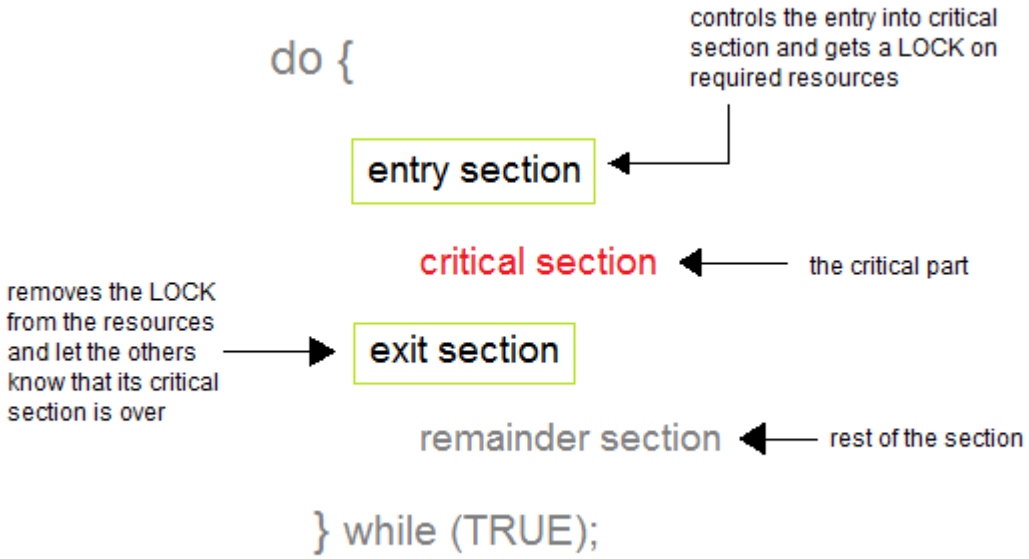
UNIT 3

SYNCHRONIZATION AND DEADLOCKS

THE CRITICAL SECTION PROBLEM:

- A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action.
- It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section.
- If any other process also wants to execute its critical section, it must wait until the first one finishes.

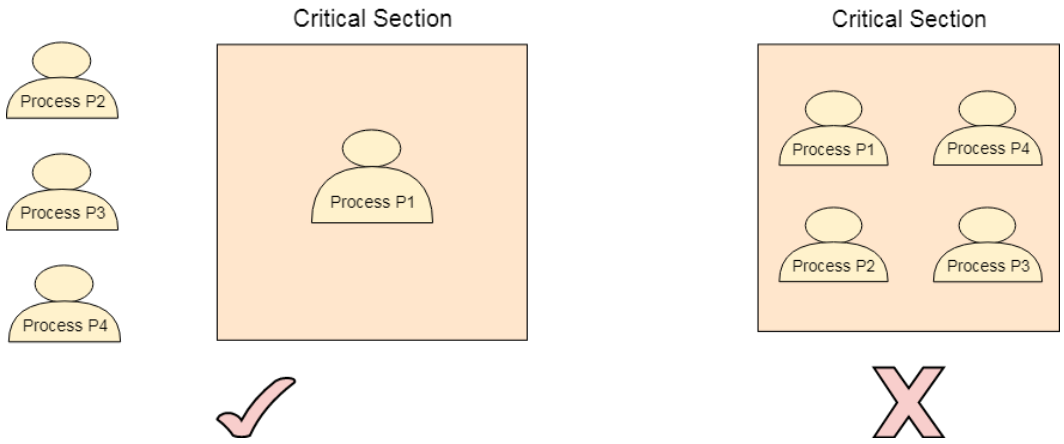


THE SOLUTION TO THE CRITICAL SECTION PROBLEM:

The main solution for the critical section problem is based on the three main ways.

1. MUTUAL EXCLUSION:

- Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.



- If process P1 is executing in its critical section, then no other processes can be executing in their critical sections.
- IMPLICATIONS:
  - Critical sections better be focused and short.
  - Better not get into an infinite loop in there.
  - If a process somehow halts/waits in its critical section, it must not interfere with other processes.

2. PROGRESS:

- If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.
- If only one process wants to enter, it should be able to.
- If two or more want to enter, one of them should succeed.

3. BOUNDED WAITING:

- After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted.
- So after the limit is reached, system must grant the process permission to get into its critical section.

TWO PROCESS SOLUTIONS:

ALGORITHM 1:

```
do {

while (turn != 1);

critical section

turn = j;

remainder section

}while (1);
```

This Algorithm satisfies Mutual exclusion whereas it fails to satisfy progress requirement since it requires strict alternation of processes in the execution of the critical section.

For example, if turn == 0 and p1 is ready to enter its critical section,p1 cannot do so, even though p0 may be in its remainder section.

ALGORITHM 2:

```
do{

flag[i] =true;

while (flag[j]);

critical section

flag[i]= false;

remainder section

}while(1);
```

In this solution, the mutual exclusion is met. But the progress is not met. To illustrate this problem, we consider the following

Execution Sequence:

To: P0 sets Flag[0] = true

T1: P1 sets Flag[1] = true

Now P0 and P1 are looping forever in their respective while statements.

ALGORITHM 3:

By combining the key ideas of algorithm 1 and 2, we obtain a correct solution.

```
Do{

Flag[i] = true;

Turn = j;

While (flag[j] && turn == j);

Critical section

Flag[i] = false;

Remainder section

}while(1);
```

The algorithm does satisfy the three essential criteria to solve the critical section problem. The three criteria are mutual exclusion, progress, and bounded waiting.

SYNCHRONIZATION HARDWARE:

- Many systems provide hardware support for critical section code.
- The critical section problem could be solved easily in a single-processor environment if we could disallow interrupts to occur while a shared variable or resource is being modified.
- In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption. Unfortunately, this solution is not feasible in a multiprocessor environment.
- Disabling interrupt on a multiprocessor environment can be time consuming as the message is passed to all the processors.

MUTEX LOCKS:

- As the synchronization hardware solution is not easy to implement for everyone, a strict software approach called Mutex Locks.

```
do{
Acquire Lock
Critical Section
Release Lock
Remainder Section
}while(TRUE);
```

MUTEX:

- Mutex is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously.
- When a program is started a mutex is created with a unique name.
- After this stage, any thread that needs the resource must lock the mutex from other threads while it is using the resource.
- The mutex is set to unlock when the data is no longer needed or the routine is finished.

