        out = (out + 1) % BUFFER_SIZE;
                count--;
                        /* consume the item in next Consumed
        }
```

2.2.1 Critical section problem:- A section of code which reads or writes shared data.

Race Condition

- The situation where two or more processes try to access and manipulate the same data and output of the process depends on the orderly execution of those processes is called as Race Condition.
- count++ could be implemented as
  register1 = count
  register1 = register1 + 1
  count = register1
- count-- could be implemented
  as register2 = count
  register2 = register2 -
  1 count = register2
- Consider this execution interleaving with "count = 5" initially:
  - S0: producer execute register1 = count {register1 = 5}
  - S1: producer execute register1 = register1 + 1 {register1 = 6}
  - S2: consumer execute register2 = count {register2 = 5}

- S3: consumer execute register2 = register2 - 1 {register2 = 4}
- S4: producer execute count = register1 {count = 6 }
- S5: consumer execute count = register2 {count = 4}

Requirements for the Solution to Critical-Section Problem

1. **Mutual Exclusion**: - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress:** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. **Bounded Waiting:** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

- To general approaches are used to handle critical sections in operating systems: (1) Preemptive Kernel (2) Non Preemptive Kernel
  - Preemptive Kernel allows a process to be preempted while it is running in kernel mode.
  - Non Preemptive Kernel does not allow a process running in kernel mode to be preempted. (these are free from race conditions)

2.2.2 Peterson's Solution

- It is restricted to two processes that alternates the execution between their critical and remainder sections.
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process Pi is ready!

Note:- Peterson's Solution is a software based solution.

Algorithm for Process P_i

```
while (true) {
  flag[i] = TRUE;
  turn = j;
  while ( flag[j] && turn == j);
      CRITICAL SECTION
  flag[i] = FALSE;
       REMAINDER SECTION
}
```

Fig: structure of process P_i in Peterson's solution

Solution to Critical Section Problem using Locks.

```
do{
        acquire lock

            Critical Section
        release lock
```

Remainder Section

```
}while (True);
```

Note:- Race Conditions are prevented by protecting the critical region by the locks.

2.2.3 Synchronization Hardware

- In general we can provide any solution to critical section problem by using a simple tool called as LOCK where we can prevent the race condition.
- Many systems provide hardware support (hardware instructions available on several systems) for critical section code.
- In UniProcessor hardware environment by disabling interrupts we can solve the critical section problem. So that Currently running code would execute without any preemption .
- But by disabling interrupts on multiprocessor systems is time taking so that it is inefficient compared to UniProcessor system.
- Now a days Modern machines provide special atomic hardware instructions that allow us to either *test memory word and set value* Or *swap contents of two memory words automatically* i.e. done through an uninterruptible unit.

Special Atomic hardware Instructions

- TestAndSet()
- Swap()
- The TestAndSet() Instruction is one kind of special atomic hardware instruction that allow us to test memory and set the value. We can provide Mutual Exclusion by using TestAndSet() instruction.
- Definition:

```
Boolean TestAndSet (Boolean *target)
{
    Boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

- Mutual Exclusion Implementation with TestAndSet()
- To implement Mutual Exclusion using TestAndSet() we need to declare Shared Boolean variable called as ‘lock’ (initialized to false) .
- Solution:

```
while (true) {
        while ( TestAndSet (&lock ))
                ;   /* do nothing
              //    critical section
        lock = FALSE;
              //      remainder section
}
```

- The Swap() Instruction is another kind of special atomic hardware instruction that allow us to swap the contents of two memory words.
- By using Swap() Instruction we can provide Mutual Exclusion.
- Definition:-

```
void Swap (Boolean *a, Boolean *b)
{
      Boolean temp = *a;
```

```
        *a = *b;
        *b = temp:
    }
```

- Shared Boolean variable called as 'lock' is to be declared to implement Mutual Exclusion in Swap() also, which is initialized to FALSE.

Solution:

```
while (true) {
        key = TRUE;
        while ( key == TRUE)
              Swap (&lock, &key );

                // critical section
         lock = FALSE;
                //      remainder section
    }
```

Note:- Each process has a local Boolean variable called as 'key'.

