UNIT -2

Synchronization: Background, The Critical-Section Problem and its solutions, Synchronization Scheduling Algorithms, Semaphores Classic Problems of Synchronization, Deadlocks: System Model, Deadlock Characterization, Methods for Handling Deadlocks, Deadlock Prevention, Deadlock Avoidance, Deadlock detection, Recovery from deadlock.

Dr. M. Sinthuja
MSRIT

What is Process Synchronization in OS?

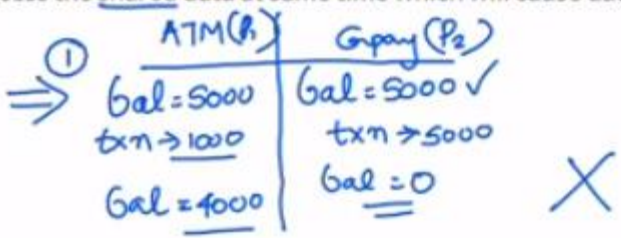
Process synchronization is very helpful when multiple processes are running at the same time and more than one process has access to the same data or resources at the same time.

Process synchronization is generally used in the multi-process system. When more than two processes have access to the same data or resources at the same time it can cause **data inconsistency** so to remove this data inconsistency processes should be synchronized with each other.

We take an example of a bank account that has a current balance of 500 and there are two users which have access to that account. User 1 and User 2 both are trying to access the balance. If process 1 is for withdrawal and process 2 is for checking the balance both will occur at the same time then user 1 might get the wrong current balance. To avoid this kind of data inconsistency process synchronization in os is very helpful.

NEED

Because cooperating processes may access the shared data at same time which will cause data inconsistency.



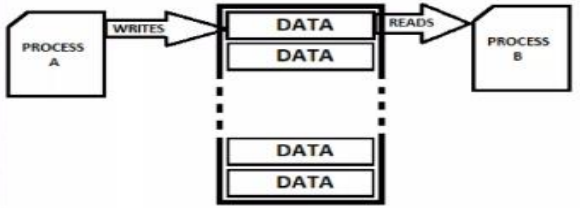
Synchronization -Background
Producer Consumer Problem

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Illustration of the problem:

Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer counter that keeps track of the number of full buffers. Initially, counter is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

WHAT IS PROCESS SYNCHRONIZATION?

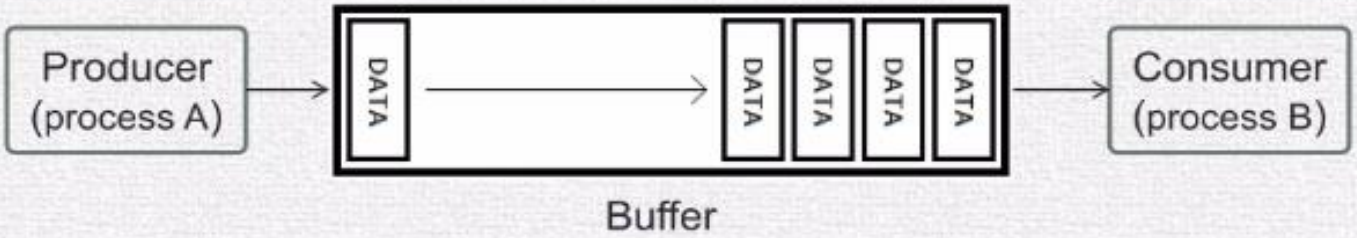
- Several Processes run in an Operating System
- Some of them share resources due to which problems like data inconsistency may arise
- For Example: One process changing the data in a memory location where another process is trying to read the data from the same memory location. It is possible that the data read by the second process will be erroneous



- Idli producer is idle when counter is full
- Foodee cannot consumes when the counter is empty
- When 2 idli added to the counter counter increases counter+2
- When Foodee consumes 2 idli the counter decreases counter-2

WHAT IS PROCESS SYNCHRONIZATION?

- **PRODUCER CONSUMER PROBLEM**
  (or Bounded-Buffer Problem)



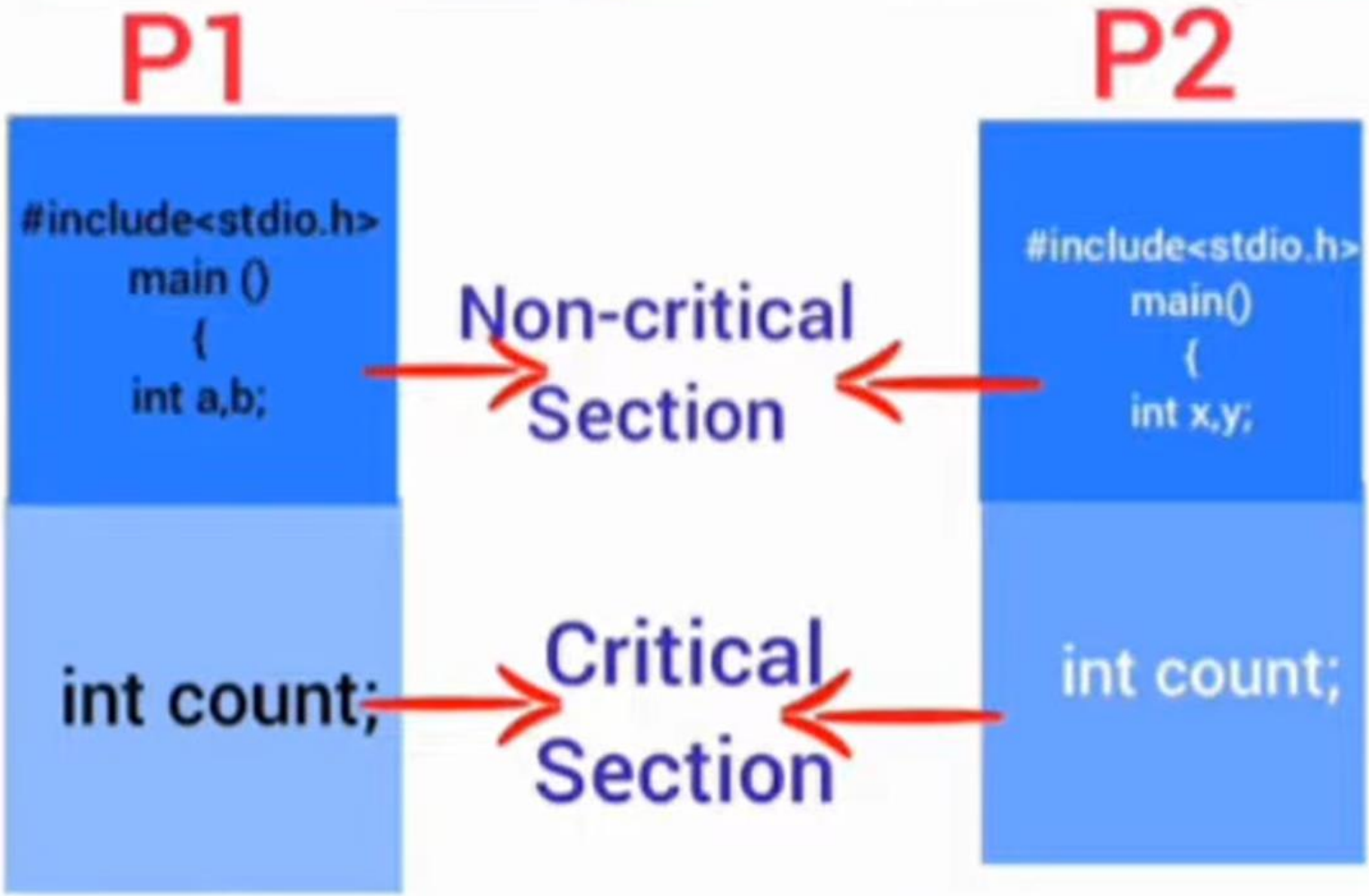
- Problem: To ensure that the ***Producer*** should not add DATA when the Buffer is *full* and the ***Consumer*** should not take data when the Buffer is *empty*

Process Synchronization

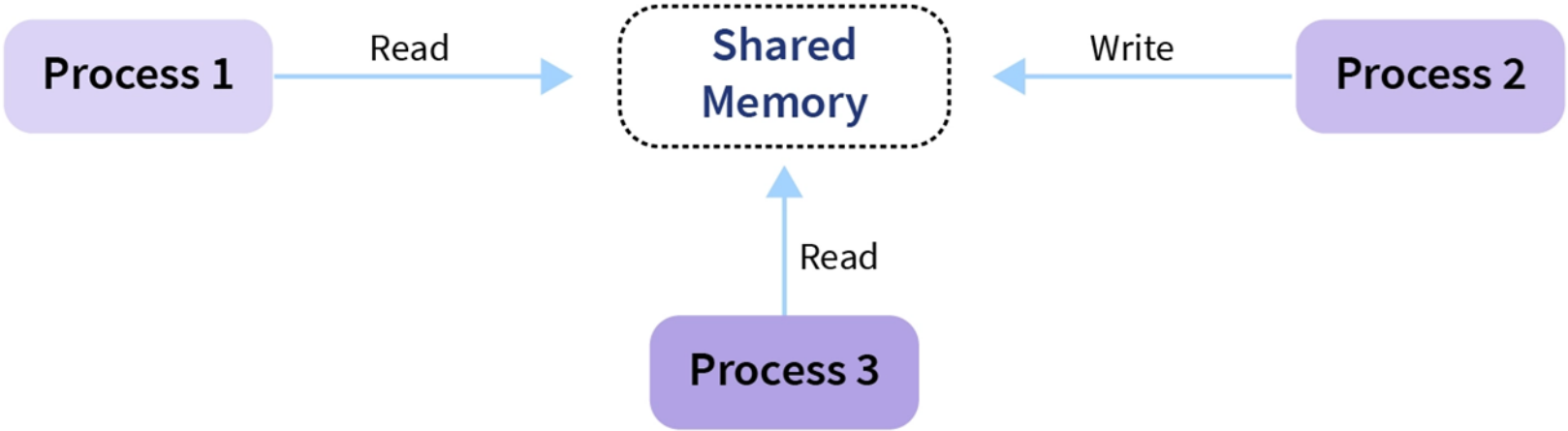
- **Independent Process**
  - Execution of one process does not affects the execution of another process.
  - Synchronization problem will not arise.
- **Co-operating Process**
  - Execution of one process does affects the execution of another process.
  - Synchronization problem will arise.

```
while (true) {
   /* produce an item in next produced */


  while (counter == BUFFER_SIZE) ;//counter size=5 initially starts with 0 if 0=5 false it moves to next line, if it is true 5=5 then loop will be there only so semi colon is used
  /* do nothing */
  buffer[in] = next_produced; //in → inorder to produce an item ‘ in’ variable is used
  in = (in + 1) % BUFFER_SIZE;
  counter++;
}
```

```
while (true) {
  while (counter == 0)//if counter=0 true it stays there that is empty buffer for that semi colon is used at the end of while loop
  ; /* do nothing */
  next_consumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE; ////out → inorder to consume an item 'out' variable is used

   counter--;
  /* consume the item in next consumed */
}
```


P1
P2
#include<stdio.h>
main ()
{
int a,b;
Non-critical
Section
#include<stdio.h>
main()
{
int x,y;
int count;
Critical
Section
int count;

Process Synchronization



Race Condition

- To prevent incorrect results when sharing data, we need to make sure that only one process at a time manipulates the shared data
  - In this example, the variable **count** should be accessed and changed only by one process at a time

A situation where several processes access and manipulate the same dta concurrently and the outcome of the execution depends on the particular order in which the access takes place is called a race condition.