SOLUTION TO THE CRITICAL - SECTION PROBLEM USING LOCKS

- Hardware features can make any programming task easier and improve system efficiency.
- The critical section problem can be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified.
- Then we can ensure that the current sequence of instructions would be allowed to execute in order without pre-emption.
- No other instructions would be run, so no unexpected modifications could be made to the shared variable.
- This solution is not feasible in a multiprocessor environment.
- Disabling interrupts on a multiprocessor can be time consuming, as the message is passed to all the processors.

- **Definition of TestAndSet:**

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

- **Shared boolean variable lock., initialized to false.**
- **Solution:**

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

- **Solution using Swap:**
- **Definition of Swap:**

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

*a = *b;

*b = temp:

}
```

- **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);
```

- **Solution with TestAndSet and bounded wait**
- **boolean waiting[n]; boolean lock; initialized to false Pi can enter critical section iff waiting[i] == false or key == false**

```
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);
```

SEMAPHORES

- It's a very significant technique for managing concurrent processes by using the value of a simple integer variable to synchronize the progress of interacting processes. This integer variable is called semaphore.
- In very simple words, semaphore is a variable which can hold only a non-negative Integer value, shared between all the threads, with operations wait and signal,
    - P(S): if S ≥ 1 then S := S - 1
      else <block and enqueue the process>;
    - V(S): if <some process is blocked on the queue>
      then <unblock a process>

else S := S + 1;

- Wait: Decrements the value of its argument S, as soon as it would become non-negative (greater than or equal to 1).
- Signal: Increments the value of its argument S, as there is no more process blocked on the queue.

PROPERTIES OF SEMAPHORES

- It's simple and always have a non-negative Integer value.
- Works with many processes.
- Can have many different critical sections with different semaphores.
- Each critical section has unique access semaphores.
- Can permit multiple processes into the critical section at once, if desirable.

TYPES OF SEMAPHORES

- **BINARY SEMAPHORE:**
  - It is a special form of semaphore used for implementing mutual exclusion, hence it is often called a **MUTEX**.
  - A binary semaphore is initialized to 1 and only takes the values 0 and 1 during execution of a program.
- **COUNTING SEMAPHORES:**
  - These are used to implement bounded concurrency.
- **EXAMPLE:**

```
Shared var mutex: semaphore = 1;
Process i
    begin
    .
    .
    P(mutex);
    execute CS;
    V(mutex);
    .
    .
    End;
```

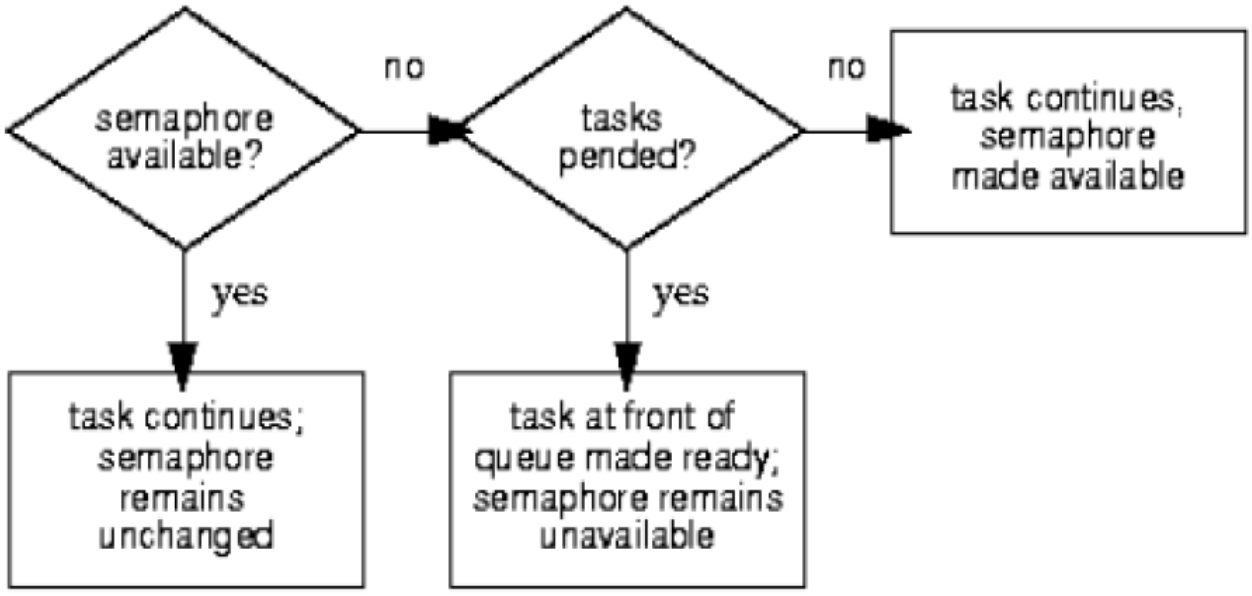
LIMITATIONS OF SEMAPHORES

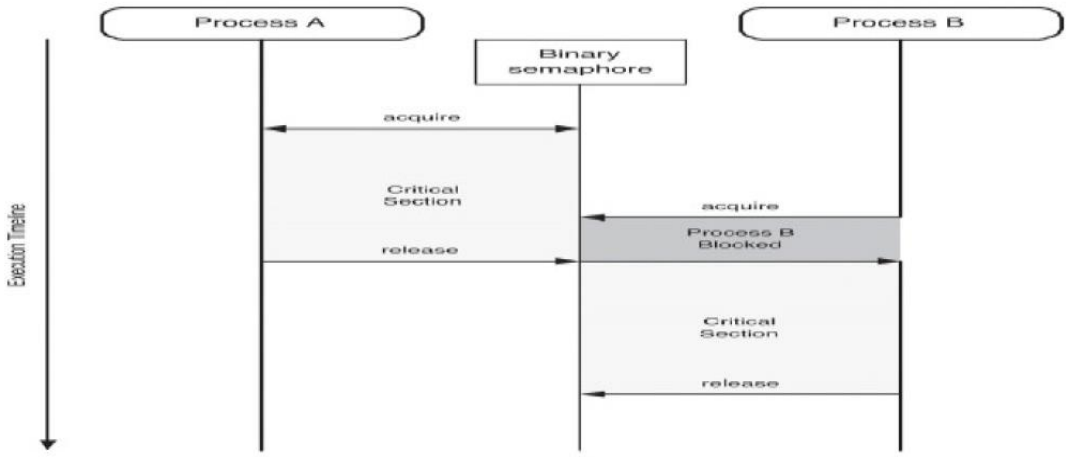
- Priority Inversion is a big limitation of semaphores.
- Their use is not enforced, but is by convention only.
- With improper use, a process may block indefinitely. Such a situation is called Deadlock. We will be studying deadlocks in details in coming lessons.

