Unit-III

Process Synchronization and Deadlocks

Concurrency: Principles of Concurrency, Inter Process Communication, Process/Thread Synchronization.

Mutual Exclusion: Requirements, Hardware and Software Support, Semaphores and Mutex, Monitors.

Classical Synchronization Problems: Readers/Writers Problem, Producer and Consumer Problem.

Principles of Deadlock: Conditions and Resource Allocation Graphs, Deadlock Prevention.

Deadlock Avoidance: Banker's Algorithm for Single & Multiple Resources, Deadlock Detection and Recovery, Dining Philosophers Problem.

Background

- The **central themes** of OS design are all **concerned with the management of processes and threads:**

- **Multiprogramming:** The management of **multiple processes** within a **uniprocessor system**

- **Multiprocessing** : The management of **multiple processes** within a **multiprocessor system**

- **Distributed processing:** The management of **multiple processes** executing on **multiple, distributed** computer systems.

  The recent proliferation of clusters is a prime example of this type of system.

- Fundamental to all of these areas and fundamental to OS design, is **concurrency.**

- Concurrency includes **design** issues, including **communication** among processes, **sharing** of and **competing** for resources.

Background Continued…

- **Concurrency arises in three different contexts**:

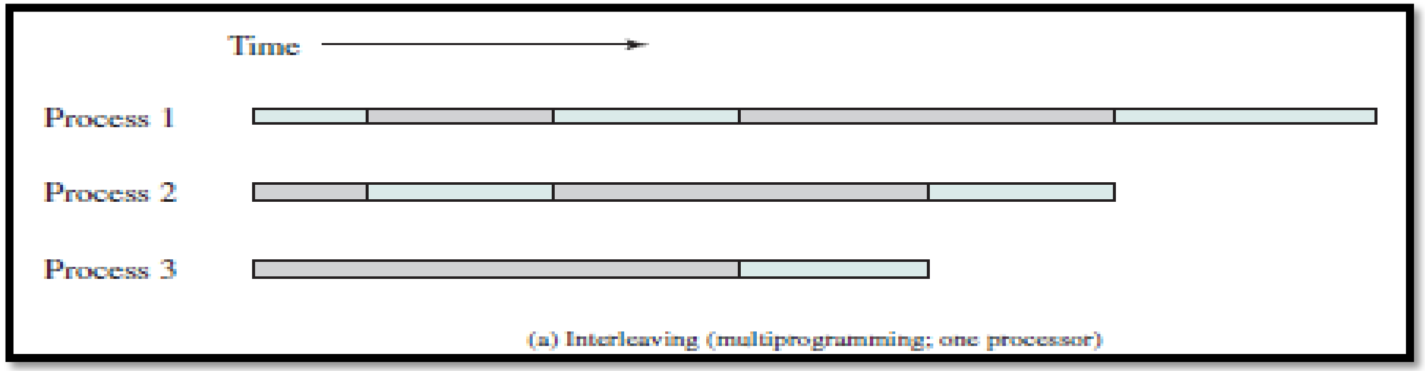
  - **Multiple applications:** Multiprogramming was invented to allow **processing time to be dynamically shared** among a number of active applications.

  - **Structured applications:** As an **extension of the principles of modular design and structured programming**, some applications can be effectively programmed as a set of concurrent processes.

  - **Operating system structure:** The same structuring advantages apply to systems programs, and we have seen that **operating systems are themselves often implemented as a set of processes or threads.**

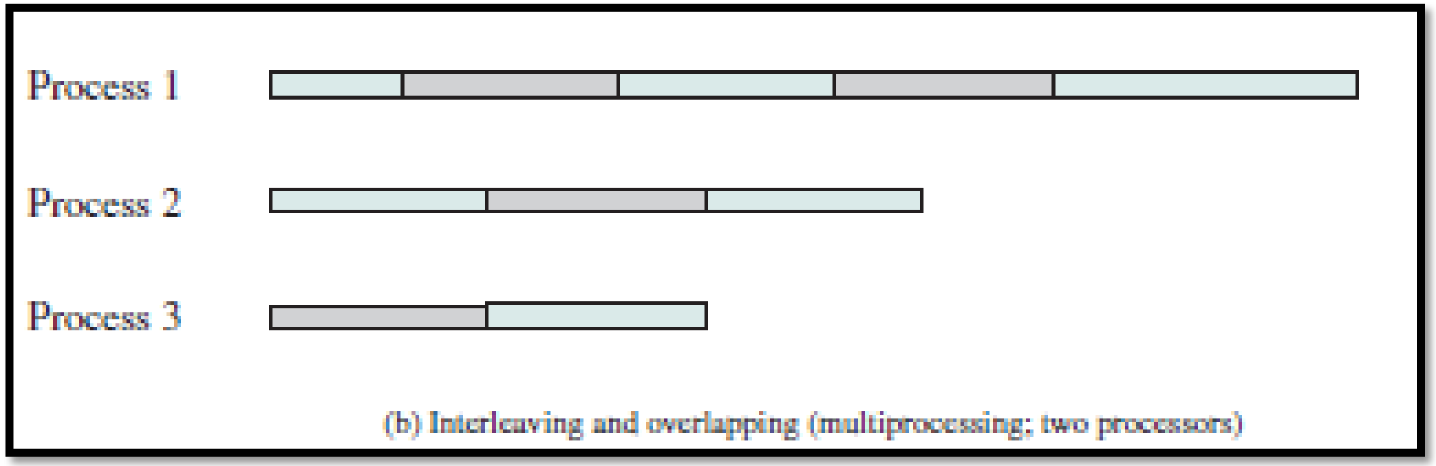
❖ Principles of Concurrency

- In a **single-processor multiprogramming system**, processes are interleaved in time to yield the appearance of simultaneous execution:



(a) Interleaving (multiprogramming; one processor)

- In a **multiple-processor system**, it is possible not only to interleave the execution of multiple processes but also **to overlap them**



(b) Interleaving and overlapping (multiprocessing; two processors)

Principles of Concurrency Continued…

- A Simple Example:

Consider the following procedure:

```
void echo( )
    {
        chin = getchar( );  //input is obtained & input character is stored in variable chin( ) .
        chout = chin;       //input transferred to variable chout( )
        putchar(chout);     //input characters displayed on screen
    }
```

- We have a **single-processor multiprogramming system** supporting a single user.
- The **procedure is shared** and loaded into a portion of memory **global to all applications**.

- **Such sharing can lead to problems**.
- Consider the following sequence:

  - **Process P1 invokes** the echo procedure and is **interrupted immediately** after getchar returns its value and stores it in chin. At this point, the most recently entered character, ‘x’, is stored in variable chin .
  - **Process P2 is activated** and invokes the echo procedure, which runs to conclusion, inputting and then displaying a single character, ‘y’, on the screen.
  - **Process P1 is resumed**. By this time, the value **x has been overwritten** in chin and therefore lost.
  - Instead, chin contains y, which is transferred to chout and displayed.

- Thus, the **first character is lost and the second character is displayed twice**.
- The essence of this problem is the **shared global variable**, **chin** .
- Multiple processes have access to this variable. If one process updates the global variable and then is interrupted, another process may alter the variable before the first process can use its value.

- Suppose, however, that **we permit only one process at a time to be in that procedure**. Then the foregoing sequence would result in the following:

  - **Process P1 invokes** the echo procedure and is **interrupted immediately** after the conclusion of the input function. At this point, the most recently entered character, **x, is stored in variable chin .**
  - **Process P2 is activated** and invokes the echo procedure. However, because P1 is still inside the echo procedure, although currently suspended, P2 is blocked from entering the procedure. Therefore, **P2 is suspended awaiting the availability of the echo procedure.**
  - At some later time, **process P1 is resumed** and completes execution of echo . The proper character, x , is displayed.
  - **When P1 exits echo** , this **removes the block on P2**. When P2 is later resumed, the echo procedure is successfully invoked.

- This example shows that **it is necessary to protect shared global variables.**

- In **a multiprocessor system**, the same problems of protected shared resources arise, and the same solution works. First, **suppose that there is no mechanism** for controlling access to the shared global variable:
  - Processes **P1 and P2 are both executing, each on a separate processor**. Both processes invoke the echo procedure.
  - The following events occur; events on the same line take place in parallel:

Process P1	Process P2
	•
chin = getchar();	•
•	chin = getchar();
chout = chin;	chout = chin;
putchar(chout);	•
•	putchar(chout);
•	•

The result is that the character input to P1 is lost before being displayed, and the character input to P2 is displayed by both P1 and P2.

- If only **one process at a time** may be in echo, then the following sequence occurs:

- Processes P1 and P2 are both executing, each on a separate processor. P1 invokes the echo procedure.

- **While P1 is inside the echo** procedure, P2 invokes echo .
  Because P1 is still inside the echo procedure (whether P1 is suspended or executing), **P2 is blocked from entering the procedure.**
  Therefore, **P2 is suspended awaiting the availability** of the echo procedure.

- At a later time, process P1 completes execution of echo, exits that procedure and continues executing.

- Immediately **upon the exit of P1 from echo , P2 is resumed** and begins executing echo .

- In the case of a **uniprocessor system**, the reason we have a problem is that **an interrupt can stop instruction execution anywhere** in a process.

- In the case of a **multiprocessor system**, we have that **same condition** and in addition, a problem can be caused because two processes may be executing simultaneously and **both trying to access the same global variable**.

- However, **the solution to both types of problem** is the same:

control access to the shared resource

❖ Race Condition

- A race condition occurs when **multiple processes or threads read and write data items** so that the final result depends on the **order of execution of instructions** in the multiple processes.

- Example 1:

  - Two processes **P1** & **P2** share global variable **a**
  - At some point, **P1** update **a** to the value **1**
  - Then at some point, **P2** updates **a** to the value **2**.
  - Thus **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.**

Race Condition Continued…

- Example 2:

  - Consider two processes, P3 and P4
  - share global variables b and c ,
  - initial values are b = 1 and c = 2 .
  - At some point, P3 executes the assignment b = b + c &
  - At some point, P4 executes the assignment c = b + c
  - The two processes update different variables.
  - Final values of the two variables depend on the order in which the two processes execute these two assignments:

➢ **If P3 executes its assignment statement first**, then the final values are **b = 3 and c = 5,**

➢ **If P4 executes its assignment statement first**, then the final values are **b = 4 and c = 3**

❖ Operating System Concerns

- **Design and management issues** are raised by the existence of concurrency:

  - The OS must be able to **keep track of** the various processes. This is done with the **use of process control blocks**.
  - The OS must **allocate and deallocate** various resources for each active process.
  - The OS must **protect the data and physical resources** of each process **against unintended interference** by other processes.
  - 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.**

- To understand how the **issue of speed independence** can be addressed, we need to look at the **ways in which processes can interact.**

▪ Process Interaction

Table: Process Interaction

Sr. No.	Degree of Awareness	Relationship	Influence that One Process Has on the Other	Potential Control Problems
1	Processes unaware of each other	**Competition** (two independent applications may both want to access the same disk or file or printer.)	• Results of one process independent of the action of others. • Timing of process may be affected.	• Mutual exclusion • Deadlock (renewable resource) • Starvation
2	Processes indirectly aware of each other (e.g., shared object)	**Cooperation by sharing**	• Results of one process may depend on information obtained from others. • Timing of process may be affected.	• Mutual exclusion • Deadlock (renewable resource) • Data coherence
3	Processes directly aware of each other (have communication primitives available to them)	**Cooperation by communication**	• Results of one process may depend on information obtained from others. • Timing of process may be affected.	• Deadlock (consumable resource) • Starvation

Process Interaction Continued..

A. Competition Among Processes for Resources

$$\begin{tikzcd}
\text{P1 \& P2 Concurrent processes} \arrow[r, "\text{Competing for}"] & \text{the same resource (say I/O device)} \arrow[dl, "\text{resource allocated}"'] \arrow[dr, "\text{access denied}"] & \\
& \text{P1 allocated printer} & & \text{P2 have to wait} \arrow[d] \\
& & & \text{So P2 will slow down} \\
& & & \text{(extreme case: may not get access ever)}
\end{tikzcd}$$

- Competing Process must face three control problems:

1. Need for mutual exclusion
2. Deadlock
3. Starvation

three control problems Continued..

