



# Operating Systems with Linux

## (MCA-105)

### Unit - 2

by

**Dr. Sunil Pratap Singh**

(Associate Professor, BVICAM, New Delhi)

2023

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.1

---

---

---

---

---

---

---

---

---

---

## CPU Scheduling

- The objective of multiprogramming is to have some process running at all times.
  - A process is executed until it must wait, typically for the completion of some I/O request.
  - When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process.
  - Every time one process has to wait, another process can take over use of the CPU.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.2

---

---

---

---

---

---

---

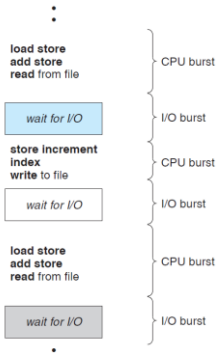
---

---

---

## CPU-I/O Burst Cycle

- Process execution consists of a cycle of CPU execution and I/O wait.
- Processes alternate between these two states.
  - Process execution begins with a CPU burst.
  - That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on.
- Eventually, the final CPU burst ends with a system request to terminate execution.



© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.3

---

---

---

---

---

---

---

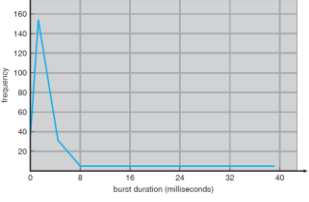
---

---

---

### CPU-I/O Burst Cycle (contd...)

- The durations of CPU bursts vary greatly from process to process and from computer to computer.
- However, it tend to have a following frequency curve:
 



- An I/O-bound program typically has many short CPU bursts.
  - A CPU-bound program might have a few long CPU bursts.
- This curve is generally characterized as exponential, with a large number of short CPU bursts and a small number of long CPU bursts.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.4

---

---

---

---

---

---

---

---

---

---

### CPU Scheduler

- 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 short-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 process.
- The ready queue is not necessarily a first-in, first-out (FIFO) queue.
- The records in the queues are generally PCBs of the processes.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.5

---

---

---

---

---

---

---

---

---

---

### Circumstances for Scheduling Decisions

- CPU-scheduling decisions may take place under the following circumstances:
  1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait for the termination of one of the child processes).
  2. When a process switches from the running state to the ready state (for example, when an interrupt occurs).
  3. When a process switches from the waiting state to the ready state (for example, at completion of I/O).
  4. When a process terminates.
- For situations 1 and 4, there is no choice in terms of scheduling.
  - A new process (if one exists in the ready queue) must be selected for execution.
  - There is a choice, however, for situations 2 and 3.
- When scheduling takes place only under circumstances 1 and 4, scheduling scheme is called **non-preemptive**; otherwise, it is **preemptive**.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.6

---

---

---

---

---


---

---

---

---

---



## Preemptive and Non-Preemptive Scheduling

- **Non-Preemptive Scheduling**
  - The CPU is allocated to the process till it terminates or switches to waiting state.
  - This scheduling method was used by Microsoft Windows 3.x.
- **Preemptive Scheduling**
  - The CPU is allocated to the processes for the limited time.
  - Windows 95 introduced preemptive scheduling, and all subsequent versions of Windows operating systems have used preemptive scheduling.
  - The Mac OS X operating system for the Macintosh also uses preemptive scheduling.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.7

---

---

---

---

---


---

---

---

---

---



## Preemptive vs. Non-Preemptive Scheduling

Preemptive Scheduling	Non-Preemptive Scheduling
The CPU is allocated to the processes for the limited time.	The CPU is allocated to the process till it terminates or switches to waiting state.
Processor can be preempted to execute a different process in the middle of execution of any current process.	Once Processor starts to execute a process it must finish it before executing the other. It cannot be paused in middle.
CPU utilization is more as compared to Non-Preemptive Scheduling.	CPU utilization is less as compared to Preemptive Scheduling.
Waiting time and Response time is less.	Waiting time and Response time is more.
If a high priority process frequently arrives in the ready queue, low priority process may starve.	If a process with long burst time is running CPU, then another process with less CPU burst time may starve.
Preemptive scheduling is flexible.	Non-preemptive scheduling is rigid.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.8

---

---

---

---

---


---

---

---

---

---



## Dispatcher

- The dispatcher (a component of CPU-scheduling function) gives control of the CPU to the process selected by the short-term scheduler.
- The functions of dispatcher involves:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart the program
- The dispatcher should be as fast as possible, since it is invoked during every process switch.
- The time taken by dispatcher to stop one process and start another running is known as the **dispatch latency**.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.9

---

---

---

---

---

---

---

---

---

---

## Scheduling Criteria

- **CPU Utilization**
  - We want to keep the CPU as busy as possible. CPU utilization can range from 0 to 100 percent.
- **Throughput**
  - If the CPU is busy executing processes, then work is being done. Throughput refers to the number of processes completed per time unit.
- **Turnaround Time**
  - The interval from the time of submission of a process to the time of completion is called turnaround time.
  - Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U2.10

---

---

---

---

---

---

---

---

## Scheduling Criteria (contd...)

- **Response Time**
  - In an interactive system, turnaround time may not be the best criterion.
  - Response time is the time it takes to start responding, not the time it takes to output the response.
- It is desirable to:
  - Maximize CPU utilization and throughput
  - Minimize turnaround time, waiting time and response time

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U2.11

---

---

---

---

---

---

---

---

## Scheduling Algorithms

- First-Come, First-Served Scheduling
- Shortest-Job-First Scheduling
- Priority Scheduling
- Round-Robin Scheduling
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling

} Refer to PDF File

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U2.12

---

---

---

---

---

---

---

---

## Multilevel Queue Scheduling

- The processes can be classified into different groups where each group has its own scheduling needs.
- A common classification is:
  - Foreground (Interactive) Processes
  - Background Processes
- These two types of processes have different requirements and so may have different scheduling needs.
- 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 size, process priority, or process type.
  - Each queue has its own scheduling algorithm.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U2.13

---

---

---

---

---

---

---

---

---

---

## Multilevel Queue Scheduling (contd...)

- Separate queues might be used for different categories of processes.

highest priority

lowest priority

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U2.14

---

---

---

---

---

---

---

---

---

---

## Multilevel Queue Scheduling (contd...)

- Scheduling among the queues is commonly implemented as fixed-priority preemptive scheduling.
  - Each queue has absolute priority over lower-priority queues.
  - For example, no process in the batch queue, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty.
  - If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.
- Another possibility is to time-slice among the queues.
  - Each queue gets a certain portion of the CPU time, which it can then schedule among its various processes.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U2.15

---

---

---

---

---


---

---

---

---

---



### Multilevel Queue Scheduling: Example

- Process : P1    P2    P3    P4
- Arrival Time : 0    0    0    10
- Burst Time : 4    3    8    5
- Queue No. : 1    1    2    1
- Priority of Queue 1 is greater than Queue 2.
- Queue 1 uses Round Robin (Time Quantum = 2) and Queue 2 uses First-Come, First-Served.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.16

---

---

---

---

---


---

---

---

---

---



### Questions

- Three process P1, P2 and P3 arrive at time zero. The total time spent by the process in the system is 10ms, 20ms, and 30ms respectively. They spent first 20% of their execution time in doing I/O and the rest 80% in CPU processing. What is the percentage utilization of CPU using FCFS scheduling algorithm?
- Three process p1, P2 and P3 arrive at time zero. Their total execution time is 10ms, 15ms, and 20ms respectively. They spent first 20% of their execution time in doing I/O, next 60% in CPU processing and the last 20% again doing I/O. For what percentage of time was the CPU free? Use Round robin algorithm with time quantum 5ms.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.17

---

---

---

---

---


---

---

---

---

---



### Multilevel Feedback Queue Scheduling

- The multilevel feedback queue scheduling algorithm allows a process to move between queues.
  - The idea is to separate processes according to the characteristics of their CPU bursts.
  - If a process uses too much CPU time, it will be moved to a lower-priority queue.
  - A process that waits too long in a lower-priority queue may be moved to a higher-priority queue.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.18

---

---

---

---

---


---

---

---

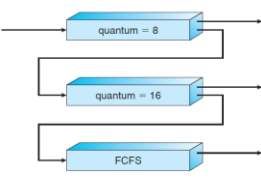
---

---



## Multilevel Feedback Queue Scheduling

- A process entering the ready queue is put in **Queue 0**.
- A process in **Queue 0** is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of **Queue 1**.
- If **Queue 0** is empty, the process at the head of **Queue 1** is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into **Queue 2**.
- Processes in **Queue 2** are run on an FCFS basis but are run only when **Queues 0** and **1** are empty.



© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.19

---

---

---

---

---


---

---

---

---

---



## Process Synchronization: Background

```

Producer   while (true) {
            /* produce an item in nextProduced */
            while (counter == BUFFER_SIZE)
                /* do nothing */
            buffer[in] = nextProduced;
            in = (in + 1) % BUFFER_SIZE;
            counter++;
        }

Consumer   while (true) {
            while (counter == 0)
                /* do nothing */
            nextConsumed = buffer[out];
            out = (out + 1) % BUFFER_SIZE;
            counter--;
            /* consume the item in nextConsumed */
        }

```

- Both, the producer and consumer routines are correct separately.
- They may not function correctly when executed concurrently.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.20

---

---

---

---

---


---

---

---

---

---



## Process Synchronization: Background

- An **incorrect state** may arrive because both processes are allowed to manipulate the variable **counter** concurrently.
- A situation, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.
- In our example, to guard against the race condition, we need to ensure that only one process at a time can manipulate the variable **counter**.
- To make such a guarantee, the processes must be synchronized in some way.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.21

---

---

---

---

---

---

---

---

---

---

## The Critical-Section Problem

- Consider a system consisting of  $n$  processes  $\{P_0, P_1, \dots, P_{n-1}\}$ .
- Each process has a **segment of code**, called a **critical section**, in which the process may be changing common variables, updating a table, etc.
- When one process is executing in its critical section, no other process is to be allowed to execute in its critical section.
  - No two processes are executing in their critical sections at the same time.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.22

---

---

---

---

---

---

---

---

---

---

## The Critical-Section Problem (contd...)

- The solution of critical-section problem involves design of a protocol that the processes can use to cooperate.

```

do {
    entry section ← controls the entry into critical section and gets a LOCK on required resources
    critical section ← the critical part
    exit section ← removes the LOCK from the resources and lets the others know that its critical section is over
    remainder section ← rest of the section
} while (TRUE);

```

- Each process must request permission to enter its critical section.
- The section of code implementing this request is the **entry section**.
- The critical section may be followed by an **exit section**.
- The remaining code is the **remainder section**.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.23

---

---

---

---

---

---

---

---

---

---

## Solution of Critical-Section Problem

- A solution to the critical-section problem must satisfy the following three requirements:
  - **Mutual Exclusion** - If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections.
  - **Progress** - If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections should decide which will enter its critical section next, in a finite time.
  - **Bounded Waiting** - After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, system must grant the process permission to get into its critical section.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.24

---

---

---

---

---

---

---

---

---

---



**Critical Section Problem: Algorithm 1**

This algorithm works only for two processes.

Given: `int turn;` (Shared Variable)

**Process  $P_i$**

```

turn = i;
do{
    while (turn!=i); <-- ENTRY SECTION
    CRITICAL SECTION
    turn = j; <-- EXIT SECTION
    REMAINDER SECTION
}while (TRUE);
    
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.25

---

---

---

---

---

---

---

---

---

---

---

---

**Critical Section Problem: Algorithm 1 (contd...)**

Given: `int turn;` (Shared Variable)

**Process  $P_i$**

```

turn = i;
do{
    while (turn!=i);
    CRITICAL SECTION
    turn = j;
    REMAINDER SECTION
}while (TRUE);
        
```

**Process  $P_j$**

```

turn = j;
do{
    while (turn!=j);
    CRITICAL SECTION
    turn = i;
    REMAINDER SECTION
}while (TRUE);
        
```

This algorithm does not satisfy the progress requirement because there is strict alternation between the processes.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.26

---

---

---

---

---

---

---

---

---

---

---

---

**Critical Section Problem: Algorithm 2**

This algorithm also works only for two processes.

Given: `boolean flag[2]; flag[0] = FALSE; flag[1] = FALSE;`

**Process  $P_i$**

```

do{
    flag[i] = TRUE;
    while (flag[j]); <-- ENTRY SECTION
    CRITICAL SECTION
    flag[i] = FALSE; <-- EXIT SECTION
    REMAINDER SECTION
}while (TRUE);
    
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.27

---

---

---

---

---

---

---

---

---

---

---

---

**Critical Section Problem: Algorithm 2 (contd...)**

Given: `boolean flag[2]; flag[0] = FALSE; flag[1] = FALSE;`

**Process P<sub>i</sub>**

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

**Process P<sub>j</sub>**

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

This algorithm can fail the **progress** requirement if both processes set their flags to true and then both execute the while loop.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.28

---

---

---

---

---

---

---

---

---

---

---

---

**Critical Section Problem: Peterson's Solution**

- Peterson's Solution is a classical software based solution to the critical section problem.
  - Peterson's algorithm is used to synchronize two processes.
  - Peterson's solution requires the two processes to share two data items:
 

```
int turn;
boolean flag[2];
```

    - **Variable turn** indicates whose turn it is to enter its critical section.
      - If `turn == i`, then process P<sub>i</sub> is allowed to execute in its critical section.
    - **Flag array** is used to indicate if a process wants to enter its critical section.
      - If `flag[i] == true`, it indicates that P<sub>i</sub> wants to enter its critical section.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.29

---

---

---

---

---

---

---

---

---

---

---

---

**Structure of Process P<sub>i</sub> in Peterson's Solution**

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

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.30

---

---

---

---

---

---

---

---

---

---

---

---

## Structure of Process $P_i$ and $P_j$

Given: `boolean flag[2];`    `int turn;`    `flag[0]=false;`    `flag[1]=false;`

**Process  $P_i$**

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

**Process  $P_j$**

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

- Peterson's algorithm satisfy the three requirements (**mutual exclusive**, **progress**, **bounded waiting**) of solution of critical section problem.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh    U2.31

---

---

---

---

---

---

---

---

---

---

## Limitations of Peterson's Solution

- **Works only for TWO Processes.**
- **Busy Waiting**
  - **Busy waiting**, also known as **spinning**, or **busy looping** is a process synchronization technique in which a process/task waits and constantly checks for a condition to be satisfied before proceeding with its execution.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh    U2.32

---

---

---

---

---

---

---

---

---

---

## Waiting Approaches

- There are two general approaches to waiting in operating systems:
  - **Busy Waiting** - A process/task can continuously check for the condition to be satisfied while **consuming the processor**.
  - **Sleeping (Blocked Waiting or Sleep Waiting)** - A process can wait **without consuming the processor**. In such a case, the process/task is alerted or awakened when the condition is satisfied.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh    U2.33

---

---

---

---

---

---

---

---

---

---

Busy Waiting

- Busy looping is usually used to achieve mutual exclusion in operating systems.
- Busy waiting can be inefficient because the looping procedure is a waste of computer resources
- Although inefficient, busy waiting can be beneficial in mutual exclusion if the waiting time is short and insignificant.
- Additionally, busy waiting is quick and simple to understand and implement.

---

- A workaround solution for the inefficiency of busy waiting that is implemented in most operating systems is the use of a delay function.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.34

---

---

---

---

---

---

---

---

Sleep Waiting

- In this case, the process/task is alerted or awakened when the condition is satisfied.
- A delay function (sleep system call) places the process involved in busy waiting into an inactive state for a specified amount of time.
  - In this case, resources are not wasted as the process is "asleep".
  - After the sleep time has elapsed, the process is awakened to continue its execution.
  - If the condition is still not satisfied, the sleep time is incremented until the condition can be satisfied.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.35

---

---

---

---

---

---

---

---

Bakery Algorithm

- Bakery Algorithm is a critical section solution for  $n$  processes.
- Each process, wanting to enter critical section, gets a token number.
  - The token numbering scheme always generates numbers in increasing order of enumeration; i.e., 1, 2, 3, 3, 4, 5, ...
- A process with lowest token number will enter the critical section.
  - The algorithm preserves the first come first serve property.
- If two processes have same token number, the process with lower process id (Pid) will enter the critical section.
- The token number of process is set to 0 when it finishes execution.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.36

---

---

---

---

---

---

---

---

**Bakery Algorithm: Structure of Process P<sub>i</sub>**

```

boolean choosing[N] = {FALSE, ..., FALSE};
int number[N] = {0, ..., 0};

Process Pi

do {
    choosing[i] = TRUE;
    number[i] = MAX(number[0], number[1], ..., number[n-1]) + 1;
    choosing[i] = FALSE;
    for (j = 0; j < n; j++) {
        while (choosing[j]);
        while ((number[j] != 0) && ((number[j],j) < (number[i],i)));
    }
    CRITICAL SECTION
    number[i] = 0;
    REMAINDER SECTION
}while(TRUE);
    
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.37

---

---

---

---

---

---

---

---

---

---

---

---

**Solution using Lock**

```

do {
    acquire lock
    critical section
    release lock
    remainder section
} while (TRUE);

do {
    while (Lock != 0);
    Lock = 1;
    CRITICAL SECTION
    Lock = 0;
    REMAINDER SECTION
}while(TRUE);
    
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.38

---

---

---

---

---

---

---

---

---

---

---

---

**Synchronization Hardware**

- **Uniprocessor Environment**
  - Critical-section problem could be solved if interrupts could be prevented from occurring (while a shared variable was being modified).
    - This is often the approach taken by non-preemptive kernels.
- **Multiprocessor Environment**
  - Disabling interrupts on a multiprocessor can be time consuming.
  - With special atomic hardware instructions, solution of critical-section problem can be done.
    - TestAndSet()
    - Swap()

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.39

---

---

---

---

---

---

---

---

---

---

---

---

## TestAndSet()

**Definition of TestAndSet()**

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

It is executed atomically (that is, as one uninterruptible unit)

If two TestAndSet() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.