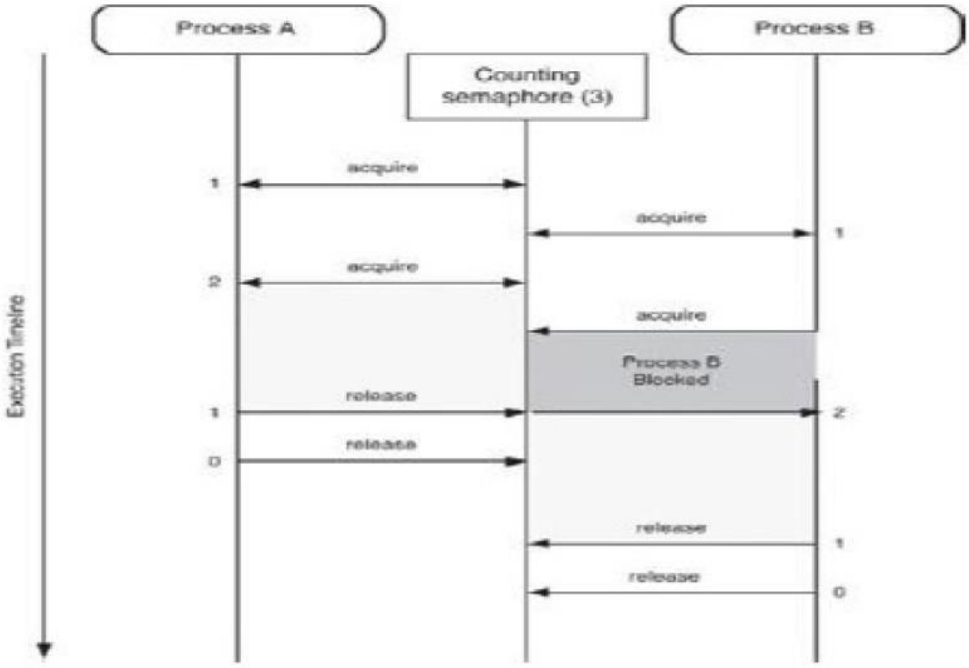
BINARY SEMAPHORE:

USAGE:

- OS's distinguish between counting and binary semaphores.
- The value of a counting semaphore can range over an unrestricted domain.
- The value of a binary semaphore can range only between 0 and 1.
- Binary semaphores are known as mutex locks as they are locks that provide mutual exclusion.
- Binary semaphores are used to deal with the critical section problem for multiple processes.
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
- The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a wait() operation on the semaphore.
- When a process releases a resource, it performs a signal() operation.







IMPLEMENTATION

- The main disadvantage of the semaphore is that it requires busy waiting. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.
- Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is called a **SPINLOCK** because the process spins while waiting for the lock.

- To overcome, the need for busy waiting the definition of wait () and signal() semaphore operations can be modified.
- When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait.

PROBLEMS WITH SEMAPHORE:

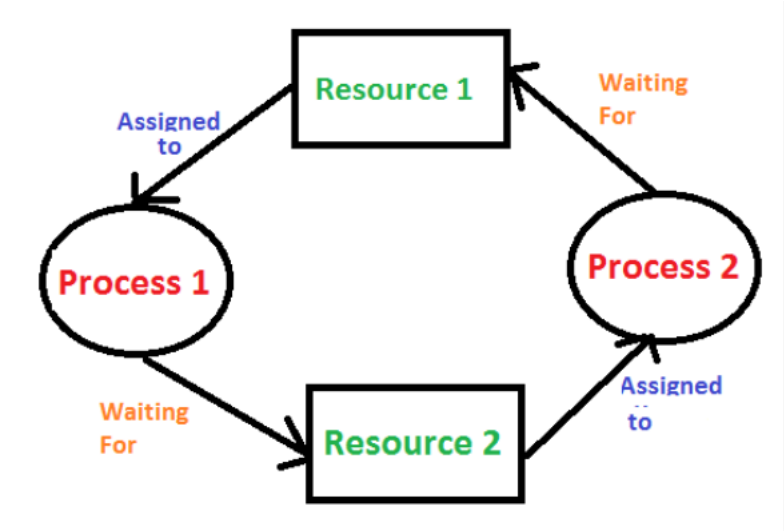
Correct use of semaphore operations:

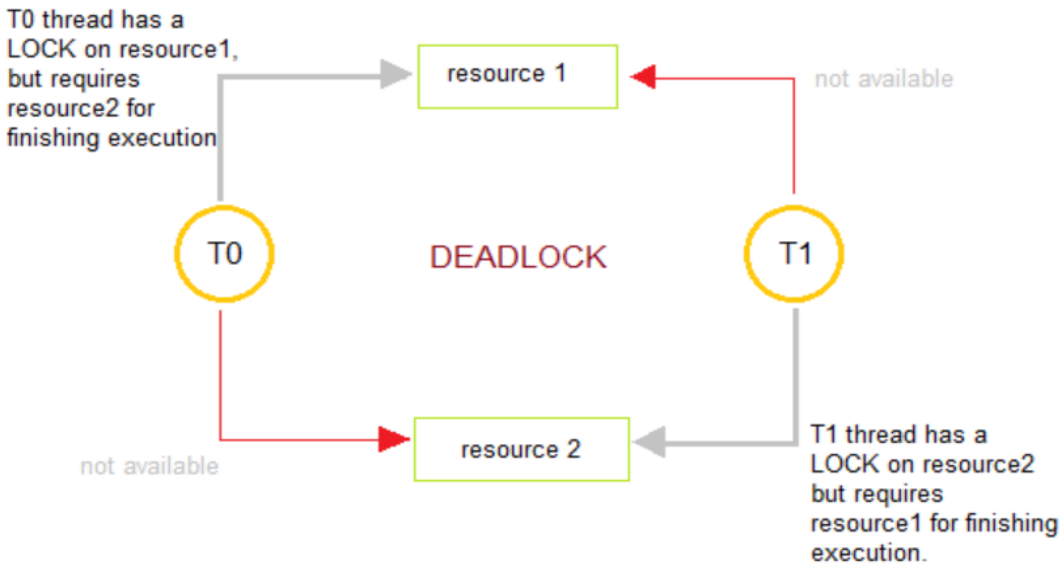
- signal (mutex) …. wait (mutex)
- wait (mutex) … wait (mutex)
- Omitting of wait (mutex) or signal (mutex) (or both)

DEADLOCKS AND STARVATION:

DEADLOCK:

- Deadlocks are a set of blocked processes each holding a resource and waiting to acquire a resource held by another process.





