UNIT III

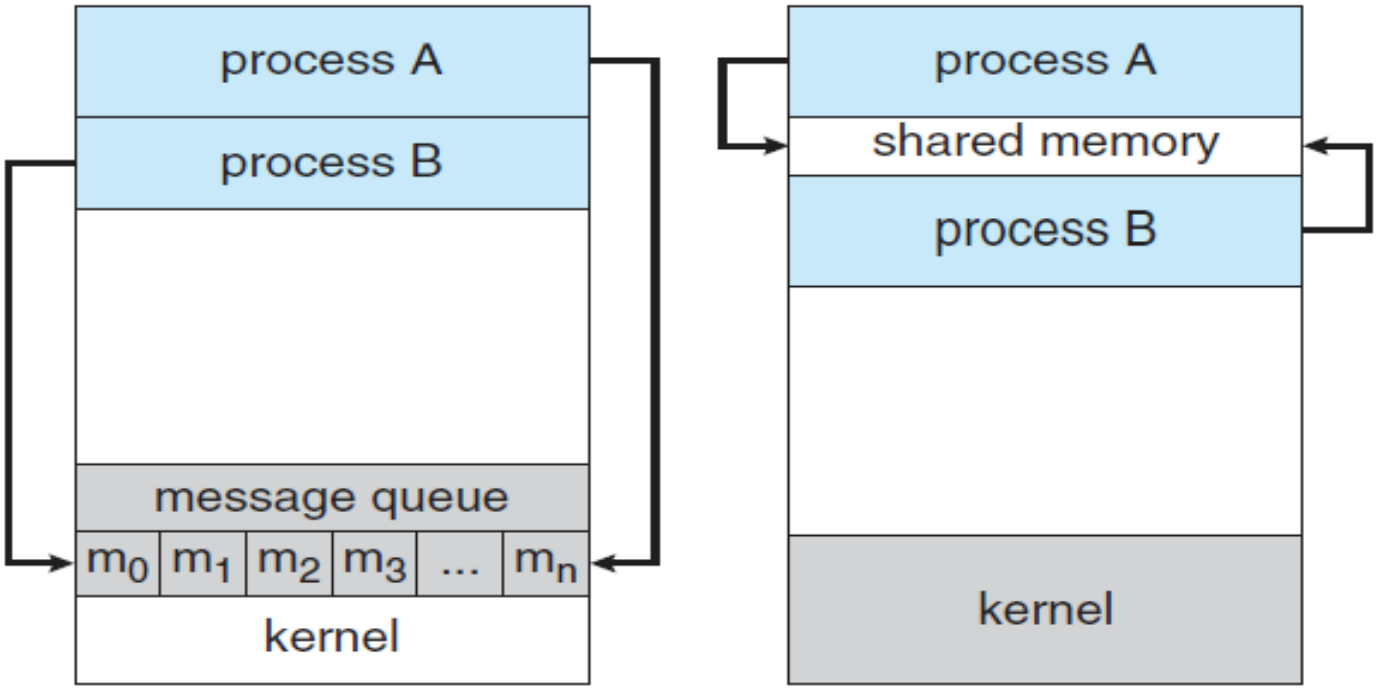
Inter-process Communication, Race Conditions, Critical Section, Mutual Exclusion, Hardware Solution, Strict Alternation, Peterson's Solution, The Producer Consumer Problem, Semaphores, Event Counters, Monitors, Message Passing, and Classical IPC Problems.

Deadlocks: Definition, Deadlock characteristics, Deadlock Prevention, Deadlock Avoidance (concepts only).

Inter-process Communication:

- Inter process communication (IPC) is a mechanism which allows processes to communicate each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them.
- Processes executing concurrently in the operating system may be either independent processes or cooperating processes.
- A process is ***independent*** if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent.
- A process is ***cooperating*** if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.
- Reasons for co-operating processes:
    - **Information sharing:** Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
    - **Computation speedup:** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.
    - **Modularity:** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
    - **Convenience:** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music and compiling in parallel.

- There are two fundamental models of inter-process communication:
  - **Shared Memory** –
    - In the shared-memory model, 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.
    - Shared memory can be faster than message passing, since message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention.
  - **Message Passing** –
    - In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.
    - Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided.
    - Message passing is also easier to implement in a distributed system than shared memory.



Message passing. Shared memory.

Shared-Memory Systems

- Inter-process communication using shared memory requires communicating processes to establish a region of shared memory.
- Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment.

- Other processes that wish to communicate using this shared-memory segment must attach it to their address space.
- Recall that, normally, the operating system tries to prevent one process from accessing another process's memory.
- Shared memory requires that two or more processes agree to remove this restriction.
- They can then exchange information by reading and writing data in the shared areas.
- The form of the data and the location are determined by these processes and are not under the operating system's control.
- The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

Race Conditions:

- A situation like this, 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**.

  - Suppose that two processes, P1 and P2, share the global variable **a**. At some point in its execution, P1 updates **a** to the value 1, and at some point in its execution, P2 updates **a** to the value 2. Thus, the two tasks are in a race to write variable **a**.
  - In this example the "loser" of the race (the process that updates last) determines the final value of a.

Therefore Operating System Concerns of following things

1. The operating system must be able to keep track of the various processes.
2. The operating system must allocate and deallocate various resources for each active process.
3. The operating system must protect the data and physical resources of each process against unintended interference by other processes.
4. The functioning of a process, and the output it produces, must be independent of the speed at which its execution is carried out relative to the speed of other concurrent processes.

❖ **Process Interaction can be defined as**

- Processes unaware of each other
- Processes indirectly aware of each other

- Processes directly aware of each other

❖ **Concurrent processes come into conflict with each other when they are**

- Competing for the use of the same resource.
- Two or more processes need to access a resource during the course of their execution.
- Each process is unaware of the existence of the other processes.
- There is no exchange of information between the competing processes.