**Implementation of TestAndSet()**

```
do {
    while (TestAndSet(&lock))
        ; // do nothing

    // critical section

    lock = FALSE;

    // remainder section
} while (TRUE);
```

This algorithm satisfies the mutual-exclusion requirement, but do not satisfy the bounded-waiting requirement.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.40

---

---

---

---

---

---

---

---

---

---

## Swap

**Definition of Swap()**

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

It is executed atomically (that is, as one uninterruptible unit)

**Implementation of Swap()**

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

    // critical section

    lock = FALSE;

    // remainder section
} while (TRUE);
```

This algorithm satisfies the mutual-exclusion requirement, but do not satisfy the bounded-waiting requirement.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.41

---

---

---

---

---

---

---

---

---

---

## Semaphore

- In 1965, proposed by Dijkstra, Semaphore is a mechanism that can be used to provide synchronization of tasks (to solve critical-section problem).
- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal().
- The initial value of S depends on how many processes are allowed in CS.

**wait()**

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

**signal()**

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

The testing of the integer value of S (S ≤ 0), as well as its possible modification (S--), must be executed without interruption.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.42

---

---

---

---

---

---

---

---

---

---

## Types of Semaphores

- **Binary Semaphore**
  - It is a special form of semaphore used for implementing mutual exclusion, hence it is often called a **Mutex Lock**.
  - A binary semaphore is **initialized to 1** and only **takes the values 0 and 1** during execution of a program.
  - It can be used to deal with the critical-section problem for multiple processes.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.43

---

---

---

---

---

---

---

---

## Types of Semaphores

- **Counting Semaphore**
  - Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
  - It is **initialized to the number of resources available**.
  - Each process that wishes to use a resource performs a **wait()** operation on the semaphore (**decrementing the count**).
  - When a process releases a resource, it performs a **signal()** operation (**incrementing the count**).
  - **When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.44

---

---

---

---

---

---

---

---

## Implementation of Semaphore

### Mutual-Exclusion Implementation with Semaphores

<pre>do{   wait(S);   CRITICAL SECTION   signal(S);   REMAINDER SECTION }while(TRUE)</pre>	<pre>wait(S){   while(S &lt;= 0);   S--; }</pre>	<pre>signal(S){   S++; }</pre>
--	--	--------------------------------

- The main disadvantage of the semaphore definition given here is that it **requires busy waiting**.
  - While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.45

---

---

---


---

---

---

---

---



## Spinlock

- Busy waiting wastes CPU cycles that some other process might be able to use productively.
- The semaphore having busy waiting mechanism is also called a **spinlock** because the process “spins” while waiting for the lock.
- Spinlock mechanism has an advantage in that no context switch is required when a process waits on a lock
  - When locks are expected to be held for short times, spinlocks are useful.
  - Spinlocks are often employed on multiprocessor systems where one thread can “spin” on one processor while another thread performs its critical section on another processor.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U2.46

---

---

---

---

---


---

---

---

---

---



## Solution of Spinlock (Busy Waiting)

- To overcome the need for busy waiting, we can **modify the definition of the wait() and signal()** semaphore operations.
- When a process executes the **wait()** operation and finds that the semaphore value is not positive, it must wait.
  - However, rather than engaging in busy waiting, the process can block itself.
  - The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U2.47

---

---

---

---

---


---

---

---

---

---



## Solution of Spinlock (Busy Waiting)

- To overcome the need for busy waiting, we can **modify the definition of the wait() and signal()** semaphore operations.

The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

```

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

wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}

```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U2.48

---

---

---

---

---

---

---

---

---

---



**Solution of Spinlock (Busy Waiting)**

- In the solution of busy waiting, semaphore values may be negative.
- Semaphore values are never negative under the classical definition of semaphores with busy waiting.
- If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.
- The list of waiting processes can be easily implemented by a link field in each process control block (PCB).
- To ensure bounded waiting, a FIFO queue or any other queuing strategy may be used.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.49

---

---

---

---

---

---

---

---

---

---

**Limitations of Semaphores**

- Deadlocks**
  - Consider a system consisting of two processes, P0 and P1, each accessing two semaphores, S and Q, set to the value 1:

$P_0$	$P_1$	
<code>wait(S);</code>	<code>wait(Q);</code>	<ul style="list-style-type: none"> <li>Suppose that P0 executes <code>wait(S)</code> and then P1 executes <code>wait(Q)</code>.</li> <li>When P0 executes <code>wait(Q)</code>, it must wait until P1 executes <code>signal(Q)</code>.</li> <li>Similarly, when P1 executes <code>wait(S)</code>, it must wait until P0 executes <code>signal(S)</code>.</li> <li>Since these <code>signal()</code> operations cannot be executed, P0 and P1 are <b>deadlocked</b>.</li> </ul>
<code>wait(Q);</code>	<code>wait(S);</code>	
<code>...</code>	<code>...</code>	
<code>signal(S);</code>	<code>signal(Q);</code>	
<code>signal(Q);</code>	<code>signal(S);</code>	

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.50

---

---

---

---

---

---

---

---

---

---

**Limitations of Semaphores**

- Indefinite Blocking or Starvation**
  - Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.51

---

---

---

---

---

---

---

---

---

---

**Classic Problems of Synchronization**

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

} Solution using Semaphore

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.52

---

---

---

---

---

---

---

---

**Bounded-Buffer Problem**

```

Producer {
    while (true) {
        /* produce an item in nextProduced */
        while (counter == BUFFER_SIZE)
            /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
    }
}

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

- The bounded-buffer problem is also known as producer-consumer problem.
- Both, the producer and consumer routines are correct separately.
- They may not function correctly when executed concurrently.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.53

---

---

---

---

---

---

---

---

**Bounded-Buffer Problem - Solution**

- Assume,
  - Buffer Size = n slots
  - semaphore mutex (for mutual exclusion)  
mutex = 1
  - semaphore empty (number of empty slots in buffer)  
empty = n
  - semaphore full (number of full slots in buffer)  
full = 0

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.54

---

---

---

---

---

---

---

---

Bounded-Buffer Problem - Solution

Producer Process	Consumer Process
<pre>do {     . . .     // produce an item in nextp     . . .     wait(empty);     wait(mutex);     . . .     // add nextp to buffer     . . .     signal(mutex);     signal(full); } while (TRUE);</pre>	<pre>do {     wait(full);     wait(mutex);     . . .     // remove an item from buffer to nextc     . . .     signal(mutex);     signal(empty);     . . .     // consume the item in nextc     . . . } while (TRUE);</pre>

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U2.55

---

---

---

---

---

---

---

---

---

---

Readers–Writers Problem

- Consider a situation where a shared resource (such as file or dataset) is to be accessed by multiple concurrent processes.
- Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.
- We distinguish between these two types of processes - **reader** and **writer**.
  - If two readers access the shared data simultaneously, no adverse effects will result.
  - If a writer and some other process (either a reader or a writer) access the database simultaneously, inconsistency may occur.
- This synchronization problem is referred to as the readers–writers problem.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U2.56

---

---

---

---

---

---

---

---

---

---

Readers–Writers Problem - Solution

- Assume,
  - **int readcount = 0**  
The readcount variable keeps track of how many processes are currently reading the object.
  - **semaphore mutex**  
mutex is used for mutual exclusion when the variable readcount is updated.  
`mutex = 1`
  - **semaphore wrt**  
wrt functions as a mutual-exclusion semaphore for the writers.  
`wrt = 1`

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U2.57

---

---

---

---

---

---

---

---

---

---

**Readers–Writers Problem - Solution**

<p><b>Write Process(s)</b></p> <pre>do {     wait(wrt);     . . .     // writing is performed     . . .     signal(wrt); } while (TRUE);</pre>	<p><b>Reader Process(s)</b></p> <pre>do {     wait(mutex);     readcount++;     if (readcount == 1)         wait(wrt);     signal(mutex);     . . .     // reading is performed     . . .     wait(mutex);     readcount--;     if (readcount == 0)         signal(wrt);     signal(mutex); } while (TRUE);</pre>
--	---

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.58

---

---

---

---

---

---

---

---

---

---

**Dining-Philosophers Problem**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.59

---

---

---

---

---

---

---

---

---

---

**Dining-Philosophers Problem**

- Consider a situation where five philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.
- They are in thinking-eating cycle.
- In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.
- A philosopher may pick up only one chopstick at a time. (Obviously, he/she cannot pick up a chopstick that is already in the hand of a neighbor).
- From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to his/her (the chopsticks that are between his/her and his/her left and right neighbors).
- The problem is how to choose two chopsticks (one his/her own and one of other)

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.60

---

---

---

---

---

---

---

---

---

---

Dining-Philosophers Problem - Solution

- The Dining-Philosopher Problem is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.
- The problem can be solved by representing each chopstick with a semaphore.
  - A philosopher tries to grab a chopstick by executing a `wait()` operation on that semaphore.
  - He/she releases his/her chopsticks by executing the `signal()` operation on the appropriate semaphores.
  - semaphore chopstick[5] = 1;

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.61

---

---

---

---

---

---

---

---

---

---

Dining-Philosophers Problem - Solution

```

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

```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.62

---

---

---

---

---

---

---

---

---

---

Limitations of Solution

- Although this solution guarantees that no two neighbors are eating simultaneously, **but it could create a deadlock.**
  - Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick.
  - All the elements of chopstick will now be equal to 0.
  - When each philosopher tries to grab her right chopstick, she will be delayed forever.
- Possible Solutions:
  - Allow at most four philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up her chopsticks only if both chopsticks are available.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U2.63

---

---

---

---

---

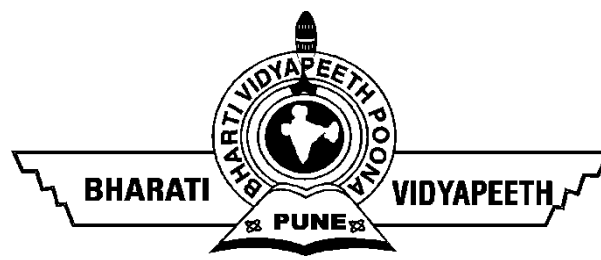
---

---

---

---

---



# Operating Systems with Linux

(MCA-105)

## Unit - 2

by

**Dr. Sunil Pratap Singh**

(Associate Professor, BVICAM, New Delhi)