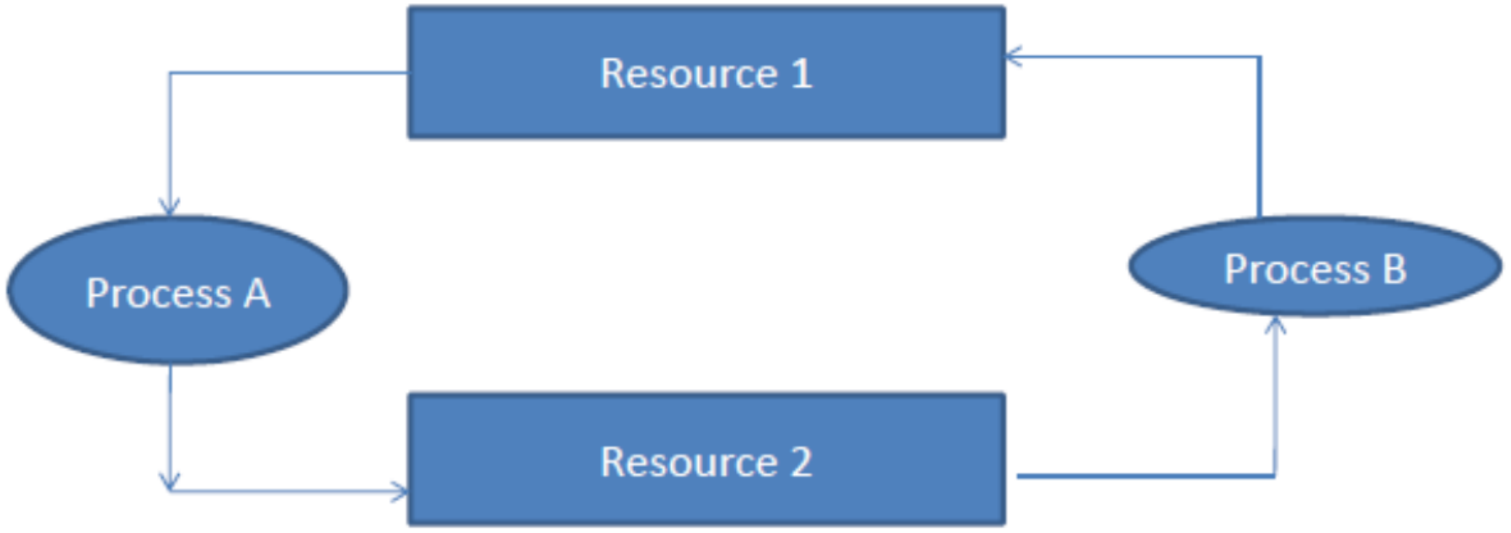
1. Mutual Exclusion

- Suppose two or more processes require **access to a single non-sharable resource**, such as a **printer.**
- This device is to be referred as a **critical resource** and the portion of the program that uses it as a **critical section** of the program.
- Only **one program at a time** be allowed in its critical section.
- The enforcement of mutual exclusion creates **two additional control problems**:

  One is that of **Deadlock** & other is **Starvation**

Process Interaction Continued..

2. Deadlock



- Consider two processes, **A and B**, and two resources, **R1 and R2**.
- Suppose that each process needs access to both resources to perform part of its function.
- The OS assigns **R1 to A**, and **R2 to B**.
- **Each process is waiting** for one of the two resources.
- Neither will release the resource that it already owns until it has acquired the other resource and performed the function requiring both resources.
- The two processes are **deadlocked**.

3. Starvation

- Suppose that three processes (P1, P2, P3) each require periodic access to resource R.
- Consider the situation in which:
- **P1 is in possession** of the resource and both P2 and P3 are delayed, waiting for that resource.
- **When P1 exits** its critical section, either P2 or P3 should be allowed access to R.
- Assume that the **OS grants access to P3** and that **P1 again requires access before P3 completes** its critical section.
- If the OS grants **access to P1** after P3 has finished and subsequently **alternately grants access to P1 and P3**, then **P2 may indefinitely be denied access** to the resource, **even though there is no deadlock** situation.

B. Cooperation among Processes by Sharing

- The case of cooperation by sharing **covers processes that interact with other processes without being explicitly aware** of them.
- **For example**, multiple processes may **have access to shared variables** or to shared **files or databases**.
- Processes **may use and update the shared data without reference to other processes** but know that **other processes may have access to the same data**.
- Thus the <u>processes must cooperate to ensure</u> that the data they share are properly managed.
- The **control mechanisms** must <u>ensure the integrity</u> of the shared data.

C. Cooperation Among Processes by Communication

- Processes directly aware of each other (**have communication primitives** available to them)
- The **communication provides a way to synchronize or coordinate the various activities**.
- Typically, communication can be characterized as **consisting of messages** of some sort.
- **Primitives for sending and receiving messages** may be provided as part of the programming language or provided by the OS kernel.

❖ Inter Process Communication (IPC)

- 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.

Inter Process Communication (IPC) Continued…..

- Reasons for providing an environment that allows process cooperation:
  - Information sharing: environment to allow concurrent access to shared file
  - Computation speed-up: task divided in subtasks executing in parallel for faster execution
  - Modularity: construct the system in a modular fashion, divide system functions into separate processes or threads
  - Convenience: individual user working on many tasks at the same time
    (editing text, listening to music, compiling in parallel)

- **Inter process Communication (IPC) provides:**

- Mechanism for processes **to communicate and to synchronize their actions**

- Two fundamental models:

    - **Shared Memory**: region of memory is shared

    - **Message Passing**: communication using message exchange

- **Shared Memory Systems:**

  - A region of shared memory is required.
  - Shared memory region resides in address space of a process
  - Two processes must agree to share the memory space
  - They can read & write data in shared area
  - e.g.

  Web server produces HTML files & images to be consumed by the client web browsers.

- **Message Passing System:**

  - Provides a mechanism to allow processes to communicate & synchronize their actions <u>without sharing address space.</u>
  - Useful in distributed environment.
  - e.g. chat program on WWW

- Provides two operations:
  - send(*message*) – message size fixed or variable
  - receive(*message*)

- If process *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive

Message Passing System Continued…

- **Naming**

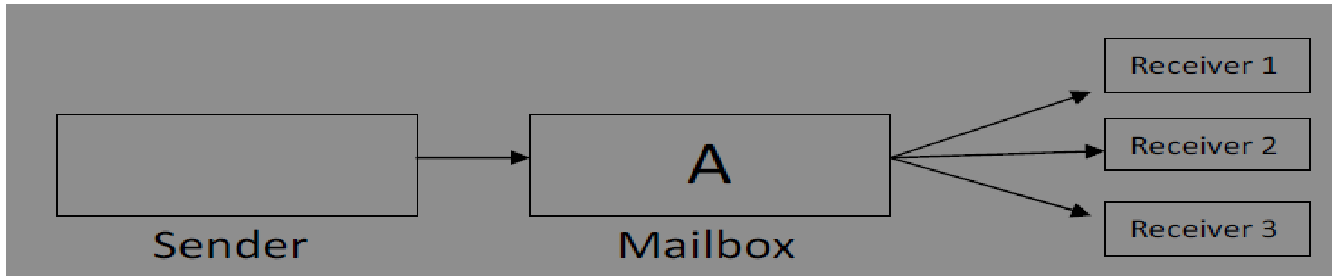
- Processes that want to communicate must have a way to refer to each other.
- They can use either direct or indirect communication.

A. 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; Pid is required.
    - A link is associated with exactly one pair of communicating processes.
    - Between each pair there exists exactly one link.
    - The link is usually bi-directional.

B. Indirect Communication



- Messages are sent to and received from **mailboxes (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 (one link- one mailbox).
  - Link may be unidirectional or bi-directional.

Indirect Communication Continued ….

- 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

C. Buffering

- Queue of messages attached to the link.
- Implemented in one of three ways:

1. Zero capacity – zero messages
   Sender must **wait for receiver**

2. Bounded capacity – finite length of *n* messages
   Sender must **wait if link full**

3. Unbounded capacity **– infinite length**
   Sender never waits

❖ Synchronization

- Communication between processes takes place through calls **send( ) and receive( ) primitives.**
- There are **different design options for implementing each primitive**.
- Message passing may be either **blocking or non-blocking** also known as **synchronous and asynchronous.**

  - **Blocking send**: The **sending process is blocked until** the message is received by the receiving process or by the mailbox.
  - **Nonblocking send**: The sending process sends the message and resumes operation.
  - **Blocking receive**: The receiver is blocked until a message is available.
  - **Nonblocking receive**: The receiver retrieves either a valid message or a null.

• **Thread Synchronization**

- All of the threads of a process share the **same address space** and other resources, such as open files.
- Any **alteration of a resource** by one thread affects the environment of the other threads in the same process.
- It is therefore **necessary to synchronize the activities** of the various threads so that they do not interfere with each other or corrupt data structures.
- **For example**,

  if two threads each try to add an element to a doubly linked list at the same time, one element may be lost or the list may end up malformed.

❖ Critical Section Problem

- *n* processes all competing to use some shared data.
- Each process has a code segment, called ***critical section***, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section

  i.e. No two processes are executing in their critical sections at the same time.

```
do
    {
        entry section
                critical section
        exit section
                remainder section

    } while (True);
```

Fig: General structure of typical process *Pi*

- The section of code implementing the request is the **entry section**.
- The critical section may be followed by an **exit section**.
- The remaining code is the **remainder section**.

• **Solution to Critical-Section Problem**

A solution to the critical-section problem **must satisfy the following three requirements**:

- **Mutual Exclusion** - If process P_i is executing in its critical section, then **no other processes** can be executing in their critical sections.

- **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then only **those processes that are not executing in their remainder sections** can participate in deciding the **selection of the processes** that will enter the critical section next & **the selection cannot be postponed indefinitely.**

- **Bounded Waiting** - A bound must exist on the **number of times that other processes are allowed to enter their critical sections** after a process has made a request to enter its critical section and before that request is granted.

- **Race Condition**

It’s a situation where:

- Two or more processes reading & writing shared data.
- Output depends on particular order of action.

- Solution: mutual exclusion

Race Condition Continued…

• Conditions acceptable for race condition problem:

- No two processes inside C.S. at the same time
- No process is made to wait for very long time before entering C.S
- Process should be in C.S. for finite time only.
- The solution should not be based on available H/W assumptions (number & speed of CPU)
- Process outside C.S. should not avoid other processes from entering C.S.

Solutions to Critical Section Problem

1. Peterson’s Solution (Only for two processes)

Software solution to critical section problem

```
/* i is this process; j is the other process */

    while (true)
    {
      state[ i ] = interested;                    /* declare interest */
      turn = j;                                   /* turn of j to access C.S.

      while (state[ j ] == interested && turn == j);     /* process i inside c.s.*/

      <<< critical section >>>

      state[ i ] = notinterested;                        /* we’re done */

      <<< code outside critical section >>>
}
```

Peterson’s Solution Continued..

Peterson's Solution (Only for two processes)

```
/* i is this process; j is the other process */
int turn                    / first data structures used
Boolean flag [2]            / second data structures used
 do
    {
        flag[i] = true;              / indicates that Pi is ready to enter its C. S.
        turn = j;                    / variable turn indicates whose turn it is to enter C. S.
        while (flag[j] ==  true && turn == j);


             <<critical section>>


        flag[i] = false;
             remainder section
    } while (true); >
```

Peterson's Solution Continued..

- 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.
- The **eventual value of turn determines** which of the two processes is allowed to enter its critical section first.