2.2.4 Mutex Locks

2.2.5 Semaphores

- As it is difficult for the application programmer to use these hardware instructions, to overcome this difficulty we use the synchronization tool called as Semaphore (that does not require busy waiting)
- Semaphore *S* _ integer variable, apart from this initialization we can access this only through two standard atomic operations called as wait() and signal().
- Originally the wait() and signal() operations are termed as P() and V() respectively. Which are termed from the Dutch words "proberen" and "verhogen".
- *The definition for wait() is as follows*:

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

- *The definition for signal() is as follows*:

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

- All the modifications to the integer value of the semaphore in the wait() and signal() atomic operations must be executed indivisibly. i.e. when one process changes the semaphore value, no other process will change the same semaphore value simultaneously.
- *Usage of semaphore*:- we have two types of semaphores
  - _ Counting semaphore
  - _ Binary Semaphore.
- The value of the Counting Semaphore can ranges over an unrestricted domain.
- The value of the Binary Semaphore can ranges between 0 and 1 only.
- In some systems the Binary Semaphore is called as Mutex locks, because, as they are locks to provide the mutual exclusion.
- We can use the Binary Semaphore to deal with critical section problem for multiple processes.
- Counting Semaphores are used to control the access of given resource each of which consists of some finite no. of instances. This counting semaphore is initialized to number of resources available.

- The process that wish to use a resource must performs the wait() operation (count is decremented)
- The process that releases a resource must performs the signal() operation (count is incremented)
- When the count for the semaphore is 0 means that all the resources are being used by some processes. Otherwise resources are available for the processes to allocate .
- When a process is currently using a resource means that it blocks the resource until the count becomes > 0.
- For example:
  - Let us assume that there are two processes p_0 and p_1 which consists of two statements s_0 & s_1 respectively.
  - Also assume that these two processes are running concurrently such that process p_1 executes the statement s_1 only after process p_0 executes the statement s_0.
  - Let us assume the process p_0 & p_1 share the same semaphore called as "synch" which is initialized to 0 by inserting the statements

```
S0;
Signal (synch);
```

in process p_0 and the statements wait

```
(synch); S1;
```

in process p_1 .

Implementation:

- The main disadvantage of the semaphore definition is, it requires the busy waiting.
- Because when one process is in critical section and if another process needs to enter in to the critical section must have to loop in the entry code continuously.

- To overcome the need of the busy waiting we have to modify the definition of wait() and signal() operations. i.e. when a process executes wait() operation and finds that it is not positive then it must wait.
- Instead of engaging the busy wait, the process block itself so that there will be a chance to the CPU to select another process for execution. It is done by block() operation.
  - Blocked processes are placed in waiting queue.
- Later the process that has already been blocked by itself is restarted by using wakeup() operation, so that the process will move from waiting state to ready state.
  - Blocked processes that are placed in waiting queue are now placed into ready queue.
- To implement the semaphore with no busy waiting we need to define the semaphore of the wait() and signal() operation by using the 'C' Struct. Which is as follows: typedef

```
struct {
        int value;
        struct process *list;
}semaphore;
```

  - i.e. each semaphore has an integer value stored in the variable "value" and the list of processes list.
  - When a process perform the wait() operation on the semaphore then it will adds list of processes to the list .
  - When a process perform the signal() operation on the semaphore then it removes the processes from the list.

Semaphore Implementation with no Busy waiting

Implementation of wait: (definition of wait with no busy waiting) wait

```
(S){
        value--;
        if (value < 0) {
```

add this process to waiting queue
block(); }

}

Implementation of signal: (definition of signal with no busy waiting)

```
Signal (S){
        value++;
        if (value <= 0) {
            remove a process P from the waiting queue
            wakeup(P); }
}
```

Deadlock and Starvation

- The implementation of semaphore with waiting queue may result in the situation where two or more processes are waiting for an event is called as Deadlocked.
- To illustrate this, let us assume two processes P_0 and P_1 each accessing two semaphores S and Q which are initialized to 1 :-

```
P0                  P1
  wait (S);                  wait (Q);
   wait (Q);                 wait (S);
.                   .
.                   .
.                   .
      signal (S); signal (Q); signal (Q); signal (S);
```

- Now process P_0 executes *wait(S)* and P_1 executes *wait(Q)*, assume that P_0 wants to execute *wait(Q)* and P_1 executes *wait(S)*. But it is possible only after process P_1 executes the *signal(Q)* and P_0 executes *signal(S)*.
- Starvation or indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