Critical Section Problem:

Consider a system consisting of n processes (P0, P1, …… ,Pn-1) each process has a segment of code which is known as critical section in which the process may be changing common variable, updating a table, writing a file and so on.

- The important feature of the system is that when the process is executing in its critical section no other process is to be allowed to execute in its critical section.
- The execution of critical sections by the processes is a mutually exclusive.
- The critical section problem is to design a protocol that the process can use to co-operate each process must request permission to enter its critical section.
- The section of code implementing this request is the entry section. The critical section is followed on exit section. The remaining code is the remainder section.

Example:

```
While (1)
{
    Entry Section;
        Critical Section;
    Exit Section;
        Remainder Section;
}
```

A solution to the critical section problem must satisfy the following three conditions.

1. **Mutual Exclusion:** If process Pi is executing in its critical section then no any other process can be executing in their critical section.

2. **Progress:** If no process is executing in its critical section and some process wish to enter their critical sections then only those process that are not executing in their remainder section can enter its critical section next.

3. **Bounded waiting:** There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request.

Mutual Exclusion:

Requirements for Mutual Exclusion:

1. Mutual exclusion must be enforced: Only one process at a time is allowed into its critical section, among all processes that have critical sections for the same resource or shared object.
2. A process that halts in its non critical section must do so without interfering with other processes.
3. It must not be possible for a process requiring access to a critical section to be delayed indefinitely: no deadlock or starvation.
4. When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.
5. No assumptions are made about relative process speeds or number of processors.
6. A process remains inside its critical section for a finite time only.

Hardware Solution:

Hardware approaches to mutual exclusion.

- **Interrupt Disabling:**

In a uniprocessor machine, concurrent processes cannot be overlapped; they can only be interleaved. Furthermore, a process will continue to run until it invokes an operating system service or until it is interrupted. Therefore, to guarantee mutual exclusion, it is sufficient to prevent a process from being interrupted. This capability can be provided in the form of primitives defined by the system kernel for disabling and enabling interrupts.

Solution to Critical-section Problem Using Locks

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

Because the critical section cannot be interrupted, mutual exclusion is guaranteed.

Disadvantages

- It works only in a single processor environment.
- Interrupts can be lost if not serviced promptly.
- A process waiting to enter its critical section could suffer from starvation.

- **Test and Set Instruction**
  - It is special machine instruction used to avoid mutual exclusion.
  - The test and set instruction can be defined as follows:

Definition:

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

The above function is carried out automatically.

Solution using TestAndSet

Shared boolean variable lock initialized to false.

Solution:

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

```
            lock = FALSE;
                        // remainder section
        } while (TRUE);
```

Advantages

1. It is simple and easy to verify.
2. It is applicable to any number of processes.
3. It can b used to support multiple critical section.

Disadvantages

1. Busy waiting is possible.
2. Starvation is also possible.
3. There may be deadlock.

Swap Instruction:

Definition:

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

Solution using Swap

Shared Boolean variable lock initialized to FALSE;
Each process has a local Boolean variable key

Solution:

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

                // critical section
            lock = FALSE;
                        // remainder section
      } while (TRUE);
```

Bounded-waiting Mutual Exclusion with TestandSet()

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = TestAndSet(&lock);
        waiting[i] = FALSE;
                // critical section
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;
        if (j == i)
                lock = FALSE;
        else
                waiting[j] = FALSE;
                // remainder section
  } while (TRUE);
```

Peterson's Solution:

The mutual exclusion problem is to devise a pre-protocol (or entry protocol) and a post-protocol (or exist protocol) to keep two or more threads from being in their critical sections at the same time. Tanenbaum examine proposals for critical-section problem or mutual exclusion problem.

Problem

When one process is updating shared modifiable data in its critical section, no other process should allowed to enter in its critical section.

❖ **Proposal 1 -Disabling Interrupts (Hardware Solution)**

Each process disables all interrupts just after entering in its critical section and re-enable all interrupts just before leaving critical section. With interrupts turned off the CPU could not be switched to other process.
Hence, no other process will enter its critical and mutual exclusion achieved.

Conclusion

Disabling interrupts is sometimes a useful interrupts is sometimes a useful technique within the kernel of an operating system, but it is not appropriate

as a general mutual exclusion mechanism for user process. The reason is that it is unwise to give user process the power to turn off interrupts.

❖ Proposal 2 -Lock Variable (Software Solution)

In this solution, we consider a single, shared, (lock) variable, initially 0. When a process wants to enter in its critical section, it first test the lock. If lock is 0, the process first sets it to 1 and then enters the critical section. If the lock is already 1, the process just waits until (lock) variable becomes 0. Thus, a 0 means that no process in its critical section, and 1 means hold your horses -some process is in its critical section.

Conclusion
The flaw in this proposal can be best explained by example. Suppose process A sees that the lock is 0. Before it can set the lock to 1 another process B is scheduled, runs, and sets the lock to 1. When the process A runs again, it will also set the lock to 1, and two processes will be in their critical section simultaneously.

❖ Proposal 1 -Strict Alternation

In this proposed solution, the integer variable 'turn' keeps track of whose turn is to enter the critical section. Initially, process A inspects turn, finds it to be 0, and enters in its critical section. Process B also finds it to be 0 and sits in a loop continually testing 'turn' to see when it becomes 1. Continuously testing a variable waiting for some value to appear is called the Busy-Waiting.

Conclusion:
Taking turns is not a good idea when one of the processes is much slower than the other. Suppose process 0 finishes its critical section quickly, so both processes are now in their noncritical section. This situation violates above mentioned condition 3.

The Producer Consumer Problem:

- A **producer process** produces information that is consumed by a **consumer process**.
- For example, a compiler may produce assembly code that is consumed by an assembler. The assembler, in turn, may produce object modules that are consumed by the loader.