This satisfies all three properties of mutual exclusion:

- **Mutual exclusion:** Only one process can be in the critical section at a time
- **Progress:** No process is forced to wait for an available resource
- **Bounded wait:** No process can wait forever for a resource

2. Synchronization Hardware

- TSL (Test & Set Lock instruction) (H/W solution to C. S. problem)

/Assume two processes; Process 0 calls enter_region

```
enter_region:                       ; before execution of TSL
TSL register, flag                  ; copy flag to register and set flag to 1(non zero)
cmp register, #0                    ; was flag zero? Flag is compared to zero
jnz enter_region                    ; if flag was non zero, lock was set, goto first instruction
{
  C.S.                              ; enter critical region
}
leave_region:
mov flag, #0                         ; store zero in flag
return                               ;return
```

- Assume, again, two processes.
- Process 0 calls enter_region.
- The TSL instruction copies the flag to a register and sets it to a non-zero value.
- The flag is now compared to zero (cmp - compare) and if found to be non-zero (jnz - jump if non-zero) the routine loops back to the top.
- Only when process 1 has set the flag to zero (or under initial conditions), by calling leave_region, will process 0 be allowed to continue.

3. Interrupt disabling & enabling

```
other instructions
.
.
.
.
disable interrupts
.
.
.
.

enable interrupts
.
.
.
.
remaining instructions
```

4. Mutex Locks

- The simplest of software tools is the **mutex lock** to solve the critical-section problem.
- *Mutex* is abbreviation for ***mut***ual ***ex***clusion.
- A process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section.
- Calls to either acquire( ) or release( ) must be performed atomically.

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

5. Semaphores

To solve problem of busy wait in previous solutions:

- Waiting process enters into blocked state to free the processor.
- Process issues proper control constructs before entering or exit from critical section.
- OS determines whether process to be blocked or awaken from blocked state.
- This mechanism is accomplished using semaphore.

▪ **Semaphore is non negative protected integer on which only initialization & individual operations ‘p’ (sleep/down/wait) and ‘v’ (wakeup/up/signal) are defined.**

Semaphore Continued..

- Principal operations of semaphore

```
    Begin
0. Initial Routine
1. Down(S)                    // down semaphore
2. C.S
3. UP(S)                      // up semaphore
4. Remaining Portion
    End
```

- Two standard atomic operations wait( ) and signal ( )

(The term **atomic operation** refers to an operation that might be **composed of multiple steps**. If the operation is performed atomically, **either all the steps are performed, or none** are performed. It must not be possible for a subset of the steps to be performed.)

```
wait (S)
          {
                while $S \leq 0$
                 ; // do no-op
                  S--;
          }

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

Semaphore Continued..

- General structure of process in semaphore

Down (S)	UP(S)
D_0 Disable Interrupt	V_0 Disable Interrupt
D_1 if $s > 0$	V_1 $s = s+1$
D_2 then $s = s-1$	V_2 if semaphore queue empty
D_3 else wait on S	V_3 then release a process
D_4 end if	V_4 end if
D_5 Enable interrupt	V5 Enable interrupt
end	end

The sleep operation

- Checks the semaphore to see if it is greater than zero.
- If it is, it decrements the value and continues.
- If the semaphore is zero the process sleeps.

The wakeup operation

- Increments the value of the semaphore.
- If one or more processes were sleeping on that semaphore then one of the processes is chosen and allowed to complete its DOWN.

- Checking and updating the semaphore must be done as an ***atomic*** action to avoid race conditions.
- **All modifications** to 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.

- Here is an example of a series of Down and Up's.
- We are assuming we have a semaphore called *mutex* (for mutual exclusion).
- It is initially set to 1.
- The subscript figure, in this example, represents the process, p, that is issuing the Down.

Down1(mutex) // p1 enters critical section (mutex = 0)

Down2(mutex) // p2 sleeps (mutex = 0)

Down3(mutex) // p3 sleeps (mutex = 0)

Down4(mutex) // p4 sleeps (mutex = 0)

Up(mutex) // mutex = 1 and chooses p3

Down3(mutex) // p3 completes its down (mutex = 0)

Up(mutex) // mutex = 1 and chooses p2

Down2(mutex) // p2 completes its down (mutex = 0)

Up(mutex) // mutex = 1 and chooses p1

Down1(mutex) // p1 completes its down (mutex = 0)

Up(mutex) // mutex = 1 and chooses p4

Down4(mutex) // p4 completes its down (mutex = 0)

Classical Problems of Synchronization

- The classical problems are synchronization problems as examples of a large class of **concurrency-control problems**.
- These problems are **used for testing** every newly proposed synchronization scheme.
- The **semaphores** are used for synchronization.

- Bounded-Buffer Problem (Producer Consumer Problem)
- Reader's-Writer's Problem
- Dining philosopher problem

❖ Producer Consumer Problem

- A Process produces the information which is consumed by consumer.
- Both the Producer & Consumer share a fixe sized common buffer.
- Producer puts an item in buffer & consumer takes it out.
- The trouble arises when producer wants to put new item in buffer but it is already full.
- Also a situation may arise when consumer wants to remove an item but the buffer is empty.

- **Producer Consumer Problem without semaphore:**

```
# define N 100
int  count = 0
Producer ( )
{
          while (true)                    ;repeat forever
         {
                    produce_item( ) ;            ; producer is producing item
                    if (count==N)                ; if buffer is full.
                    sleep ( );
         else
         {          enter_item( );               ; enter item in buffer
                    count ++;
         }
                    If (count==1)                ; wakes up consumer after producing first item
                    wakeup(consumer);
         }
   }
```

```
Consumer
{
      while (true)
     {
      if (count==0)
      sleep ( );
     else
             {
                    remove_item ( );                    ; remove item from buffer
                 count  - -;
              }
        If (count==(N-1))                               ; wakes up consumer after producing first item
        wakeup(Producer);

      consume(item);
     }
}
```

- **Producer Consumer Problem with semaphore:**

```
# define N 100
typedef int semaphore
Semaphore=1                                  //if mutex = 1 enter C.S.
Semaphore_empty = N                          //N= Buffer Size; count for empty & full buffer slot
Semaphore_full = 0
void producer(void)
{
        int item;
        while(TRUE)
        {
        produce_item(&item );                //generate some items;
        down(&empty);                        //decrement empty count
        down(&mutex);                        //entering in C.S.
        enter_item(item);                    //put item in buffer
        up(&mutex);                          //leave C.S.
        up(&full);                           //increment count for full slots
 }
}
```

```
void consumer (void)
{
   int item;
   while(TRUE)
   {
      down(&full);                      // decrement full count is
     down(&mutex);                      //enter in C.S.
      remove_item(&item);               //remove items from buffer
      up(&mutex);                       //leaving C.S.
      up(&empty);                       //increment count of empty slots
       consume_item(&item);             //print item
    }
}
```

❖ Reader’s-Writer’s Problem using Semaphore

- A data set is shared among a number of concurrent processes
- Readers – only read the data set; they do not perform any updates
- Writers – can both read and write
- Multiple readers to read at the same time can be allowed
- Problem – if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may arise.
- Only one single writer can access the shared data at the same time
- e.g. Airline Reservation System
    - Many Companies read & write d/b
    - Multiple processes read d/b at the same time

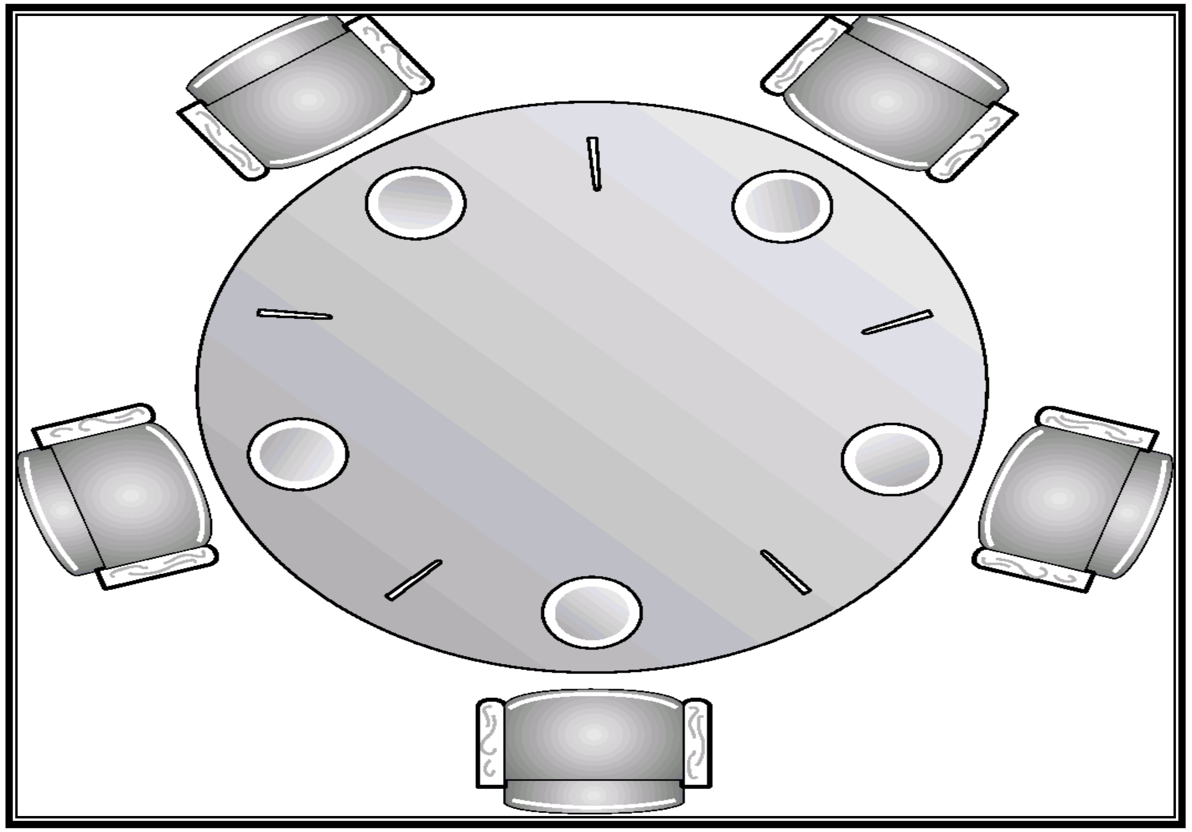
- **Reader's-Writer's Problem using Semaphore**

```
typedef int semaphore
semaphore mutex=1                          //control access to rc
semaphore db=1                             //control access to d/b
int rc=0                                   //rc is read count
void reader(void)
{
 while (TRUE)
        {
                   DOWN(mutex);            //access to rc
                   rc=rc+1
                  If   (rc==1)             // if it is the first reader then
                  DOWN(&db);               // closes the lock of d/b
                  UP (&mutex);             //opens the lock of semaphore for reading
                  read_db( );
                  DOWN(& mutex);          // locks the semaphore after reading
```

Continued….