To prevent race conditions, concurrent processes must be **synchronized**

- **`counter++`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter`	{register1 = 5}
S1: producer execute `register1 = register1 + 1`	{register1 = 6}
S2: consumer execute `register2 = counter`	{register2 = 5}
S3: consumer execute `register2 = register2 - 1`	{register2 = 4}
S4: producer execute `counter = register1`	{counter = 6 }
S5: consumer execute `counter = register2`	{counter = 4}

Critical Section Problem

Critical section problem is a part of the program where shared memory is accessed. Shared memory means a variable or a shared file.

If we take producer consumer problem accessing the count is critical section. Count denotes how many items present in the buffer. Producer produces the item that is stored in buffer with increment **count++/count=count+1** likewise consumer consumes the item from the buffer **count--./count=count-1**. **PC-problem accessing the count variable is critical section problem.**

Why it is denoted as a CS problem? If producer tries to access CS then consumer not allowed to access CS. If the consumer tries to access CS then producer not allowed to access CS. If both producer and consumer are accessing the count variable then the results will be wrong. To solve that problem we are going to CS-problem

To solve CS-Problem

- **Peterson's Solution (software)**
- **Synchronization Hardware**
- **Mutex Locks**

In Order to solve critical section problem we have 3 requirements. Any solution to CS-problem should satisfy this requirements

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections (one process is only executed in CS-Problem)

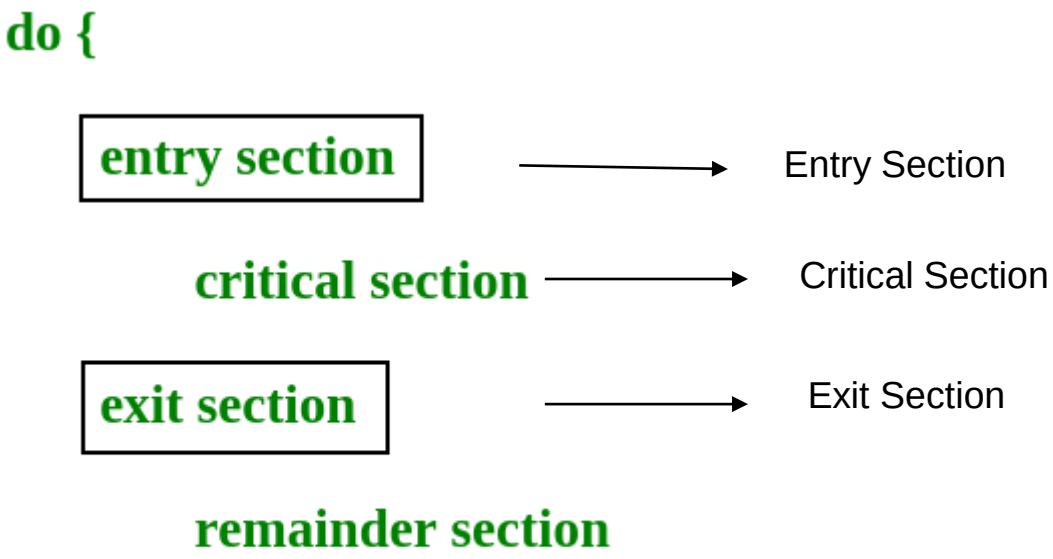
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

3. **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

~~A~~ssume that each process executes at a nonzero speed

~~No~~ assumption concerning **relative speed** of the ***n*** processes

General structure of process p_i



1. **Entry section contains a code whether a process enter a CS or not**

3. If the CS is free and if the process wants to enter the CS the corresponding author locks the CS as it will not allow other process to enter. Here mutual exclusion is achieved

2. Whenever the process enters the critical section it checks whether the critical section is free or occupied by any of the process.

4.If the critical section is free the corresponding process enters the critical section or else it has to wait until the process completes its execution

5. In exit section unlock the critical section.

6. Remainder section means remaining part of the program

Humble solution

Peterson's Solution

Peterson's Solution

- A classic software-based solution to the critical-section problem.
- May not work correctly on modern computer architectures.
- However, it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting requirements.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. Let's call the processes P_i and P_j

Peterson's solution requires two data items to be shared between the two processes:

int turn

→ Indicates whose turn it is to enter its critical section.

boolean flag [2]

→ Used to indicate if a process is ready to enter its critical section.

int turn

Indicates whose turn it is to enter its critical section.

boolean flag [2]

Used to indicate if a process is ready to enter its critical section.

Structure of process P_i in Peterson's solution

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

        critical section

    flag [i] = false ;

        remainder section

} while (TRUE) ;
```

int turn

Indicates whose turn it is to enter its critical section.

boolean flag [2]

Used to indicate if a process is ready to enter its critical section.

Structure of process P_i in Peterson's solution

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

        critical section

    flag [i] = false ;

        remainder section

    } while (TRUE) ;
```

Structure of process P_j in Peterson's solution

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

        critical section

    flag [ j ] = false ;

        remainder section

    } while (TRUE) ;
```

Hardware Synchronization

Test and Set Lock

- A hardware solution to the synchronization problem.
- There is a shared lock variable which can take either of the two values, 0 or 1.
- Before entering into the critical section, a process inquires about the lock.
- If it is locked, it keeps on waiting till it becomes free.
- If it is not locked, it takes the lock and executes the critical section.

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

Lock variable (0 or 1)

Value whatever is passed is returned

The definition of the TestAndSet () instruction

Atomic Operation

single operation which will not be interrupted is atomic operation

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

Atomic Operation

The definition of the TestAndSet () instruction

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

Process P1

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

Atomic Operation

The definition of the TestAndSet () instruction

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


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

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **acquire()** a lock then **release()** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **acquire()** and **release()** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires busy waiting
  - This lock therefore called a spinlock

acquire() and release()

available

P_1

$\overline{R_1}$

acquire R_1 → avai → True → false ⊗

CS

release

avail = True

- acquire() { available = false

spinlock

```
while (!available)
    ; /* busy wait */ ✓
available = false;
```

```
}
```

- release() {

```
available = true;
}
```

- do {

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

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;;
}
release() {
    available = true;
}
do {      /* the use of mutex lock in a process */
   acquire lock
      critical section
   release lock
     remainder section
} while (true);
```

Semaphores

Semaphore proposed by Edsger Dijkstra, is a technique to manage concurrent processes by using a simple integer value, which is known as a semaphore.

It is an integer variable shared, i.e shared by multiple process in the system (all 5 process can use the variable.)

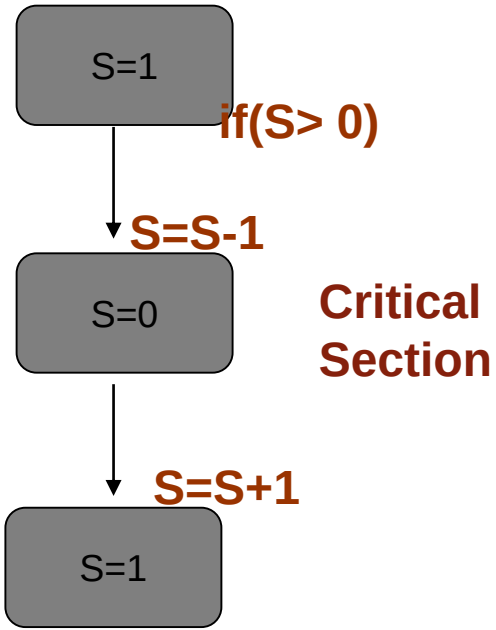
Can initialize semaphore variable

Can access the semaphore variable by 2 standard operations: **wait and signal;** these are called **atomic operations.**

wait ( ) → P [from the Dutch word proberen, which means "to test"]

signal ( ) → V [from the Dutch word verhogen, which means " to increment"]

https://www.youtube.com/watch?v=LIzTbA3cAWY



wait()
S=S-1

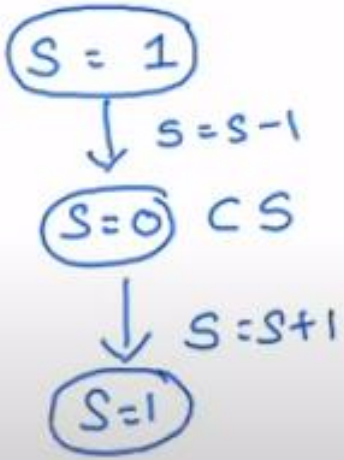
Critical Section

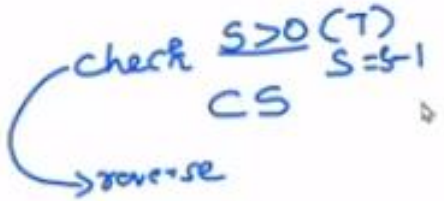
signal()
S=S+1

Initially, semaphore should be 1

More than mutex semaphores are used.