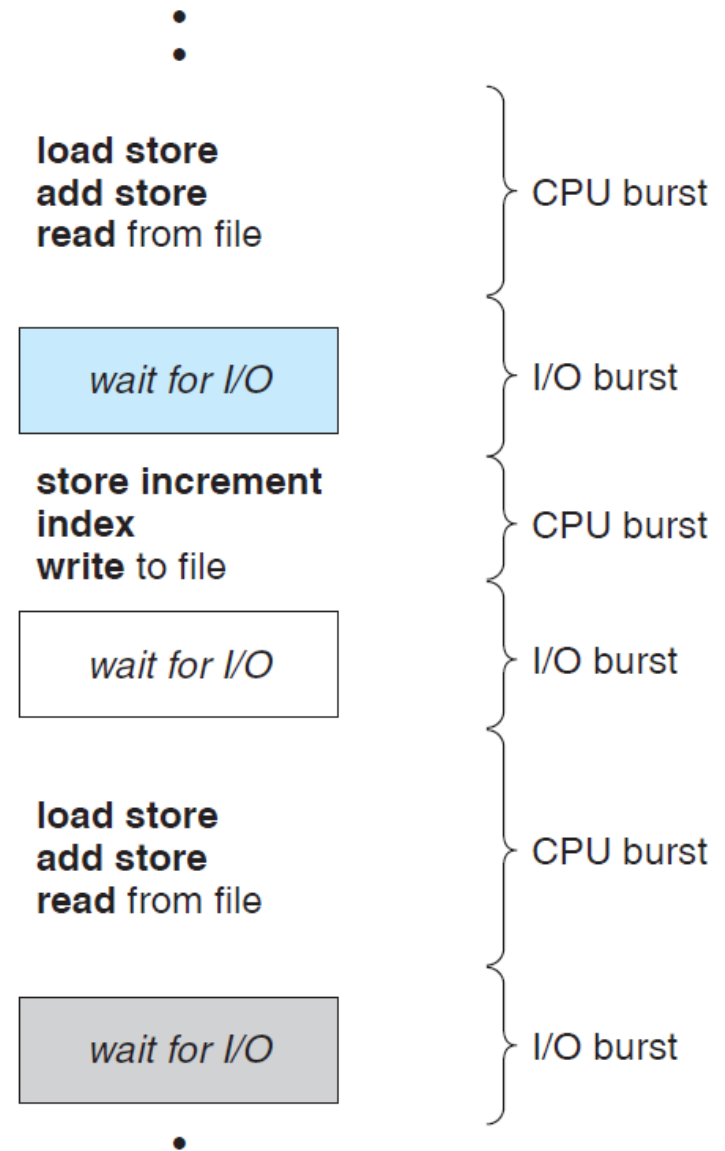
2023

# CPU Scheduling

- The objective of multiprogramming is to have some process running at all times.
  - A process is executed until it must wait, typically for the completion of some I/O request.
  - When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process.
  - Every time one process has to wait, another process can take over use of the CPU.

# CPU-I/O Burst Cycle

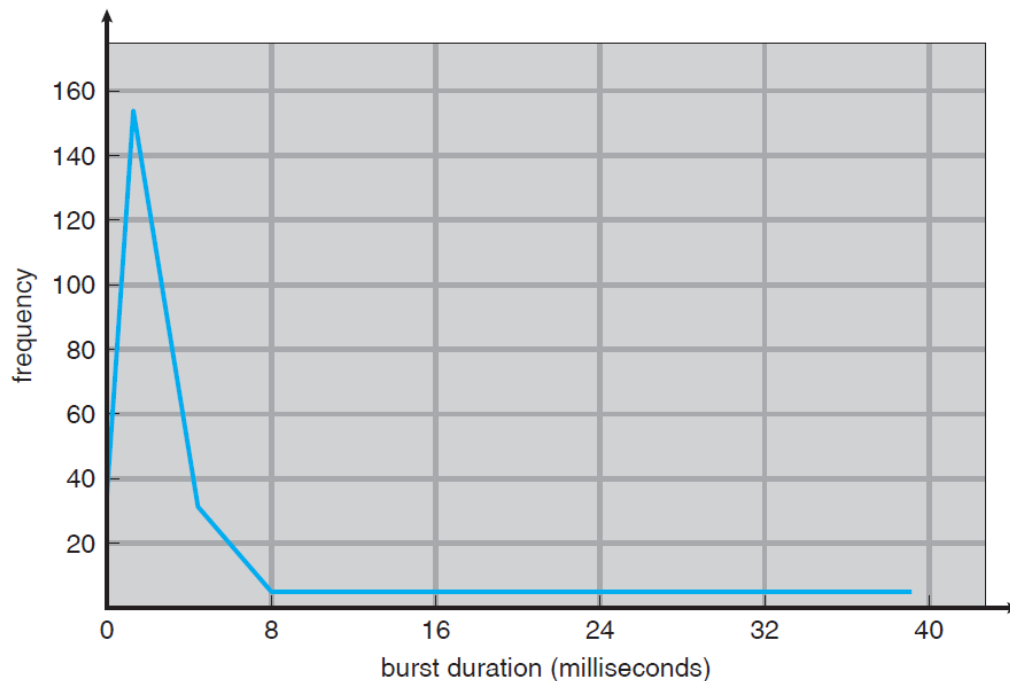
- Process execution consists of a **cycle** of CPU execution and I/O wait.
- Processes **alternate** between these two states.
  - Process execution begins with a CPU burst.
  - That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on.
- Eventually, the final CPU burst ends with a system request to terminate execution.





# CPU-I/O Burst Cycle (contd...)

- The durations of CPU bursts vary greatly from process to process and from computer to computer.
- However, it tends to have a following frequency curve:



- An I/O-bound program typically has many short CPU bursts.
- A CPU-bound program might have a few long CPU bursts.

- This curve is generally characterized as exponential, with a large number of short CPU bursts and a small number of long CPU bursts.

# CPU Scheduler

- 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 short-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 process.
- The ready queue is not necessarily a first-in, first-out (FIFO) queue.
- The records in the queues are generally PCBs of the processes.

# Circumstances for Scheduling Decisions

- CPU-scheduling decisions may take place under the following circumstances:
  1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait for the termination of one of the child processes).
  2. When a process switches from the running state to the ready state (for example, when an interrupt occurs).
  3. When a process switches from the waiting state to the ready state (for example, at completion of I/O).
  4. When a process terminates.
- For situations 1 and 4, there is no choice in terms of scheduling.
  - A new process (if one exists in the ready queue) must be selected for execution.
  - There is a choice, however, for situations 2 and 3.
- When scheduling takes place only under circumstances 1 and 4, scheduling scheme is called **non-preemptive**; otherwise, it is **preemptive**.

## ■ Non-Preemptive Scheduling

- The CPU is allocated to the process till it terminates or switches to waiting state.
- This scheduling method was used by Microsoft Windows 3.x.

## ■ Preemptive Scheduling

- The CPU is allocated to the processes for the limited time.
- Windows 95 introduced preemptive scheduling, and all subsequent versions of Windows operating systems have used preemptive scheduling.
- The Mac OS X operating system for the Macintosh also uses preemptive scheduling.

# Preemptive vs. Non-Preemptive Scheduling

Preemptive Scheduling	Non-Preemptive Scheduling
The CPU is allocated to the processes for the limited time.	The CPU is allocated to the process till it terminates or switches to waiting state.
Processor can be preempted to execute a different process in the middle of execution of any current process.	Once Processor starts to execute a process it must finish it before executing the other. It cannot be paused in middle.
CPU utilization is more as compared to Non-Preemptive Scheduling.	CPU utilization is less as compared to Preemptive Scheduling.
Waiting time and Response time is less.	Waiting time and Response time is more.
If a high priority process frequently arrives in the ready queue, low priority process may starve.	If a process with long burst time is running CPU, then another process with less CPU burst time may starve.
Preemptive scheduling is flexible.	Non-preemptive scheduling is rigid.

# Dispatcher

- The dispatcher (a component of CPU-scheduling function) gives control of the CPU to the process selected by the short-term scheduler.
- The functions of dispatcher involves:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart the program
- The dispatcher should be as fast as possible, since it is invoked during every process switch.
- The time taken by dispatcher to stop one process and start another running is known as the **dispatch latency**.

# Scheduling Criteria

## ■ CPU Utilization

- We want to keep the CPU as busy as possible. CPU utilization can range from 0 to 100 percent.

## ■ Throughput

- If the CPU is busy executing processes, then work is being done. Throughput refers to the number of processes completed per time unit.

## ■ Turnaround Time

- The interval from the time of submission of a process to the time of completion is called turnaround time.
- Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

# Scheduling Criteria (contd...)

## ■ Response Time

- In an interactive system, turnaround time may not be the best criterion.
- Response time is the time it takes to start responding, not the time it takes to output the response.
- It is desirable to:
  - Maximize CPU utilization and throughput
  - Minimize turnaround time, waiting time and response time



# Scheduling Algorithms

- First-Come, First-Served Scheduling
- Shortest-Job-First Scheduling
- Priority Scheduling
- Round-Robin Scheduling
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling

Refer to PDF File

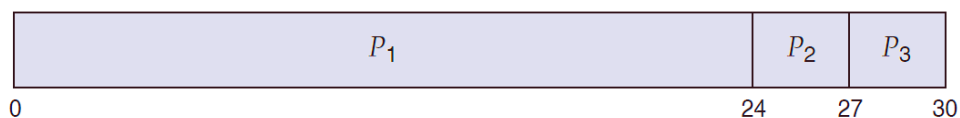
## First-Come, First-Served Scheduling

- The process that requests the CPU first is allocated the CPU first.
- The implementation of the FCFS policy is 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.

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

Process	:	P1	P2	P3
Burst Time	:	24	3	3

- If the processes arrive in the order P1, P2, P3, and are served in FCFS order, we get the result shown in the following Gantt Chart:



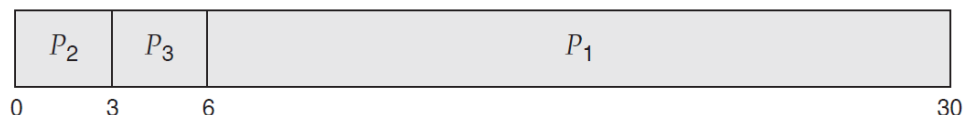
Waiting Time for P1 = 0 Millisecond

Waiting Time for P2 = 24 Milliseconds

Waiting Time for P3 = 27 Milliseconds

**Average Waiting Time =  $(0 + 24 + 27)/3 = 17$  Milliseconds**

- If the processes arrive in the order P2, P3, P1, and are served in FCFS order, we get the result shown in the following Gantt Chart:



Waiting Time for P1 = 6 Milliseconds

Waiting Time for P2 = 0 Millisecond

Waiting Time for P3 = 3 Milliseconds

**Average Waiting Time =  $(6 + 0 + 3)/3 = 3$  Milliseconds**

- The average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly.
- The FCFS scheduling algorithm is non-preemptive.
  - Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.
  - The FCFS algorithm is particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals.

