Operating Systems

Unit-III

Inter-process Communication: Race conditions, Critical Regions, Mutual exclusion with busy waiting, Sleep and wakeup, Semaphores, Mutexes, Monitors, Message passing, Classical IPC Problems - Dining philosophers problem, Readers and writers problem.

Deadlocks: Resources, Conditions for resource deadlocks, Ostrich algorithm, Deadlock detection and recovery, Deadlock avoidance, Deadlock prevention.

INTERPROCESS COMMUNICATION

- **Processes frequently need to communicate with other processes. For example, in a shell pipeline, the output of the first process must be passed to the second process.**

  - How one process can pass information to another?
  - Making sure two or more processes do not get in each other's way
  - Poper sequencing when dependencies are present

INTERPROCESS COMMUNICATION

Race Conditions

In some operating systems, processes that are working together may share some common storage that each one can read and write. The shared storage may be in main memory (possibly in a kernel data structure) or it may be a shared file.

To see how interprocess communication works in practice, consider below example

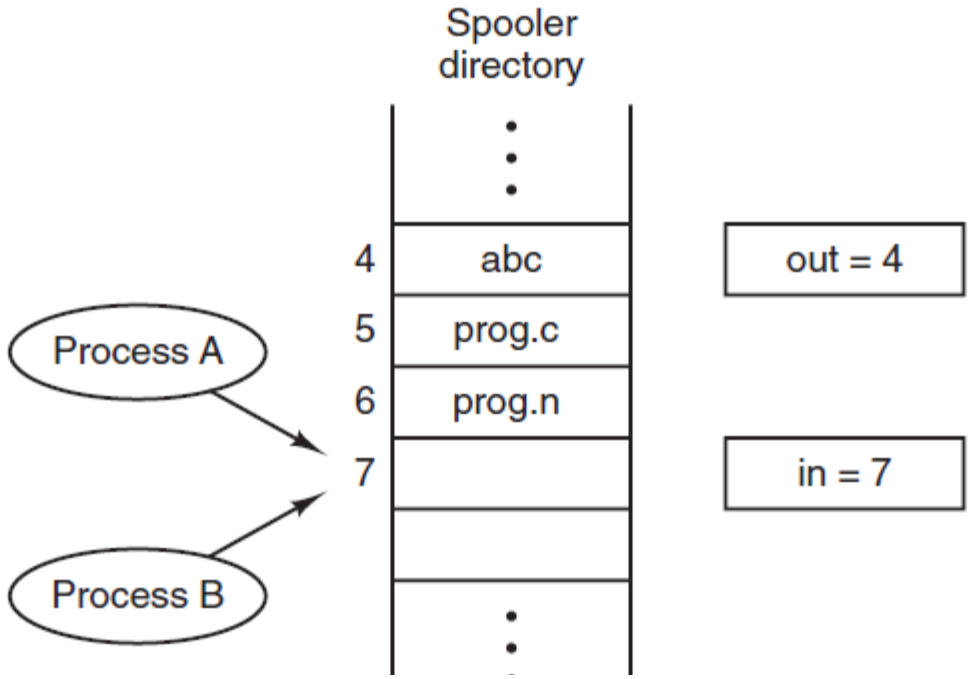
Example: print spooler

The print spooler is an executable file that **manages the printing process**. Management of printing involves retrieving the location of the correct printer driver, loading that driver, spooling high-level function calls into a print job, scheduling the print job for printing, and so on.

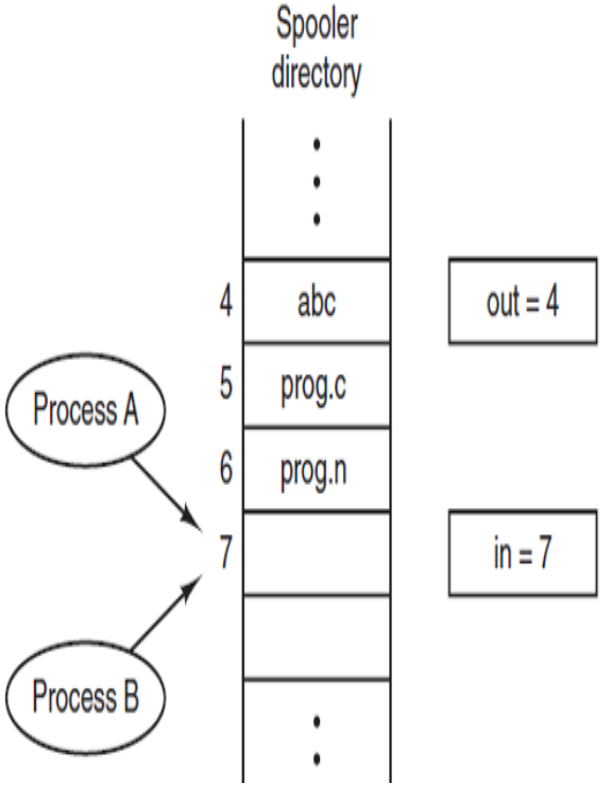
- When a process wants to print a file, it enters the file name in a special **spooler directory (C:\Windows\System32\spool\PRINTERS)**

- Another process, the **printer daemon**, periodically checks to see if there are any files to be printed, and if there are, it prints them and then removes their names from the directory.

Imagine that our spooler directory has a very large number of slots, numbered 0, 1, 2, ..., each one capable of holding a file name. Also imagine that there are two shared variables, *out*, which points to the next file to be printed, and *in*, which points to the next free slot in the directory. These two variables might well be kept in a two-word file available to all processes. At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files queued for printing). More or less simultaneously, processes *A* and *B* decide they want to queue a file for printing. This situation is shown in Fig.



Two processes want to access shared memory at the same time



In jurisdictions where Murphy's law† is applicable, the following could happen. Process *A* reads *in* and stores the value, 7, in a local variable called *next_free_slot*. Just then a clock interrupt occurs and the CPU decides that process *A* has run long enough, so it switches to process *B*. Process *B* also reads *in* and also gets a 7. It, too, stores it in *its* local variable *next_free_slot*. At this instant both processes think that the next available slot is 7.

Process *B* now continues to run. It stores the name of its file in slot 7 and updates *in* to be an 8. Then it goes off and does other things.

Eventually, process *A* runs again, starting from the place it left off. It looks at *next_free_slot*, finds a 7 there, and writes its file name in slot 7, erasing the name that process *B* just put there. Then it computes *next_free_slot* + 1, which is 8, and sets *in* to 8. The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process *B* will never receive any output. User *B* will hang around the printer for years, wistfully hoping for output that

Race conditions

- A race condition is an undesirable situation that occurs when a processes, or threads attempt to access the same resource at the same time and cause problems in the system.
- Race conditions are considered a common issue for multithreaded applications.

Critical Regions

How to avoid race conditions?