Semaphore → integer variable





When the S<0 some process is in CS
When S>0 it is S= S-1

***wait* (S):** while S <= 0 do no-op;
S = S - 1; **as soon as S > 0**

***signal* (S):** S = S + 1;

- ***S*** is initialized to a value >= 1 **why not 0? when >0?**
- ***wait* & *signal*** are executed **atomically**
  - if one process is modifying the semaphore, no other process can do so.
  - in the ***wait***, the testing of the semaphore and the modification is un-interruptible.

Types of Semaphore

1. Binary Semaphore:

The value of a binary semaphore can range only between 0 and 1. On some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion.

Value is 0: process has to wait for the execution
Value is 1: process can avail critical section

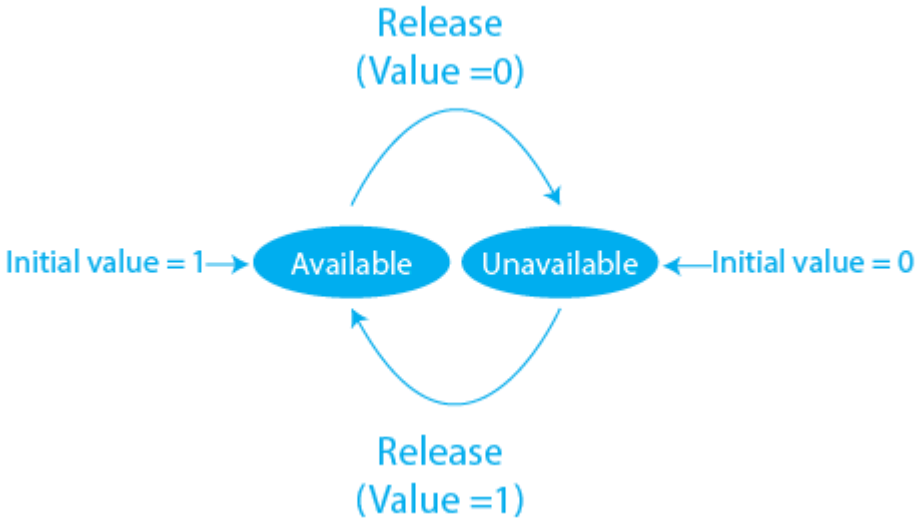
Same as mutex but in mutex true or false is used

2. Counting Semaphore:

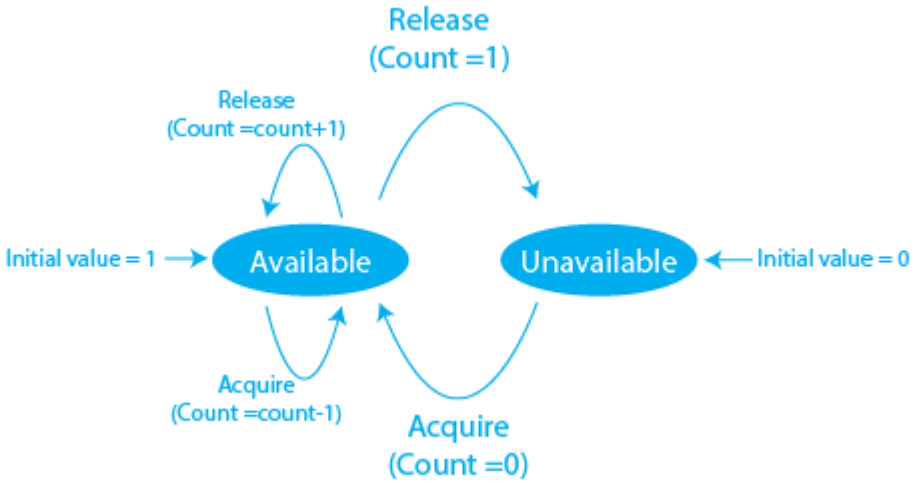
Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Non negative value
More number
Efficient use of multiple resource

Binary Semaphores: Binary semaphores are also known as mutual exclusion. A binary semaphore can only have two states: 0 or 1. It is typically used to implement mutual exclusion, which ensures that only one process or thread can access a shared resource at any given time.



Counting Semaphores: Counting semaphores can have any non-negative integer value. They are typically used to manage a pool of resources that can be accessed by multiple processes or threads.



Advantages of Semaphore

- To prevent race condition
- To prevent deadlock
- To implement mutual exclusion
- Efficient use of resources

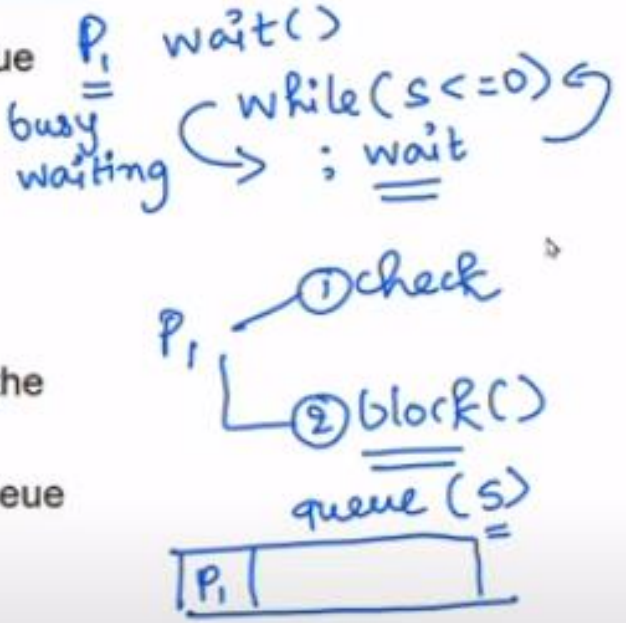
Disadvantage:

- Improper use of semaphore leads to deadlock. More number semaphore leads to high maintenance.
- It is difficult to debug if any synchronization issues occur.

Semaphore Implementation

Semaphore Implementation with no Busy waiting

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



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

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

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

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

Volume

$S = 1$

$P_1 \rightarrow S = S-1 = 0 \Rightarrow$ CS ✓

$P_2 \rightarrow S = S-1 = -1 \Rightarrow$ queue

$P_3 \rightarrow S = S-1 = -2 \Rightarrow$ queue

P_2 | P_3

$S = -2$

$P_1 \rightarrow S = S+1 = -1$

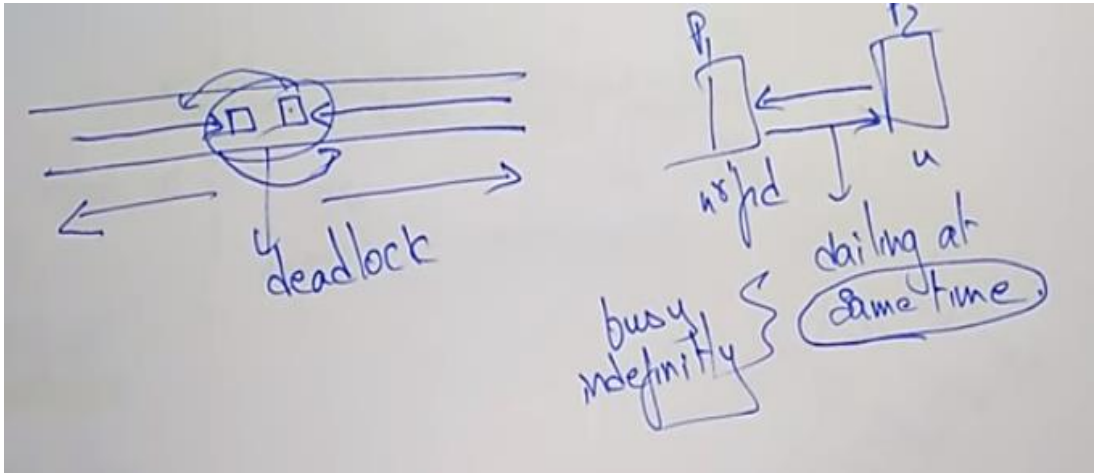
P_2 | P_3

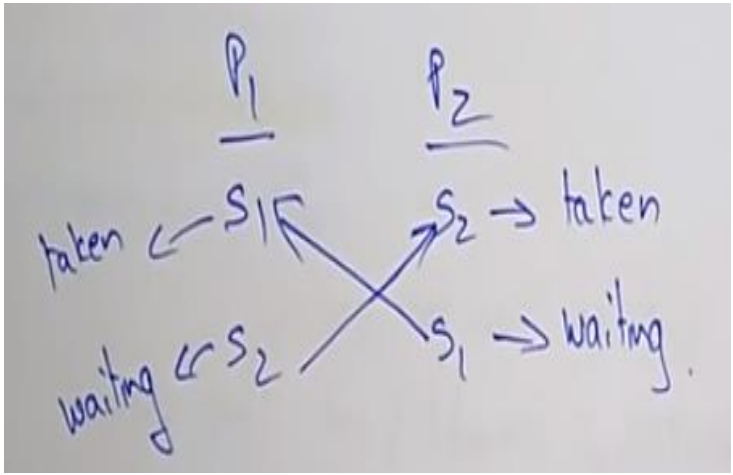
P_2 → Ready queue

⇒ No busy waiting

Deadlock & Starvation

- Deadlock occurs When 2 or more processes try to get the same multiple resources at a same time. Ex: thin road, calling ur friend at same time