2.2.6 Classical Problems of Synchronization

- **Bounded-Buffer Problem**
- **Readers and Writers Problem**
- **Dining-Philosophers Problem**

Bounded-Buffer Problem

- Let us assume *N* buffers, each can hold only one item.
- Semaphore mutex initialized to the value 1 which is used to provide mutual exclusion.
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.
- Semaphore full and empty are used to count the number of buffers.
- The structure of the producer process

```
while (true) {
        //  produce an
    item wait (empty);
    wait (mutex);
        //  add the item to the buffer
     signal (mutex);
     signal (full);
}
```

The structure of the consumer process

```
while (true) {
     wait (full);
     wait (mutex);
```

```
            // remove an item from buffer
        signal (mutex);
        signal (empty);

            // consume the removed item
    }
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set, do not perform any updates
  - Writers – can both read and write the data set (perform the updates).
- If two readers read the shared data simultaneously, there will be no problem. If both a reader(s) and writer share the same data simultaneously then there will be a problem.
- In the solution of reader-writer problem, the reader process share the following data structures: Semaphore Mutex, wrt;
int readcount;
- Where → Semaphore mutex is initialized to 1.

→ Semaphore wrt is initialized to 1.
→ Integer readcount is initialized to 0.

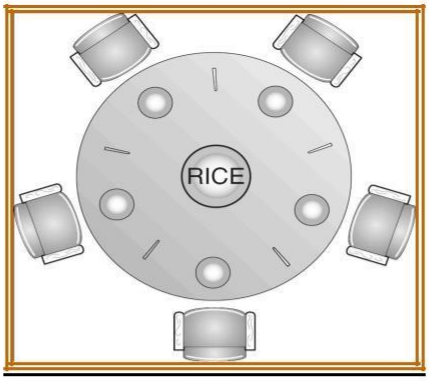
The structure of a writer process

```
while (true) {
        wait (wrt) ;

            //    writing is
        performed signal (wrt) ;
}
```

The structure of a reader process

```
while (true) {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1) wait (wrt) ;
        signal (mutex)

            // reading is performed
        wait (mutex) ; readcount -
        - ;
        if (readcount == 0) signal (wrt) ;
        signal (mutex) ;
}
```

Dining-Philosophers Problem



- **Shared data**
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem

- **The structure of Philosopher *i*:**

```
While (true) {
    wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );

            // eat
        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

       // think
}
```

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex) → Case 1
  - wait (mutex) … wait (mutex) → Case 2
  - Omitting of wait (mutex) or signal (mutex) (or both) → Case 3
- As the semaphores used incorrectly as above may results the timing errors.
- Case 1 → Several processes may execute in critical section by violating the mutual exclusion requirement.
- Case 2 → Dead lock will occur.
- Case 3 → either mutual exclusion is violated or dead lock will occur
- To deal with such type of errors, researchers have developed high-level language constructs.
- One type of high-level language constructs that is to be used to deal with the above type of errors is → the *Monitor* type.

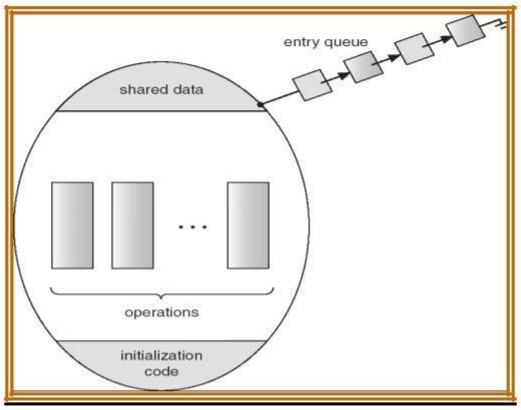
2.2.7 Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization.
- A procedure can access only those variables that are declared in a monitor and formal parameters .
- Only one process may be active within the monitor at a time

Syntax of the monitor :-