```
          rc=rc-1;
                    if (rc==0)
                    UP(&db);                   //opens the lock of d/b
                    UP(& mutex);              // opens the semaphore for future use
                    use data_read( );
          }
}
void writer(void)
{
          while(TRUE);
          {
                    think_update( );            //requires data updation
                    DOWN(&db);                  //get exclusive access to db
                    write_database( );
                    UP(&db);
          }
}
```

Dining-Philosophers Problem



Concurrency control problem

Dining-Philosophers Problem Continued….

- Five philosophers, each either eats or thinks
- Share a circular table with five chopsticks
- Thinking: do nothing
    - Eating => need two chopsticks, try to pick up two closest chopsticks
    - Block if neighbor has already picked up a chopstick
- After eating, put down both chopsticks and go back to thinking

Dining-Philosophers Problem Continued….

Shared data

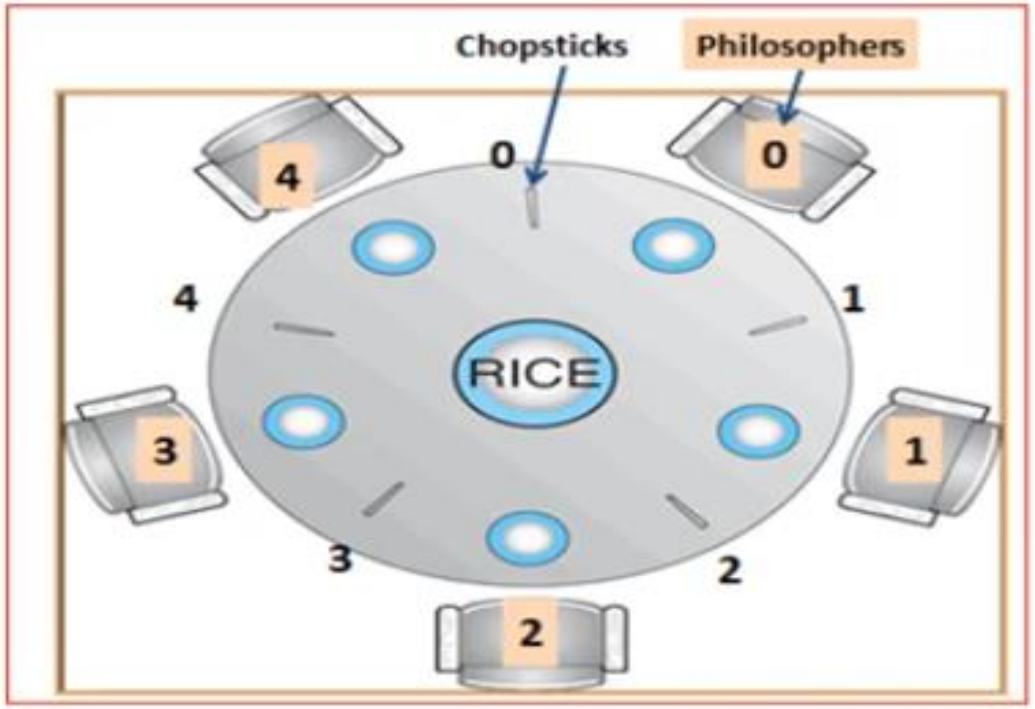
semaphore chopstick[5];

Each chopstick = one semaphore
Wait ( ) – grab chopstick & signal ( )– release chopsticks

- The five Philosophers are represented as P0, P1, P2, P3, & P4 and five chopsticks by C0, C1, C2, C3, and C4.

- Philosopher *i*:

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



Dining-Philosophers Problem Continued….

- Let value of i = 0(initial value), Suppose **Philosopher P0** wants to eat, it will enter in Philosopher() function, and execute **Wait(chopstickC[i]);** by doing this it holds **C0 chopstick** and reduces **semaphore C0 to 0;**
- After that it execute **Wait(chopstickC[(i+1) % 5]);** by doing this it holds **C1 chopstick** (since i =0, therefore (0 + 1) % 5 = 1) and **reduces semaphore C1 to 0.**
- Suppose now **Philosopher P1** wants to eat, it will enter in Philosopher() function, and execute **Wait(chopstickC[i])**; by doing this it will try to hold C1 chopstick but will not be able to do that, since the value of semaphore C1 has already been set to 0 by philosopher P0, therefore it will **enter into an infinite loop** because of which philosopher P1 will not be able to pick chopstick C1.
- Now if **Philosopher P2** wants to eat, it will enter in Philosopher() function, and execute Wait(chopstickC[i]); by doing this it **holds C2 chopstick** and reduces **semaphore C2 to 0;**
- After that, it executes Wait(chopstickC[(i+1) % 5]); by doing this it holds **C3 chopstick**(since i =2, therefore (2 + 1) % 5 = 3) and reduces **semaphore C3 to 0**.

- Hence the above code is providing a **solution** to the dining philosopher problem.
- A philosopher **can only eat if both immediate left and right chopsticks of the philosopher are available** else philosopher needs to wait.
- Also **at one go two independent philosophers can eat simultaneously** (i.e., philosopher P0 and P2, P1 and P3 & P2 and P4 can eat simultaneously as all are the independent processes and they are following the above constraint of dining philosopher problem)

- Semaphore solution

- No two philosophers eating simultaneously.
- Philosopher tries to grab chopstick executes wait( ) and execute signal( ) when leaves the chopstick.
- All elements of chopsticks initialized to 1.
- What happen next?
    - If all philosophers grabs her left chopstick
    - Philosopher delayed forever (deadlock)

- **Solution (free from deadlock)**
  - Allow 4 philosophers at a time.
  - Allow philosophers to pick two chopsticks if available.
  - Asymmetric solution: Even-Odd

  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.

❖ Monitors

- Using semaphores incorrectly can result in timing errors that are difficult to detect.
- A ***monitor type*** is an ADT that **includes a set of programmer-defined operations that are provided with mutual exclusion within the monitor.**
- **Set of multiple routines** which are **protected by a mutual exclusion lock**.
- Routine executed **iff** thread acquires lock.
- **Only one thread can execute within the monitor at a time**.
- Other threads must wait.
- Thread can actually suspend itself inside a monitor and then wait for an event

  x.wait( ); process suspended till another process invokes x.signal( );
- Monitors are **simpler to use than semaphores**.
- Monitors, unlike semaphores, **automatically acquire the necessary locks**.

ADT*—encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT. eg: List, Stack, Queue*

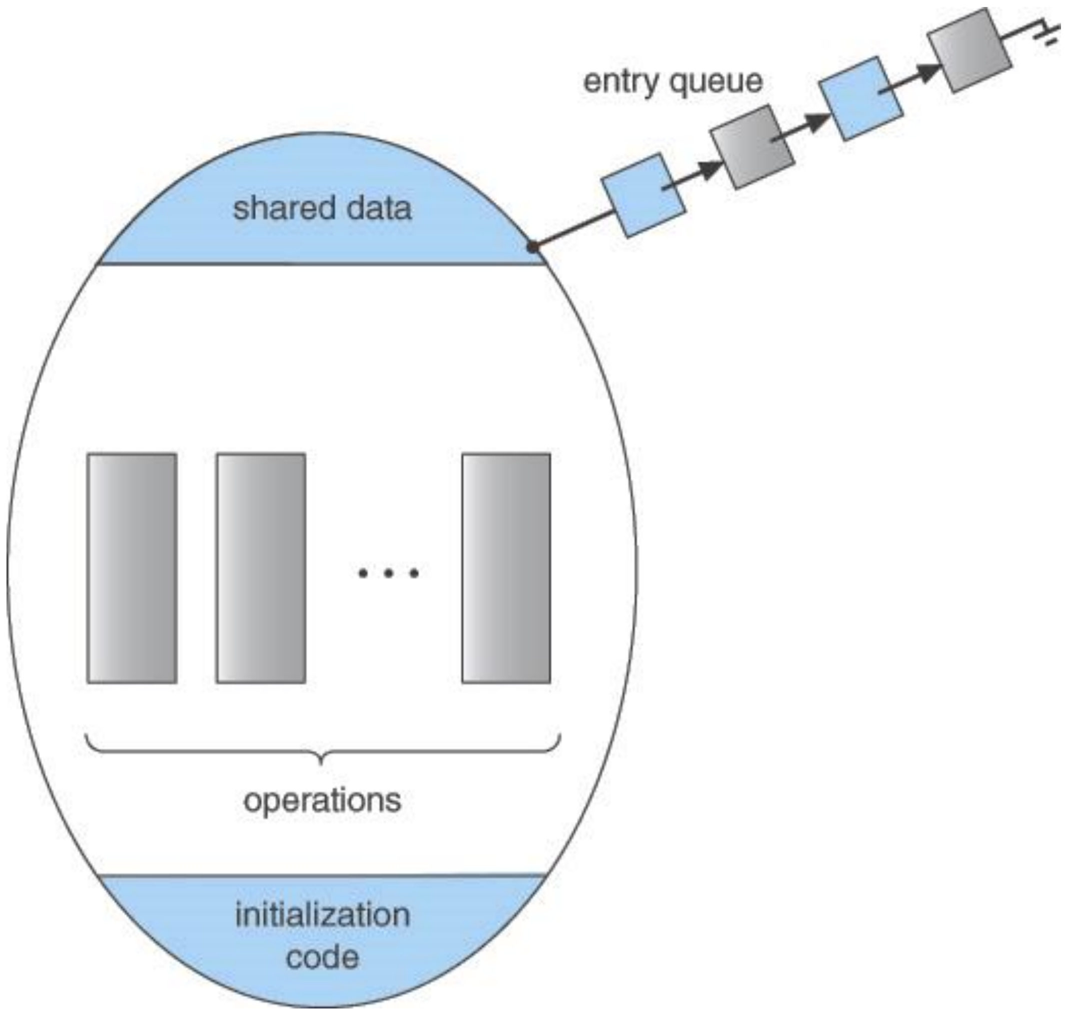


Figure: Schematic view of a monitor