- find some way to prohibit more than one process from reading and writing the shared data at the same time- **“ Mutual Exclusion”.**

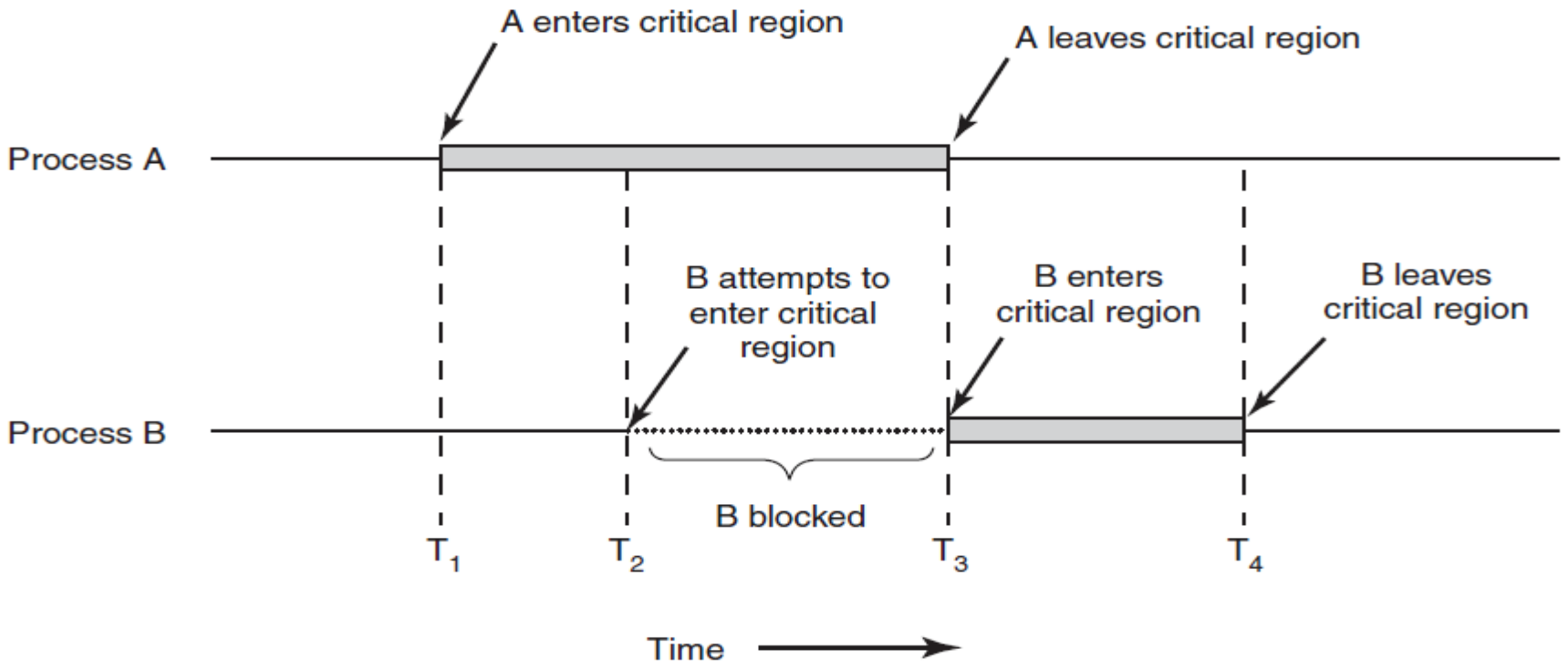
Mutual Exclusion: if one process is using a shared variable or file, the other processes will be excluded from doing the same thing.

- The part of the program where the shared memory is accessed is called the **critical region or critical section.**
- If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid races.

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

Critical Regions

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



Mutual exclusion using critical regions.

Mutual Exclusion with Busy Waiting

Disabling Interrupts

- On a single-processor system, the simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it.
- This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts.
- On the other hand, it is frequently convenient for the kernel itself to disable interrupts for a few instructions while it is updating variables or especially lists.
- **Disabling interrupts is often a useful technique within the operating system itself but is not appropriate as a general mutual exclusion mechanism for user processes.**

Mutual Exclusion with Busy Waiting

Lock Variables

- It is a software solution.

- Consider having a single, shared (lock) variable, initially 0. When a process wants to enter its critical region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.
- **Drawback: Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time.**

Mutual Exclusion with Busy Waiting

Strict Alternation

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

(a)

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

(b)

A proposed solution to the critical-region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

The integer variable ***turn***, initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory. Initially, process 0 inspects ***turn***, finds it to be 0, and enters its critical region. Process 1 also finds it to be 0 and therefore sits in a tight loop continually testing turn to see when it becomes 1. Continuously testing a variable until some value appears is called busy waiting. It should usually be avoided, since it wastes CPU time. Only when there is a reasonable expectation that the wait will be short is busy waiting used. A lock that uses busy waiting is called a spin lock.

Mutual Exclusion with Busy Waiting

Peterson's Solution

- It is a software based solution.

Peterson's solution requires the two processes to share two data items:

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

- The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process *Pi* is allowed to execute in its critical section. The flag array is used to indicate if a process is ready to enter its critical section. For example, if flag[i] is true, this value indicates that *Pi* is ready to enter its critical section.
- To enter the critical section, process *Pi* first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately. The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

The structure of process P_i in Peterson's solution

Sleep and Wakeup

- Peterson's solution is correct, but it has the defect of requiring busy waiting.
- when a process wants to enter its critical region, it checks to see if the entry is allowed. If it is not, the process just sits in a tight loop waiting until it is.
- This approach waste CPU time, but it can also have unexpected effects.
- some interprocess communication primitives that block instead of wasting CPU time when they are not allowed to enter their critical regions. One of the simplest is the pair **sleep and wakeup.**

- **Sleep** is a system call that causes the caller to block, that is, be suspended until another process wakes it up.
- The **wakeup** call has one parameter, the process to be awakened.
- Alternatively, both sleep and wakeup each have one parameter, a memory address used to match up sleeps with wakeups.

Producer-Consumer problem

- The Producer-Consumer problem is a classical multi-process synchronization problem, that is we are trying to achieve synchronization between more than one process.
- There is one Producer in the producer-consumer problem, Producer is producing some items, whereas there is one Consumer that is consuming the items produced by the Producer. The same memory buffer is shared by both producers and consumers which is of fixed-size.
- The task of the Producer is to produce the item, put it into the memory buffer, and again start producing items. Whereas the task of the Consumer is to consume the item from the memory buffer.

•The producer should produce data only when the buffer is not full. In case it is found that the buffer is full, the producer is not allowed to store any data into the memory buffer.

•Data can only be consumed by the consumer if and only if the memory buffer is not empty. In case it is found that the buffer is empty, the consumer is not allowed to use any data from the memory buffer.

•Accessing memory buffer should not be allowed to producer and consumer at the same time.

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

void producer(void)
{
        int item;

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


void consumer(void)
{
        int item;

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

The producer-consumer problem with a fatal race condition.

Semaphores

- Using an integer variable to count the number of wakeups saved for future use
- A semaphore could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending.
- Two operations on semaphores: **down and up(or wait and signal).**
- The **down operation** on a semaphore checks to see if the value is greater than 0. If so, it decrements the value (i.e., uses up one stored wakeup) and just continues. If the value is 0, the process is put to sleep without completing the down for the moment.
- The **up operation** increments the value of the semaphore addressed. If one or more processes were sleeping on that semaphore, unable to complete an earlier down operation, one of them is chosen by the system (e.g., at random) and is allowed to complete its down.
- Thus, after an up on a semaphore with processes sleeping on it, the semaphore will still be 0, but there will be one fewer process sleeping on it.