```
monitor monitor-name
{
        // shared variable declarations
        procedure P1 (…) { …. }
                …
        procedure Pn (…) {……}
    Initialization code ( ….) { … }
                …
        }
```

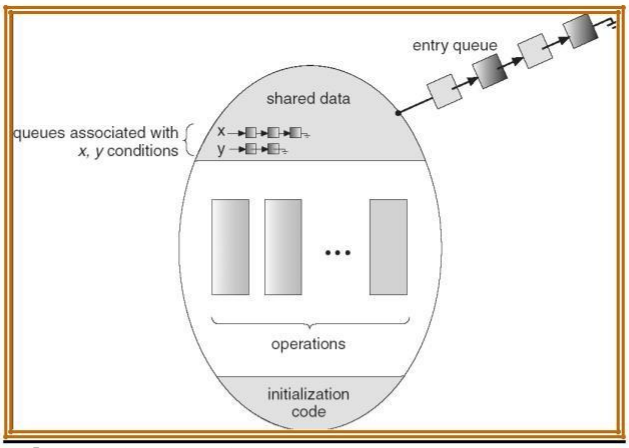
Schematic view of a Monitor



Condition Variables

- Synchronization scheme is not effective within the monitors.
- A programmer who needs to write the synchronization scheme can define one or more variables of type *Condition*
- condition x, y;
- The only operations that can be invoked on a condition variable are wait() and signal().
- The operations are
  - x.wait () – a process that request an operation is suspended until another process invokes x.signal ()
  - x.signal () – resumes only one suspended processes (if any) that invoked x.wait ()
- Now suppose that when x.signal() operation is invoked by a process P, there i s a suspended process Q associated with condition x. if Q is allowed to resume its execution, the signaling process P must wait. Else both P and Q would be active simultaneously with in a monitor.
- There are two possibilities
  - 1) signal and wait:- P either waits until Q leaves the monitor or waits for another condition.
  - 2) signal and continue:- Q either waits Until P leaves the monitor or waits for another condition.

Monitor with Condition Variables



Solution to Dining Philosophers using Monitors

```
monitor DP
  {
        enum { THINKING, HUNGRY, EATING} state [5] ;
        condition self [5];
        void pickup (int i) {
              state[i] = HUNGRY;
              test(i);
              if (state[i] != EATING) self [i].wait;
        }
              void putdown (int i) {
              state[i] = THINKING;
             // test left and right
               neighbors test((i + 4) % 5);
               test((i + 1) % 5);
      }
void test (int i) {
               if ( (state[(i + 4) % 5] != EATING) &&
               (state[i] == HUNGRY) &&
               (state[(i + 1) % 5] != EATING) ) {
                   state[i] = EATING ;
                      self[i].signal () ;
                }
          }
     initialization_code() {
              for (int i = 0; i < 5; i++)
              state[i] = THINKING;
        }
}
```

Each philosopher *i* invokes the operations pickup()
and putdown() in the following sequence:

```
dp.pickup (i)
    EAT
dp.putdown (i)
```

Monitor Implementation Using Semaphores

Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;     // (initially  = 0)
```

```
int next-count = 0;
```

Each procedure *F* will be replaced by

```
wait(mutex);
    …
        body of F;
    …
if (next-count > 0)
        signal(next)
else
        signal(mutex);
```

Mutual exclusion within a monitor is ensured.

Monitor Implementation

For each condition variable *x*, we have:

```
semaphore x-sem; // (initially = 0)
int x-count = 0;
```

The operation x.wait can be implemented as:

```
x-count++;
if (next-count > 0)
        signal(next);
else
        signal(mutex);
wait(x-sem);
x-count--;
```

The operation x.signal can be implemented as:

```
if (x-count > 0) {
        next-count++;
        signal(x-sem);
        wait(next);
        next-count--;
}
```

2.2.8 Synchronization Examples

- **Windows XP**
- **Linux**

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
  - An event acts much like a condition variable

Linux Synchronization

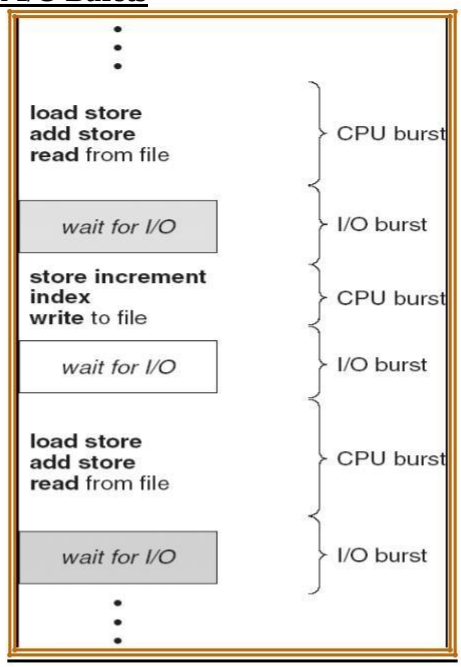
- Linux:
  - disables interrupts to implement short critical sections
- Linux provides:
  - semaphores
  - spin locks