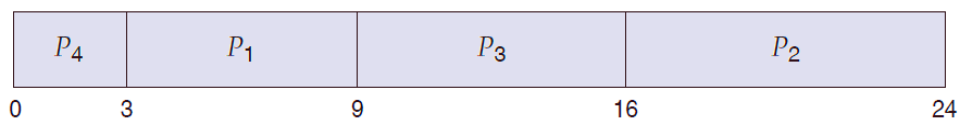
## Shortest-Job-First Scheduling

- This algorithm associates, with each process, the length of the process's next CPU burst.
- When the CPU is available, it is assigned to the process that has the smallest next CPU burst.
- If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.
- A more appropriate term for this scheduling method would be the shortest-next-CPU-burst algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.

**Example:** Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	:	P1	P2	P3	P4
Burst Time	:	6	8	7	3

- Using SJF scheduling, we would schedule these processes according to the following Gantt chart:



Waiting Time for P1 = 3 Milliseconds

Waiting Time for P2 = 16 Milliseconds

Waiting Time for P3 = 9 Milliseconds

Waiting Time for P4 = 0 Millisecond

**Average Waiting Time = (3 + 16 + 9 + 0)/4 = 7 Milliseconds**

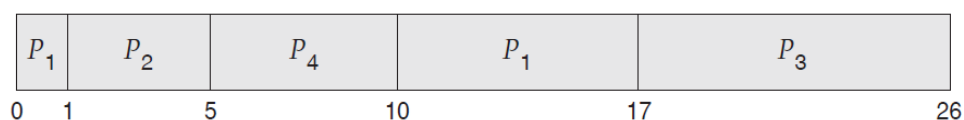
**If we use FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.**

- The SJF scheduling gives the minimum average waiting time for a given set of processes.
- The real difficulty with the SJF algorithm is to know the length of the next CPU request.
- SJF scheduling is used frequently in long-term scheduling.
  - For long-term (job) scheduling in a batch system, we can use the length as the process time limit that a user specifies when he submits the job.
  - Thus, users are motivated to estimate the process time limit accurately, since a lower value may mean faster response. (Too low a value will cause a time-limit-exceeded error and require resubmission.)
  - SJF scheduling is used frequently in long-term scheduling.
- Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling.
  - With short-term scheduling, there is no way to know the length of the next CPU burst.
  - One approach is to try to approximate SJF scheduling – We expect that the next CPU burst will be similar in length to the previous ones.
- The SJF algorithm can be either preemptive or non-preemptive.
  - The choice arises when a new process arrives at the ready queue while a previous process is still executing.
  - The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process.
  - Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first scheduling**.

**Example of Preemptive SJF Scheduling:** Consider the following set of processes, with the length of the CPU burst given in milliseconds:

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

- If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt Chart:



Waiting Time for P1 = 10 – 1 = 9 Milliseconds

Waiting Time for P2 = 1 – 1 = 0 Millisecond

Waiting Time for P3 = 17 – 2 = 15 Milliseconds

Waiting Time for P4 = 5 – 3 = 2 Milliseconds

**Average Waiting Time (Preemptive Scheduling) = (9 + 0 + 15 + 2)/4 = 6.5 Milliseconds**

**Average Waiting Time (Non-Preemptive Scheduling) = 7.75 Milliseconds**

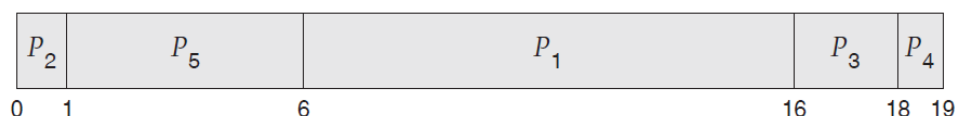
## Priority Scheduling

- The SJF algorithm is a special case of the general priority scheduling algorithm.
- A priority is associated with each process, and the CPU is allocated to the process with the highest priority.
- Equal-priority processes are scheduled in FCFS order.
- 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.
- Some systems use low numbers to represent low priority; others use low numbers for high priority.
- **We assume that low numbers represent high priority.**

**Example:** Consider the following set of processes, assumed to have arrived at time 0 in the order P1, P2, . . . , P5, with the length of the CPU burst given in milliseconds:

Process	:	P1	P2	P3	P4	P5
Burst Time	:	10	1	2	1	5
Priority	:	3	1	4	5	2

- Using priority scheduling, the processes are scheduled according to the following Gantt Chart:



Waiting Time for P1 = 6 Milliseconds

Waiting Time for P2 = 0 Millisecond

Waiting Time for P3 = 16 Milliseconds

Waiting Time for P4 = 18 Milliseconds

Waiting Time for P5 = 1 Millisecond

**Average Waiting Time =  $(6 + 0 + 16 + 18 + 1)/5 = 8.2$  Milliseconds**

- **Priorities can be defined either internally or externally.**
  - Internally defined priorities use some measurable quantity or quantities to compute the priority of a process.
  - For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities.
  - External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use.
- **Priority scheduling can be either preemptive or non-preemptive.**
  - When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.
  - 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.
  - A non-preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.
- **A major problem with priority scheduling algorithms is indefinite blocking, or starvation.**
  - A priority scheduling algorithm can leave some low priority processes waiting indefinitely.
- **A solution to the problem of indefinite blockage of low-priority processes is aging.**
  - Aging is a technique of gradually increasing 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 increase the priority of a waiting process by 1 every 15 minutes.

## Round-Robin Scheduling

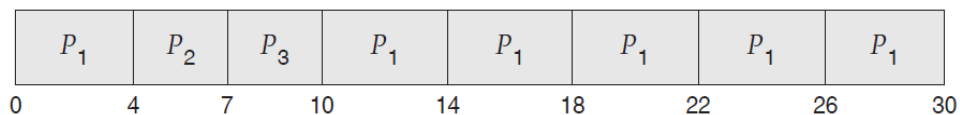
- The round-robin (RR) scheduling algorithm is designed especially for time-sharing systems.
- It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes.

- A small unit of time, called a time quantum or time slice, is defined.
- A time quantum is generally from 10 to 100 milliseconds in length.
- The ready queue is treated as a circular queue.
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.
- New processes are added to the tail of the ready queue (FCFS).
- The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

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

Process	:	P1	P2	P3
Burst Time	:	24	3	3
Time Quantum = 4 Milliseconds				

- Using RR scheduling (with time quantum of 4 milliseconds), the processes are scheduled according to the following Gantt Chart:



Waiting Time for P1 = (10 – 4) = 6 Milliseconds

Waiting Time for P2 = 4 Millisecond

Waiting Time for P3 = 7 Milliseconds

**Average Waiting Time = (6 + 4 + 7)/3 = 5.66 Milliseconds**

- In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process).
- If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue.
  - The RR scheduling algorithm is thus preemptive.
- The performance of the RR algorithm depends heavily on the size of the time quantum.

- If the time quantum is extremely large, the RR policy is the same as the FCFS policy.
- If the time quantum is extremely small (say, 1 millisecond), the RR approach is called processor sharing.
- We need also to consider the effect of context switching on the performance of RR scheduling.
- In practice, most modern systems have time quantum ranging from 10 to 100 milliseconds.
- The time required for a context switch is typically less than 10 microseconds.

**Question 1:** Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	:	P1	P2	P3	P4	P5
Burst Time	:	5	24	16	10	3

Determine the average waiting time, average turnaround time and throughput using First-Come, First Served and Shortest Job First scheduling.

**Question 2:** Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	:	P1	P2	P3
Burst Time	:	30	6	8

Determine the average waiting time, average turnaround time and throughput using **Round Robin** scheduling. Assume time quantum = 5 milliseconds.

**Question 3:** Consider the following set of processes with the length of the CPU burst given in milliseconds:

Process	:	P1	P2	P3	P4	P5
Burst Time	:	3	6	4	5	2
Arrival Time	:	0	2	4	6	8

Determine the average waiting time, average turnaround time and throughput using **Shortest Remaining Time First** scheduling.

**Question 4:** Consider the following set of processes with the length of the CPU burst given in milliseconds:

Process	:	P1	P2	P3	P4	P5
Burst Time	:	6	12	1	3	4
Priority	:	2	4	5	1	3

Determine the average waiting time, average turnaround time and throughput using **Priority** scheduling.