```
monitor monitor name
{
        /* shared variable declarations */
        function P1 ( . . . ) {
        . . .
        }
        function P2 ( . . . ) {
        . . .
        }
        .
        .
        function Pn ( . . . ) {
        . . .
        }
        initialization code ( . . . ) {
        . . .
        }
}
```

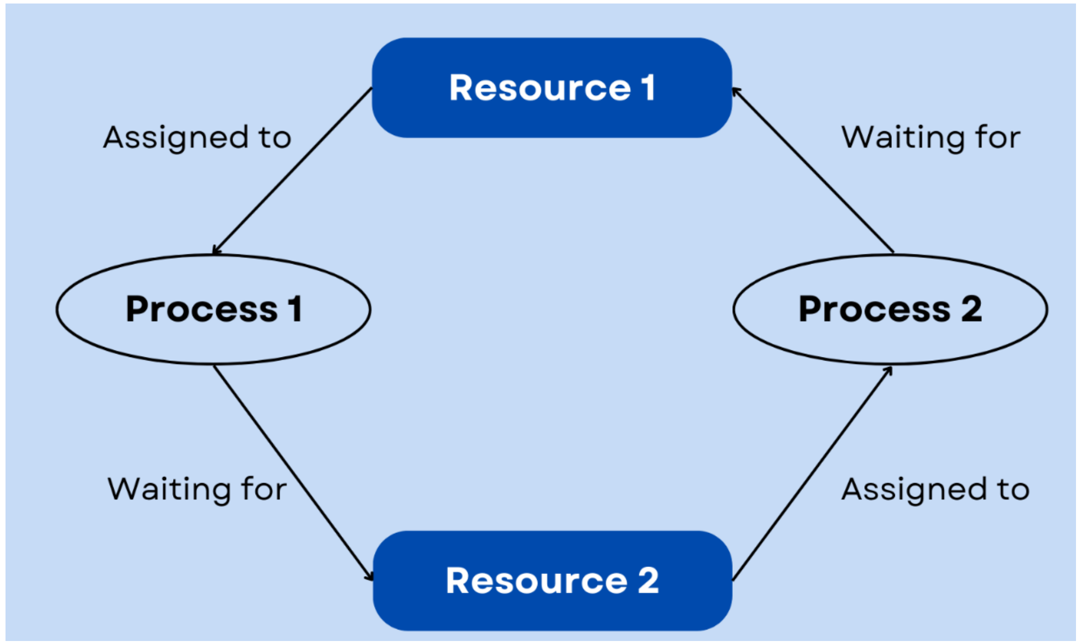
Syntax of a monitor

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**.

Deadlock

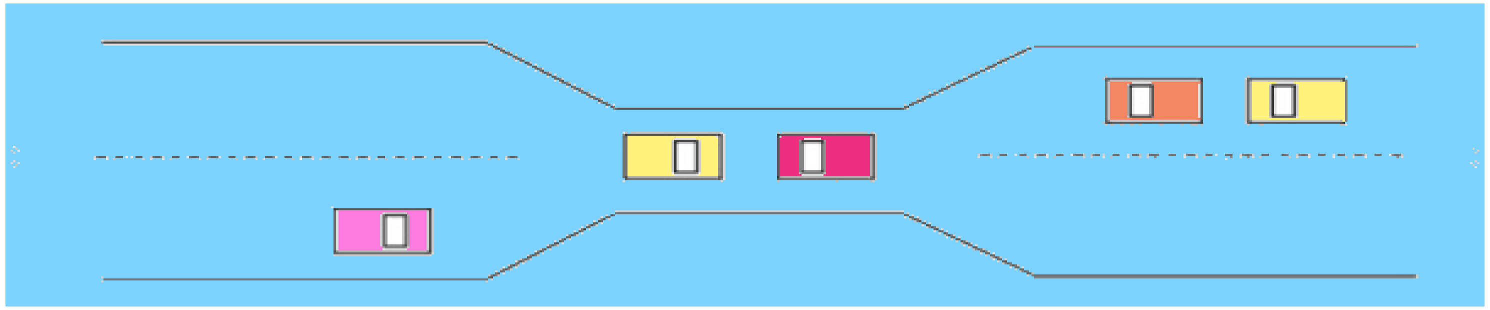
❖ The Deadlock Problem:

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - System has 2 disk drives
  - Process 1 and 2 each hold one disk drive and each needs another one

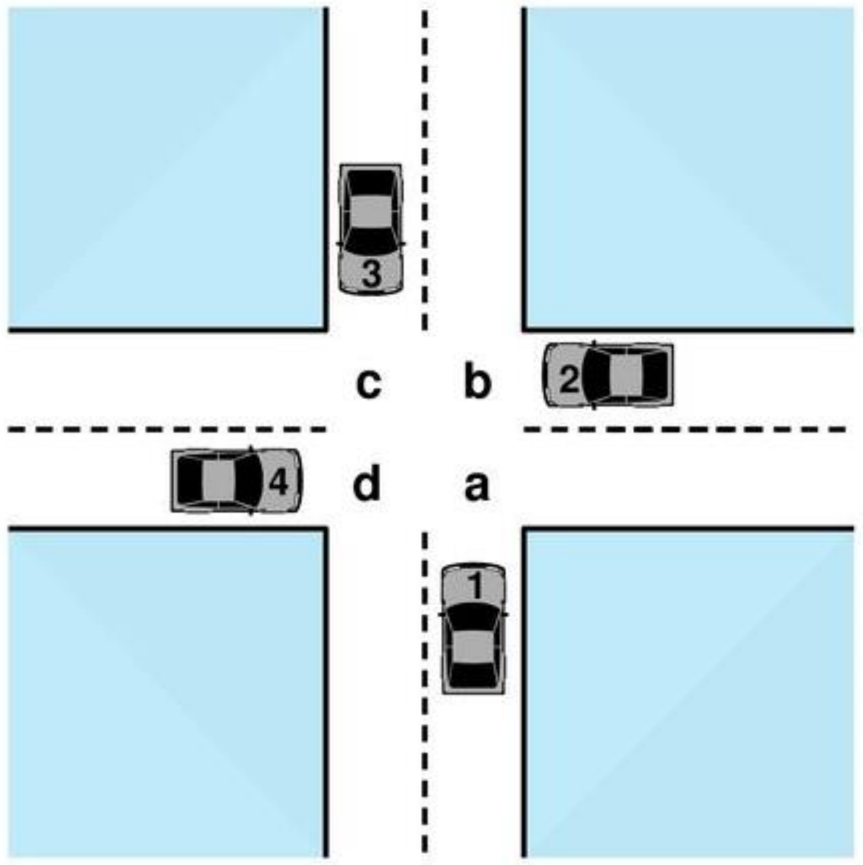


DEADLOCKS

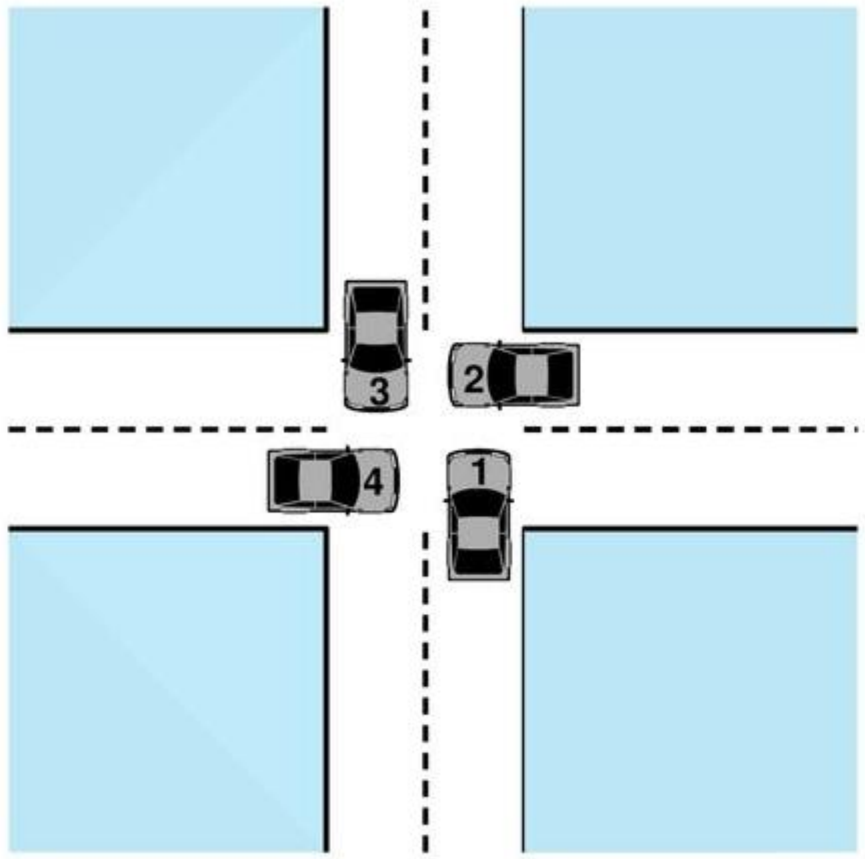
Bridge Crossing Example



- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.



(a) Deadlock possible



(b) Deadlock

❖ **System Model**

- System consists of resources
- Sequence of resource utilization by a process:
  - Request
  - Use
  - Release

Request → Use → Release

e.g.

Consider a system with one printer and one DVD drive.

Suppose that process **P1 is holding the DVD** and process **P2 is holding the printer**.

If P1, requests the printer and P2 requests the DVD drive, a deadlock occurs.

❖ Deadlock Characterization

Necessary Conditions to occur deadlock:

A deadlock can arise if all the following four conditions hold simultaneously in a system:

- Mutual exclusion: at least one resource cannot be shared; only one process at a time can use the resource.

  Processes claim exclusive control of resources.

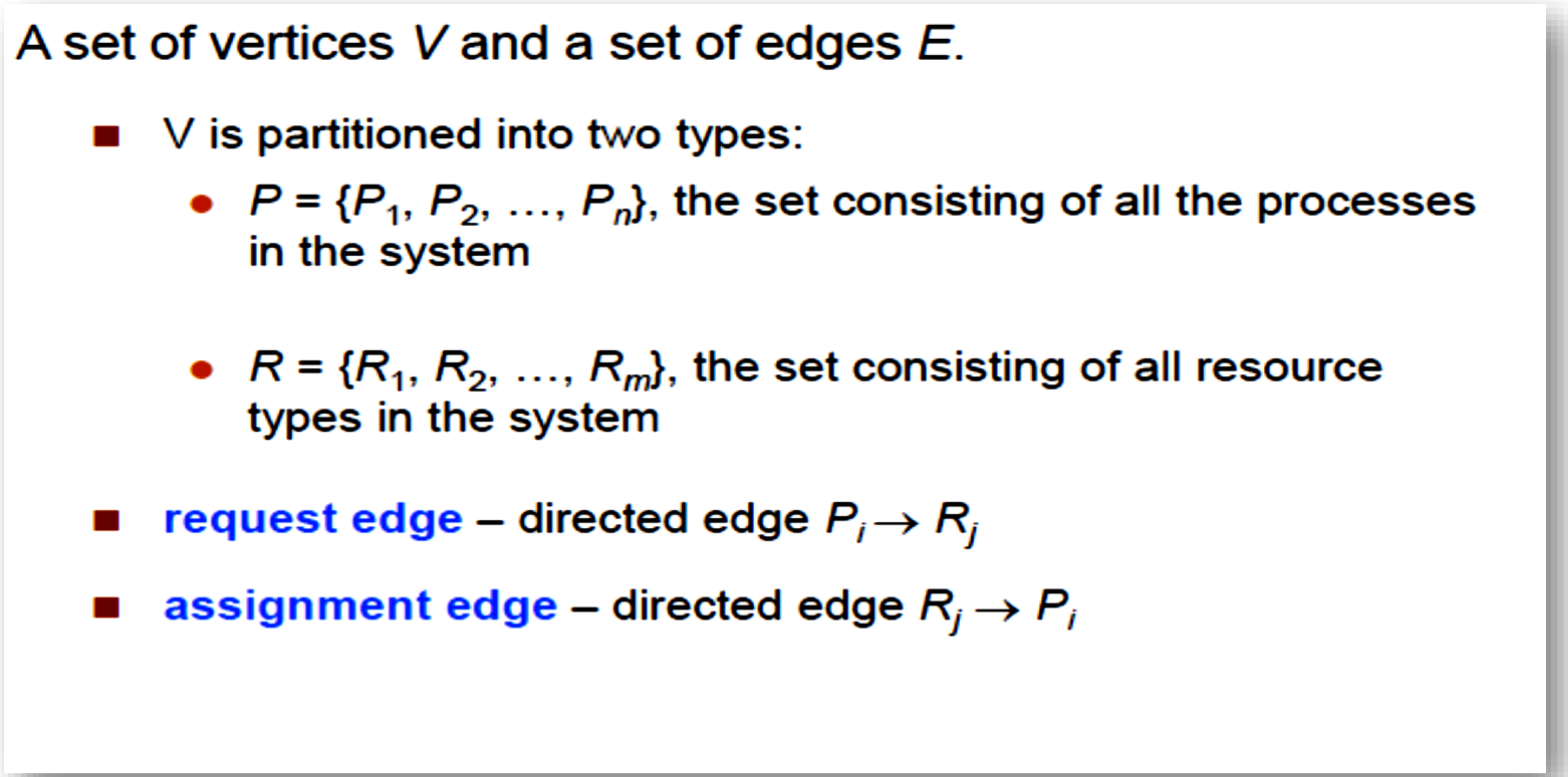
- Hold and wait: a process holding at least one resource & waiting to acquire additional resources held by other processes.

- No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task.

- Circular wait: circular chain of processes exists in which each holds one or more resources requested by the next process in the chain. {P0, P1, P2,…Pn-1, Pn}

❖ Resource-Allocation Graph

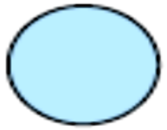
- **A directed graph can describe deadlock more precisely**



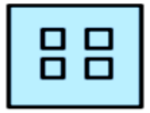
A set of vertices *V* and a set of edges *E*.

- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

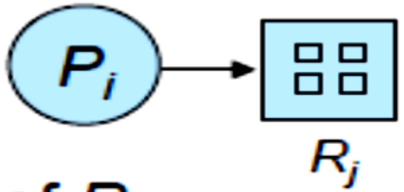
- Process



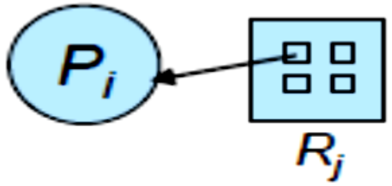
- Resource Type with 4 instances



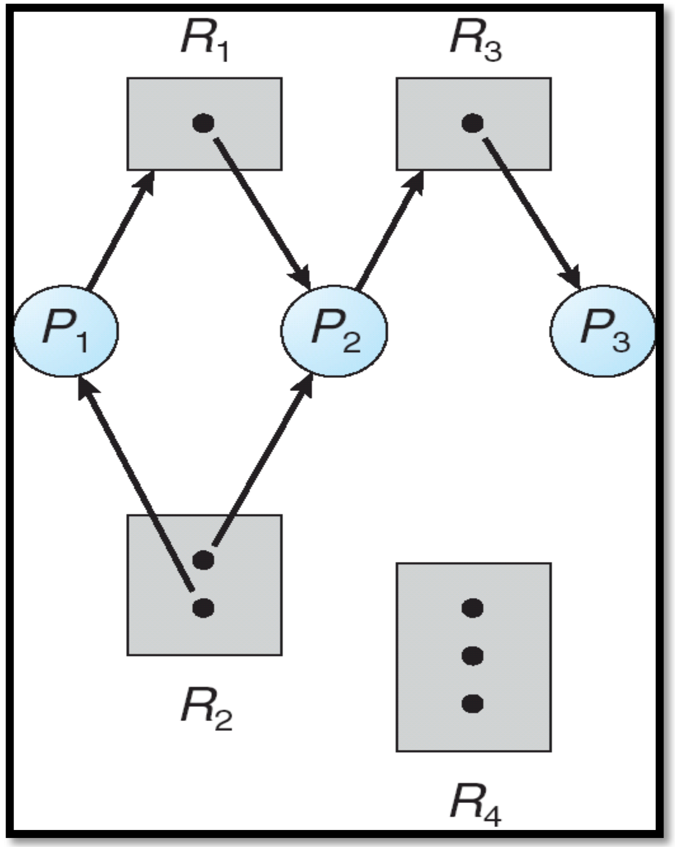
- P_i requests instance of R_j



- P_i is holding an instance of R_j



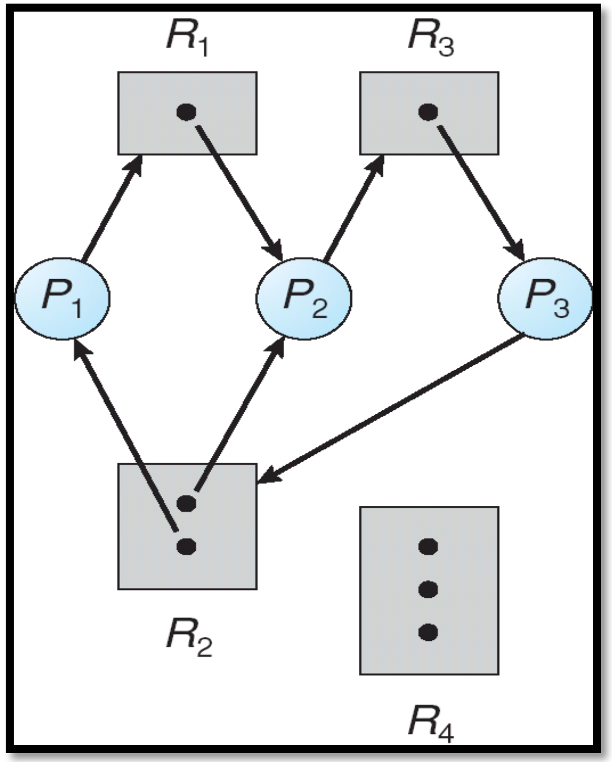
• Example of Resource-Allocation Graph



$P_i \rightarrow R_j$ – request edge
$R_j \rightarrow P_i$ – assignment edge

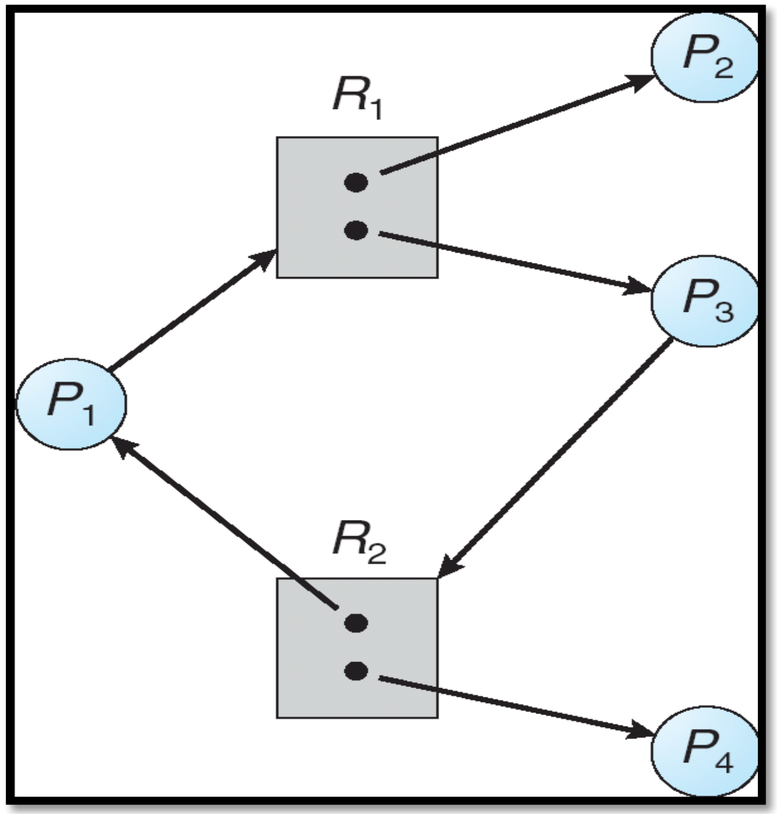
If the graph has a cycle?

- **Resource Allocation Graph With A Deadlock**
  - If graph contains **no cycl**es ⇒ **no deadlock**
  - If graph contains a cycle ⇒
    - if only one instance per resource type, then deadlock occurs
    - if several instances per resource type, possibility of deadlock



Cycle 1: P1 $\longrightarrow$ R1 $\longrightarrow$ P2 $\longrightarrow$ R3 $\longrightarrow$ P3 $\longrightarrow$ R2 $\longrightarrow$ P1
Cycle 2: P2 $\longrightarrow$ R3 $\longrightarrow$ P3 $\longrightarrow$ R2 $\longrightarrow$ P2

- **Graph With A Cycle But No Deadlock**



Cycle: P1 $\longrightarrow$ R1 $\longrightarrow$ P3 $\longrightarrow$ R2 $\longrightarrow$ P1

❖ **Methods for Handling Deadlocks**

- *Ignore* the problem/deadlock and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.
- Use a protocol to ensure that the system will *never* enter a deadlock state.
- Allow the system to enter a deadlock state and then *recover*.

❖ Deadlock Prevention

By ensuring at least one of the following condition can’t hold, deadlock can be prevented.

- Mutual Exclusion: We cannot prevent deadlocks by denying the mutual-exclusion condition; not possible due to intrinsically non-sharable devices (eg - Read-only files)
- Hold and Wait: Resource requesting process should not hold any other resource. Two protocols are used:
  1: Allocating all the needed resources when starting a process.
  2: A process is allowed to request a resource only if it does not hold any resource.
- No Preemption: Currently held resources are preempted & implicitly released.
- Circular Wait: Process must request resources in linear ascending order.

- **Universally accepted solution for deadlock prevention cant be available**

❖ Deadlock Avoidance

- A method for avoiding deadlocks is to require additional information about how resources are to be requested.

- **Banker’s Algorithm for Deadlock Avoidance**

▪ **Safe State:** If system can allocate resources to each requesting process.