⇒ Deadlock and Starvation

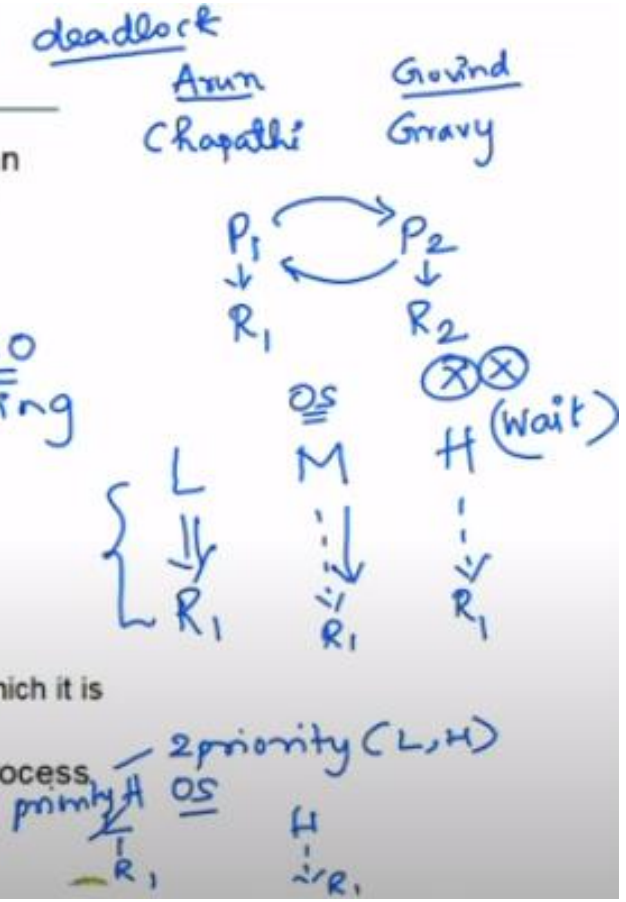
- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let ***S*** and ***Q*** be two semaphores initialized to 1

P_0 S=1 Q=1 ⟺ P_1

```
wait(S);   → S=0
wait(Q);   waiting
...
signal(S);
signal(Q);
```

```
wait(Q);   → Q=0
wait(S);   waiting
...
signal(Q);
signal(S);
```

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol** (2)



Starvation

Indefinite blocking

A process may never be removed from a semaphore queue in which it is suspended.

This causes priority inversion

https://www.youtube.com/watch?v=N2VECl8F_Pc

Priority inversion

Scheduling problem when low priority process holds a lock needed by the higher priority process.

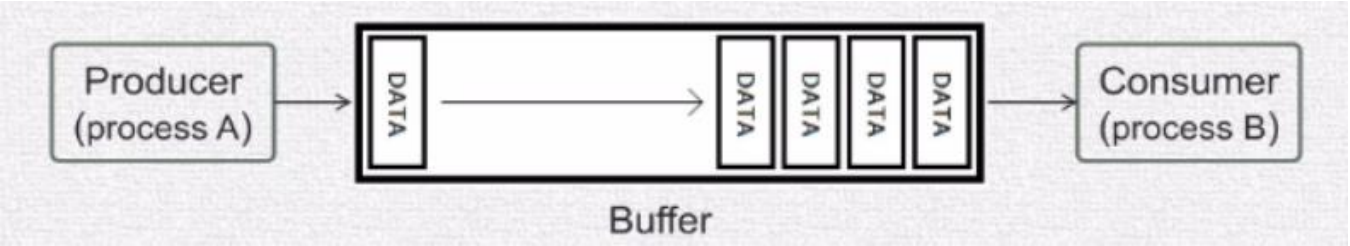
This is solved by **priority inheritance protocol.**

Classic Problems of Synchronization

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

The Bounded Buffer Problem or producer consumer problem

The concept of bounded buffer problem is producer will produce the item to the buffer and CONSUMER CONSUMES ITEM from the buffer. If the buffer if full producer the producer cannot produce item if the buffer is empty consumer cannot consume the item. This is the problem statement. How semaphore is here is the concept.



Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

n=4
mutex=1
full=0
empty=n

The Bounded Buffer Problem or producer consumer problem

Each buffer holds one item if 5 buffer it can store 5 items

N buffer n items can be stored

3 Semaphore

1. Semaphore mutex

   mutex=1

1. Semaphore full

   full=0

1. Semaphore empty

   empty=n

3 Semaphore

1. ***Semaphore mutex***

   mutual exclusion,

   for the security using mutex that is only one process is allowed in critical section,

   initial value of mutex is 1 it means no process in the critical section

   If mutex is 0 already some process is there in the critical section

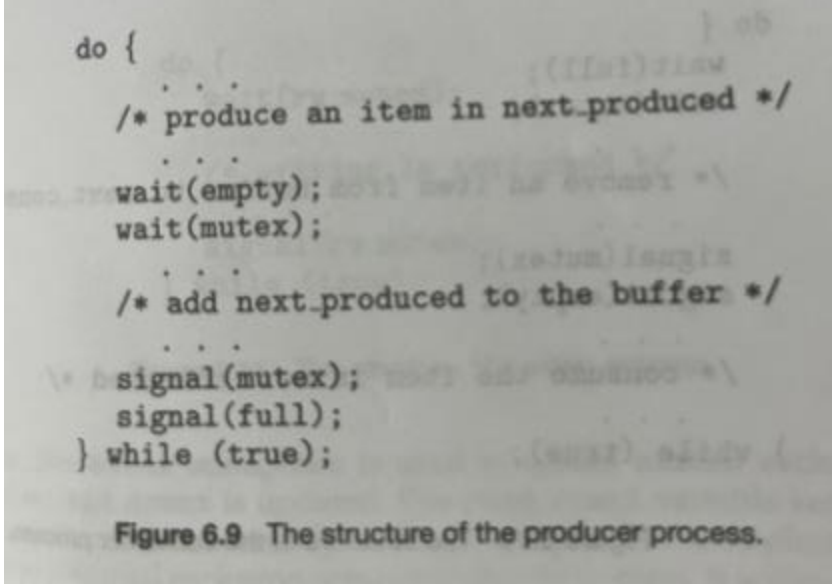
   mutex=1

1. ***Semaphore full=0***

   How many buffers are full initial the buffer is empty so full =0

1. ***Semaphore empty=n***

   How many buffers are empty, initially the buffer is empty so the value of empty=n. Foe ex n=5.



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

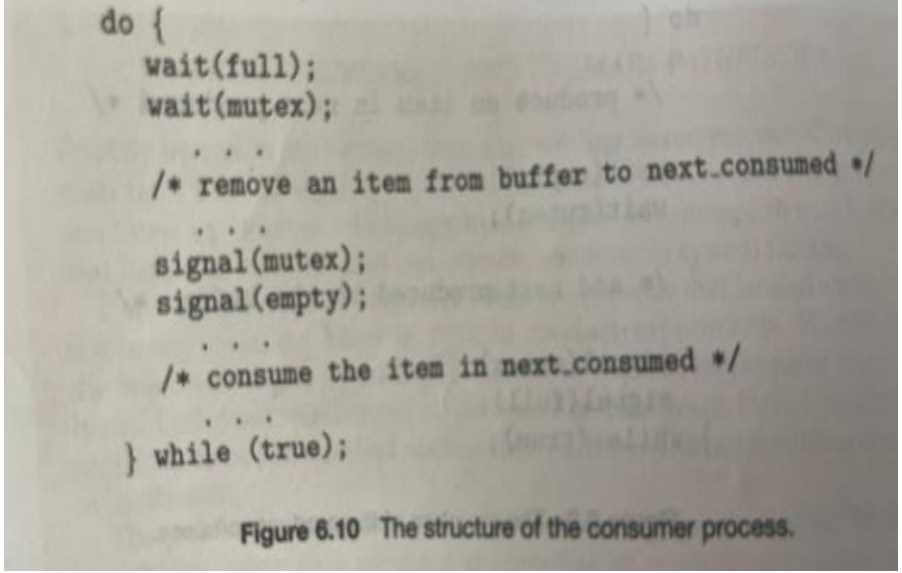
Figure 6.9 The structure of the producer process.

If the condition is true it will enter first it will check **wait(empty)** it means it check whether the empty is 0 or not. If 0 it means the buffer is full. If empty=1 then one buffer is free. **For ex consider empty =5 it is decremented to 4**
Next **wait(mutex)** initially the mutex=1 it means the critical section is free so the process enters the CS. **So decrement the mutex=0.**
Now the process is added to the buffer with value 10.

Signal means incremented by 1.

Once the producer produces the item it has to come out of CS. so execute signal(mutex) it means increment the mutex. **The value of mutex was 0 now increment to 1.**

Next signal(full) full=0 no buffer is full. Now signal means increment by 1. **So increment the signal(full)=1**



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

Figure 6.10 The structure of the consumer process.

Now to consume the item from buffer it checks wait(full) it means whether the buffer is full if the buffer =0 its not full u cant consume.
In this example buffer full value is 1. So condition true.
As consuming the item the value decremented to 0.

Next wait(mutex) here mutex=1 it means the process can enter CS. now decrement to 0. Now CS is busy.

Now consumes removes item from the buffer.

signal(mutex)-->increment to 1
signal(empty)-->increment to 5

The Readers-Writers Problem

- The readers-writers problem is used to manage synchronization so that there are no problems with the object data. **For example - If two readers access the object at the same time there is no problem. However if two writers or a reader and writer access the object at the same time, there may be problems.**
- To solve this situation, a writer should get exclusive access to an object i.e. when a writer is accessing the object, no reader or writer may access it. However, multiple readers can access the object at the same time.
- This can be implemented using semaphores.