2.2.9 Alternative approaches

2.3 CPU Scheduling

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait
- CPU burst distribution

Alternating Sequence of CPU & I/O Bursts



CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state
2. Switches from running to ready state
3. Switches from waiting to ready
4. Terminates

- Scheduling under 1 and 4 is *nonpreemptive*
- All other scheduling is *preemptive*

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- *Dispatch latency* – time it takes for the dispatcher to stop one process and start another running

2.3.1 Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – No. 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, not output (for time-sharing environment)

Optimization Criteria

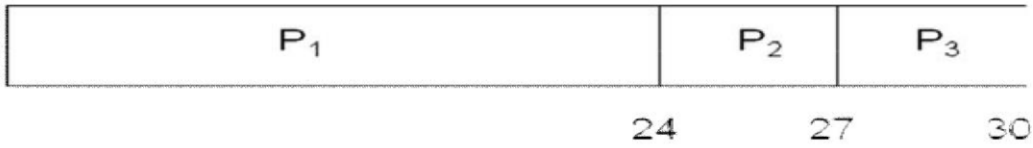
- Max CPU utilization
- Max throughput

- Min turnaround time
- Min waiting time
- Min response time

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

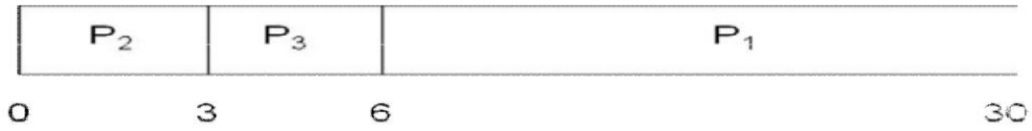
Suppose that the processes arrive in the order: P_1 , P_2 , P_3
The Gantt Chart for the schedule is:



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

- Suppose that the processes arrive in the order

- The Gantt chart for the schedule is:



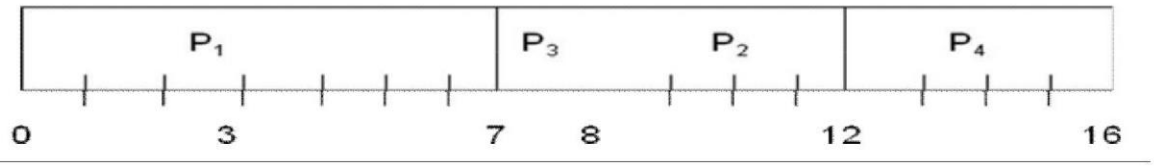
- Waiting time for P_1 = 6; P_2 = 0; P_3 = 3
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- *Convoy effect* short process behind long process

Shortest-Job-First (SJF) Scheduling

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

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

SJF (non-preemptive)

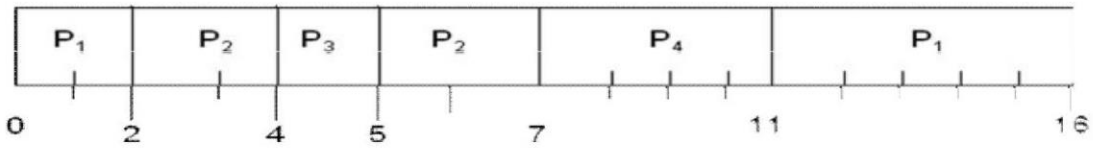


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

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

- **SJF (preemptive)**



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

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Non preemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ Starvation – low priority processes may never execute
- Solution ≡ Aging - as time progresses increase the priority of the process (means Aging increases the priority of the processes so that to terminate in finite amount of time).