- **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 process P_i is currently allocated k instances of R_j.

- Need: n x m matrix.

  If Need[i, j] = k, then process P_i may need k more instances of R_j to complete its task.

$$\boxed{\textbf{Need [i, j] = Max[i, j] – Allocation [i, j]}}$$

- **Safety Algorithm**

We can now present the algorithm for finding out **whether or not a system is in a safe state.**

This algorithm can be described as follows:

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) **$\text{Need}_i \leq \text{Work}$**

   If no such i exists, go to step 4.

3. **$\text{Work} = \text{Work} + \text{Allocation}_i$**
   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**

$\boldsymbol{Request_i}$ = request vector for process P_i.

If $\boldsymbol{Request_i[j] = k}$ then process P_i wants $\boldsymbol{k}$ instances of resource type R_j

1. If $\boldsymbol{Request_i \leq Need_i}$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $\boldsymbol{Request_i \leq Available}$, 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

Numerical on Banker's Algorithm

- Consider the following snapshot of a system:

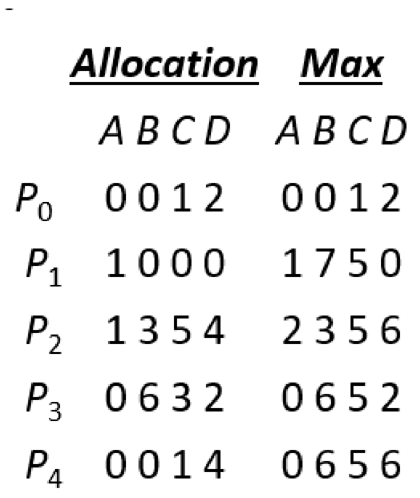
	Allocation	***Max***	***Available***
	A B C D	*A B C D*	*A B C D*
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

i. What are the contents of Need matrix?
ii. Is system in safe state? Write possible safe sequence.
iii. If a request from process P1 arrives for (0 4 2 0), can the request be granted immediately?

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

Need = Max – Allocation.

	Need
	A B C D
P_0	0 0 0 0
P_1	0 7 5 0
P_2	1 0 0 2
P_3	0 0 2 0
P_4	0 6 4 2



	Allocation	***Max***
	A B C D	*A B C D*
P_0	0 0 1 2	0 0 1 2
P_1	1 0 0 0	1 7 5 0
P_2	1 3 5 4	2 3 5 6
P_3	0 6 3 2	0 6 5 2
P_4	0 0 1 4	0 6 5 6

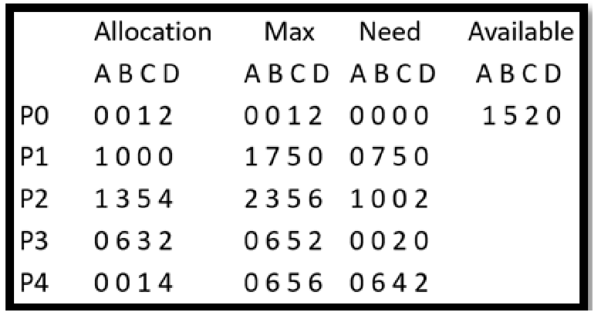
	Allocation	Max	Need	Available
	A B C D	A B C D	A B C D	A B C D
P0	0 0 1 2	0 0 1 2	0 0 0 0	1 5 2 0
P1	1 0 0 0	1 7 5 0	0 7 5 0	
P2	1 3 5 4	2 3 5 6	1 0 0 2	
P3	0 6 3 2	0 6 5 2	0 0 2 0	
P4	0 0 1 4	0 6 5 6	0 6 4 2	

Work = Available
$Need_i \leq$ Work
Work = Work + $Allocation_i$

	Allocation	Max	Need	Available
	A B C D	A B C D	A B C D	A B C D
P0	0 0 1 2	0 0 1 2	0 0 0 0	1 5 2 0
P1	1 0 0 0	1 7 5 0	0 7 5 0	
P2	1 3 5 4	2 3 5 6	1 0 0 2	
P3	0 6 3 2	0 6 5 2	0 0 2 0	
P4	0 0 1 4	0 6 5 6	0 6 4 2	

		Available	Available + Allocation (Pi)	
		1 5 2 0		
			1 5 2 0	Work
			+ 0 0 1 2	(P_0) Allocation
Process	P_0	1 5 3 2	1 5 3 2	(New) Work
			1 5 3 2	Work
			+ 1 3 5 4	(P_2) Allocation
Process	P_2	2 8 8 6	2 8 8 6	(New) Work
			2 8 8 6	Work
			+ 0 6 3 2	(P_3) Allocation
Process	P_3	2 14 11 8	2 14 11 8	(New) Work
			2 14 11 8	Work
			+ 0 0 1 4	(P_4) Allocation
Process	P_4	2 14 12 12	2 14 12 12	
			2 14 12 12	Work
			+ 1 0 0 0	(P1) Allocation
Process	P1	3 14 12 12	3 14 12 12	(New) Work

The system is in safe state with the sequence P0 -> P2 -> P3 -> P4 -> P1



	Allocation	Max	Need	Available
	A B C D	A B C D	A B C D	A B C D
P0	0 0 1 2	0 0 1 2	0 0 0 0	1 5 2 0
P1	1 0 0 0	1 7 5 0	0 7 5 0	
P2	1 3 5 4	2 3 5 6	1 0 0 2	
P3	0 6 3 2	0 6 5 2	0 0 2 0	
P4	0 0 1 4	0 6 5 6	0 6 4 2	

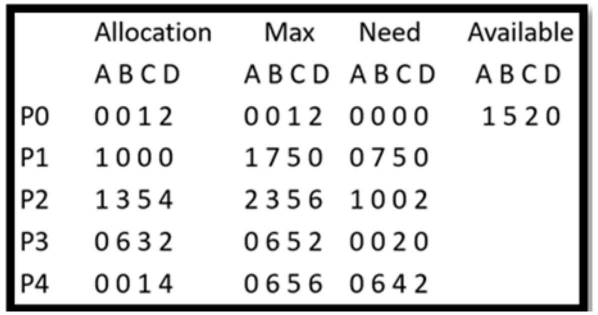
- **Safety Algo: Finish[i] = true**

❖Resource-Request Algorithm for Process P_i

- ***New Available* = Previous *Available* – *Request*$_i$**

= 1520 – 0420

= 1100



	Allocation	Max	Need	Available
	A B C D	A B C D	A B C D	A B C D
P0	0 0 1 2	0 0 1 2	0 0 0 0	1 5 2 0
P1	1 0 0 0	1 7 5 0	0 7 5 0	
P2	1 3 5 4	2 3 5 6	1 0 0 2	
P3	0 6 3 2	0 6 5 2	0 0 2 0	
P4	0 0 1 4	0 6 5 6	0 6 4 2	

	Allocation	**Max**	**Available**
	A B C D	A B C D	A B C D
P0	0 0 1 2	0 0 1 2	**1 1 0 0**
P1	**1 4 2 0**	1 7 5 0	
P2	1 3 5 4	2 3 5 6	
P3	0 6 3 2	0 6 5 2	
P4	0 0 1 4	0 6 5 6	

P1 = 1000(Allocation) + 0420 (Request i)

= 1420

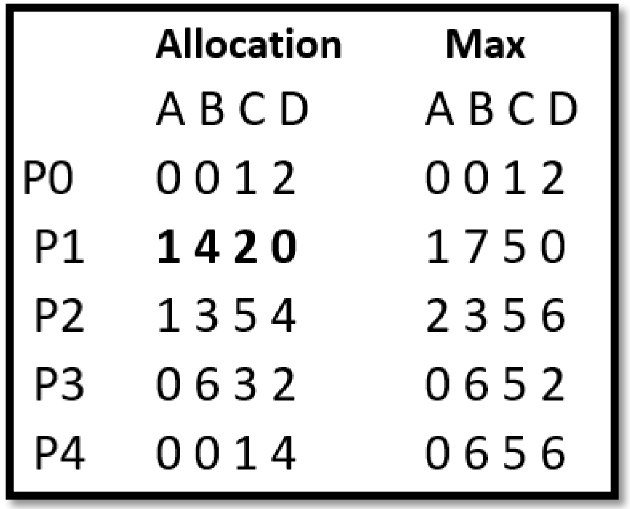
$$\boxed{Allocation_i = Allocation_i + Request_i}$$

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

Need = Max – Allocation.

Need

	A B C D
P_0	0 0 0 0
P_1	**0 3 3 0**
P_2	1 0 0 2
P_3	0 0 2 0
P_4	0 6 4 2

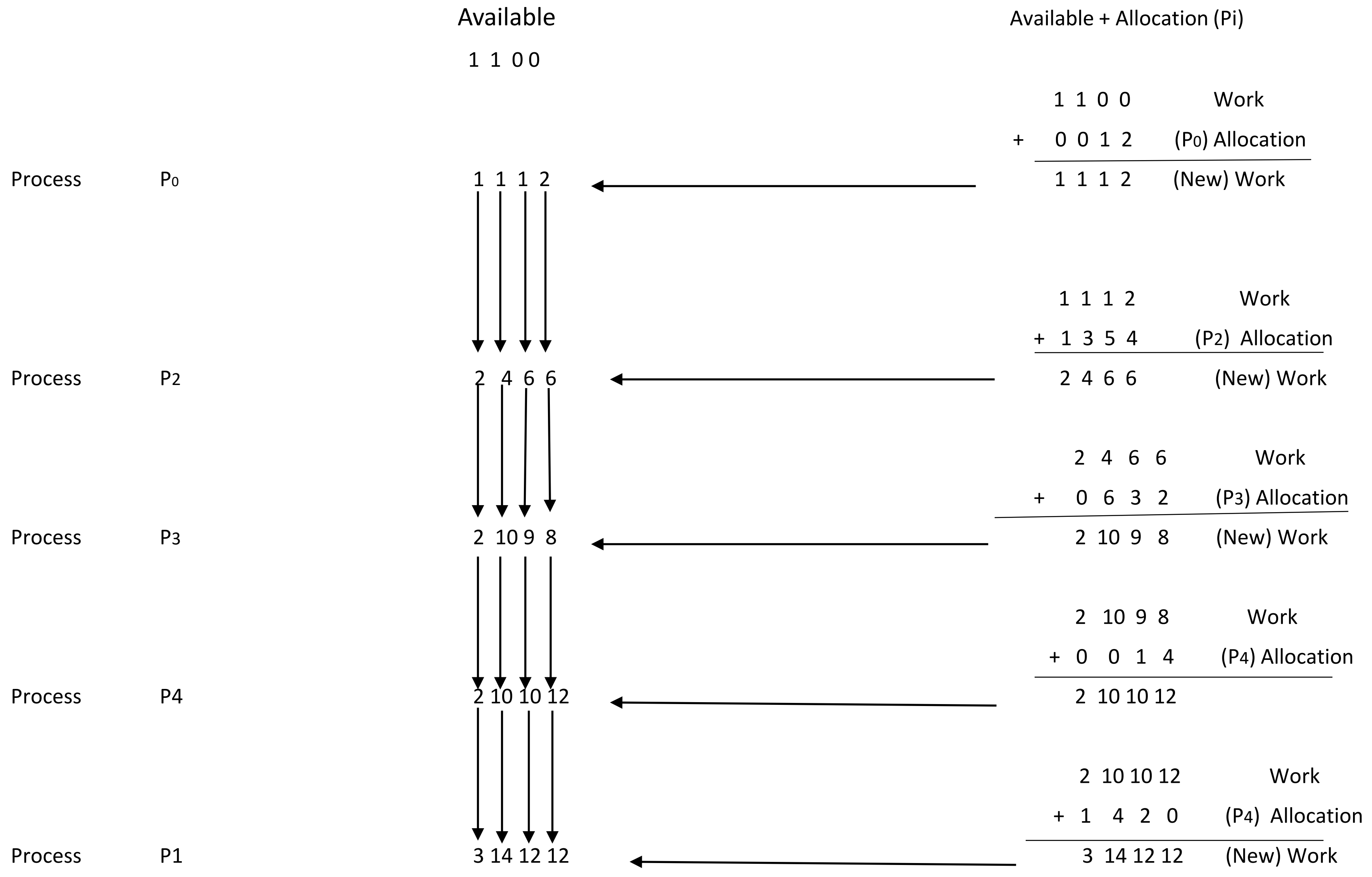


	Allocation	**Max**
	A B C D	A B C D
P0	0 0 1 2	0 0 1 2
P1	**1 4 2 0**	1 7 5 0
P2	1 3 5 4	2 3 5 6
P3	0 6 3 2	0 6 5 2
P4	0 0 1 4	0 6 5 6

	Allocation	Max	Need	Available
	A B C D	A B C D	A B C D	A B C D
P0	0 0 1 2	0 0 1 2	0 0 0 0	**1 1 0 0**
P1	**1 4 2 0**	1 7 5 0	**0 3 3 0**	
P2	1 3 5 4	2 3 5 6	1 0 0 2	
P3	0 6 3 2	0 6 5 2	0 0 2 0	
P4	0 0 1 4	0 6 5 6	0 6 4 2	

Work = Available
Need$_i$ ≤ Work
Work = Work + Allocation$_i$

	Allocation	Max	Need	Available
	A B C D	A B C D	A B C D	A B C D
P0	0 0 1 2	0 0 1 2	0 0 0 0	**1 1 0 0**
P1	**1 4 2 0**	1 7 5 0	**0 3 3 0**	
P2	1 3 5 4	2 3 5 6	1 0 0 2	
P3	0 6 3 2	0 6 5 2	0 0 2 0	
P4	0 0 1 4	0 6 5 6	0 6 4 2	