STARVATION

- Starvation, process with high priorities continuously uses the resources preventing low priority process to acquire the resources.
- Starvation can be defined as when a process request for a resource and that resource has been continuously used by the other processes then the requesting process faces starvation.
- In starvation, a process ready to execute waits for CPU to allocate the resource. But the process has to wait indefinitely as the other processes continuously block the requested resources.

AGING

- Aging can resolve the problem of starvation. Aging gradually increases the priority of the process that has been waiting long for the resources. Aging prevents a process with low priority to wait indefinitely for a resource.

DIFFERENCE BETWEEN DEADLOCKS & STARVATION

BASIS FOR COMPARSION	DEADLOCK	STARVATION
Basic	Deadlock is where no process proceeds, and get blocked.	Starvation is where low priority processes get blocked, and high priority process proceeds.
Arising condition	The occurrence of Mutual exclusion, Hold and wait, No preemption and Circular wait simultaneously.	Enforcement of priorities, uncontrolled resource management.
Other name	Circular wait.	Lifelock.
Resources	In deadlocked, requested resources are blocked by the other processes.	In starvation, the requested resources are continuously used by high priority processes.

CLASSIC PROBLEMS OF SYNCHRONIZATION

- These synchronization problems are examples of large class of concurrency control problems. In solutions to these problems, we use semaphores for synchronization.
    - **Bounded buffer problem / Producer consumer problem**
    - **Readers-writer problem**
    - **Dining-philosophers problem**

BOUNDED BUFFER / PRODUCER CONSUMER PROBLEM

- This problem is generalized in terms of the Producer Consumer problem, where a finite buffer pool is used to exchange messages between producer and consumer processes.
- Because the buffer pool has a maximum size, this problem is often called the Bounded buffer problem.

- Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively.
- The code below can be interpreted as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

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

- The structure of the Producer Process

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

- The structure of the Consumer Process

EXAMPLE CODE:

```
#include<stdio.h>
#include<conio.h>
int main()
{
int s,n,b=0,p=0,c=0;
clrscr();
printf("\n producer and consumer problem");
do
{
printf("\n menu");
printf("\n 1.producer an item");
printf("\n 2.consumer an item");
printf("\n 3.add item to the buffer");
printf("\n 4.display status");
printf("\n 5.exit");
printf("\n enter the choice");
scanf("%d",&s);
switch(s)
{
case 1:
p=p+1;
printf("\n item to be produced");
break;
case 2:
if(b!=0)
{
c=c+1;
```

```
b=b-1;
printf("\n item to be consumed");
}
else
{
printf("\n the buffer is empty please wait...");
}
break;
case 3:
if(b<n)
{
if(p!=0)
{
b=b+1;
printf("\n item added to buffer");
}
else
printf("\n no.of items to add...");
}
else
printf("\n buffer is full,please wait");
break;
case 4:
printf("no.of items produced :%d",p);
printf("\n no.of consumed items:%d",c);
printf("\n no.of buffered item:%d",b);
break;
case 5:exit(0);
}
```

```
}
while(s<=5);
getch();
return 0;
}
```

THE READERS WRITERS PROBLEM

- In this problem there are some processes(called readers) that only read the shared data, and never change it, and there are other processes(called writers) who may change the data in addition to reading, or instead of reading it.
- There are various type of readers-writers problem, most centered on relative priorities of readers and writers.

SOLUTION:

- From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.
- Here, we use one mutex m and a semaphore w. An integer variable **sread** is used to maintain the number of readers currently accessing the resource. The variable **swrite** is initialized to 0. A value of 1 is given initially to m and w.
- Instead of having the process to acquire lock on the shared resource, we use the mutex m to make the process to acquire and release lock whenever it is updating the **sread** variable.

EXAMPLE CODE

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
void main()
{
typedef int semaphore;
semaphore sread=0, swrite=0;
int ch,r=0;
```

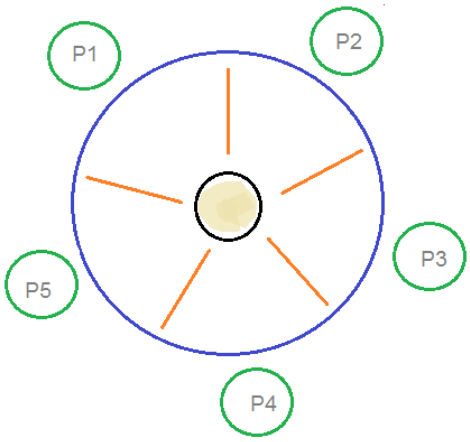
```
clrscr();
printf("\nReader writer");
do
{
printf("\nMenu");
printf("\n\t 1.Read from file");
printf("\n \t2.Write to file");
printf("\n \t 3.Exit the reader");
printf("\n \t 4.Exit the writer");
printf("\n \t 5.Exit");
printf("\nEnter your choice:");
scanf("%d",&ch);
switch(ch)
{
case 1: if(swrite==0)
{
sread=1;
r+=1;
printf("\nReader %d reads",r);
}
else
{
printf("\n Not possible");
}
break;
case 2: if(sread==0 && swrite==0)
{
swrite=1;
printf("\nWriter in Progress");
```

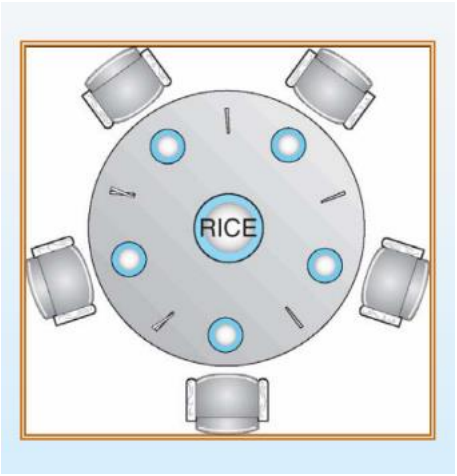
```
}
else if(swrite==1)
{
printf("\nWriter writes the files");
}
else if(sread==1)
{
printf("\nCannot write while reader reads the file");
}
else
printf("\nCannot write file");
break;
case 3: if(r!=0)
{
printf("\n The reader %d closes the file",r);
r-=1;
}
else if(r==0)
{
printf("\n Currently no readers access the file");
sread=0;
}
else if(r==1)
{
printf("\nOnly 1 reader file");
}
else
printf("%d reader are reading the file\n",r);
break;
```