- The producer–consumer problem also provides a useful metaphor for the client–server paradigm.
- One solution to the producer–consumer problem uses shared memory
- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.
- Two types of buffers can be used.
    - Unbounded buffer
    - Bounded buffer
- The unbounded buffer places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.
- The bounded buffer assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.
- Let's look more closely at how the bounded buffer illustrates inter-process communication using shared memory. The following variables reside in a region of memory shared by the producer and consumer processes.

```
#define BUFFER SIZE 10

typedef struct {

. . .

}item;

item buffer[BUFFER SIZE];

int in = 0;

int out = 0;
```

- The shared buffer is implemented as a circular array with two logical pointers: **in and out**.
- The variable **in** points to the next free position in the buffer;
- The variable **out** points to the first full position in the buffer.

- The buffer is **empty** when **in ==out**;
- The buffer is **full** when **((in + 1) % BUFFER SIZE) == out.**
- The producer process has a local variable **next produced** in which the new item to be produced is stored.
- The consumer process has a local variable **next consumed** in which the item to be consumed is stored.

```
item next produced;
while (true) {
/* produce an item in next produced */
while (((in + 1) % BUFFER SIZE) == out)
; /* do nothing */
buffer[in] = next produced;
in = (in + 1) % BUFFER SIZE;
}
```

The producer process using shared memory.

```
item next consumed;
while (true) {
while (in == out)
; /* do nothing */
next consumed = buffer[out];
out = (out + 1) % BUFFER SIZE;
/* consume the item in next consumed */
}
```

The consumer process using shared memory.

➢ To write a C program to implement the Producer & consumer Problem (Semaphore)

ALGORITHM:

Step 1: The Semaphore mutex, full & empty are initialized.
Step 2: In the case of producer process
i) Produce an item in to temporary variable.

ii) If there is empty space in the buffer check the mutex value for enter into the critical section.

iii) If the mutex value is 0, allow the producer to add value in the temporary variable to the buffer.

Step 3: In the case of consumer process

i) It should wait if the buffer is empty

ii) If there is any item in the buffer check for mutex value, if the mutex==0, remove item from buffer

iii) Signal the mutex value and reduce the empty value by 1.

iv) Consume the item.

Step 4: Print the result

PROGRAM CODING

```
#include<stdio.h>
int mutex=1,full=0,empty=3,x=0; main()
{
int n;
void producer();
void consumer();
int wait(int);
int signal(int);
printf("\n 1.producer\n2.consumer\n3.exit\n"); while(1)
{
printf(" \nenter ur choice");
scanf("%d",&n);
switch(n)
{
case 1:     if((mutex==1)&&(empty!=0))
            producer();
      else
            printf("buffer is full\n");
            break;

case 2:     if((mutex==1)&&(full!=0))
            consumer();
      else
            printf("buffer is empty");
            break;

case 3:     exit(0);
            break;
}
```

```
}
}
int wait(int s)
{
        return(--s);
}
int signal(int s)
{
        return (++s);
}

void producer()
{
        mutex=wait(mutex);
        full=signal(full);
        empty=wait(empty);
        x++;
        printf("\n producer produces the items %d",x);
        mutex=signal(mutex);
}

void consumer()
{
        mutex=wait(mutex);
        full=wait(full);
        empty=signal(empty);
        printf("\n consumer consumes the item %d",x);
        x--;
        mutex=signal(mutex);
}
```

Semaphores:

- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal().

- The wait() operation was originally termed P (from the Dutch proberen, “to test”); signal() was originally called V (from verhogen, “to increment”).

- The definition of wait() is as follows:

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

- The definition of signal() is as follows:

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

The integer value of the semaphore in the wait and signal operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
In addition, in the case of the wait(S), the testing of the integer value of S (S 0) and its possible modification (S := S – 1), must also be executed without interruption.

Semaphore as General Synchronization Tool

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1;
- Can be simpler to implement
- Also known as mutex locks Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion Semaphore mutex;
  // initialized to

```
do {
        wait (mutex);
            // Critical Section
        signal (mutex);
  } while (TRUE);
```

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.

- Could now have busy waiting in critical section implementation But implementation code is short Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

Semaphore Implementation with no Busy waiting:

- With each semaphore there is an associated waiting queue.
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Implementation of wait:

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

Implementation of signal:

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

Semaphores are not provided by hardware. But they have several attractive properties:

1. Semaphores are machine independent.
2. Semaphores are simple to implement.
3. Correctness is easy to determine.

4. Can have many different critical sections with different semaphores.
5. Semaphore acquires many resources simultaneously.

Drawback of Semaphore

1. They are essentially shared global variables.
2. Access to semaphores can come from anywhere in a program.
3. There is no control or guarantee of proper usage.
4. There is no linguistic connection between the semaphore and the data to which the semaphore controls access.
5. They serve two purposes, mutual exclusion and scheduling constraints.

Monitors:

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect, since these errors happen only if particular execution sequences take place and these sequences do not always occur.