Problem:

1. **Allows multiple reader to read the file simultaneously**

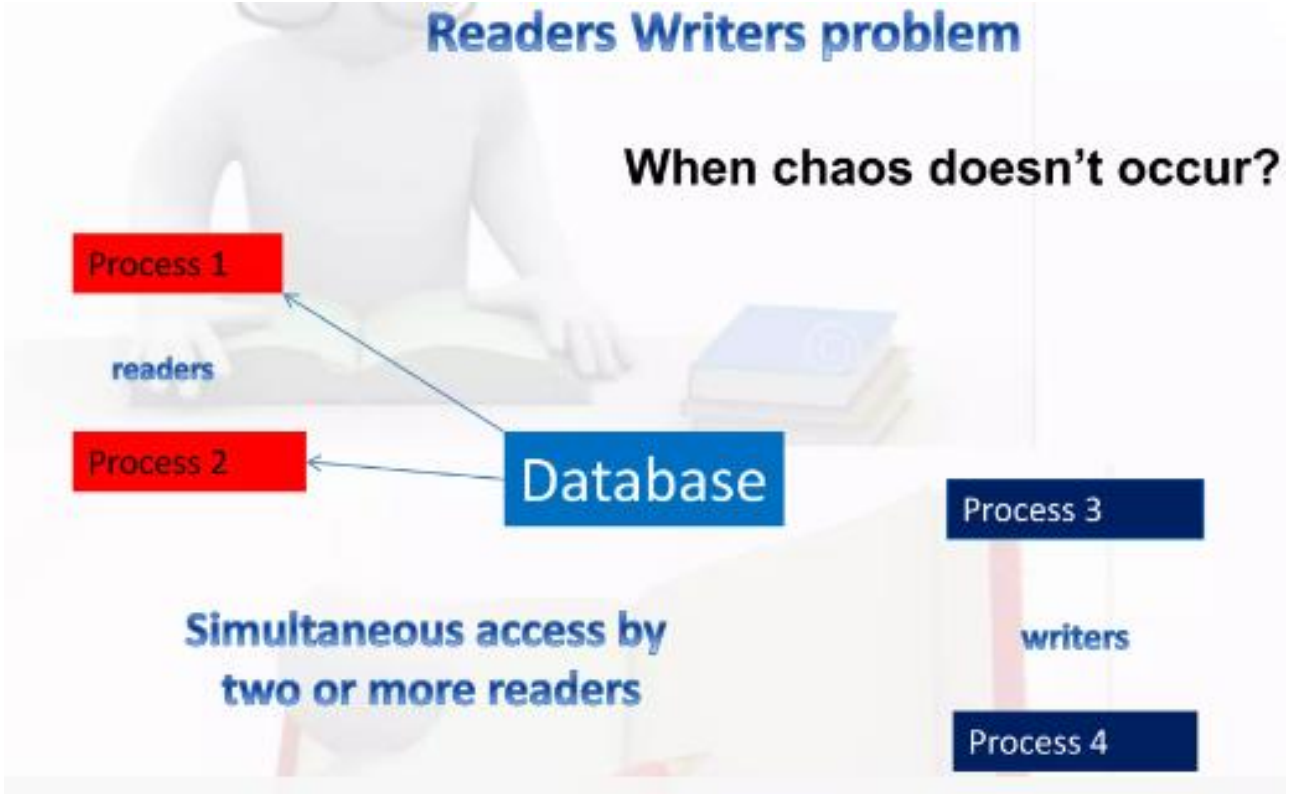
i.e., Read Read is possible

1. **When reader process is accessing the file and the writer process is not allowed to access the file**

i.e., RW not possible to read (R in critical section and W trying to enter critical section is not allowed)

1. **When writer process is accessing the file no other process is allowed to access the file.**

i.e., WR, WW both not possible


Readers Writers problem
When chaos doesn't occur?
Process 1
readers
Process 2
Database
Process 3
writers
Process 4
Simultaneous access by
two or more readers

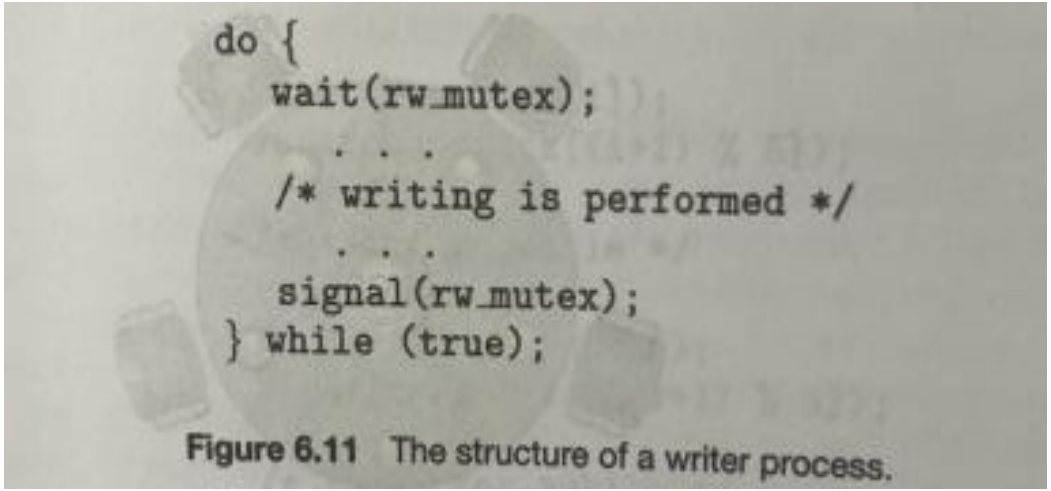
When chaos occur?

Reader writer problem or classic problem of synchronization

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

Shared data

- ➔ Dataset
- ➔ **Semaphore rw_mutex** initialized to 1 (to protect the shared variable) (decision to enter Critical section)
- ➔ integer **semaphore read_count** initialized to 0. (no. of readers)
- ➔ **Semaphore mutex** initialized to 1 (mutex-mutual exclusion -to protect the read count variable)



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

Figure 6.11 The structure of a writer process.

wait(rw_mutex) rw_mutex=1

Wait so decrement 1-1=0 (it enters CS)

signal(rw_mutex) (0+1)=1

Any other process can enter

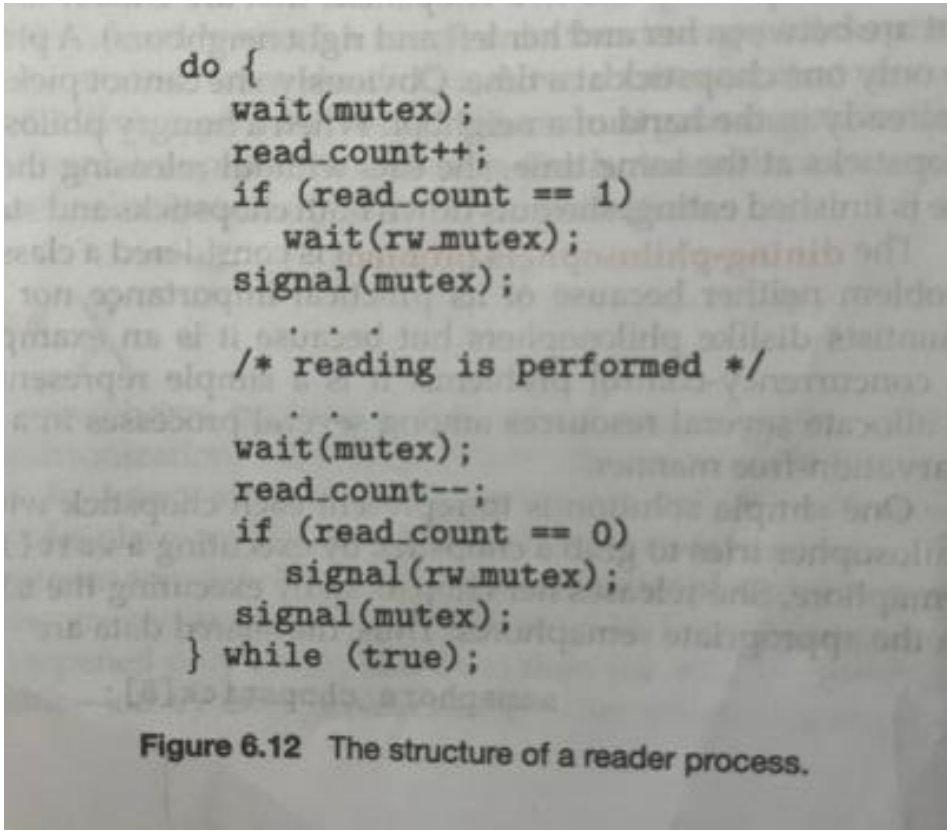
It can perform read or write operation

Initially the condition will be true

It checks for **wait(rw_mutex) initially the mutex=1** that is CS accepts process so the process enters the CS so the mutex is changed to 0

Once the file is read the process needs to come out so it enters the exit section signal(rw_mutex) signal increments the mutex so now the mutex increments from 0 to 1.

Here write write is not possible when the mutex is in 0 another write process cannot enter as CS is busy with another process.



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

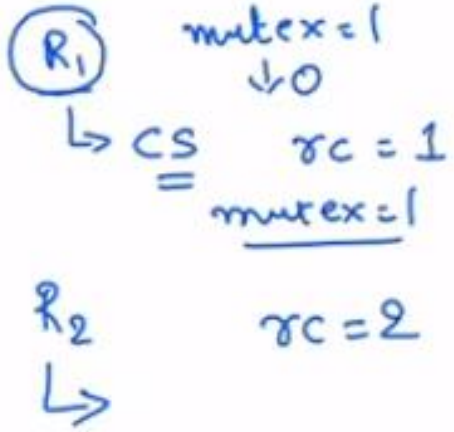
Figure 6.12 The structure of a reader process.