Semaphores

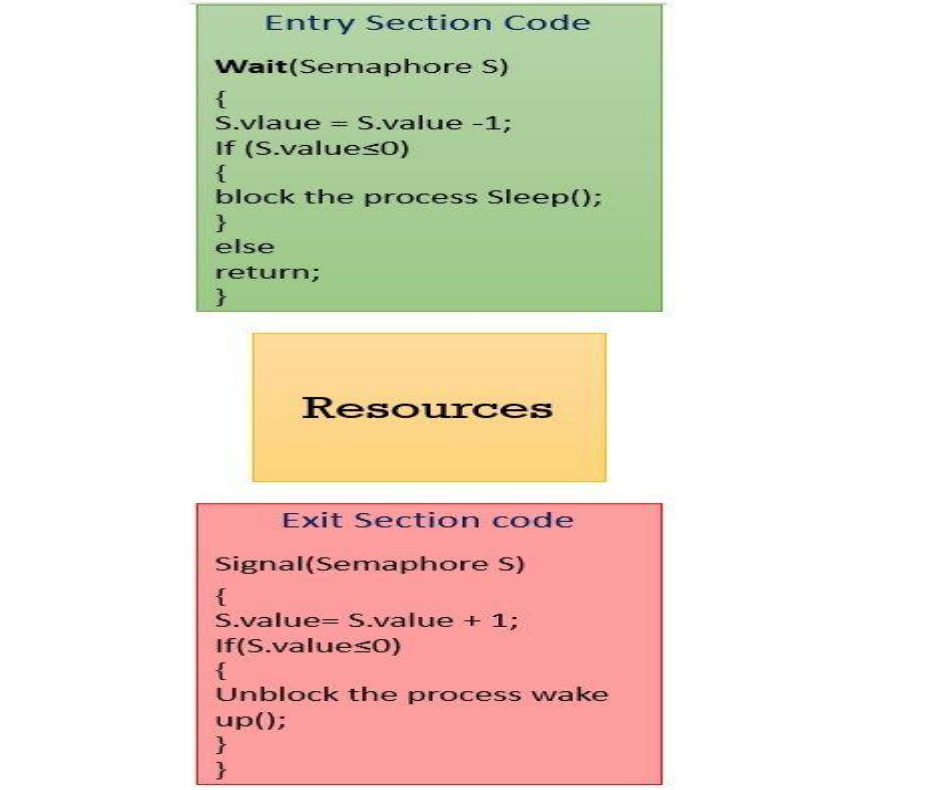
- Semaphore in OS is an integer value that indicates whether the resource required by the process is available or not. The value of a semaphore is modified by wait() or signal() operation where the wait() operation decrements the value of semaphore and the signal() operation increments the value of the semaphore.
- The wait() operation is performed when the process wants to access the resources and the signal() operation is performed when the process want to release the resources. The semaphore can be binary semaphore or the counting semaphore.
- Semaphore is a process synchronization tool that prevents race condition that may occur when multiple cooperative processes try to access the same resources. Two or more process can synchronise by means of the signal. This means the process, trying to access the same shared resource can be forced to stop at a specific place until it is signalled to access the resource.

Semaphores

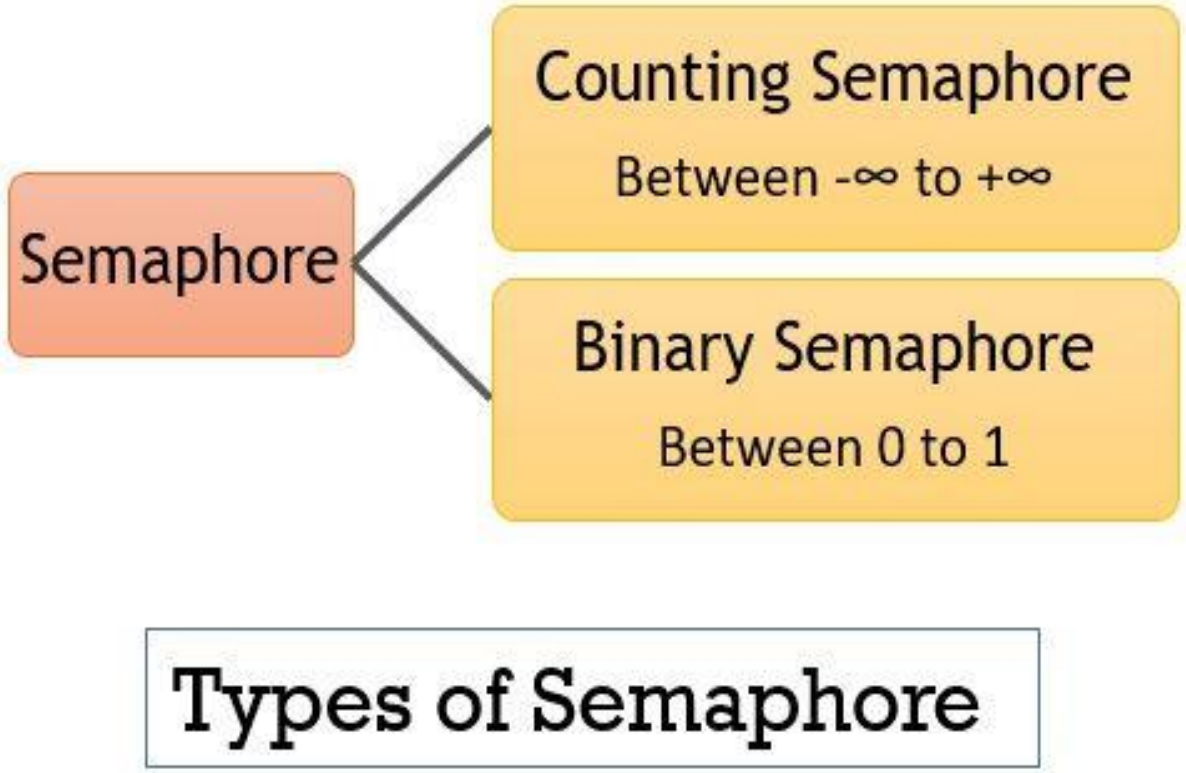
- The wait() operation decrements the value of semaphore and if the semaphore value becomes negative then the process executing wait() would be blocked. The signal() operation increments the value of semaphore and if the semaphore value is less than or equal to 0 then a process blocked by wait() is unblocked.
- The wait() and signal() operation must be performed indivisibly i.e. if a process is modifying the semaphore value no other process must simultaneously modify the semaphore value. Along with that the execution of wait() and signal() must not be interrupted in between.

Implementation of Semaphore

- Any processes that want to access the shared resources have to first execute the entry section where the wait() operation decrements the value of the semaphore. If the value of the semaphore is 0 that means all the resources are in use and now onwards the processes trying to execute the entry section would be blocked.



Implementation of Semaphore



CountingSemaphore

The counting semaphores controls access to resources that have finite instances. Therefore the value of counting semaphore is not restricted to a certain domain.

BinarySemaphores

Binary semaphores are generally used to access the critical section. As we know that only one process can enter a critical section at a time therefore the value of binary semaphores ranges over 0 to 1.