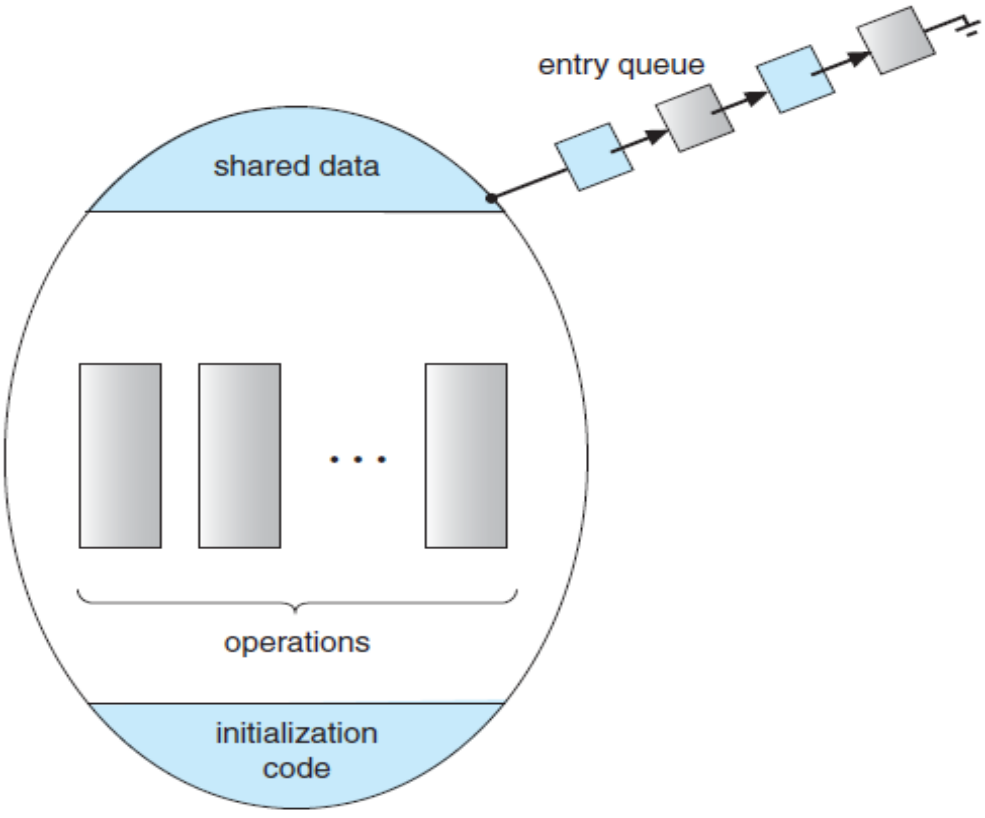
- The monitor is a programming-language construct that provides equivalent functionality to that of semaphores and that is easier to control.
- The monitor construct has been implemented in a number of programming languages, including Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, and Java.
- An **abstract data type**—or **ADT**—encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT.
- A monitor type is an ADT that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor.
- The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables.
- It has also been implemented as a program library.
- This allows programmers to put monitor locks on any object.

```
monitor monitor_name
{
    /* shared variable declarations */
    function P1 ( . . . ) {
    . . .
    }
```

```
        function P2 ( . . . ) {
        . . .
        }
        .
        .
        .
        function Pn ( . . . ) {
        . . .
        }
        initialization code ( . . . ) {
        . . .
        }
}
```

Figure: Syntax of a monitor.

- Thus, a function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.
- Similarly, the local variables of a monitor can be accessed by only the local functions.



Schematic view of a monitor.

- The monitor construct ensures that only one process at a time is active within the monitor

- However, the monitor construct, as defined so far, is not sufficiently powerful for modeling some synchronization schemes.
- For this purpose, we need to define additional synchronization mechanisms.

condition x, y;

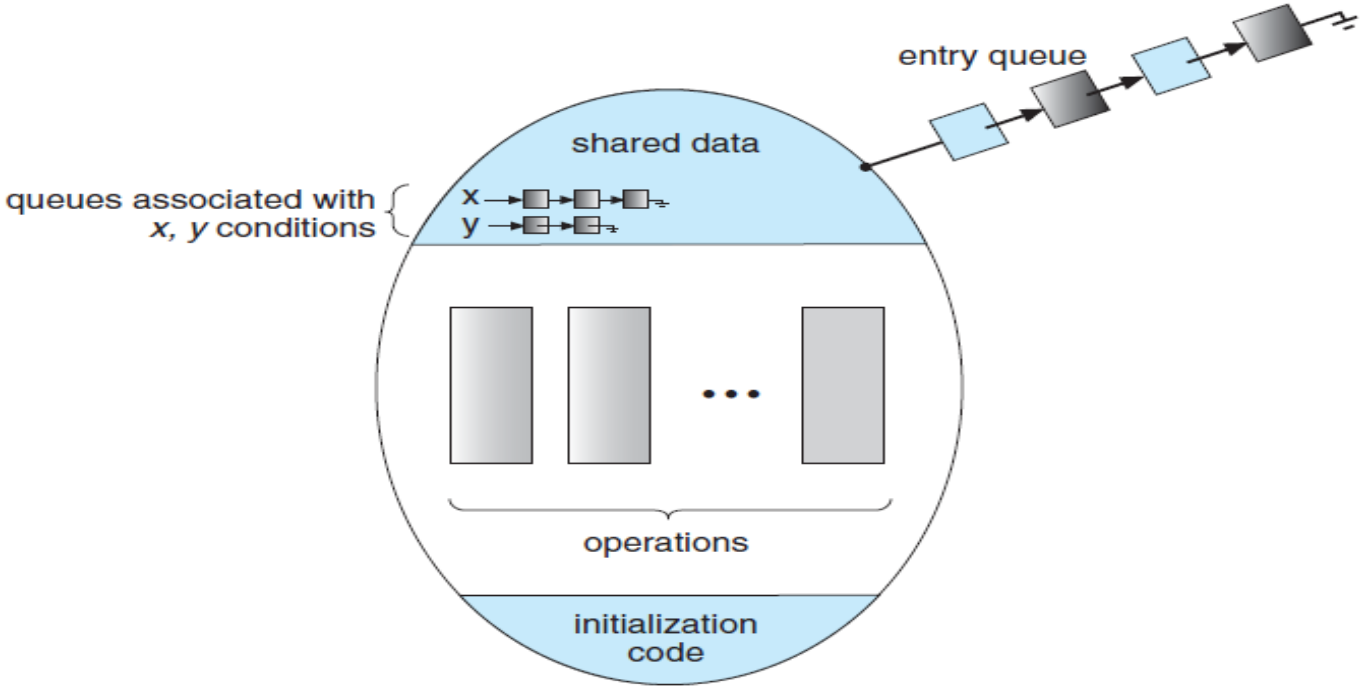
- The only operations that can be invoked on a condition variable are wait() and signal().

x.wait();

- The operation means that the process invoking this operation is suspended until another process invokes

x.signal();

- The x.signal() operation resumes exactly one suspended process.
- If no process is suspended, then the signal() operation has no effect;



Monitor with condition variables.