The code is organized into 3 sections Entry section Critical section and Exit section

- ➔ **Semaphore rw_mutex** = 1
- ➔ **semaphore read_count** initialized to 0.
- ➔ **Semaphore mutex** =1

The structure of a reader process

```
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);

        ...
    /* reading is performed */
        ...

    wait(mutex);
        read_count--;
        if (read_count == 0)
    signal(rw_mutex);
    signal(mutex);
} while (true);
```

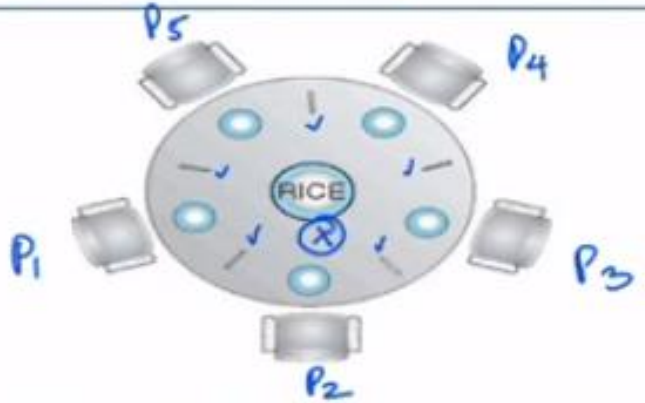


- **Semaphore rw_mutex** = 1
- **semaphore read_count** initialized to 0.
- **Semaphore mutex** =1

The Dining- Philosophers Problem

Dining-Philosophers Problem

1) think/eat



- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem

1) think/eat

2) eat = left + right chopstick

- Philosophers spend their lives alternating thinking and eating ✓
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5] );

                // eat

    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );

                // think

} while (TRUE);
```

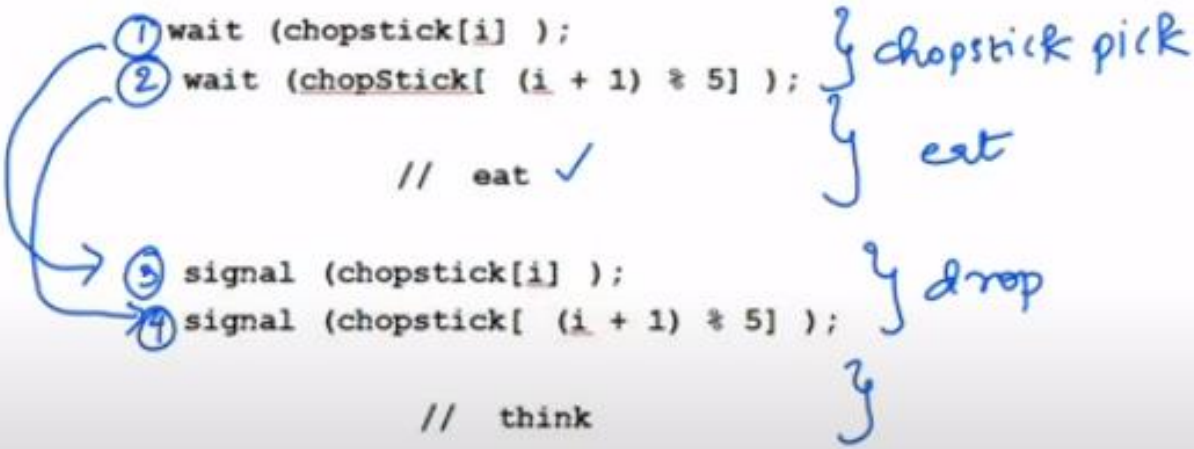
As it is circle

- What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm

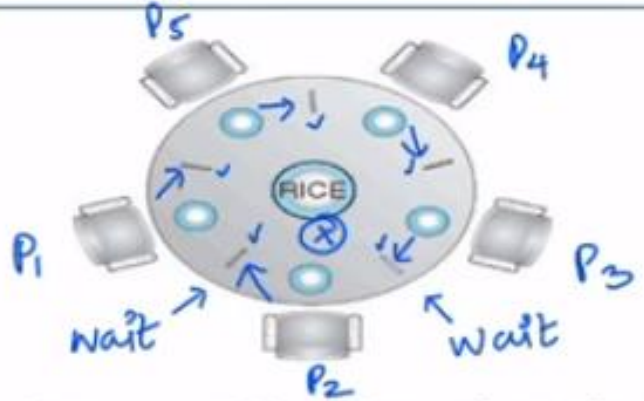
- The structure of Philosopher *i*:

```
do {
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5] );

        // eat

    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );

        // think

} while (TRUE);
```



- What is the problem with this algorithm?

Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem

1) think/eat

2) eat = left + right chopstick

$P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_5 \rightarrow P_1$

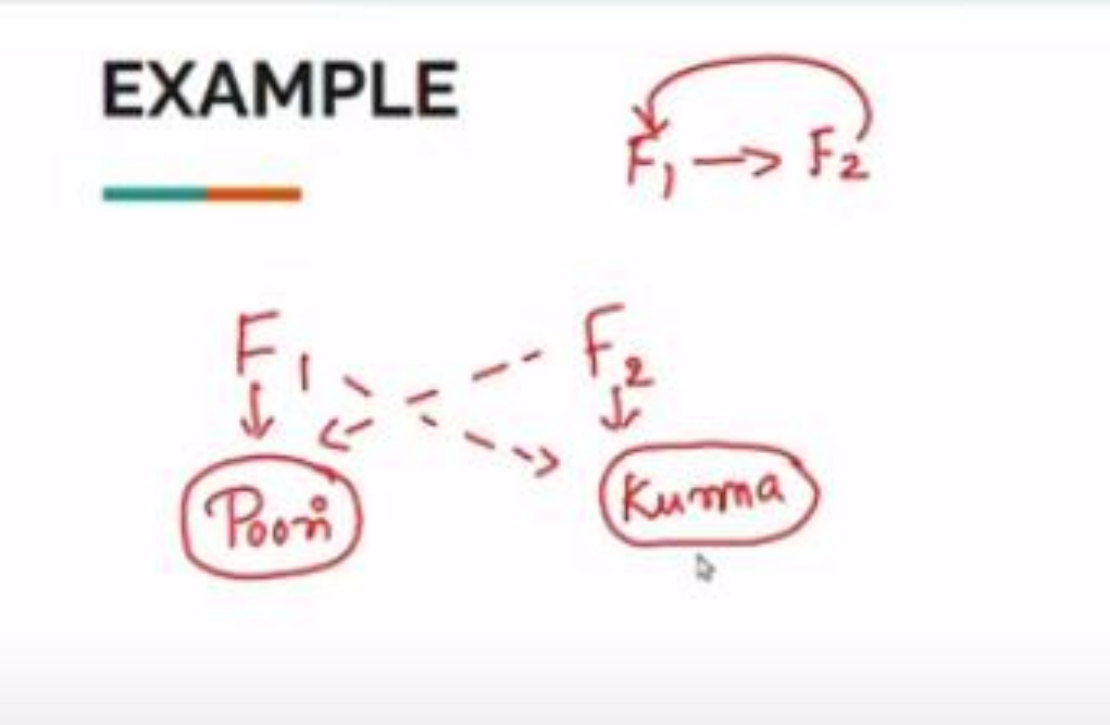
deadlock

- Philosophers spend their lives alternating thinking and eating ✓
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Deadlock handling ✓

1. Allow at most 4 philosophers to be sitting simultaneously at the table.
2. Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
3. Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Deadlock



Deadlock

What is a Dead-Lock ?

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

Realtime-Example of Deadlock

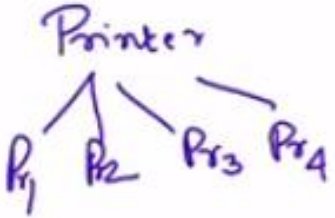
- Two Trains are Travelling on different Tracks. If a Crossing came, A Train Must Hold or wait for sometime to continue the Journey. That waiting situation Is Referred as “Deadlock” Situation.

- While, playing Chess – CHECKMATE is the deadlock situation. Where, there is no chance of escaping and have to hold for sometime to move.

DEADLOCK SYSTEM MODEL

System Model

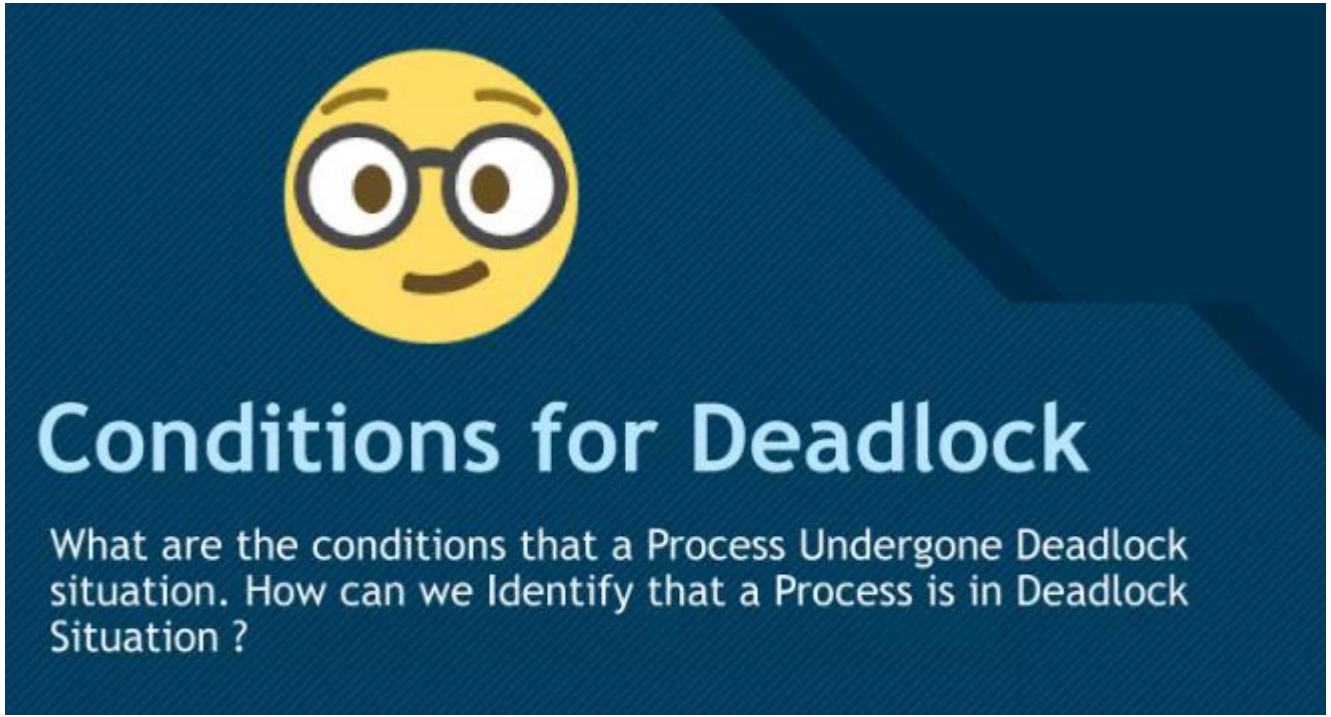
- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$
  - → *CPU cycles, memory space, I/O devices*
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**



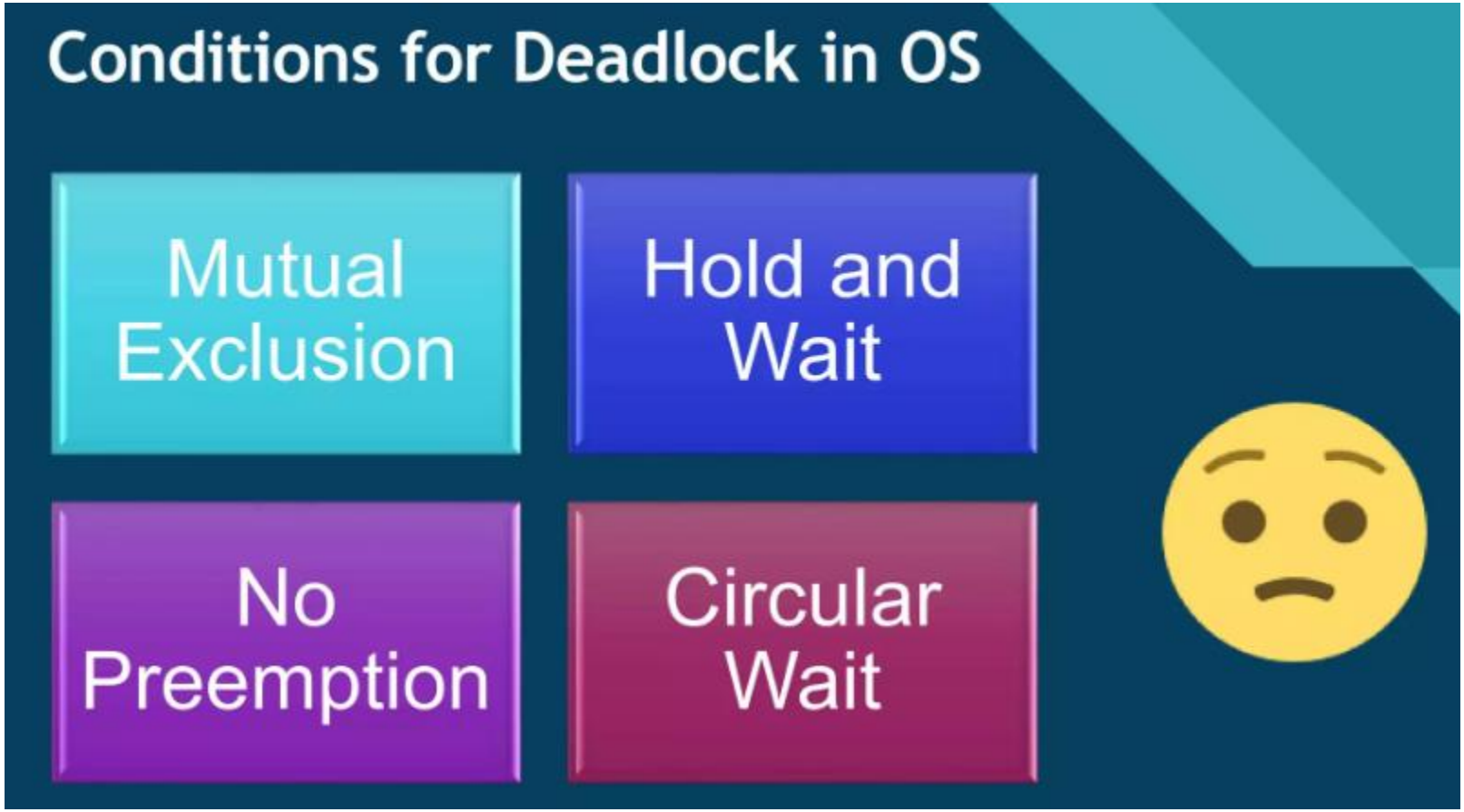
Each process utilizes a resource as follows:

- **request**
- **use**
- **release**

Deadlock Characterization

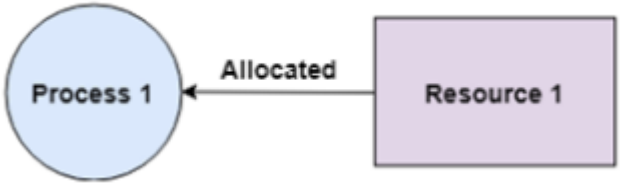


Deadlock can arise if four conditions arise simultaneously



Mutual Exclusion

There should be a resource that can only be held by one process at a time. In the diagram below, there is a single instance of Resource 1 and it is held by Process 1 only.

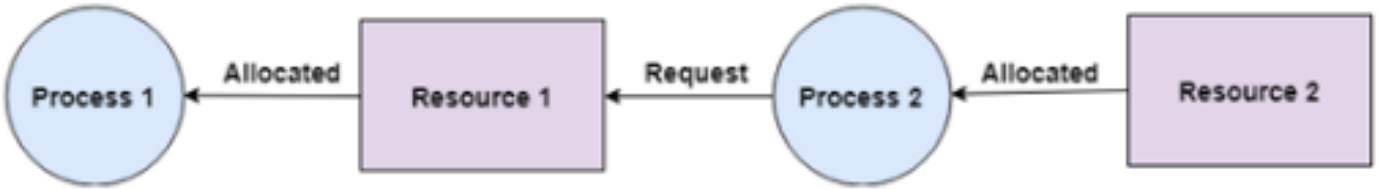


If it is shareable resource, then we can break the mutual exclusion (such as: Read-only file)

If it is not a shareable resource, then mutual exclusion must hold (such as: Printer)

No Preemption

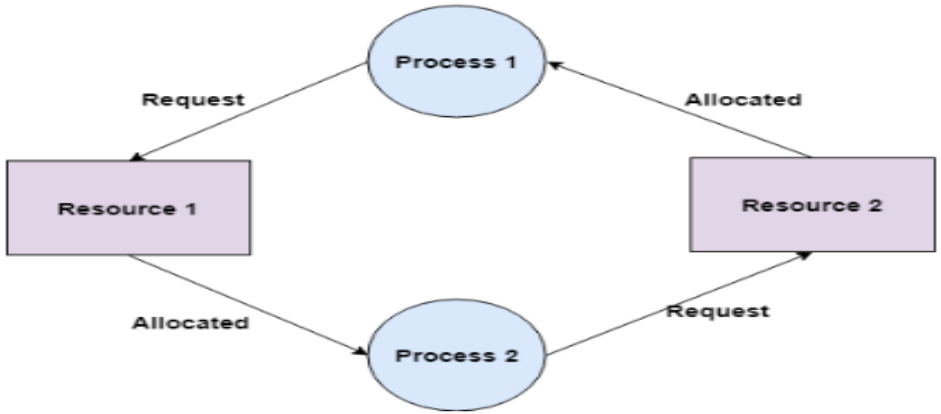
A resource cannot be preempted from a process by force. A process can only release a resource voluntarily. **In the diagram below, Process 2 cannot preempt Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntari**



Circular Wait

A process is waiting for the resource held by the second process, which is waiting for the resource held by the third process and so on, till the last process is waiting for a resource held by the first process. This forms a circular chain.

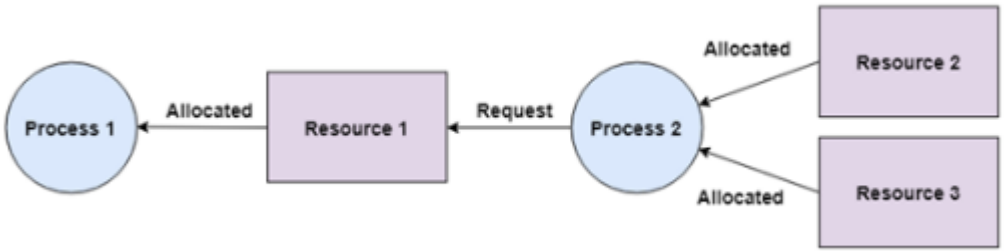
For example: Process 1 is allocated Resource2 and it is requesting Resource 1. Similarly, Process 2 is allocated Resource 1 and it is requesting Resource 2. This forms a circular wait loop.



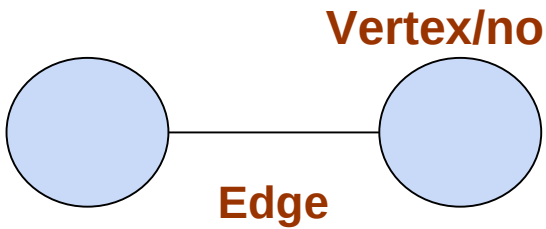
Hold and Wait

A process can hold one or multiple resources and still request more resources from other processes which are holding them.

In the diagram given below, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.



Resource Allocation Graph



- 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

Describing deadlock with the help of graph

G(V,E)

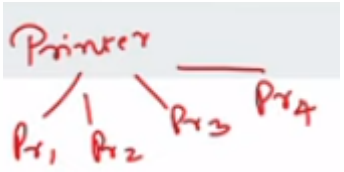
Vertex —> Process & Resources
Edges—->Request & Assignment

E is also partitioned into two types:

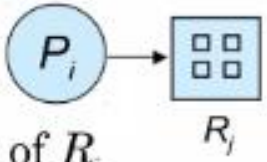
- request edge –directed edge $P1 \rightarrow Rj$
- assignment edge –directed edge $Rj \rightarrow Pi$

- Process

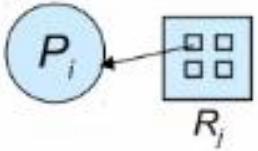
- Resource Type with 4 instances



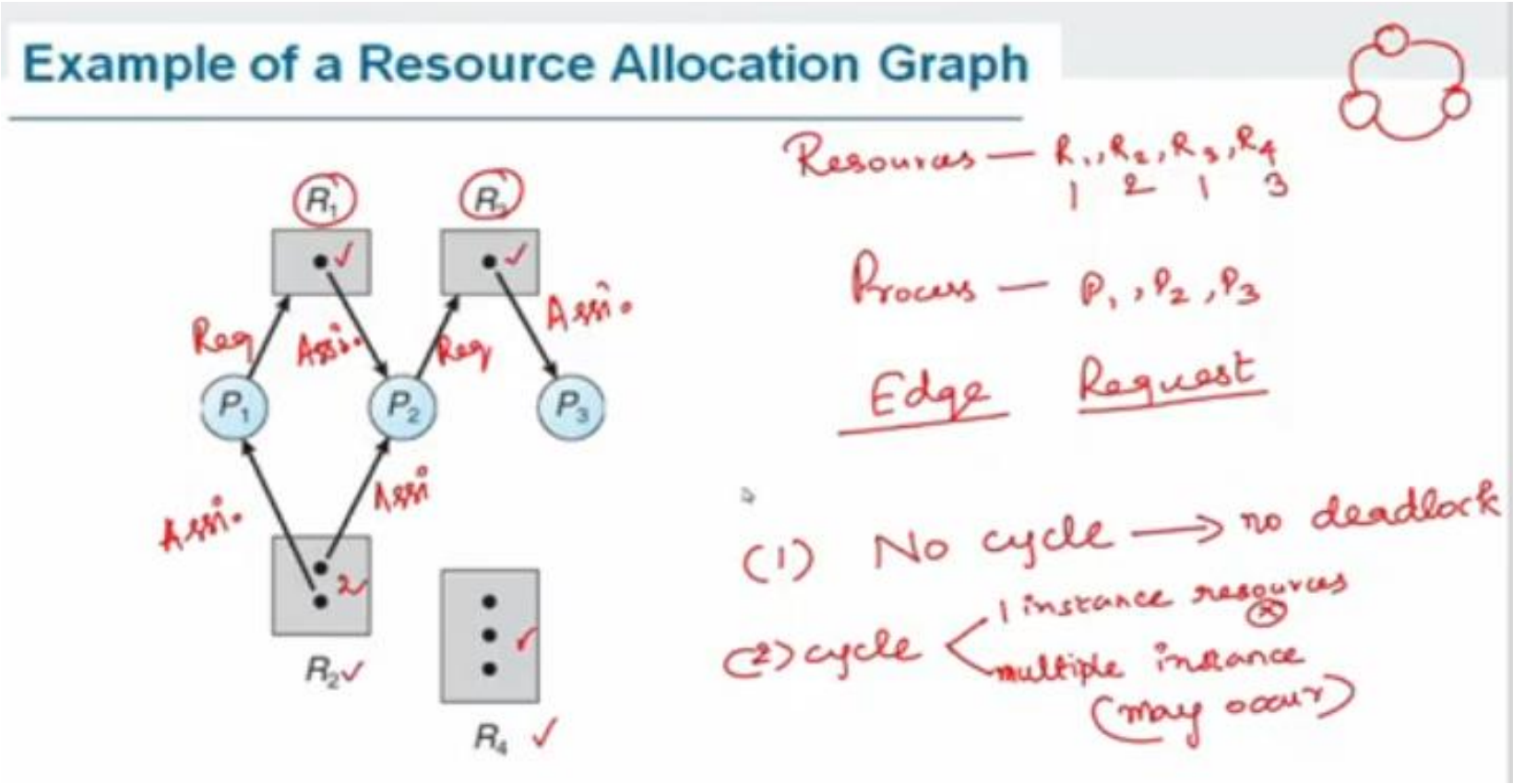