Round Robin (RR)

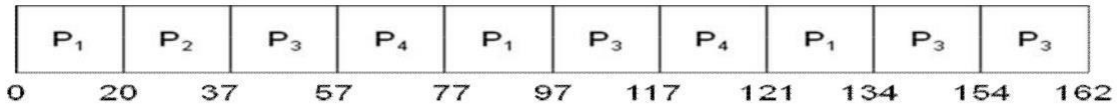
- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time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

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68

P_4 24

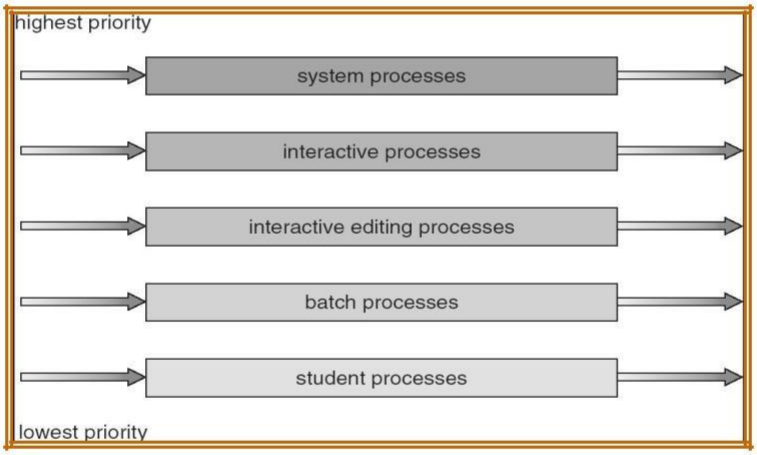
The Gantt chart is:



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

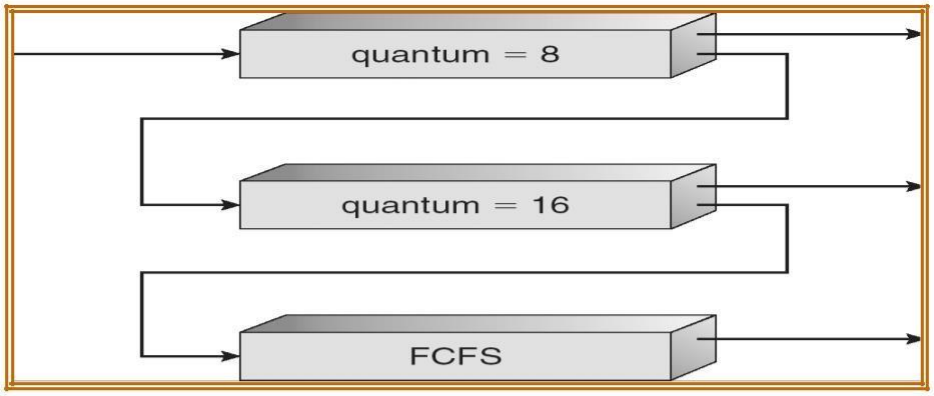
Multilevel Queue Scheduling

- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS



Multilevel Feedback Queue Scheduling

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service



2.3.2 Scheduling Algorithms

2.3.3 Thread Scheduling

- Local Scheduling – How the threads library decides which thread to put onto an available LWP
- Global Scheduling – How the kernel decides which kernel thread to run next

2.3.4 Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- *Homogeneous processors* within a multiprocessor
- *Load sharing*
- *Asymmetric multiprocessing* – only one processor accesses the system data structures, alleviating the need for data sharing

Operating System Examples

Windows XP Priorities

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Linux Scheduling

- Two algorithms: time-sharing and real-time
- Time-sharing
    - Prioritized credit-based – process with most credits is scheduled next
    - Credit subtracted when timer interrupt occurs
    - When credit = 0, another process chosen
    - When all processes have credit = 0, recrediting occurs
        - Based on factors including priority and history
- Real-time
    - Soft real-time
    - Posix.1b compliant – two classes
        - FCFS and RR

- Highest priority process always runs first

Java Thread Scheduling

- JVM Uses a Preemptive, Priority-Based Scheduling Algorithm
- FIFO Queue is Used if There Are Multiple Threads With the Same

Priority JVM Schedules a Thread to Run When:

1. The Currently Running Thread Exits the Runnable State
2. A Higher Priority Thread Enters the Runnable State

* Note _ the JVM Does Not Specify Whether Threads are Time-Sliced or Not