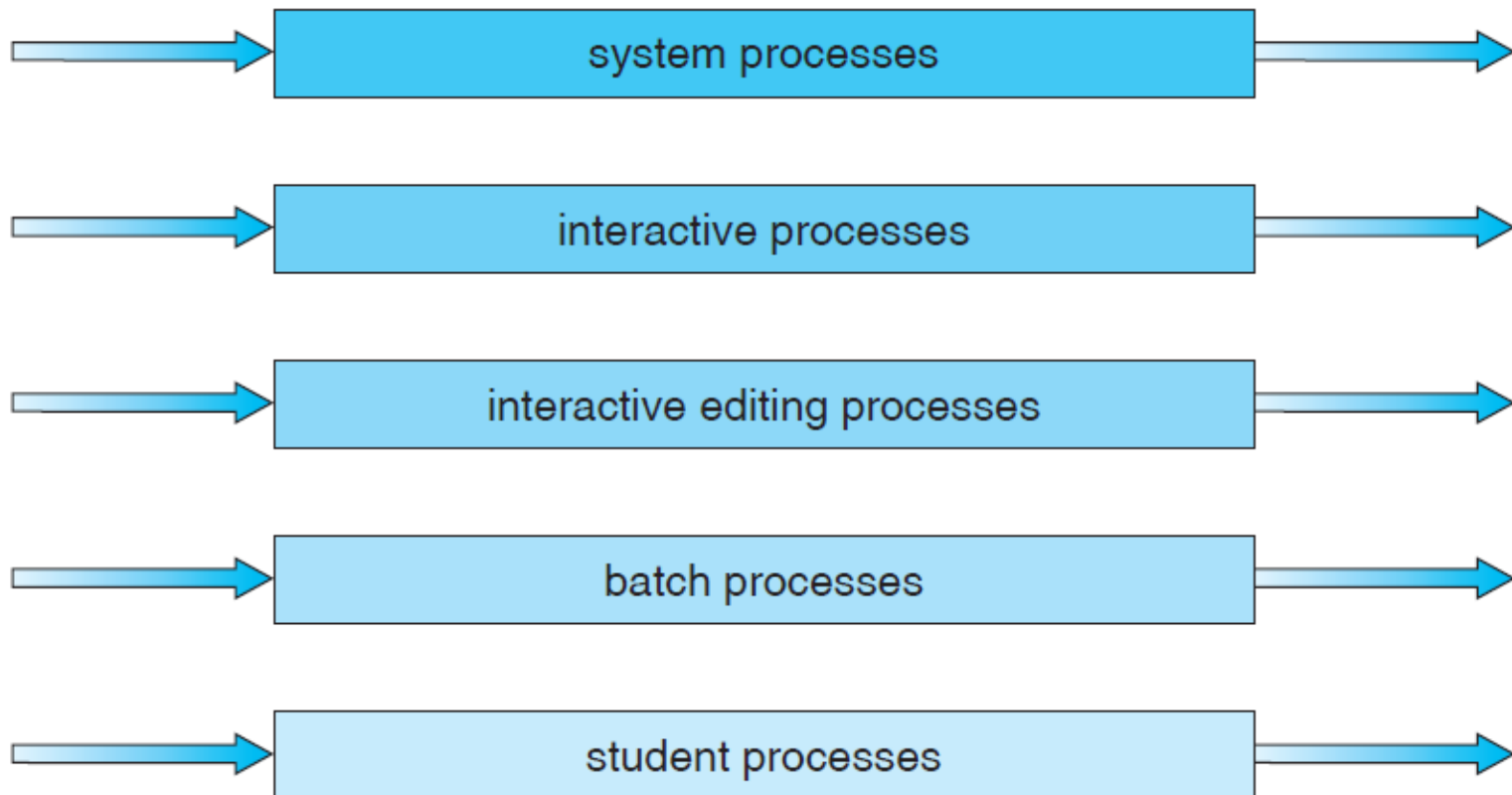
# Multilevel Queue Scheduling

- The processes can be classified into different groups where each group has its own scheduling needs.
- A common classification is:
  - Foreground (Interactive) Processes
  - Background Processes
- These two types of processes have different requirements and so may have different scheduling needs.
- 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 size, process priority, or process type.
  - Each queue has its own scheduling algorithm.

# Multilevel Queue Scheduling (contd...)

- Separate queues might be used for different categories of processes.

highest priority



lowest priority

# Multilevel Queue Scheduling (contd...)

- Scheduling among the queues is commonly implemented as fixed-priority preemptive scheduling.
  - Each queue has absolute priority over lower-priority queues.
  - For example, no process in the batch queue, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty.
  - If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.
- Another possibility is to time-slice among the queues.
  - Each queue gets a certain portion of the CPU time, which it can then schedule among its various processes.

# Multilevel Queue Scheduling: Example

- |                |   |    |    |    |    |
|----------------|---|----|----|----|----|
| ▪ Process      | : | P1 | P2 | P3 | P4 |
| ▪ Arrival Time | : | 0  | 0  | 0  | 10 |
| ▪ Burst Time   | : | 4  | 3  | 8  | 5  |
| ▪ Queue No.    | : | 1  | 1  | 2  | 1  |
- Priority of Queue 1 is greater than Queue 2.
  - Queue 1 uses Round Robin (Time Quantum = 2) and Queue 2 uses First-Come, First-Served.

# Questions

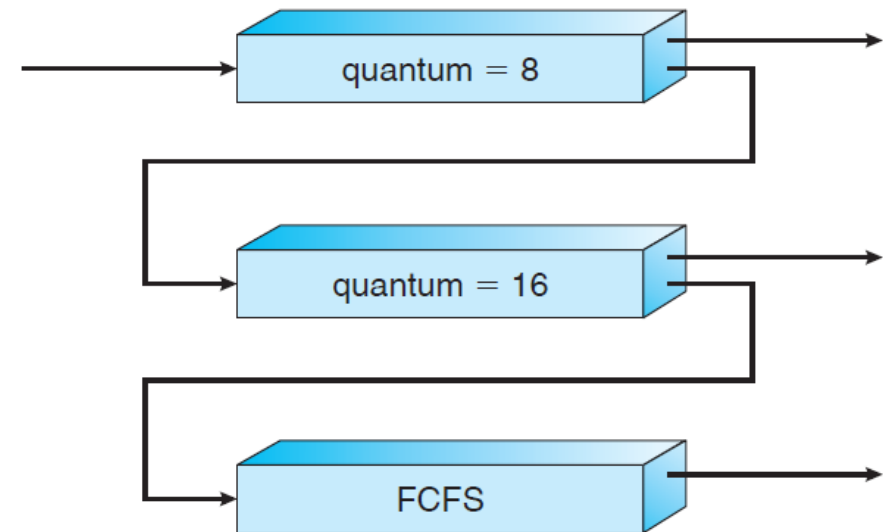
- Three process P1, P2 and P3 arrive at time zero. The total time spent by the process in the system is 10ms, 20ms, and 30ms respectively. They spent first 20% of their execution time in doing I/O and the rest 80% in CPU processing. What is the percentage utilization of CPU using FCFS scheduling algorithm?
- Three process p1, P2 and P3 arrive at time zero. Their total execution time is 10ms, 15ms, and 20ms respectively. They spent first 20% of their execution time in doing I/O, next 60% in CPU processing and the last 20% again doing I/O. For what percentage of time was the CPU free? Use Round robin algorithm with time quantum 5ms.

# Multilevel Feedback Queue Scheduling

- The multilevel feedback queue scheduling algorithm allows a process to move between queues.
  - The idea is to separate processes according to the characteristics of their CPU bursts.
  - If a process uses too much CPU time, it will be moved to a lower-priority queue.
  - A process that waits too long in a lower-priority queue may be moved to a higher-priority queue.

# Multilevel Feedback Queue Scheduling

- A process entering the ready queue is put in **Queue 0**.
- A process in **Queue 0** is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of **Queue 1**.
- If **Queue 0** is empty, the process at the head of **Queue 1** is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into **Queue 2**.
- Processes in **Queue 2** are run on an FCFS basis but are run only when **Queues 0** and **1** are empty.



# Process Synchronization: Background

## Producer

```
while (true) {
    /* produce an item in nextProduced */
    while (counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

## Consumer

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

- Both, the producer and consumer routines are correct separately.
- They may not function correctly when executed concurrently.



# Process Synchronization: Background

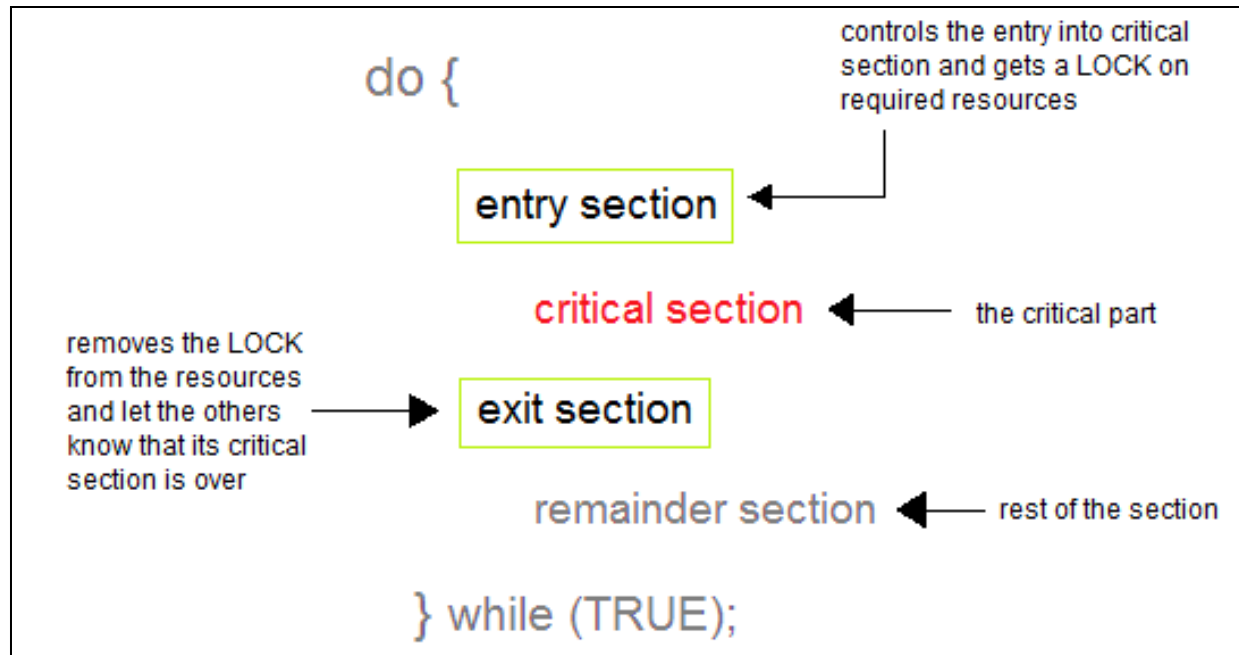
- An **incorrect state** may arrive because **both processes** are allowed to **manipulate the variable counter concurrently**.
- A situation, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.
- In our example, to guard against the race condition, we need to ensure that only one process at a time can manipulate the variable **counter**.
- To make such a guarantee, the processes must be synchronized in some way.

# The Critical-Section Problem

- Consider a system consisting of  $n$  processes  $\{P_0, P_1, \dots, P_{n-1}\}$ .
- Each process has a **segment of code**, called a **critical section**, in which the process may be changing common variables, updating a table, etc.
- When one process is executing in its critical section, no other process is to be allowed to execute in its critical section.
  - No two processes are executing in their critical sections at the same time.

# The Critical-Section Problem (contd...)

- The solution of critical-section problem involves design of a protocol that the processes can use to cooperate.



- Each process must request permission to enter its critical section.
- The section of code implementing this request is the **entry section**.
- The critical section may be followed by an **exit section**.
- The remaining code is the **remainder section**.

# Solution of Critical-Section Problem

- A solution to the critical-section problem must satisfy the following three requirements:
  - **Mutual Exclusion** - If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections.
  - **Progress** - If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections should decide which will enter its critical section next, in a finite time.
  - **Bounded Waiting** - After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, system must grant the process permission to get into its critical section.

# Critical Section Problem: Algorithm 1

This algorithm works only for two processes.

Given: `int turn;` (Shared Variable)

## Process $P_i$