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

void producer(void)
{
        int item;

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


void consumer(void)
{
        int item;

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

Mutexes

- When the semaphore's ability to count is not needed, a simplified version of the semaphore, called a mutex, is sometimes used.
- Mutexes are good only for managing mutual exclusion to some shared resource or piece of code. They are easy and efficient to implement, which makes them especially useful in thread packages that are implemented entirely in user space.
- A mutex is a shared variable that can be in one of two states: unlocked or locked.
- 0 meaning unlocked and all other values meaning locked.
- Two procedures are used with mutexes. When a thread (or process) needs access to a critical region, it calls **mutex_lock(acquire()).** if the mutex is already locked, the calling thread is blocked until the thread in the critical region is finished and calls **mutex_unlock(release()).**
- We use the mutex lock to protect critical regions and thus prevent race conditions.
- Process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section.

The definition of `acquire()` is as follows:

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;;
}
```

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

Solution to the critical-section problem using mutex locks.

Monitors

CONT…

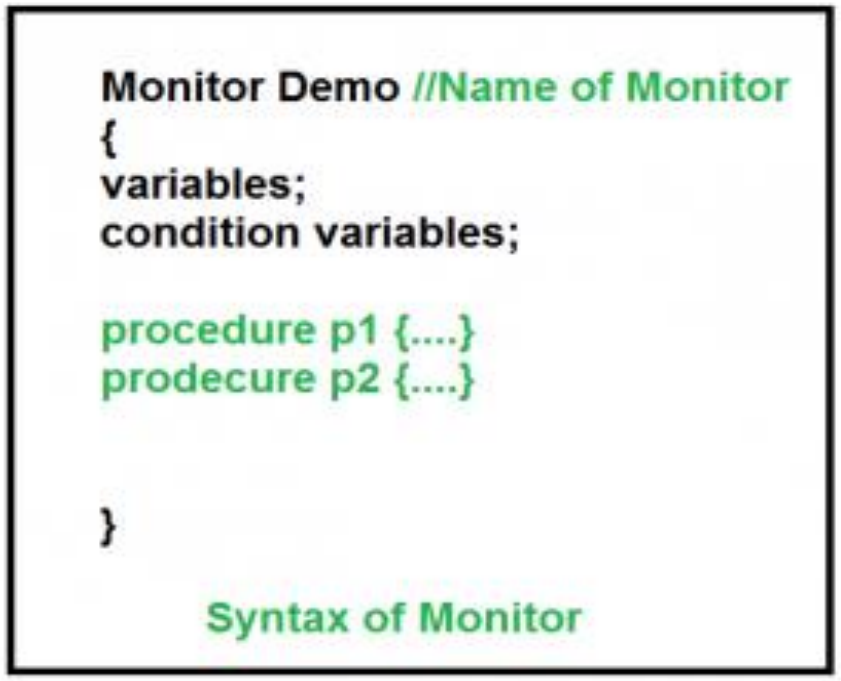
- The monitor is one of the ways to achieve Process synchronization. The monitor is supported by programming languages to achieve mutual exclusion between processes. For example Java Synchronized methods. Java provides wait() and notify() constructs.

- It is the collection of condition variables and procedures combined together in a special kind of module or a package.
- Characteristics of Monitors.

1.Inside the monitors, we can only execute one process at a time.
2.Monitors are the group of procedures, and condition variables that are merged together in a special type of module.
3. If the process is running outside the monitor, then it cannot access the monitor's internal variable. But a process can call the procedures of the monitor.
4. Monitors offer high-level of synchronization
5. Monitors were derived to simplify the complexity of synchronization problems.
6. There is only one process that can be active at a time inside the monitor.

Monitors

Syntax of monitor



Condition Variables

There are two types of operations that we can perform on the condition variables of the monitor:

1.Wait

2.Signal

Wait Operation

a.wait(): - The process that performs wait operation on the condition variables are suspended and locate the suspended process in a block queue of that condition variable.

Signal Operation

a.signal() : - If a signal operation is performed by the process on the condition variable, then a chance is provided to one of the blocked processes.

Monitors

Advantages

1.Mutual exclusion is automatic in monitors.
2.Monitors are less difficult to implement than semaphores.
3.Monitors may overcome the timing errors that occur when semaphores are used.
4.Monitors are a collection of procedures and condition variables that are combined in a special type of module.

DisAdvantages

1.Monitors must be implemented into the programming language.
2.The compiler should generate code for them.
3.It gives the compiler the additional burden of knowing what operating system features is available for controlling access to crucial sections in concurrent processes.

Cooperative and Independent processes

- Processes that executing concurrently in the operating system may be either **independent processes or cooperating processes**. Any process that does not share any data with any other process is **independent**.
- If a process cannot affect or be affected by the other processes executing in the system then the process is said to be independent. So any process that does not share any data with any other process is independent.
- A process is said to be cooperating if it can affect or be affected by the other processes executing in the system. So it is clear that, any process which shares its data with other processes is a cooperating process.
- A cooperating process shares data with another.

Advantages of Cooperating Process in Operating System

Information Sharing: Cooperating processes can be used to share information between various processes. It could involve having access to the same files. A technique is necessary so that the processes may access the files concurrently.

Modularity: Modularity refers to the division of complex tasks into smaller subtasks. Different cooperating processes can complete these smaller subtasks. As a result, the required tasks are completed more quickly and efficiently.

Computation Speedup: Cooperating processes can be used to accomplish subtasks of a single task simultaneously. It improves computation speed by allowing the task to be accomplished faster. Although, it is only possible if the system contains several processing elements.

Convenience: There are multiple tasks that a user requires to perform, such as printing, compiling, editing, etc. It is more convenient if these activities may be managed through cooperating processes.

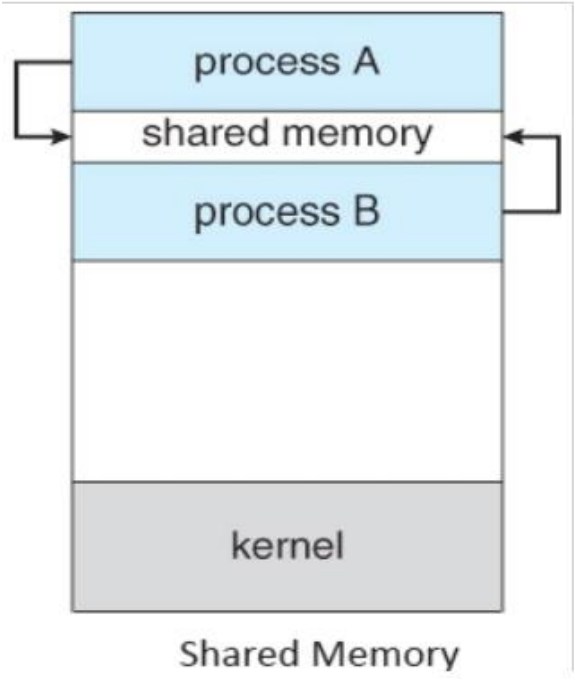
Communication models of cooperative processes (Inter Process Communication)

Two basic models of inter-process communication are

- **shared memory**
- **message passing**

Shared Memory

- A region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.
- Processes can then exchange their information by reading and writing data to the shared region.
- It is easier to implement in a distributed system than shared memory.
- Once shared memory is initiated, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.



Shared Memory

Message Passing

In message-passing systems, processors communicate with one another by sending and receiving messages over a communication channel.

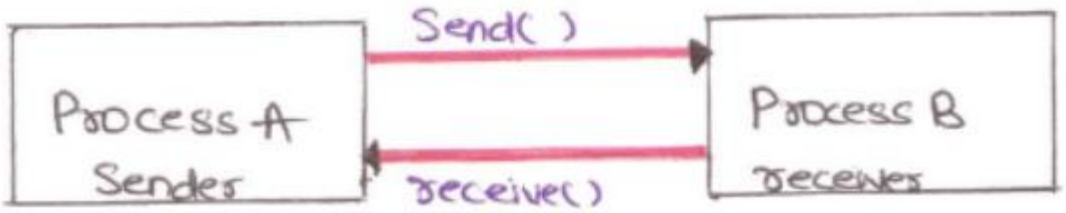
Message passing provides two operations which are as follows –

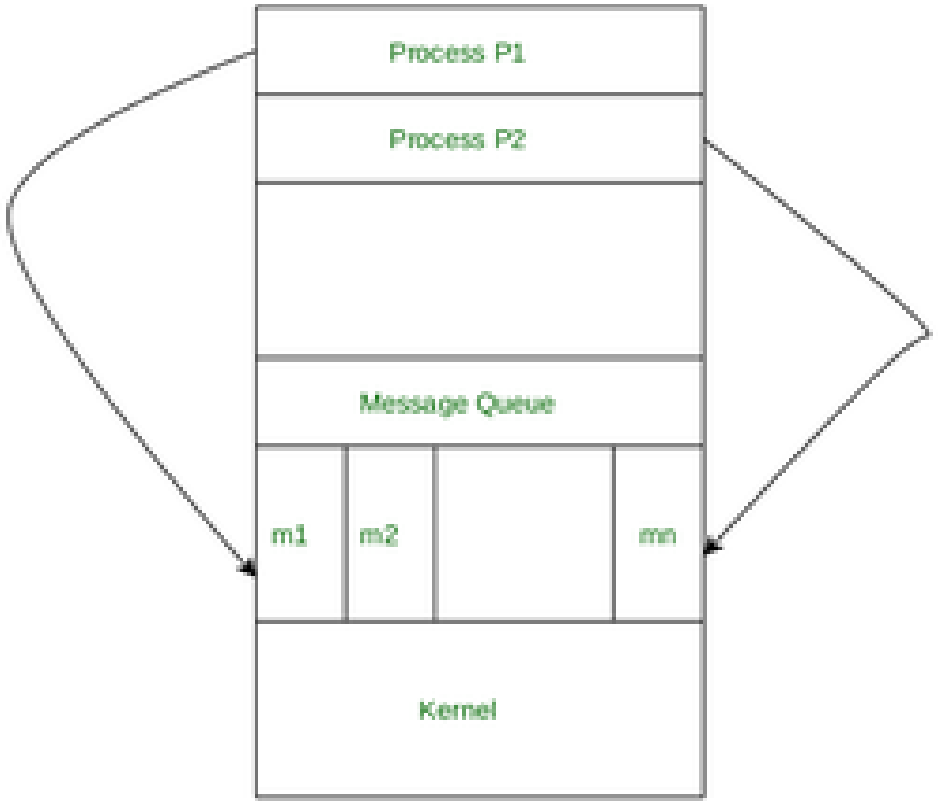
- Send message
- Receive message