```
case 4: if (swrite==1)
{
printf("\nWriter close the file");
swrite=0;
}
else
printf("\nThere is no writer in the file");
break;
case 5: exit(0);
}
}
while(ch<6);
getch();
}
```

DINING PHILOSOPHERS PROBLEM:

- The dining philosopher's problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.
- There are five philosophers sitting around a table, in which there are five chopsticks/forks kept beside them and a bowl of rice in the center.
- When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right.
- When a philosopher wants to think, he keeps down both chopsticks at their original place.





SOLUTION:

- From the problem statement, it is clear that a philosopher can think for an indefinite amount of time.
- But when a philosopher starts eating, he has to stop at some point of time.
- The philosopher is in an endless cycle of thinking and eating.
- An array of five semaphores, stick[5], for each of the five chopsticks.

EXAMPLE CODE:

```
#include<stdio.h>
#include<conio.h>
#define LEFT (i+4) %5
#define RIGHT (i+1) %5
#define THINKING 0
#define HUNGRY 1
#define EATING 2
int state[5];
void put_forks(int);
void test(int);
void take_forks(int);
void philosopher(int i)
{
```

```
if(state[i]==0)
{
take_forks(i);
if(state[i]==EATING)
printf("\n Eating in process....");
put_forks(i);
}
}
void put_forks(int i)
{
state[i]=THINKING;
printf("\n philosopher %d completed its works",i);
test(LEFT);
test(RIGHT);
}
void take_forks(int i)
{
state[i]=HUNGRY;
test(i);
}
void test(int i)
{
if(state[i]==HUNGRY && state[LEFT]!=EATING && state[RIGHT]!=EATING)
{
printf("\n philosopher %d can eat",i);
state[i]=EATING;
}
}
void main()
```

```
{
int i;
clrscr();
for(i=1;i<=5;i++)
state[i]=0;
printf("\n\t\t\t Dining Philosopher Problem");
printf("\n\t\t.....");
for(i=1;i<=5;i++)
{
printf("\n\n the philosopher %d falls hungry\n",i);
philosopher(i);
}
getch();
}
```

CRITICAL REGIONS:

- A critical region is a section of code that is always executed under mutual exclusion.
- Critical regions shift the responsibility for enforcing mutual exclusion from the programmer (where it resides when semaphores are used) to the compiler.
- They consist of two parts:
    - Variables that must be accessed under mutual exclusion.
    - A new language statement that identifies a critical region in which the variables are accessed.

EXAMPLE

```
var
v : shared T;
...
region v do
begin
...
end;
```

- All critical regions that are ‘tagged’ with the same variable have compiler-enforced mutual.
- Exclusion so that only one of them can be executed at a time:

```
Process A:
region V1 do
begin
{ Do some stuff. }
end;
region V2 do
begin
{ Do more stuff. }
end;

Process B:
region V1 do
begin
{ Do other stuff. }
end;
```

- Here process A can be executing inside its V2 region while process B is executing inside its V1 region, but if they both want to execute inside their respective V1 regions only one will be permitted to proceed.
- Each shared variable (V1 and V2 above) has a queue associated with it. Once one process is executing code inside a region tagged with a shared variable, any other processes that attempt to enter a region tagged with the same variable are blocked and put in the queue.

CONDITIONAL CRITICAL REGIONS:

- Critical regions aren’t equivalent to semaphores.
- As described so far, they lack condition synchronization.
- We can use semaphores to put a process to sleep until some condition is met (e.g. see the bounded-buffer Producer-Consumer problem), but we can’t do this with critical regions.
- Conditional critical regions provide condition synchronization for critical regions

```
region v when B do
begin
...
end;
```

where B is a boolean expression (usually B will refer to v).

- Conditional critical regions work as follows:
    - A process wanting to enter a region for v must obtain the mutex lock. If it cannot, then it is queued.
    - Once the lock is obtained the boolean expression B is tested. If B evaluates to true then the process proceeds, otherwise it releases the lock and is queued. When it next gets the lock it must retest B.

IMPLEMENTATION:

- Each shared variable now has two queues associated with it.
- The **MAIN QUEUE** is for processes that want to enter a critical region but find it locked.
- The **EVENT QUEUE** is for the processes that have blocked because they found the condition to be false.
- When a process leaves the conditional critical region the processes on the event queue join those in the main queue.

LIMITATIONS:

- Conditional critical regions are still distributed among the program code.
- There is no control over the manipulation of the protected variables — no information hiding or encapsulation.
- Once a process is executing inside a critical region it can do whatever it likes to the variables it has exclusive access to.
- Conditional critical regions are more difficult to implement efficiently than semaphores.

MONITORS:

- Consist of private data and operations on that data.
- Can contain types, constants, variables and procedures.
- Only the procedures explicitly marked can be seen outside the monitor.
- The monitor body allows the private data to be initialized.
- The compiler enforces mutual exclusion on a particular monitor.
- Each monitor has a boundary queue, and processes wanting to call a monitor routine join this queue if the monitor is already in use.
- Monitors are an improvement over conditional critical regions because all the code that accesses the shared data is localized.

```
Signal (mutex);
.....
Critical section
.....
Wait (mutex);
```

- 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 some particular execution sequences take place and these sequences do not always occur.
- Suppose that a process interchanges the order in which the wait () and signal () operations on the semaphore mutex are executed.
- Here several processes may be executing in their critical sections simultaneously, violating the mutual exclusion requirement.
- Suppose that a process replaces signal (mutex) with wait (mutex) that is it executes

```
Wait(mutex);
.....
Critical section
.....
Wait(mutex);
```

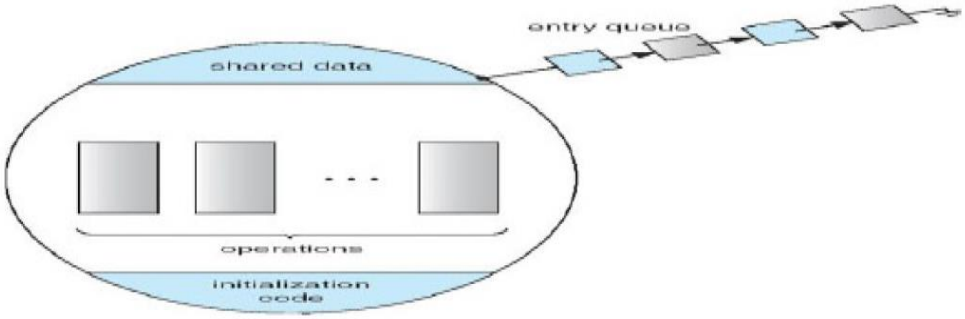
- Here a deadlock will occur.
- Suppose that a process omits the wait(mutex), or the signal(mutex) or both. Here, either mutual exclusion is violated or a dead lock will occur.
- To deal with such errors, a fundamental high level synchronization construct called **MONITORS** type is used.

USAGE OF MONITORS:

- A monitor type presents a set of programmer defined operations that are provided mutual exclusion within the monitor.
- The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of the procedures or functions that operate on those variables.
- The representation of a monitor type cannot be used directly by the various processes.
- Thus, a procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.
- The local variables of a monitor can be accessed by only the local procedures.