- Now suppose that, when the x.signal() operation is invoked by a process P, there exists a suspended process Q associated with condition x.
- Clearly, if the suspended process Q is allowed to resume its execution, the signaling process P must wait.
- Otherwise, both P and Q would be active simultaneously within the monitor. Note, however, that conceptually both processes can continue with their execution.

Two possibilities exist:

1. **Signal and wait.** P either waits until Q leaves the monitor or waits for another condition.

2. **Signal and continue.** Q either waits until P leaves the monitor or waits for another condition.

- On the one hand, since P was already executing in the monitor, the ***signal and continue*** method seems more reasonable. On the other, if we allow thread P to continue, then by the time Q is resumed, the logical condition for which Q was waiting may no longer hold. When thread P executes the signal operation, it immediately leaves the monitor. Hence, Q is immediately resumed.

Message Passing:

- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.
- It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.
- A message-passing facility provides at least two operations:
  - send(message)
  - receive(message)
- If P and Q wish to communicate, they need to establish a communication link between them exchange messages via send/receive Implementation of communication link **physical** (e.g., shared memory, hardware bus) **logical** (e.g., logical properties)

Direct Communication

- Processes must name each other explicitly:
  - send (P, message) – send a message to process P
  - receive(Q, message) – receive a message from process Q
- Properties of communication link
- Links are established automatically
- A link is associated with exactly one pair of communicating processes
- Between each pair there exists exactly one link
- The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
- Each mailbox has a unique id
- Processes can communicate only if they share a mailbox

- Properties of communication link
- Link established only if processes share a common mailbox
- A link may be associated with many processes
- Each pair of processes may share several communication links
- Link may be unidirectional or bi-directional

Operations

- create a new mailbox
- send and receive messages through mailbox
- destroy a mailbox
- Primitives are defined as:
    - send(A, message) – send a message to mailbox A
    - receive(A, message) – receive a message from mailbox A
- Mailbox sharing

Problem: Now suppose that processes P1, P2, and P3 all share mailbox A. Process P1 sends a message to A, while both P2 and P3 execute a receive() from A. Which process will receive the message sent by P1?

Solutions

- Allow a link to be associated with at most two processes
- Allow only one process at a time to execute a receive operation
- Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
- **Blocking send** has the sender block until the message is received
- **Blocking receive** has the receiver block until a message is available
- Non-blocking is considered asynchronous
- **Non-blocking send** has the sender send the message and continue
- **Non-blocking receive** has the receiver receive a valid message or null

Buffering

Queue of messages attached to the link; implemented in one of **three** ways

1. Zero capacity – 0 messages Sender must wait for receiver (rendezvous)
2. Bounded capacity – finite length of n messages Sender must wait if link full
3. Unbounded capacity – infinite length Sender never waits

Classical IPC Problems:

These problems are used for testing nearly every newly proposed synchronization scheme. In our solutions to the problems, we use semaphores for synchronization, since that is the traditional way to present such solutions. However, actual implementations of these solutions could use mutex locks in place of binary semaphores.

1. The Bounded-Buffer Problem
2. The Readers–Writers Problem
3. The Dining-Philosophers Problem

1. The Bounded-Buffer Problem:

- It is commonly used to illustrate the power of synchronization primitives
- In our problem, the producer and consumer processes share the following data structures:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

- We assume that the pool consists of n buffers, each capable of holding one item.
- The *mutex* semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1.
- The empty and full semaphores count the number of empty and full buffers.
- The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

```
do {
. . .
/* produce an item in next produced */
. . .
wait(empty);
wait(mutex);
. . .
/* add next produced to the buffer */
. . .
signal(mutex);
signal(full);
} while (true);
```

Fig: The structure of the producer process.

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

Fig: The structure of the consumer process.

2. The Readers–Writers Problem:

- Suppose that a database is to be shared among several 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.
- 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, problem may arise.
- No reader should wait for other readers to finish simply because a writer is waiting.
- Once a writer is ready, that writer performs its write as soon as possible.
- A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve.
- In the solution to the first readers–writers problem, the reader processes share the following data structures:

```
semaphore rw mutex = 1;
semaphore mutex = 1;
int read count = 0;
```

- The semaphore ***rw_mutex*** is common to both reader and writer processes. The ***mutex*** semaphore is used to ensure mutual exclusion when the variable read count is updated.
- The ***read_count*** variable keeps track of how many processes are currently reading the object.

- The semaphore ***rw_mutex*** functions as a mutual exclusion semaphore for the writers
- **The code for a writer process:**

```
do {
wait(rw mutex);
. . .
/* writing is performed */
. . .
signal(rw mutex);
} while (true);
```

- **The code for a reader process:**

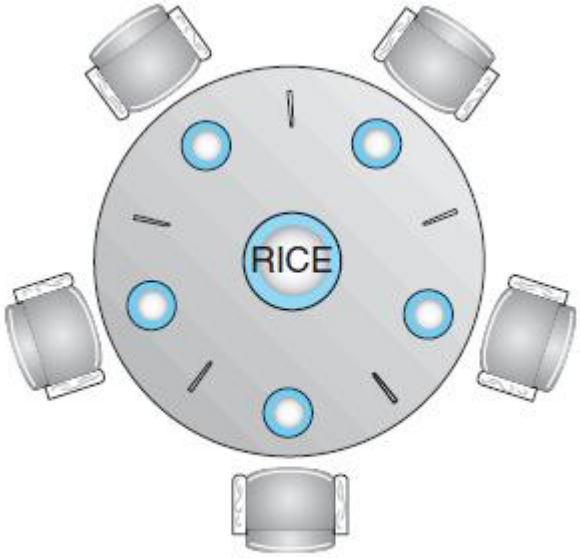
```
do {
wait(mutex);
read count++;
if (read count == 1)
wait(rw mutex);
signal(mutex);
. . .
/* reading is performed */
. . .
wait(mutex);
read count--;
if (read count == 0)
signal(rw mutex);
signal(mutex);
} while (true);
```

- If a writer is in the critical section and n readers are waiting, then one reader is queued on rw_mutex, and n − 1 readers are queued on mutex.
- When a writer executes signal (rw_mutex), we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

3. The Dining-Philosophers Problem:

- The dining-philosophers problem is considered a classic synchronization problem.

- Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks



The situation of the dining philosophers.

- When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).
- A philosopher may pick up only one chopstick at a time.
- She cannot pick up a chopstick that is already in the hand of a neighbor.
- When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks.
- When she is finished eating, she puts down both chopsticks and starts thinking again.
- A philosopher tries to grab a chopstick by executing a ***wait()*** operation on that semaphore.
- She releases her chopsticks by executing the ***signal()*** operation on the appropriate semaphores.
- Thus, the shared data are where all the elements of chopstick are initialized to 1.

```
semaphore chopstick[5];
do {
wait(chopstick[i]);
wait(chopstick[(i+1) % 5]);
. . .
/* eat for awhile */
. . .
```

```
signal(chopstick[i]);
signal(chopstick[(i+1) % 5]);
. . .
/* think for awhile */
. . .
} while (true);
```

Fig: The structure of philosopher i.

- Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock.
- Suppose that all five philosophers become hungry at the same time 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.
- Several possible remedies to the deadlock problem are replaced by:
    - 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 (to do this, she must pick them up in a critical section).
    - Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick.

Deadlocks

Definition: In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock.

SYSTEM MODEL

• A system may consist of finite number of resources and is distributed among number of processes. There resources are partitioned into several instances each with identical instances.

• A process must request a resource before using it and it must release the resource after using it. It can request any number of resources to carry out a designated task. The amount of resource requested may not exceed the total number of resources available.

A process may utilize the resources in only the following sequences:

1. **Request:-** If the request is not granted immediately then the requesting process must wait it can acquire the resources.
2. **Use:-** The process can operate on the resource.
3. **Release:-** The process releases the resource after using it.

Deadlock may involve different types of resources.

Example:-

Consider a system with one printer and one tape drive. If a process Pi currently holds a printer and a process Pj holds the tape drive. If process Pi request a tape drive and process Pj request a printer then a deadlock occurs.

Multithread programs are good candidates for deadlock because they compete for shared resources.

Deadlock characteristics:

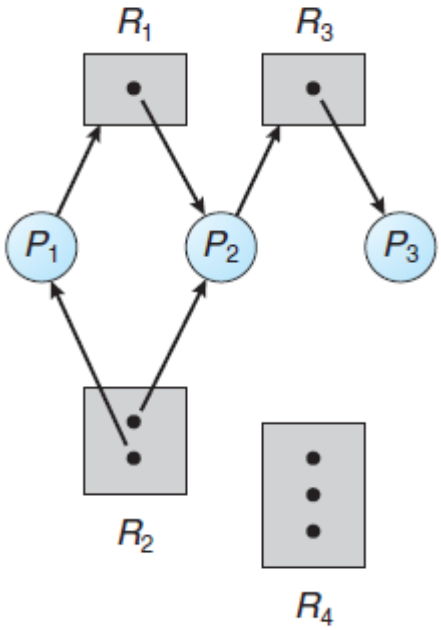
Necessary Conditions: A deadlock situation can occur if the following 4 conditions occur simultaneously in a system:-

1. **Mutual Exclusion:** Only one process must hold the resource at a time. If any other process requests for the resource, the requesting process must be delayed until the resource has been released.

2. **Hold and Wait:-** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by the other process.
3. **No Preemption:-** Resources can't be preempted i.e., only the process holding the resources must release it after the process has completed its task.
4. **Circular Wait:-** A set {P0,P1……..Pn} of waiting process must exist such that P0 is waiting for a resource i.e., held by P1, P1 is waiting for a resource i.e., held by P2. Pn-1 is waiting for resource held by process Pn and Pn is waiting for the resource i.e., held by P1. All the four conditions must hold for a deadlock to occur.

Resource Allocation Graph:

1. Deadlocks are described by using a directed graph called system resource allocation graph. The graph consists of set of vertices (v) and set of edges (e).
2. The set of vertices (v) can be described into two different types of nodes P={P1,P2……..Pn} i.e., set consisting of all active processes and R={R1,R2…….Rn} i.e., set consisting of all resource types in the system.
3. A directed edge from process Pi to resource type Rj denoted by Pi->Ri indicates that Pi requested an instance of resource Rj and is waiting. This edge is called Request edge.
4. A directed edge Ri-> Pj signifies that resource Rj is held by process Pi. This is called Assignment edge.
5. If the graph contain no cycle, then no process in the system is deadlock. If the graph contains a cycle then a deadlock may exist.
6. If each resource type has exactly one instance than a cycle implies that a deadlock has occurred. If each resource has several instances then a cycle do not necessarily implies that a deadlock has occurred.



Resource-allocation graph.

The sets *P, R,* and *E*:

- $P = \{P1, P2, P3\}$
- $R = \{R1, R2, R3, R4\}$
- $E = \{P1 \rightarrow R1, P2 \rightarrow R3, R1 \rightarrow P2, R2 \rightarrow P2, R2 \rightarrow P1, R3 \rightarrow P3\}$

Resource instances:

- One instance of resource type $R1$
- Two instances of resource type $R2$
- One instance of resource type $R3$
- Three instances of resource type $R4$

Process states:

- Process $P1$ is holding an instance of resource type $R2$ and is waiting for an instance of resource type $R1$.
- Process $P2$ is holding an instance of $R1$ and an instance of $R2$ and is waiting for an instance of $R3$.
- Process $P3$ is holding an instance of $R3$.

Methods for Handling Deadlocks:

There are three ways to deal with deadlock problem

- We can use a protocol to prevent deadlocks ensuring that the system will never enter into the deadlock state.
- We allow a system to enter into deadlock state, detect it and recover from it.
- We ignore the problem and pretend that the deadlock never occur in the system. This is used by most OS including UNIX.