```
send(destination, &message);

receive(source, &message);
```

- The former call sends a message to a given destination and the latter one receives a message from a given source (or from *ANY*, if the receiver does not care). If no message is available, the receiver can block until one arrives





Message Passing

Advantages:

1.Easier to implement.
2.Quite tolerant of high communication latencies.
3.Easier to build massively parallel hardware.
4.It is more tolerant of higher communication latencies.
5.Message passing libraries are faster and give high performance.

Disadvantages of Message Passing Model :

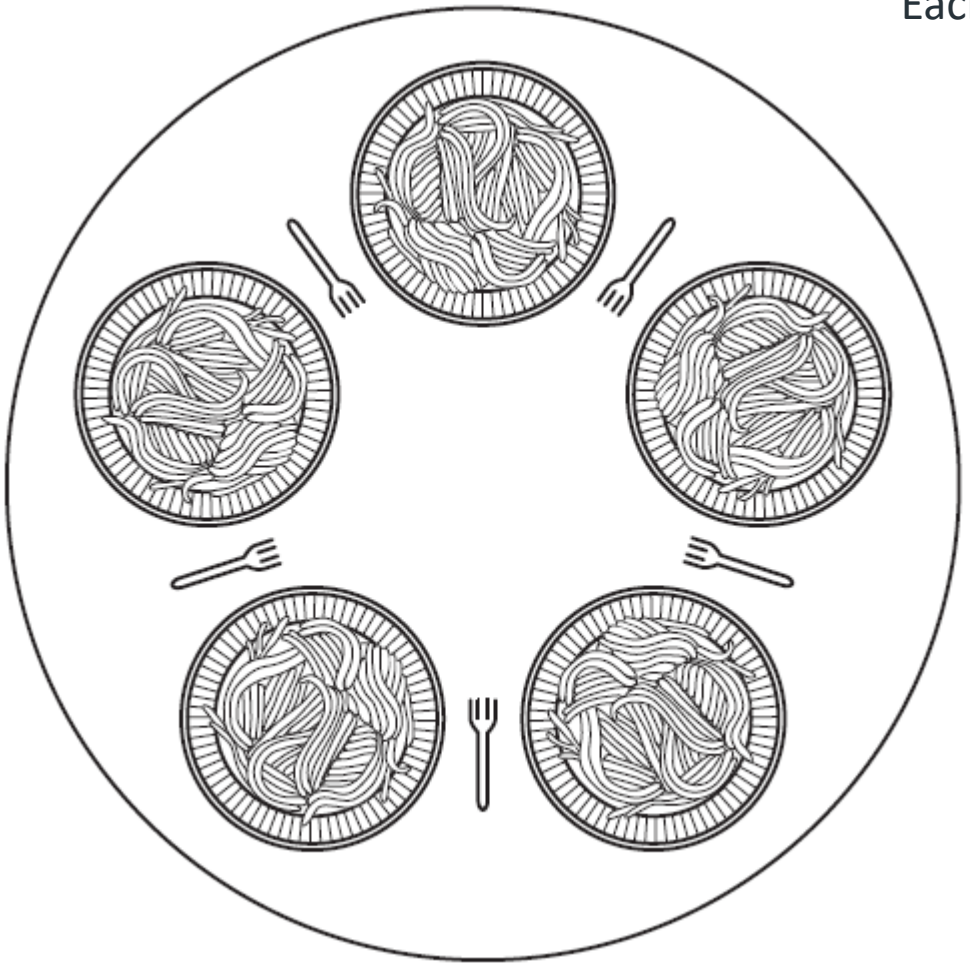
1.Programmer has to do everything.
2.Connection setup takes time that's why it is slower.
3.Data transfer usually requires cooperative operations which can be difficult to achieve.
4.It is difficult for programmers to develop portable applications using this model because message-passing implementations commonly comprise a library of subroutines that are embedded in source code. Here again, the programmer has to do everything on his own.

CLASSICAL IPC PROBLEMS

1.The Dining Philosophers Problem
2.The Readers and Writers Problem

Dining Philosophers Problem

The dining philosophers problem states that there are 5 philosophers sharing a circular table and they eat and think alternatively. There is a bowl of rice for each of the philosophers and 5 chopsticks. A philosopher needs both their right and left chopstick to eat. A hungry philosopher may only eat if there are both chopsticks available.Otherwise a philosopher puts down their chopstick and begin thinking again.



Semaphore Solution to Dining Philosopher

Each philosopher is represented by the following pseudocode:

```
process P[i]
 while true do
  {  THINK;
     PICKUP(CHOPSTICK[i], CHOPSTICK[i+1 mod 5]);
     EAT;
     PUTDOWN(CHOPSTICK[i], CHOPSTICK[i+1 mod 5])
  }
```

READERS WRITERS PROBLEM

The readers-writers problem is a classical problem of process synchronization, it relates to a data set such as a file that is shared between more than one process at a time.

•If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
•However if some person is reading the file, then others may read it at the same time.
Precisely in OS we call this situation as the **readers-writers problem.**

The readers-writers problem is used for managing synchronization among various reader and writer process so that there are no problems with the data sets, i.e. no inconsistency is generated.

READERS WRITERS PROBLEM

possibility of reading and writing

Case	Process 1	Process 2	Allowed/Not Allowed
Case 1	Writing	Writing	Not Allowed
Case 2	Writing	Reading	Not Allowed
Case 3	Reading	Writing	Not Allowed
Case 4	Reading	Reading	Allowed

Semaphore Solution to reader writer problem

Three variables are used: **mutex, wrt, readcnt**

1.semaphore mutex; // semaphore **mutex** is used to ensure mutual exclusion when **readcnt** is updated i.e. when any reader enters or exit from the critical section
2.semaphore wrt; **wrt** is used by both readers and writers
3.int readcnt; // **readcnt** tells the number of processes performing read in the critical section, initially 0

Functions for semaphore :
– wait() : decrements the semaphore value.
– signal() : increments the semaphore value.

Semaphore Solution to reader writer problem

Writer process:

1.Writer requests the entry to critical section.
2.If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
3.It exits the critical section.

```
do {
   // writer requests for critical section
   wait(wrt);

   // performs the write

   // leaves the critical section
   signal(wrt);

} while(true);
```

Semaphore Solution to reader writer problem

Reader process:

Reader requests the entry to critical section.
If allowed:

- it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the wrt semaphore to restrict the entry of writers if any reader is inside.
- It then, signals mutex as any other reader is allowed to enter while others are already reading.
- After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore “wrt” as now, writer can enter the critical section.

If not allowed, it keeps on waiting.

Semaphore Solution to reader writer problem

Reader process:

```
do {

  // Reader wants to enter the critical section
  wait(mutex);

  // The number of readers has now increased by 1
  readcnt++;

  // there is atleast one reader in the critical section
  // this ensure no writer can enter if there is even one reader
  // thus we give preference to readers here
  if (readcnt==1)
    wait(wrt);

  // other readers can enter while this current reader is inside
  // the critical section
  signal(mutex);

  // current reader performs reading here
  wait(mutex);   // a reader wants to leave

  readcnt--;

  // that is, no reader is left in the critical section,
  if (readcnt == 0)
     signal(wrt);        // writers can enter