```
name: monitor
   ...local declarations
   ...initialize local data
   proc1 (...parameters)
      ...statement list
   proc2 (...parameters)
      ...statement list
   proc3 (...parameters)
      ...statement list
```

- When x.signal() operation is invoked by a process P, there is a suspended process Q associated with condition x.
- If 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.



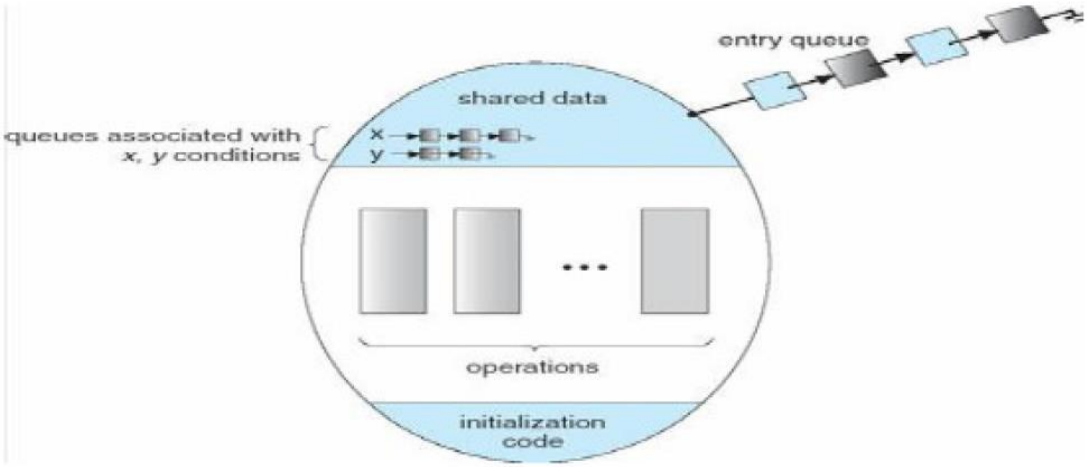
- However, both processes can conceptually continue with their execution. Two possibilities exist:
    - **Signal and wait** – P either waits until Q leaves the monitor or waits for another condition.
    - **Signal and condition** – Q either waits until P leaves the monitor or waits for another condition.

DINING PHILOSOPHERS SOLUTION USING MONITORS:

- This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.
- To code this solution, we need to distinguish among three states in which we may find a philosopher.

- For this purpose, we use this data structure **enum {thinking, hungry, eating } state[5];**

enum {thinking, hungry, eating } state[5];




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

```
void test (int i) {
      if ( (state[(i + 4) % 5] != EATING) &&
      (state[i] == HUNGRY) &&
      (state[(i + 1) % 5] != EATING) ) {
          state[i] = EATING ;
          self[i].signal () ;
       }
}
       initialization_code() {
            for (int i = 0; i < 5; i++)
            state[i] = THINKING;
       }
 }
```

Monitor solution to Dining Philosophers problem

IMPLEMENTING A MONITOR USING SEMAPHORES

- For each monitor, a semaphore mutex initialized to 1 is provided.
- A process must execute wait(mutex) before entering the monitor and must execute(signal) after leaving the monitor.
- Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore next initialized to 0, on which the signaling processes may suspend themselves.
- An integer variable next_count is used to count the number of processes suspended on next.

```
wait(mutex);
body of F
if (next_count > 0)
signal(next);
else
signal(mutex);
```

- We can now describe how condition variables are implemented.
- For each condition x, we introduce a semaphore x_sem and an integer variable x_count, both initialized to 0.
- The operation x. wait () can now be implemented as

```
x_count++;
if (next_count > 0)
signal(next);
else
signal(mutex);
wait(x_sem);
x_count—;
```

The operation x. signal () can be implemented as

```
if (x_count > 0) {
next_count++;
signal(x_sem);
wait(next) ;
next_count—;
}
```

RESUMING PROCESSES WITHIN A MONITOR

- If several processes are suspended on condition x, and an x.signal() operation is executed by some process, then for determining which suspended process should be resumed next, we use FCFS ordering so that the process waiting the longest is resumed first.
- Or conditional wait construct() can be used as x.wait(c); where c is an integer expression that is evaluated when the wait () operation is executed.
- The value of c, which is called a **PRIORITY NUMBER**, is then stored with the name of the process that is suspended.
- When x. signal () is executed, the process with the smallest associated priority number is resumed next.

A MONITOR TO ALLOCATE A SINGLE RESOURCE

```
monitor ResourceAllocator
boolean busy;
condition x;
void acquire(int time)
if (busy)
x.wait(time);
busy = TRUE;
void release() {
busy = FALSE;
x.signal();
initialization_code
busy = FALSE;
```

- A process that needs to access the resource in question must observe the following sequence:

```
R.acquire(t);
access the resource;
R. release O ;
```

- where R is an instance of type Resource Allocator.
- Unfortunately, the monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur:
    - A process might access a resource without first gaining access permission to the resource.
    - A process might never release a resource once it has been granted access to the resource.
- **Synchronization Example**
    - A process might attempt to release a resource that it never request
    - A process might request the same resource twice (without first releasing the resource).

DEADLOCKS:

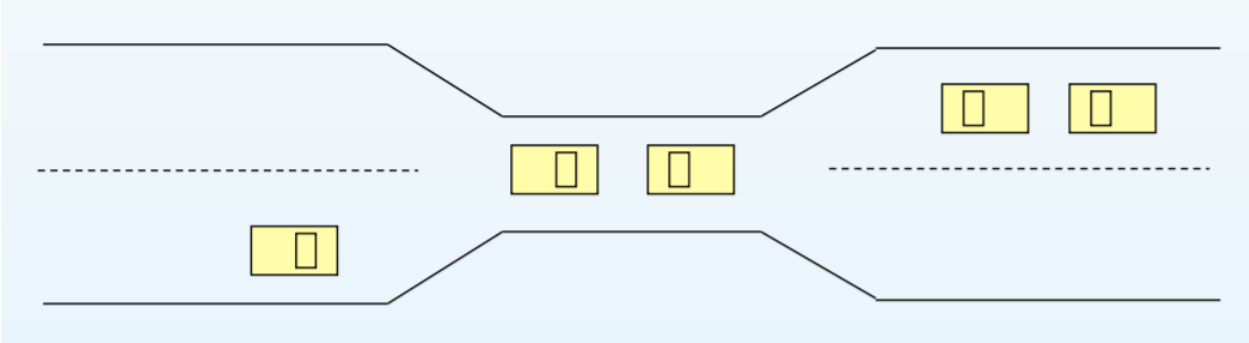
- The Deadlock Problem
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

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 tape drives.
  - P0 and P1 each hold one tape drive and each needs another one.
- Example
  - semaphores A and B, initialized to 1

P0	P1
wait (A);	wait(B)
wait (B);	wait(A)

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.

SYSTEM MODEL

- Resource types R1, R2, . . ., Rm
- CPU cycles, memory space, I/O devices
- Each resource type Ri has Wi instances.
- Each process utilizes a resource as follows:
    - request
    - use
    - release

DEADLOCK CHARACTERIZATION

- Deadlock can arise if four conditions hold simultaneously.
- **Mutual exclusion**:

  Only one process at a time can use a resource.
- **Hold and wait:**

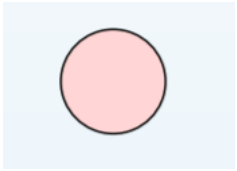
  A process holding at least one resource is waiting to acquire additional resource held by other processes.
- **No preemption:**

  A resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait:**

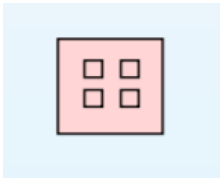
  There exists a set {P0, P1, …, P0} of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …, Pn–1 is waiting for a resource that is held by Pn, and P0 is waiting for a resource that is held by P0.

RESOURCE-ALLOCATION GRAPH

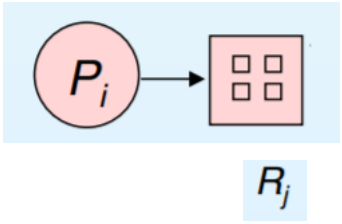
- A set of vertices V and a set of edges E.
- V is partitioned into two types:
  - P = {P1, P2, …, Pn}, the set consisting of all the processes in the system.
  - R = {R1, R2, …, Rm}, the set consisting of all resource types in the system.
- Request edge – directed edge P1 → Rj
- Assignment edge – directed edge Rj → Pi
- Process



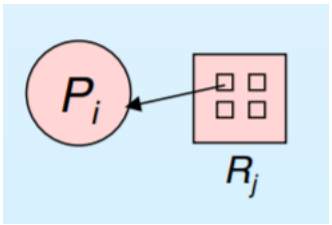
- Resource Type with 4 instances



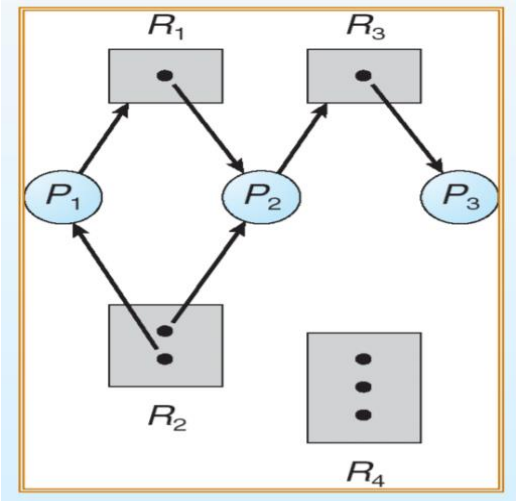
- Pi requests instance of Rj



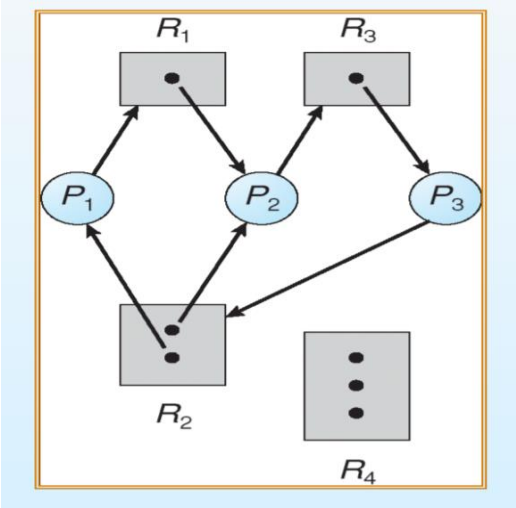
- Pi is holding an instance of Rj



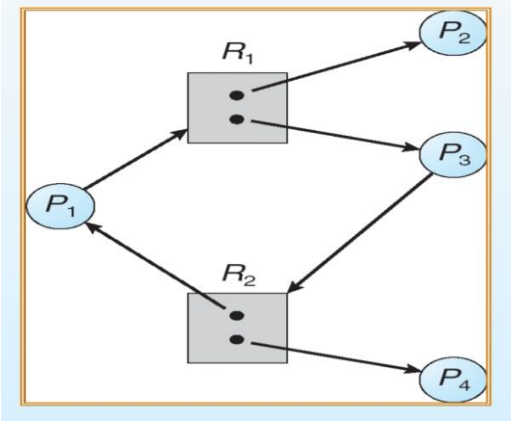
EXAMPLE OF A RESOURCE ALLOCATION GRAPH



Will there be a deadlock here?



Resource Allocation Graph With A Cycle But No Deadlock



Basic Facts

- If graph contains no cycles ⇒ no deadlock.
- If graph contains a cycle ⇒
    - if only one instance per resource type, then deadlock.
    - if several instances per resource type, possibility of deadlock.

METHODS FOR HANDLING DEADLOCKS

- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Prevention

DEADLOCK DETECTION

- Restrain the ways request can be made.
- **Mutual Exclusion** – not required for sharable resources; must hold for non-sharable resources.
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources. Low resource utilization; starvation possible.
- **No Preemption** – If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released. Preempted resources are added to the list of resources for which the process is waiting. Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

DEADLOCK AVOIDANCE

- Requires that the system has some additional a priori information available.
- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

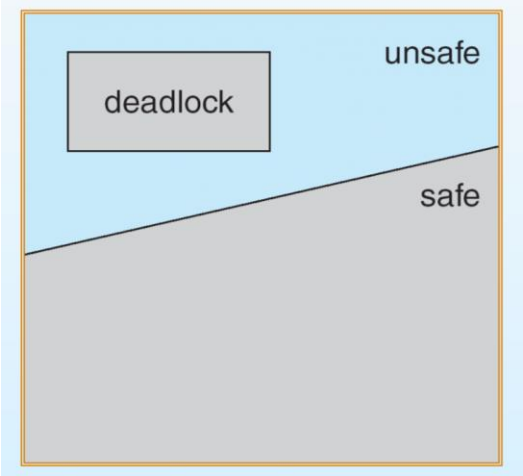
DEADLOCK DETECTION

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Basic Facts

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