```
  1 1 0 0        Work
+ 0 0 1 2        (P0) Allocation
  1 1 1 2        (New) Work

  1 1 1 2        Work
+ 1 3 5 4        (P2)  Allocation
  2 4 6 6        (New) Work

  2 4 6 6        Work
+ 0 6 3 2        (P3) Allocation
  2 10 9 8       (New) Work

  2 10 9 8       Work
+ 0 0 1 4        (P4) Allocation
  2 10 10 12

  2 10 10 12     Work
+ 1 4 2 0        (P4)  Allocation
  3 14 12 12     (New) Work
```

Hence new request can be immediately granted in the safe state if sequence is P0 -> P2 -> P3 -> P4 -> P1

❖ Consider following snapshot of system.

	Allocation	Max	Available
P_0	2 0 1 0	3 2 1 1	1 1 2 1
P_1	1 1 0 0	1 2 0 2	
P_2	1 1 0 0	1 1 2 0	
P_3	1 0 1 0	3 2 1 0	
P_4	0 1 0 1	2 1 0 1	

i) What are the contents of Need matrix?

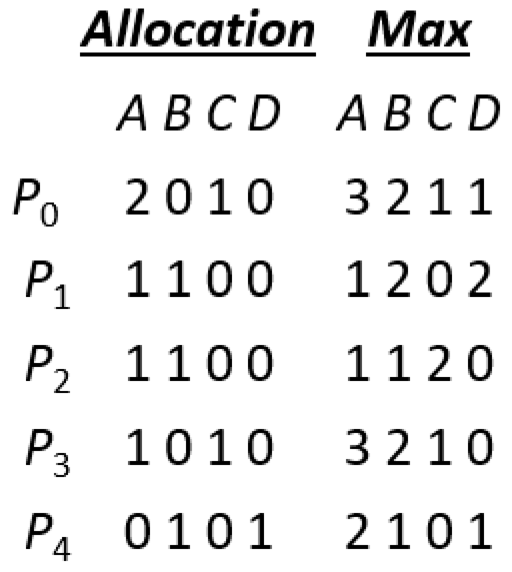
ii) Is system in safe state? Write possible safe sequence.

iii) If a request from process P1 arrives for (0 1 0 1), can the request be granted immediately?

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

Need = Max – Allocation.

	Need
	A B C D
P_0	1 2 0 1
P_1	0 1 0 2
P_2	0 0 2 0
P_3	2 2 0 0
P_4	2 0 0 0



	Allocation	***Max***
	A B C D	*A B C D*
P_0	2 0 1 0	3 2 1 1
P_1	1 1 0 0	1 2 0 2
P_2	1 1 0 0	1 1 2 0
P_3	1 0 1 0	3 2 1 0
P_4	0 1 0 1	2 1 0 1

	Allocation	Max	Need	Available
	A B C D	A B C D	A B C D	A B C D
P0	2 0 1 0	3 2 1 1	1 2 0 1	1 1 2 1
P1	1 1 0 0	1 2 0 2	0 1 0 2	
P2	1 1 0 0	1 1 2 0	0 0 2 0	
P3	1 0 1 0	3 2 1 0	2 2 0 0	
P4	0 1 0 1	2 1 0 1	2 0 0 0	

Work = Available
$Need_i \leq Work$
$Work = Work + Allocation_i$

	Allocation	Max	Need	Available
	A B C D	A B C D	A B C D	A B C D
P0	2 0 1 0	3 2 1 1	1 2 0 1	1 1 2 1
P1	1 1 0 0	1 2 0 2	0 1 0 2	
P2	1 1 0 0	1 1 2 0	0 0 2 0	
P3	1 0 1 0	3 2 1 0	2 2 0 0	
P4	0 1 0 1	2 1 0 1	2 0 0 0	

Available

1 1 2 1

Available + Allocation (Pi)

```
                                                      1 1 2 1      Work
                                                    + 1 1 0 0      (P2) Allocation
                                                    ---------
Process   P2        2 2 2 1   <-----------------      2 2 2 1      (New) Work