  signal(mutex); // reader leaves

} while(true);
```

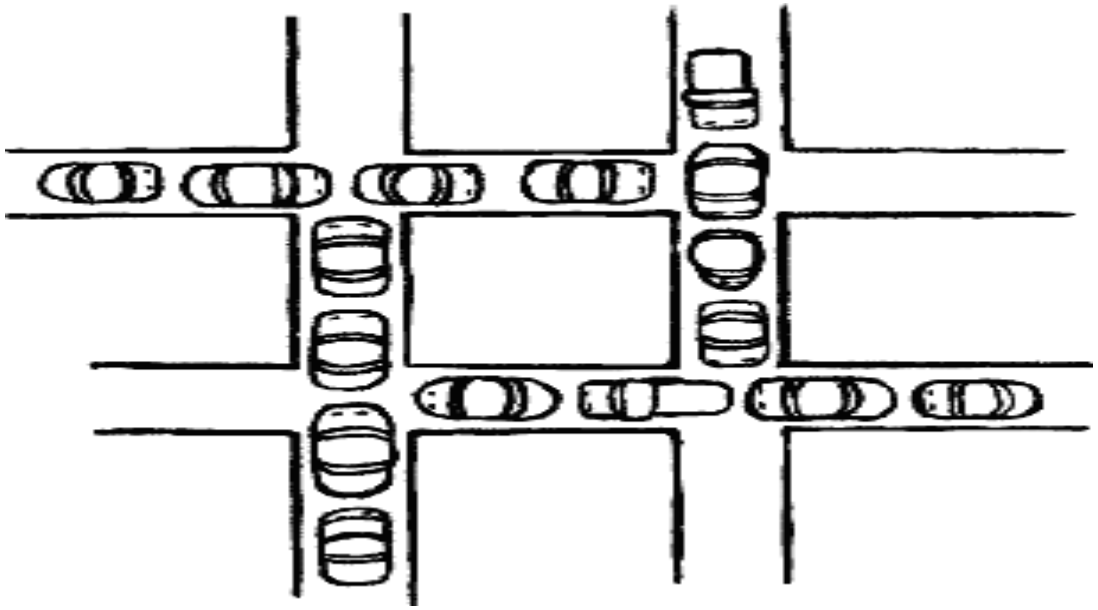
Deadlock

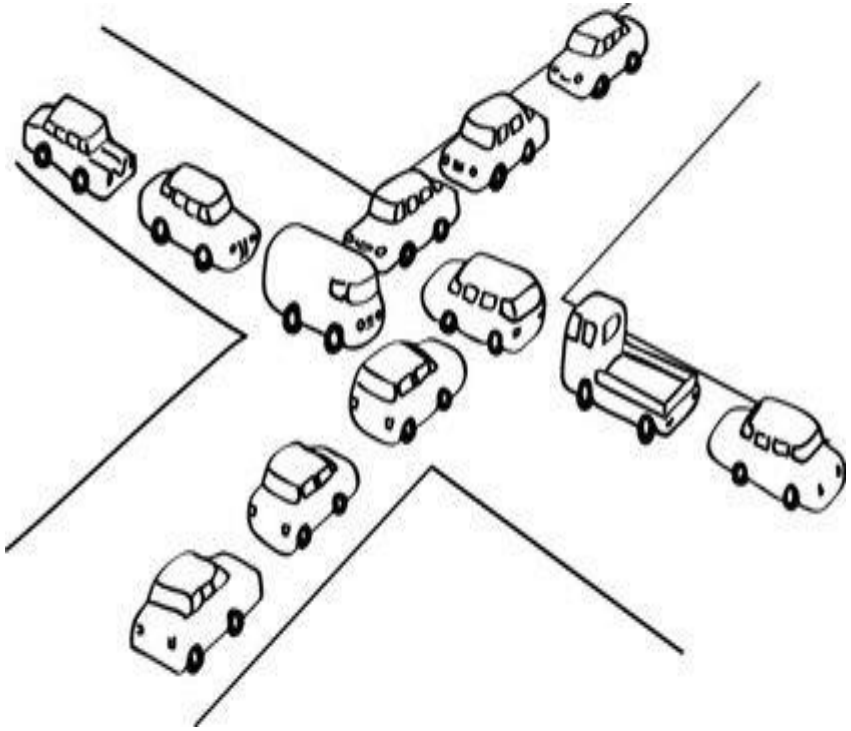
A process in operating system uses resources in the following way.

1) Requests a resource
2) Use the resource
3) Releases the resource

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

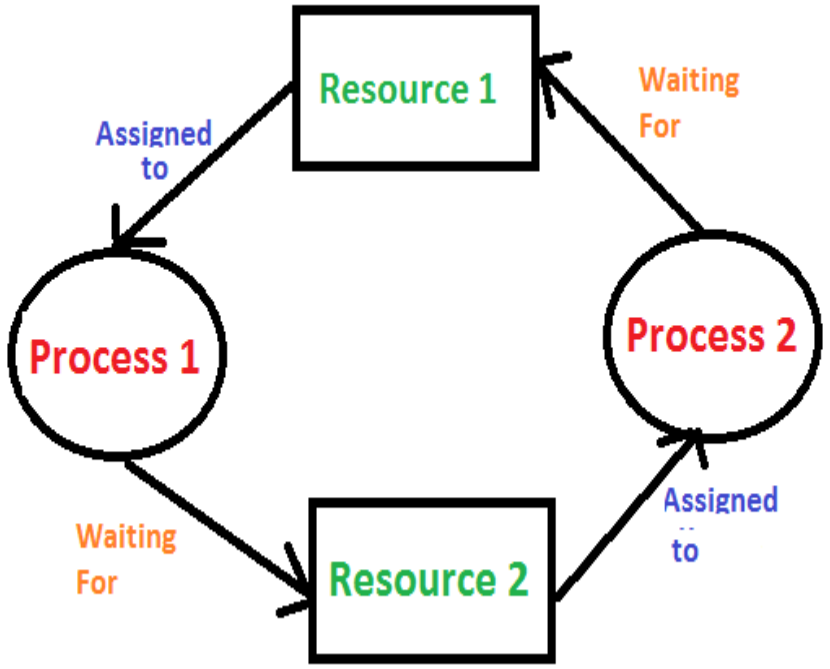
Traffic Deadlock





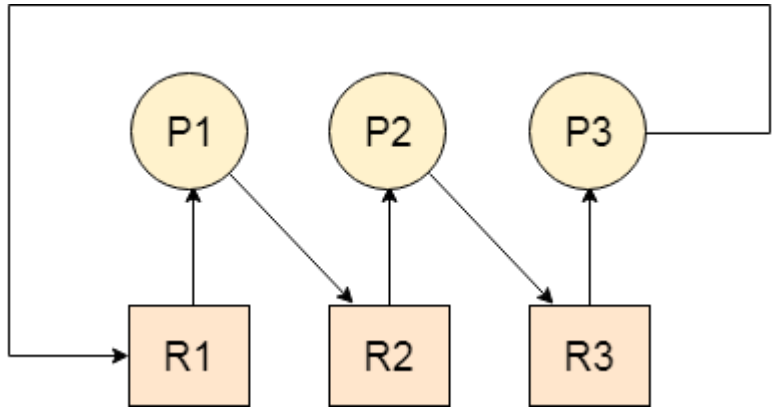
Deadlock

Consider an example when two trains are coming toward each other on the same track and there is only one track, none of the trains can move once they are in front of each other. A similar situation occurs in operating systems when there are two or more processes that hold some resources and wait for resources held by other(s). For example, in the diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.



Deadlock

Let us assume that there are three processes P1, P2 and P3. There are three different resources R1, R2 and R3. R1 is assigned to P1, R2 is assigned to P2 and R3 is assigned to P3.
After some time, P1 demands for R1 which is being used by P2. P1 halts its execution since it can't complete without R2. P2 also demands for R3 which is being used by P3. P2 also stops its execution because it can't continue without R3. P3 also demands for R1 which is being used by P1 therefore P3 also stops its execution.



Necessary conditions for Deadlocks

Deadlock can arise if the following four conditions hold simultaneously

Mutual Exclusion
A resource can only be shared in mutually exclusive manner. It implies, if two process cannot use the same resource at the same time.

Hold and Wait
A process waits for some resources while holding another resource at the same time.

No preemption
The process which once scheduled will be executed till the completion. No other process can be scheduled by the scheduler meanwhile.

Circular Wait
All the processes must be waiting for the resources in a cyclic manner so that the last process is waiting for the resource which is being held by the first process.

Methods for handling deadlock

There are 4 ways to handle deadlock

1.Deadlock Ignorance(OSTRICH ALGORITHM)

Deadlock Ignorance is the most widely used approach among all the mechanism. This is being used by many operating systems mainly for end user uses. In this approach, the Operating system assumes that deadlock never occurs. It simply ignores deadlock. This approach is best suitable for a single end user system where User uses the system only for browsing and all other normal stuff.

2.Deadlock prevention

Deadlock happens only when Mutual Exclusion, hold and wait, No preemption and circular wait holds simultaneously. If it is possible to violate one of the four conditions at any time then the deadlock can never occur in the system. The idea behind the approach is very simple that we have to fail one of the four conditions but there can be a big argument on its physical implementation in the system.

Methods for handling deadlock

There are 4 ways to handle deadlock

3.Deadlock avoidance

In deadlock avoidance, the operating system checks whether the system is in safe state or in unsafe state at every step which the operating system performs. The process continues until the system is in safe state. Once the system moves to unsafe state, the OS has to backtrack one step.In simple words, The OS reviews each allocation so that the allocation doesn't cause the deadlock in the system.