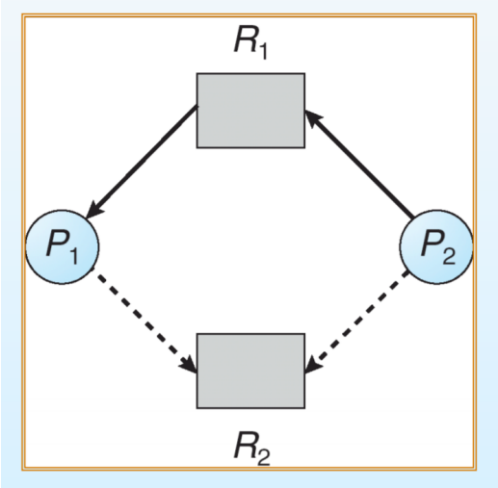
Safe, Unsafe , Deadlock State



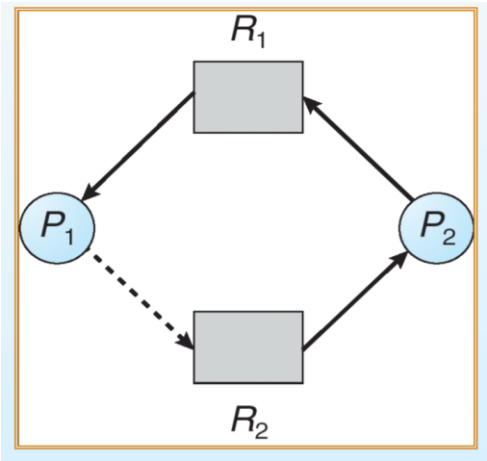
RESOURCE-ALLOCATION GRAPH ALGORITHM

- Claim edge Pi → Rj indicated that process Pj may request resource Rj; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed a priori in the system.

Resource-Allocation Graph For Deadlock Avoidance



Unsafe State In Resource-Allocation Graph



EXAMPLE:

- Banker's Algorithm
- Resource-Request Algorithm

BANKER'S ALGORITHM

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

Data Structures for the Banker's Algorithm

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

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

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

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize:
   $Work = Available$
   $Finish[i] = false$ for i - 1,3, …, n.
2. Find and i such that both:
   (a) $Finish[i] = false$
   (b) $Need_i \leq Work$
   If no such i exists, go to step 4.
3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   go to step 2.
4. If $Finish[i]$ == true for all i, then the system is in a safe state.

Resource-Request Algorithm for Process P_i

Request = request vector for process P_i

- If **$Request_i$ [j]** = k then process P_i wants k instances of resource type R_j.
- If **$Request_i \leq Need_i$** go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
- If **$Request_i \leq Available$**, go to step 3. Otherwise P_i must wait, since resources are not available.

$$Available = Available = Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

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

Example of Banker's Algorithm

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

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

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

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

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

Example request (contd)

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

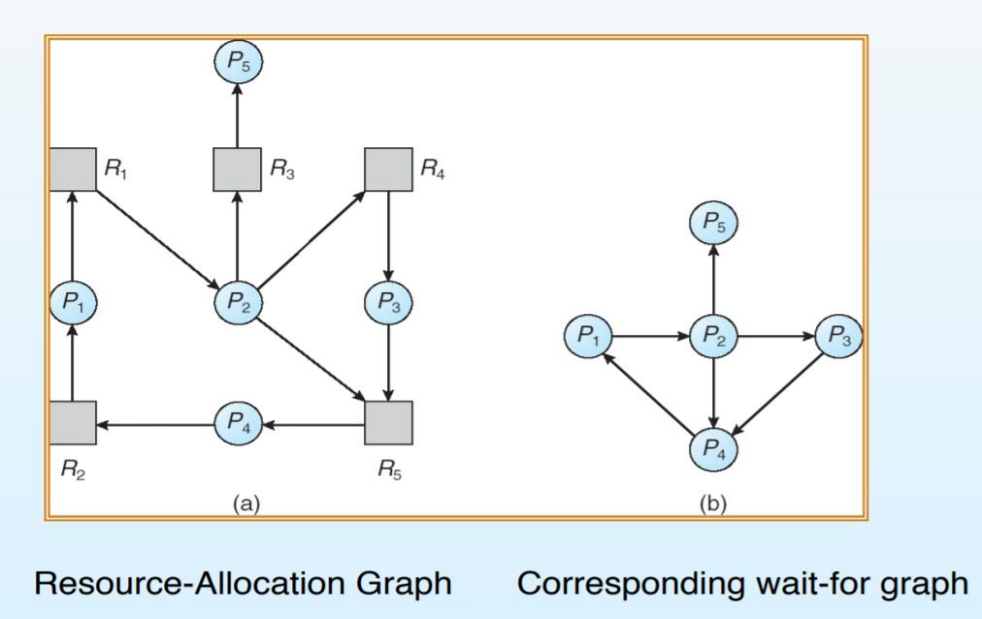
	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence <P1, P3, P4, P0, P2> satisfies safety requirement.
- Can request for (3,3,0) by P4 be granted?
- Can request for (0,2,0) by P0 be granted?

DEADLOCK DETECTION

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Resource-Allocation Graph and Wait-for Graph



Several Instances of a Resource Type

- **Available**: A vector of length m indicates the number of available resources of each type.
- **Allocation**: An n x m matrix defines the number of resources of each type currently allocated to each process.
- **Request**: An n x m matrix indicates the current request of each process. If Request $[i_j] = k$, then process P_i is requesting k more instances of resource type. R_j.

DETECTION ALGORITHM

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
   (a) *Work* = *Available*
   (b) For *i* = 1,2, …, *n*, if $Allocation_i \neq 0$, then *Finish*[i] = false;otherwise, *Finish*[i] = *true*.
2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$

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

3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   go to step 2.

4. If $Finish[i] == false$, for some i, $1 \le i \le n$, then the system is in deadlock state. Moreover, if $Finish[i] == false$, then P_i is deadlocked.

DETECTION-ALGORITHM USAGE

- When, and how often, to invoke depends on:
- How often a deadlock is likely to occur?
- How many processes will need to be rolled back? one for each disjoint cycle If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

RECOVERY FROM DEADLOCK: PROCESS TERMINATION

- 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?

RECOVERY FROM DEADLOCK: 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.