                                                      2 2 2 1      Work
                                                    + 1 0 1 0      (P3) Allocation
                                                    ---------
Process   P3        3 2 3 1   <-----------------      3 2 3 1      (New) Work

                                                      3 2 3 1      Work
                                                    + 0 1 0 1      (P4) Allocation
                                                    ---------
Process   P4        3 3 3 2   <-----------------      3 3 3 2      (New) Work

                                                      3 3 3 2      Work
                                                    + 2 0 1 0      (P0) Allocation
                                                    ---------
Process   P0        5 3 4 2   <-----------------      5 3 4 2

                                                      5 3 4 2      Work
                                                    + 1 1 0 0      (P1) Allocation
                                                    ---------
Process   P1        6 4 4 2   <-----------------      6 4 4 2      (New) Work
```

The system is in safe state with the sequence P2 -> P3 -> P4 -> P0 -> P1

	Allocation	Max	Need	Available
	A B C D	A B C D	A B C D	A B C D
P0	2 0 1 0	3 2 1 1	1 2 0 1	1 1 2 1
P1	1 1 0 0	1 2 0 2	0 1 0 2	
P2	1 1 0 0	1 1 2 0	0 0 2 0	
P3	1 0 1 0	3 2 1 0	2 2 0 0	
P4	0 1 0 1	2 1 0 1	2 0 0 0	

- **Safety Algo: Finish[i] = true**

❖**Resource-Request Algorithm for Process P_i**

- ***New Available* = Previous *Available* – *Request*$_i$**

= 1121 – 0101

= 1020

	Allocation	Max	Need	Available
	A B C D	A B C D	A B C D	A B C D
P0	2 0 1 0	3 2 1 1	1 2 0 1	1 1 2 1
P1	1 1 0 0	1 2 0 2	0 1 0 2	
P2	1 1 0 0	1 1 2 0	0 0 2 0	
P3	1 0 1 0	3 2 1 0	2 2 0 0	
P4	0 1 0 1	2 1 0 1	2 0 0 0	

	Allocation	**Max**	**Available**
	A B C D	A B C D	A B C D
P0	2 0 1 0	3 2 1 1	**1 0 2 0**
P1	**1 2 0 1**	1 2 0 2	
P2	1 1 0 0	1 1 2 0	
P3	1 0 1 0	3 2 1 0	
P4	0 1 0 1	2 1 0 1	

$$\boxed{Allocation_i = Allocation_i + Request_i}$$

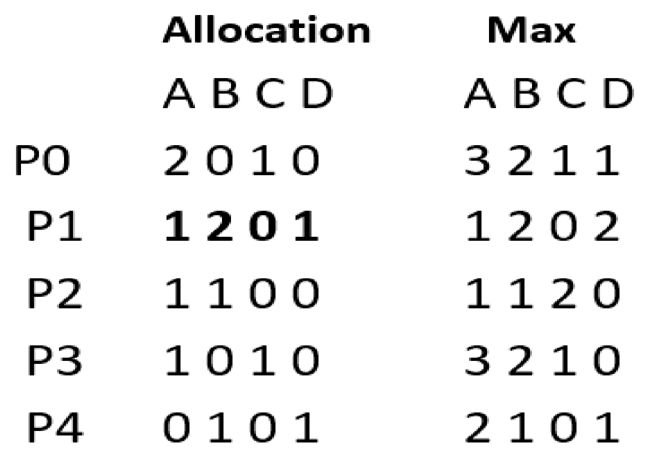
P1 = 1100(Allocation) + 0101 (Request $_i$)

= 1201

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

Need = Max – Allocation.

	<u>Need</u>
	A B C D
P_0	1 2 0 1
P_1	**0 0 0 1**
P_2	0 0 2 0
P_3	2 2 0 0
P_4	2 0 0 0



	Allocation	**Max**
	A B C D	A B C D
P0	2 0 1 0	3 2 1 1
P1	**1 2 0 1**	1 2 0 2
P2	1 1 0 0	1 1 2 0
P3	1 0 1 0	3 2 1 0
P4	0 1 0 1	2 1 0 1

	Allocation	Max	Need	Available
	A B C D	A B C D	A B C D	A B C D
P0	2 0 1 0	3 2 1 1	1 2 0 1	**1 0 2 0**
P1	**1 2 0 1**	1 2 0 2	**0 0 0 1**	
P2	1 1 0 0	1 1 2 0	0 0 2 0	
P3	1 0 1 0	3 2 1 0	2 2 0 0	
P4	0 1 0 1	2 1 0 1	2 0 0 0	

Work = Available
$\text{Need}_i \leq$ Work
Work = Work + Allocation_i

	Allocation	Max	Need	Available
	A B C D	A B C D	A B C D	A B C D
P0	2 0 1 0	3 2 1 1	1 2 0 1	**1 0 2 0**
P1	**1 2 0 1**	1 2 0 2	**0 0 0 1**	
P2	1 1 0 0	1 1 2 0	0 0 2 0	
P3	1 0 1 0	3 2 1 0	2 2 0 0	
P4	0 1 0 1	2 1 0 1	2 0 0 0	

		Available	Available + Allocation (Pi)	
		1 0 2 0		
			1 0 2 0	Work
			+ 1 1 0 0	(P_2) Allocation
Process	P_2	2 1 2 0	2 1 2 0	(New) Work
			2 1 2 0	Work
			+ 0 1 0 1	(P_4) Allocation
Process	P_4	2 2 2 1	2 2 2 1	(New) Work
			2 2 2 1	Work
			+ 2 0 1 0	(P_0) Allocation
Process	P_0	4 2 3 1	4 2 3 1	(New) Work
			4 2 3 1	Work
			+ 1 2 0 1	(P1) Allocation
Process	P1	5 4 3 2	5 4 3 2	
			5 4 3 2	Work
			+ 1 0 1 0	(P3) Allocation
Process	P3	6 4 4 2	6 4 4 2	(New) Work

The system is in safe state with the sequence P2 -> P4 -> P0 -> P1 -> P3

❖ Consider system with 5 processes $P_0 – P_4$. There are 3 resource type A, B, C.

Where A = 10 instances

B = 5 instances

C = 7 instances

Suppose following is snapshot of system at particular instance

	Allocation	Max
	A B C	A B C
P_0	0 1 0	7 5 3
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

- Whether the system in safe state?

- Calculate Available Resources

$A = \text{total given} – (A_{P_0}+A_{P_1}+A_{P_2}+A_{P_3}+A_{P_4})$

$= 10 – (0+2+3+2+0)$

$= 3$

$B = \text{total given} – (B_{P_0}+B_{P_1}+B_{P_2}+B_{P_3}+B_{P_4})$

$= 5- (1+0+0+1+0)$

$= 3$

$C = \text{total given} – (C_{P_0}+C_{P_1}+C_{P_2}+C_{P_3}+C_{P_4})$

$= 7 – (0+0+2+1+2)$

$= 2$

Thus,

A= 3, B= 3, C=2

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

Need = 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

	Allocation	**Max**	**Need**	**Available**
	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	2 0 0	3 2 2	1 2 2	
P2	3 0 2	9 0 2	6 0 0	
P3	2 1 1	2 2 2	0 1 1	
P4	0 0 2	4 3 3	4 3 1	

$Need_i \leq Work$

$Work = Work + Allocation_i$

	Allocation	Max	Need	Available
	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	2 0 0	3 2 2	1 2 2	
P2	3 0 2	9 0 2	6 0 0	
P3	2 1 1	2 2 2	0 1 1	
P4	0 0 2	4 3 3	4 3 1	

Available

3 3 2

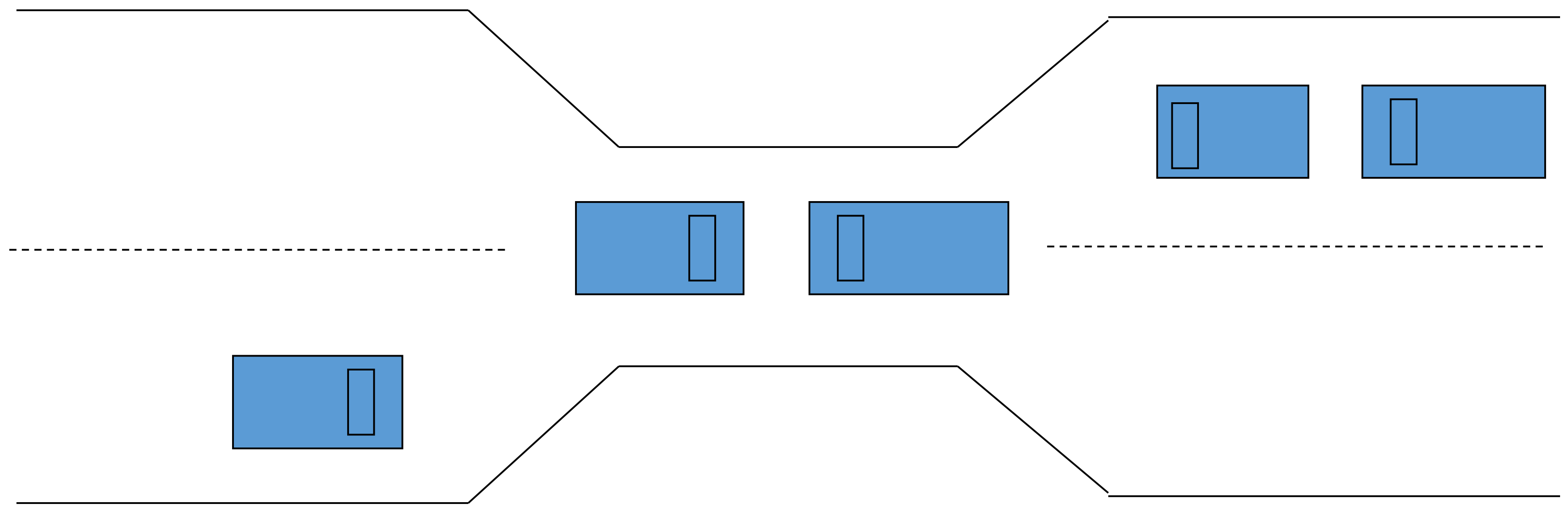
Available + Allocation (Pi)

Process	P_1	5 3 2
Process	P_3	7 4 3
Process	P_4	7 4 5
Process	P_0	7 5 5
Process	P_2	10 5 7

```
  3 3 2      Work
+ 2 0 0      (P1) Allocation
  -----
  5 3 2      (New) Work

  5 3 2      Work
+ 2 1 1      (P3) Allocation
  -----
  7 4 3      (New) Work

  7 4 3      Work
+ 0 0 2      (P4) Allocation
  -----
  7 4 5      (New) Work

  7 4 5      Work
+ 0 1 0      (P0) Allocation
  -----
  7 5 5

  7 5 5      Work
+ 3 0 2      (P2) Allocation
  -----
 10 5 7      (New) Work
```

Hence new request can be immediately granted in the safe state if sequence is P1 -> P3 -> P4 -> P0 -> P2

❖ Recovery from Deadlock



Two Options:

1) Kill one or more processes to break circular wait.

2) Preempt some resources from one or more of the deadlocked processes.

- Process Termination
  - **Two methods** are used for killing a process.
  - In both the methods, system reclaims all allotted resources.

    i) Kill/Abort all deadlocked processes - breaks deadlock but expensive

    ii) Kill/Abort one process at a time – kills one process at a time till deadlock is eliminated.

    - has considerable overhead as after each kill, deadlock detection algorithm is to be invoked.

    - if process is at middle of updating a file, killing in middle can leave the file in incorrect state.

- Resource Preemption
  - Some resources are successively preempted to give them to other processes until deadlock cycle is broken.
  - For this preemption, 3 issues are to be addressed:

  - **Selecting a victim** – which resources & processes are to be preempted are determined to minimize the cost

  - **Roll Back** – what to be done to process after preemption – roll back to safe state & restart from that safe state. Safe state must be determined appropriately.

  - **Starvation** – if cost factor based selection of victim, may a specific process may be impacted.

    A process must be picked as a victim for finite no of times only