4.Deadlock detection and recovery

This approach let the processes fall in deadlock and then periodically check whether deadlock occur in the system or not. If it occurs then it applies some of the recovery methods to the system to get rid of deadlock.

Deadlock Avoidance

Requires that the system has some additional ***a priori*** information available

- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

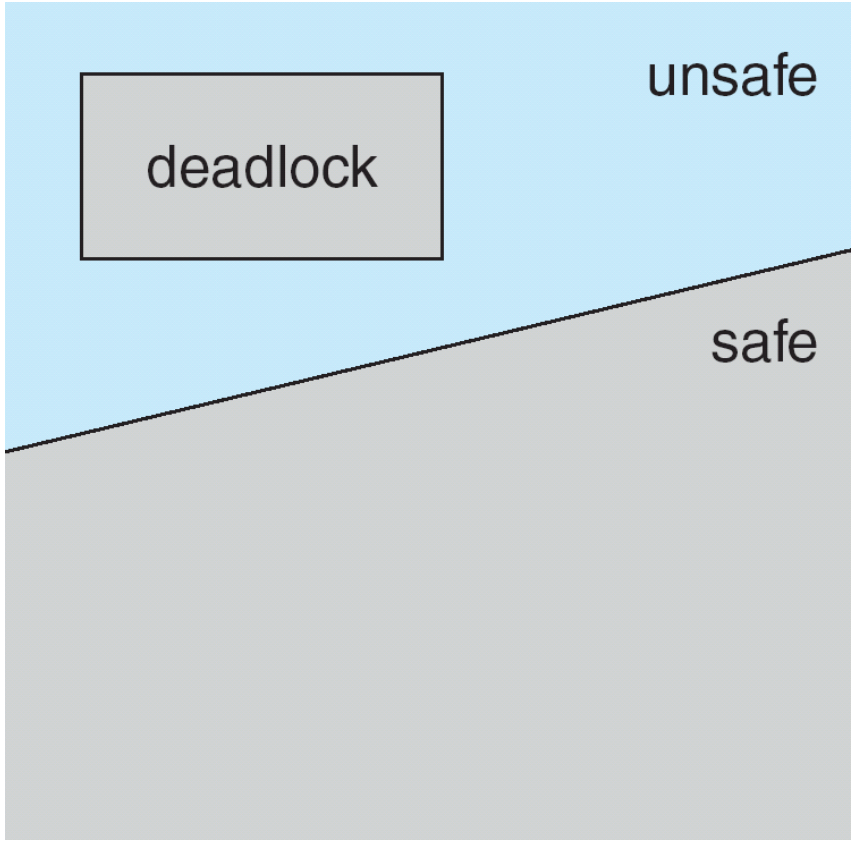
Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < I$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Basic Facts

- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

Safe, Unsafe, Deadlock State



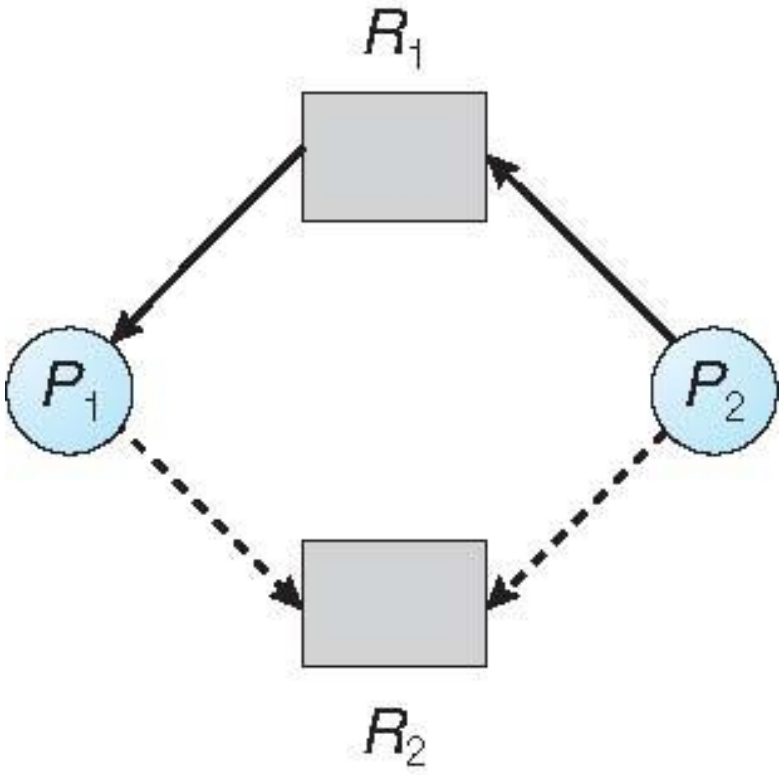
Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

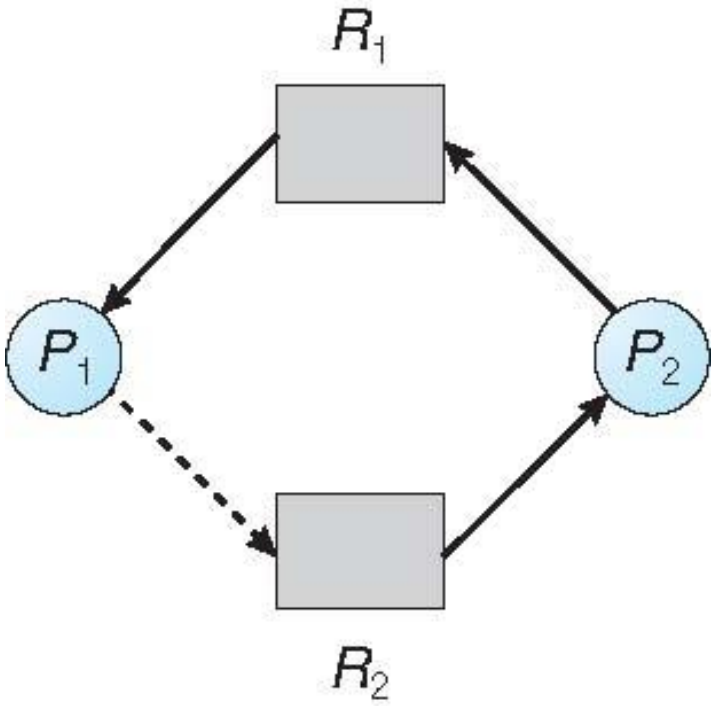
Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker's Algorithm

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- **Available**: Vector of length m. If available [j] = k, there are k instances of resource type R_j available
- **Max**: n x m matrix. If Max [i,j] = k, then process P_i may request at most k instances of resource type R_j
- **Allocation**: n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j
- **Need**: n x m matrix. If $Need$[i,j] = k, then P_i may need k more instances of R_j to complete its task

$$Need\ [i,j] = Max[i,j] - Allocation\ [i,j]$$

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize:

   Work* = *Available

   ***Finish* [*i*] = *false* for *i* = 0, 1, …, *n*- 1**

2. Find an ***i*** such that both:

   (a) ***Finish* [*i*] = *false***

   (b) $Need_i \leq Work$

   If no such ***i*** exists, go to step 4

3. $Work = Work + Allocation_i$

   Finish*[*i*] = *true

   go to step 2

4. If ***Finish* [*i*] == *true*** for all ***i***, then the system is in a safe state

Resource-Request Algorithm for Process P_i

Request$_i$ = request vector for process ***P***$_i$. If ***Request***$_i$ **[*j*] = *k*** then process ***P***$_i$ wants ***k*** instances of resource type ***R***$_j$

1. If ***Request***$_i \leq$ ***Need***$_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If ***Request***$_i \leq$ ***Available***, go to step 3. Otherwise ***P***$_i$ must wait, since resources are not available
3. Pretend to allocate requested resources to ***P***$_i$ by modifying the state as follows:

Available = Available – Request$_i$***;***

Allocation$_i$ ***= Allocation***$_i$ ***+ Request***$_i$***;***

Need$_i$ ***= Need***$_i$ ***– Request***$_i$***;***

- If safe $\Rightarrow$ the resources are allocated to ***P***$_i$
- If unsafe $\Rightarrow$ ***P***$_i$ must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

- 5 processes P_0 through P_4;
  3 resource types:
  A (10 instances), B (5instances), and C (7 instances)
- Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (Cont.)

- The content of the matrix ***Need*** is defined to be ***Max – Allocation***

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence < $\boldsymbol{P_1}$, $\boldsymbol{P_3}$, $\boldsymbol{P_4}$, $\boldsymbol{P_0}$, $\boldsymbol{P_2}$> satisfies safety requirement
- Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?
- Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?