- To ensure that the deadlock never occur the system can use either deadlock avoidance or a deadlock prevention.
- Deadlock prevention is a set of method for ensuring that at least one of the necessary conditions does not occur.
- Deadlock avoidance requires the OS is given advance information about which resource a process will request and use during its lifetime.
- If a system does not use either deadlock avoidance or deadlock prevention then a deadlock situation may occur. During this it can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and algorithm to recover from deadlock.
- Undetected deadlock will result in deterioration of the system performance.

Deadlock Prevention:

For a deadlock to occur each of the four necessary conditions must hold. If at least one of the conditions does not hold then we can prevent occurrence of deadlock.

1. **Mutual Exclusion:** This holds for non-sharable resources.
   Eg:-A printer can be used by only one process at a time.

   Mutual exclusion is not possible in sharable resources and thus they cannot be involved in deadlock. Read-only files are good examples for sharable resources. A process never waits for accessing a sharable resource. So we cannot prevent deadlock by denying the mutual exclusion condition in non-sharable resources.

2. **Hold and Wait:** This condition can be eliminated by forcing a process to release all its resources held by it when it request a resource i.e., not available.
   - One protocol can be used is that each process is allocated with all of its resources before its start execution.
   - *Eg*:-consider a process that copies the data from a tape drive to the disk, sorts the file and then prints the results to a printer. If all the resources are allocated at the beginning then the tape drive, disk files and printer are assigned to the process. The main problem with this is it leads to low resource utilization because it requires printer at the last and is allocated with it from the beginning so that no other process can use it.

- Another protocol that can be used is to allow a process to request a resource when the process has none. i.e., the process is allocated with tape drive and disk file. It performs the required operation and releases both. Then the process once again request for disk file and the printer and the problem and with this is starvation is possible.

3. **No Preemption**: To ensure that this condition never occurs the resources must be preempted. The following protocol can be used.
   - If a process is holding some resource and request another resource that cannot be immediately allocated to it, then all the resources currently held by the requesting process are preempted and added to the list of resources for which other processes may be waiting. The process will be restarted only when it regains the old resources and the new resources that it is requesting.
   - When a process request resources, we check whether they are available or not. If they are available we allocate them else we check that whether they are allocated to some other waiting process. If so we preempt the resources from the waiting process and allocate them to the requesting process. The requesting process must wait.

4. **Circular Wait:** The fourth and the final condition for deadlock is the circular wait condition. One way to ensure that this condition never, is to impose ordering on all resource types and each process requests resource in an increasing order.

Let R={R1,R2,…….Rn} be the set of resource types. We assign each resource type with a unique integer value. This will allows us to compare two resources and determine whether one precedes the other in ordering. *Eg*:-we can define a one to one function

F:R->N as follows :-F(disk drive)=5 F(printer)=12 F(tape drive)=1

Deadlock can be prevented by using the following protocol:

- Each process can request the resource in increasing order. A process can request any number of instances of resource type say Ri and it can request instances of resource type Rj only F(Rj) > F(Ri).
- Alternatively when a process requests an instance of resource type Rj, it has released any resource Ri such that F(Ri) >= F(Rj). If these two protocols are used then the circular wait can't hold.

Deadlock Avoidance (concepts only):

- Deadlock prevention algorithm may lead to low device utilization and reduces system throughput.
- Avoiding deadlocks requires additional information about how resources are to be requested. With the knowledge of the complete sequences of requests and releases we can decide for each requests whether or not the process should wait.
- For each requests it requires to check the resources currently available, resources that are currently allocated to each processes future requests and release of each process to decide whether the current requests can be satisfied or must wait to avoid future possible deadlock.
- A deadlock avoidance algorithm dynamically examines the resources allocation state to ensure that a circular wait condition never exists. The resource allocation state is defined by the number of available and allocated resources and the maximum demand of each process.

Safe State:

- A state is a safe state in which there exists at least one order in which all the process will run completely without resulting in a deadlock.
- A system is in safe state if there exists a safe sequence.
- A sequence of processes <P1,P2,……..Pn> is a safe sequence for the current allocation state if for each Pi the resources that Pi can request can be satisfied by the currently available resources.
- If the resources that Pi requests are not currently available then Pi can obtain all of its needed resource to complete its designated task.
- A safe state is not a deadlock state.
- Whenever a process request a resource i.e., currently available, the system must decide whether resources can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in safe state.
- In this, if a process requests a resource i.e., currently available it must still have to wait. Thus resource utilization may be lower than it would be without a deadlock avoidance algorithm.

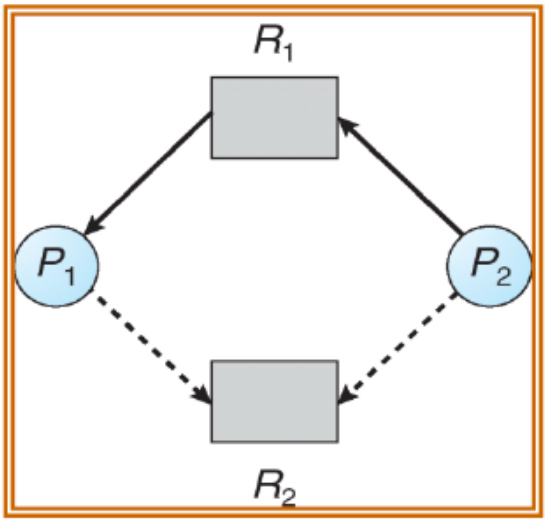
Resource Allocation Graph Algorithm:

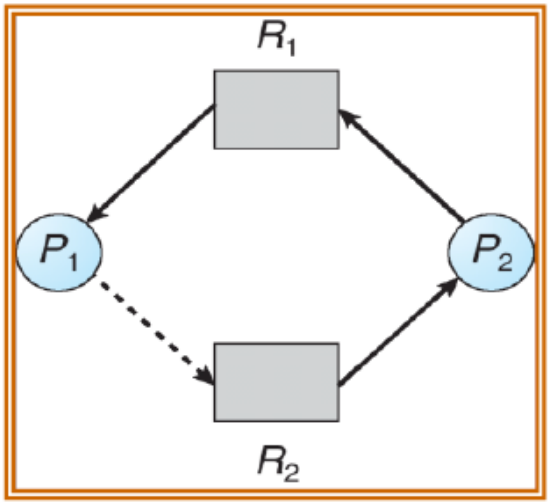
This algorithm is used only if we have one instance of a resource type. In addition to the request edge and the assignment edge a new edge called claim edge is used.