```
turn = i;
do{
    while(turn!=i); <-- ENTRY SECTION
    CRITICAL SECTION
    turn = j; <-- EXIT SECTION
    REMAINDER SECTION
}while(TRUE);
```

# Critical Section Problem: Algorithm 1 (contd...)

Given: `int turn;` (Shared Variable)

Process  $P_i$

```
turn = i;
do{
    while (turn!=i) ;
    CRITICAL SECTION
    turn = j;
    REMAINDER SECTION
}while (TRUE) ;
```

Process  $P_j$

```
turn = j;
do{
    while (turn!=j) ;
    CRITICAL SECTION
    turn = i;
    REMAINDER SECTION
}while (TRUE) ;
```

This algorithm does not satisfy the progress requirement because there is strict alternation between the processes.

# Critical Section Problem: Algorithm 2

This algorithm also works only for two processes.

Given: `boolean flag[2];`    `flag[0] = FALSE;`    `flag[1] = FALSE;`

## Process $P_i$

```
do{  
    flag[i] = TRUE;  
    while(flag[j]); <-- ENTRY SECTION  
        CRITICAL SECTION  
    flag[i] = FALSE; <-- EXIT SECTION  
    REMAINDER SECTION  
}while(TRUE);
```

# Critical Section Problem: Algorithm 2 (contd...)

Given: `boolean flag[2];`    `flag[0] = FALSE;`    `flag[1] = FALSE;`

Process  $P_i$

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

Process  $P_j$

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

This algorithm can fail the **progress** requirement if both processes set their flags to true and then both execute the while loop.



# Critical Section Problem: Peterson's Solution

- Peterson's Solution is a classical software based solution to the critical section problem.

- Peterson's algorithm is used to synchronize two processes.
- Peterson's solution requires the two processes to share two data items:

```
int turn;  
boolean flag[2];
```

- **Variable turn indicates whose turn it is to enter its critical section.**
  - If  $\text{turn} == i$ , then process  $P_i$  is allowed to execute in its critical section.
- **Flag array is used to indicate if a process wants to enter its critical section.**
  - If  $\text{flag}[i] == \text{true}$ , it indicates that  $P_i$  wants to enter its critical section.

# Structure of Process P<sub>i</sub> in Peterson's Solution

```
do {
```

```
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);
```

critical section

```
    flag[i] = FALSE;
```

remainder section

```
} while (TRUE);
```

# Structure of Process $P_i$ and $P_j$

Given:    `boolean flag[2];`                      `int turn;`                      `flag[0]=false;`                      `flag[1]=false;`

## Process $P_i$

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

## Process $P_j$

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

- Peterson's algorithm satisfy the three requirements (**mutual exclusive**, **progress**, **bounded waiting**) of solution of critical section problem.

# Limitations of Peterson's Solution

- Works only for TWO Processes.
- Busy Waiting
  - **Busy waiting**, also known as **spinning**, or **busy looping** is a process synchronization technique in which a process/task waits and constantly checks for a condition to be satisfied before proceeding with its execution.

# Waiting Approaches

- There are two general approaches to waiting in operating systems:
  - **Busy Waiting** - A process/task can continuously check for the condition to be satisfied while **consuming the processor**.
  - **Sleeping (Blocked Waiting or Sleep Waiting)** - A process can wait **without consuming the processor**. In such a case, the process/task is alerted or awakened when the condition is satisfied.

# Busy Waiting

- Busy looping is usually used to achieve mutual exclusion in operating systems.
  - Busy waiting can be inefficient because the looping procedure is a waste of computer resources
  - Although inefficient, busy waiting can be beneficial in mutual exclusion if the waiting time is short and insignificant.
  - Additionally, busy waiting is quick and simple to understand and implement.
- 
- A workaround solution for the inefficiency of busy waiting that is implemented in most operating systems is the use of a delay function.

# Sleep Waiting

- In this case, the process/task is alerted or awakened when the condition is satisfied.
- A **delay function (sleep system call)** places the process involved in busy waiting into an inactive state for a specified amount of time.
  - In this case, resources are not wasted as the process is “asleep”.
  - After the sleep time has elapsed, the process is awakened to continue its execution.
  - If the condition is still not satisfied, the sleep time is incremented until the condition can be satisfied.

# Bakery Algorithm

- Bakery Algorithm is a critical section solution for **n processes**.
- Each process, wanting to enter critical section, gets a **token number**.
  - The token numbering scheme always generates numbers in increasing order of enumeration; i.e., 1, 2, 3, 3, 4, 5, ...
- **A process with lowest token number will enter the critical section.**
  - The algorithm preserves the first come first serve property.
- If two processes have same token number, the process with **lower process id (PId)** will enter the critical section.
- The token number of process is set to 0 when it finishes execution.



```
boolean choosing[N] = {FALSE, ..., FALSE};  
int number[N] = {0, ..., 0};
```

## Process $P_i$

```
do {  
    choosing[i] = TRUE;  
    number[i] = MAX(number[0], number[1], ..., number[n-1]) + 1;  
    choosing[i] = FALSE;  
    for (j = 0; j < n; j++) {  
        while (choosing[j]);  
        while ((number[j] != 0) && ((number[j],j) < (number[i],i)));  
    }  
    CRITICAL SECTION  
    number[i] = 0;  
    REMAINDER SECTION  
}while(TRUE);
```

# Solution using Lock

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (TRUE);
```

```
do {  
    while (Lock != 0);  
    Lock = 1;  
    CRITICAL SECTION  
    Lock = 0;  
    REMAINDER SECTION  
}while(TRUE);
```

## ■ Uniprocessor Environment

- Critical-section problem could be solved if interrupts could be prevented from occurring (while a shared variable was being modified).
  - This is often the approach taken by non-preemptive kernels.

## ■ Multiprocessor Environment

- Disabling interrupts on a multiprocessor can be time consuming.
- With special **atomic hardware instructions**, solution of critical-section problem can be done.
  - TestAndSet()
  - Swap()

# TestAndSet()

## Definition of TestAndSet()

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

It is executed atomically  
(that is, as one  
uninterruptible unit)

If two TestAndSet() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.

## Implementation of TestAndSet()

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

This algorithm satisfies the mutual-exclusion requirement, but do not satisfy the bounded-waiting requirement.

# Swap

## Definition of Swap()

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

It is executed atomically  
(that is, as one  
uninterruptible unit)

## Implementation of Swap()

```
do {  
    key = TRUE;  
    while (key == TRUE)  
        Swap(&lock, &key);  
  
    // critical section  
  
    lock = FALSE;  
  
    // remainder section  
} while (TRUE);
```

This algorithm satisfies the mutual-exclusion requirement, but do not satisfy the bounded-waiting requirement.

# Semaphore

- In 1965, proposed by Dijkstra, Semaphore is a mechanism that can be used to provide synchronization of tasks (to solve critical-section problem).
- A semaphore  $S$  is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: `wait()` and `signal()`.
- The initial value of  $S$  depends on how many processes are allowed in CS.

## `wait()`

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

The testing of the integer value of  $S$  ( $S \leq 0$ ), as well as its possible modification ( $S--$ ), must be executed without interruption.

## `signal()`

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

# Types of Semaphores

## ■ Binary Semaphore

- It is a special form of semaphore used for implementing mutual exclusion, hence it is often called a **Mutex Lock**.
- A binary semaphore is initialized to 1 and only takes the values 0 and 1 during execution of a program.
- It can be used to deal with the critical-section problem for multiple processes.

# Types of Semaphores

- Counting Semaphore

- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
- It is initialized to the number of resources available.
- Each process that wishes to use a resource performs a `wait()` operation on the semaphore (`decrementing the count`).
- When a process releases a resource, it performs a `signal()` operation (`incrementing the count`).
- When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.



## Mutual-Exclusion Implementation with Semaphores

```
do {  
    wait(S);  
    CRITICAL SECTION  
    signal(S);  
    REMAINDER SECTION  
}while(TRUE)
```

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

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

- The main disadvantage of the semaphore definition given here is that it requires busy waiting.
  - While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.

# Spinlock

- Busy waiting wastes CPU cycles that some other process might be able to use productively.
- The semaphore having busy waiting mechanism is also called a **spinlock** because the process “spins” while waiting for the lock.
- Spinlock mechanism has an advantage in that no context switch is required when a process waits on a lock
  - When locks are expected to be held for short times, spinlocks are useful.
  - Spinlocks are often employed on multiprocessor systems where one thread can “spin” on one processor while another thread performs its critical section on another processor.

# Solution of Spinlock (Busy Waiting)

- To overcome the need for busy waiting, we can modify the definition of the `wait()` and `signal()` semaphore operations.
- When a process executes the `wait()` operation and finds that the semaphore value is not positive, it must wait.
  - However, rather than engaging in busy waiting, the process can block itself.
  - The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

# Solution of Spinlock (Busy Waiting)

- To overcome the need for busy waiting, we can modify the definition of the `wait()` and `signal()` semaphore operations.

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

The `block()` operation suspends the process that invokes it. The `wakeup(P)` operation resumes the execution of a blocked process `P`. These two operations are provided by the operating system as basic system calls.

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

# Solution of Spinlock (Busy Waiting)

- In the solution of busy waiting, semaphore values may be negative.
- Semaphore values are never negative under the classical definition of semaphores with busy waiting.
- If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.
- The list of waiting processes can be easily implemented by a link field in each process control block (PCB).
- To ensure bounded waiting, a FIFO queue or any other queuing strategy may be used.

# Limitations of Semaphores

- Deadlocks

- Consider a system consisting of two processes, P0 and P1, each accessing two semaphores, S and Q, set to the value 1:

$P_0$	$P_1$
<code>wait(S);</code>	<code>wait(Q);</code>
<code>wait(Q);</code>	<code>wait(S);</code>
<code>.</code>	<code>.</code>
<code>.</code>	<code>.</code>
<code>.</code>	<code>.</code>
<code>signal(S);</code>	<code>signal(Q);</code>
<code>signal(Q);</code>	<code>signal(S);</code>

- Suppose that P0 executes `wait(S)` and then P1 executes `wait(Q)`.
- When P0 executes `wait(Q)`, it must wait until P1 executes `signal(Q)`.
- Similarly, when P1 executes `wait(S)`, it must wait until P0 executes `signal(S)`.
- Since these `signal()` operations cannot be executed, P0 and P1 are **deadlocked**.

# Limitations of Semaphores

- Indefinite Blocking or Starvation
  - Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

# Classic Problems of Synchronization

- Bounded-Buffer Problem
- Readers–Writers Problem
- Dining-Philosophers Problem

**Solution using Semaphore**



# Bounded-Buffer Problem

Producer

```
while (true) {  
    /* produce an item in nextProduced */  
    while (counter == BUFFER_SIZE)  
        ; /* do nothing */  
    buffer[in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

Consumer

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

- The bounded-buffer problem is also known as producer-consumer problem.
- Both, the producer and consumer routines are correct separately.
- They may not function correctly when executed concurrently.

- Assume,
  - **Buffer Size = n slots**
  - **semaphore mutex** (for mutual exclusion)  
mutex = 1
  - **semaphore empty** (number of empty slots in buffer)  
empty = n
  - **semaphore full** (number of full slots in buffer)  
full = 0

# Bounded-Buffer Problem - Solution

## Producer Process

```
do {  
    . . .  
    // produce an item in nextp  
    . . .  
    wait(empty);  
    wait(mutex);  
    . . .  
    // add nextp to buffer  
    . . .  
    signal(mutex);  
    signal(full);  
} while (TRUE);
```

## Consumer Process

```
do {  
    wait(full);  
    wait(mutex);  
    . . .  
    // remove an item from buffer to nextc  
    . . .  
    signal(mutex);  
    signal(empty);  
    . . .  
    // consume the item in nextc  
    . . .  
} while (TRUE);
```

# Readers–Writers Problem

- Consider a situation where a shared resource (such as file or dataset) is to be accessed by multiple concurrent processes.
- Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.
- We distinguish between these two types of processes - **reader** and **writer**.
  - If two readers access the shared data simultaneously, no adverse effects will result.
  - If a writer and some other process (either a reader or a writer) access the database simultaneously, inconsistency may occur.
- This synchronization problem is referred to as the readers–writers problem.

# Readers–Writers Problem - Solution

- Assume,

- `int readcount = 0`

The readcount variable keeps track of how many processes are currently reading the object.

- `semaphore mutex`

mutex is used for mutual exclusion when the variable readcount is updated.

`mutex = 1`

- `semaphore wrt`

wrt functions as a mutual-exclusion semaphore for the writers.

`wrt = 1`

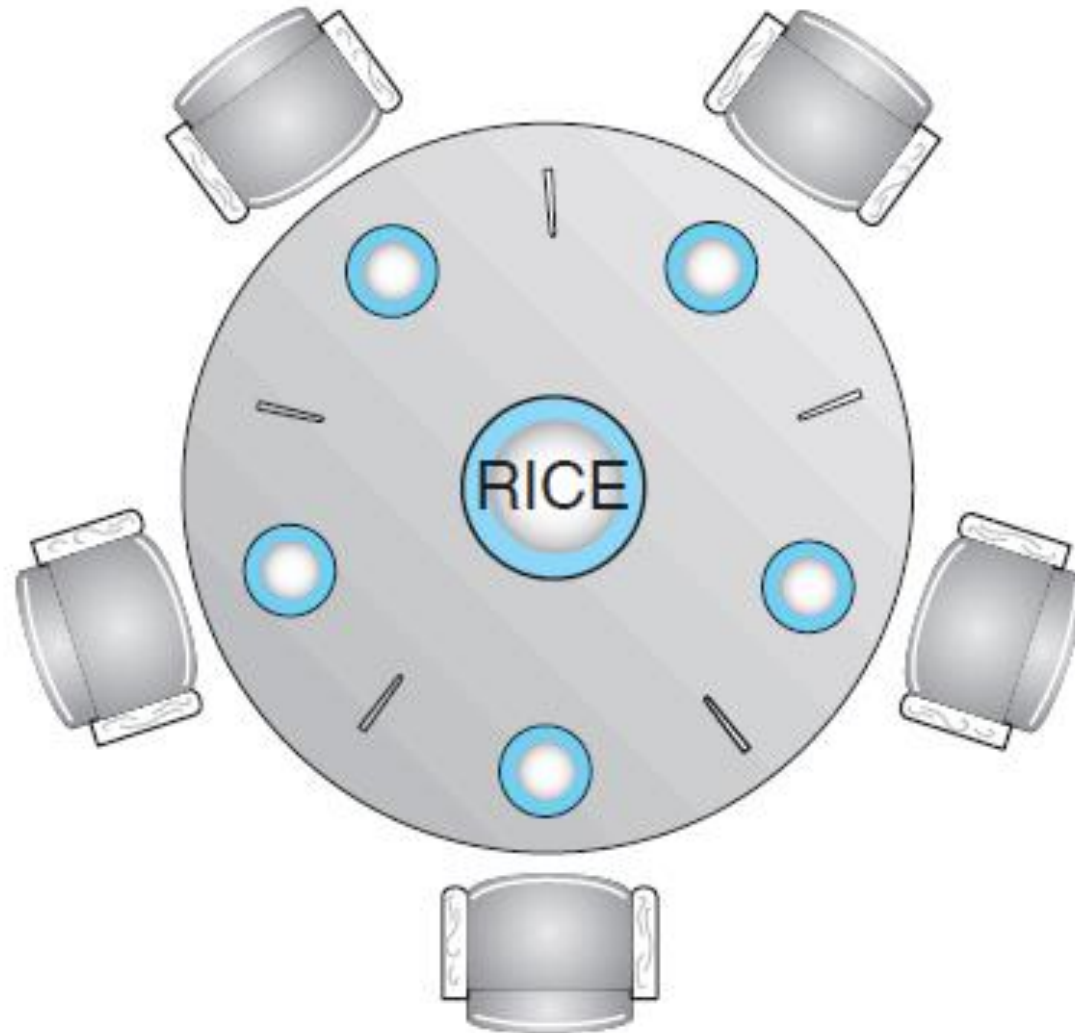
## Write Process(s)

```
do {  
    wait(wrt);  
    . . .  
    // writing is performed  
    . . .  
    signal(wrt);  
} while (TRUE);
```

## Reader Process(s)

```
do {  
    wait(mutex);  
    readcount++;  
    if (readcount == 1)  
        wait(wrt);  
    signal(mutex);  
    . . .  
    // reading is performed  
    . . .  
    wait(mutex);  
    readcount--;  
    if (readcount == 0)  
        signal(wrt);  
    signal(mutex);  
} while (TRUE);
```

# Dining-Philosophers Problem



# Dining-Philosophers Problem

- Consider a situation where five philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.
- They are in thinking-eating cycle.
- In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.
- A philosopher may pick up only one chopstick at a time. (Obviously, he/she cannot pick up a chopstick that is already in the hand of a neighbor).
- From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to his/her (the chopsticks that are between his/her and his/her left and right neighbors).
- The problem is how to choose two chopsticks (one his/her own and one of other)



# Dining-Philosophers Problem - Solution

- The Dining-Philosopher Problem is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.
- The problem can be solved by representing each chopstick with a semaphore.
  - A philosopher tries to grab a chopstick by executing a **wait()** operation on that semaphore.
  - He/she releases his/her chopsticks by executing the **signal()** operation on the appropriate semaphores.
  - **semaphore chopstick[5] = 1;**

# Dining-Philosophers Problem - Solution

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

# Limitations of Solution

- Although this solution guarantees that no two neighbors are eating simultaneously, **but it could create a deadlock.**
  - Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick.
  - All the elements of chopstick will now be equal to 0.
  - When each philosopher tries to grab her right chopstick, she will be delayed forever.
- Possible Solutions:
  - Allow at most four philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up her chopsticks only if both chopsticks are available.