***Ex* :-** A claim edge Pi->Rj indicates that process Pi may request Rj in future. The claim edge is represented by a dotted line.

- When a process Pi requests the resource Rj, the claim edge is converted to a request edge.
- When resource Rj is released by process Pi, the assignment edge Rj->Pi is replaced by the claim edge Pi->Rj.

When a process Pi requests Rj the request is granted only if converting the request edge Pi->Rj to as assignment edge Rj->Pi do not result in a cycle. Cycle detection algorithm is used to detect the cycle. If there are no cycles then the allocation of the resource to process leave the system in safe state





Banker's Algorithm:

- This algorithm is applicable to the system with multiple instances of each resource types, but this is less efficient then the resource allocation graph algorithm.
- When a new process enters the system it must declare the maximum number of resources that it may need. This number may not exceed the total number of resources in the system. The system must determine that whether the allocation of the resources will leave the system in a safe state or not. If it is so resources are allocated else it should wait until the process release enough resources.
- Several data structures are used to implement the banker's algorithm. Let 'n' be the number of processes in the system and 'm' be the number of resources types.

We need the following data structures:

- **Available**:-A vector of length m indicates the number of available resources.
  If Available[i]=k, then k instances of resource type Rj is available.

- **Max**:-An n*m matrix defines the maximum demand of each process if Max[i,j]=k, then Pi may request at most k instances of resource type Rj.

- **Allocation**:-An n*m matrix defines the number of resources of each type currently allocated to each process.
  If Allocation[i,j]=k, then Pi is currently k instances of resource type Rj.

- **Need**:-An n*m matrix indicates the remaining resources need of each process.
  If Need[i,j]=k, then Pi may need k more instances of resource type Rj to compute its task.
  So Need[i,j]=Max[i,j]-Allocation[i]

Safety Algorithm:-

This algorithm is used to find out whether or not a system is in safe state or not.

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

Work = Available
For i = 1,2, …, n,
if Allocation$_i$ ^{1} 0, then
Finish[i] = false;otherwise, Finish[i] = true.

Step 2: Find an index i such that both:
Finish[i] == false
Requesti <= Work
If no such i exists, go to step 4.

Step 3: Work = Work + Allocation
Finish [i] = true
Go to step 2

Step 4: If Finish [i] = false, for some i, 1<=i<= n, then the system is in a deadlock state.
Moreover, if Finish[i] = false, then process Pi is deadlocked.

Resource Allocation Algorithm:

Request = request vector for process P_i. If $Request_i$ [j] = k then process Pi wants k instances of resource type R_j.

Step 1: If $Request_i$ <= $Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.

Step 2: If $Request_i$ <= Available, go to step 3. Otherwise Pi must wait, since resources are not available.

Step 3: Pretend to allocate requested resources to Pi by modifying the state as follows:

Available = Available – Request;
$Allocation_i$ = $Allocation_i$ + $Request_i$;
$Need_i$ = $Need_i$ – $Request_i$;

· If safe ⇒ the resources are allocated to Pi.
· If unsafe ⇒ Pi must wait and the old resource-allocation state is restored

Example 1:

- 5 processes P0 through P4;
- 3 resource types: A (10 instances), B (5instances), and C (7 instances).
- Snapshot at time T0:

	Allocation	Max
	A B C	A B C
P0	0 1 0	7 5 3
P1	2 0 0	3 2 2
P2	3 0 2	9 0 2
P3	2 1 1	2 2 2
P4	0 0 2	4 3 3

- Check that $Need_i$ <= Available
- To calculate first Available matrix is (10 5 7) – (7 2 5) = **3 3 2**

	Allocation	Max	Request/Need (Max-Allocation)	Available/Work (Total-Allocation)
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3	3 3 2
P1	3 0 2	3 2 2	0 2 0	5 3 2
P2	3 0 1	9 0 2	6 0 0	7 4 3
P3	2 1 1	4 2 2	0 1 1	7 4 5
P4	0 0 2	5 3 3	4 3 1	7 5 5

10 5 7

Total = 7 2 5

- To find safe state using Banker's Algorithm:

	$Need_i$	<=	Available	=>	Work = Work+Allocation
P0	7 4 3	<=	3 3 2		X
P1	1 2 2	<=	3 3 2		Work = 3 3 2 + 2 0 0 = 5 3 2
P2	6 0 0	<=	5 3 2		X
P3	2 1 1	<=	5 3 2		Work = 5 3 2 + 2 1 1 = 7 4 3
P4	5 3 1	<=	7 4 3		Work = 7 4 3 + 0 0 2 = 7 4 5
P0	7 4 3	<=	7 4 5		Work = 7 4 5 + 0 1 0 = 7 5 5
P1	6 0 0	<=	7 5 5		Work = 7 5 5 + 3 0 2 = 10 5 7

- Executing safety algorithm shows that sequence <P1, P3, P4, P0, P2> satisfies safety requirement.
- Can request for (3,3,0) by P4 be granted? –NO
- Can request for (0,2,0) by P0 be granted? –NO (Results Unsafe)

Recovery from Deadlock

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.

In which order should we choose to abort?

- Priority of the process.
- How long process has computed, and how much longer to completion.
- Resources the process has used.
- Resources process needs to complete.
- How many processes will need to be terminated?
- Is process interactive or batch?.

Resource Preemption

- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.