- P_i requests instance of R_j



- P_i is holding an instance of R_j



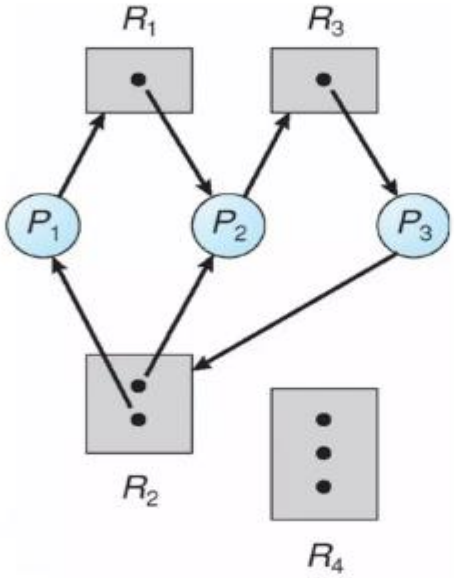
Example of a Resource Allocation Graph



P3 is holding the resource instance of R3, P1 Holding the resource instance of R2 & P2 holding the resource instance of R2.

Now we will check deadlock occur or not. There is no deadlock it doesn't have any cycle. So no deadlock.

RESOURCE ALLOCATION GRAPH WITH A DEADLOCK



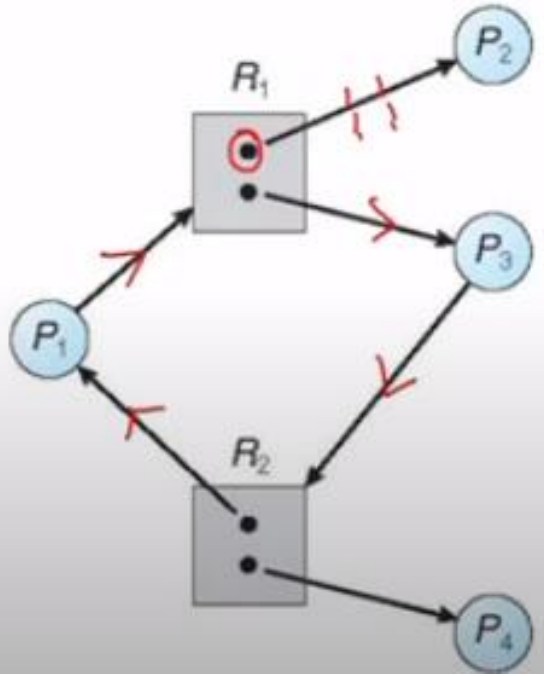
$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_1 \rightarrow R_1 (P_2) \rightarrow R_3 (P_3) \rightarrow R_2 (P_1, P_2)$

$P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow$

It contains cycle so it is deadlock P3-PR2-P2-R3 cycle formed

Graph With A Cycle But No Deadlock



$P_1 - R_1 - P_3 - R_2 - P_1$

$P_1 - R_1(P_2, P_3) - R_2(P_1, P_4)$

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 Deadlock

Deadlock Prevention
Deadlock Avoidance

Taking medicine before the disease

Deadlock Detection
Deadlock Recovery

Taking medicine after the disease

Operating system designers will take necessary precautions before designing to avoid deadlock

Once the deadlock is detected, they will decide how to detect and recover from deadlock.

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidence
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

Best example: Taking medicine before disease
Operating system designers will take necessary precautions before designing to avoid deadlock.

If all four are there, then definitely deadlock is there; then how to avoid it?
If we break any one of them, then automatically deadlock can be prevented

If these four conditions occur simultaneously in the system then deadlock may occur. Our target is we should eliminate(false) any one of these condition or all the conditions if we eliminate then deadlock will not occur.

Mutual Exclusion:

Only one resource is sharable at a time. All resources are not sharable by default.It is not possible to share a resource simultaneously among multiple processors.
To make false or to eliminate mutual exclusion just share resources to the multiple processor.

Mutual Exclusion

Attacking mutual exclusion

Mutual Exclusion = While a process is using a resource no other process can use the same resource

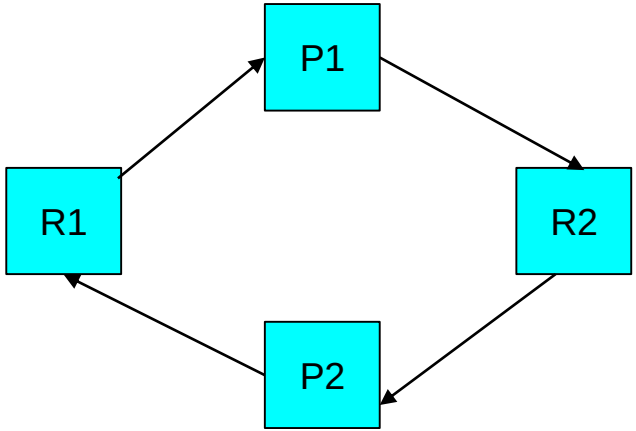
Problem

- Some resources inherently require mutual exclusion

Example

- Printers
- Files for updates
- CD-Rs

Hold & Wait



R1 is allocated to P1 (P1 holding R1) and P1 is requesting R2 (P1 waiting for R2)

P2 is allocated with R2 and P2 is requesting R1
It is called as hold and wait.
There is possibility of occurring deadlock.

Attacking hold & wait

① Require all processes to acquire all the needed resources at once

(a) Resource request does not have to be at the beginning of a process

(b) If any of the required resource is not available, drop everything (then try it again later)

(c) Any process can not make a request for any new resource, while the process currently holds some resource

② A process does NOT have to request all the resources at once

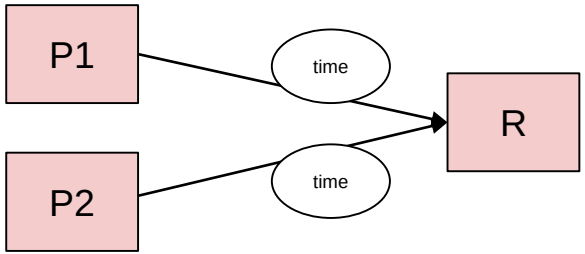
(a) Whenever an additional resource is needed, drop all what are possessed by this process and request everything at once

(b) If any process steals your resource(s) when you temporarily have to wait until all resources become available

No Preemption→ make this a preemption

Process cannot release the resource until it completes the execution. To make is false allow preemption

Preemption is achieved by time quantum method

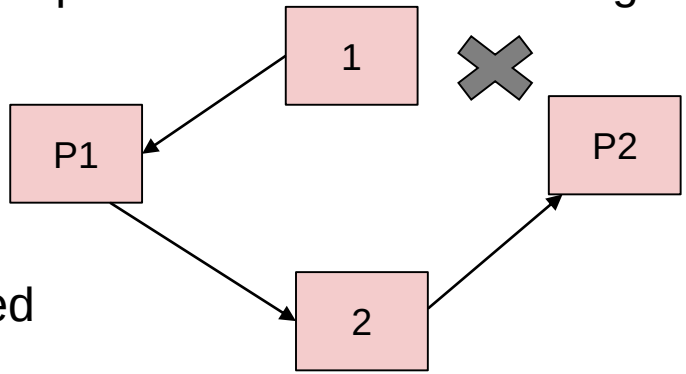


By using preemption we can eliminate deadlock

Once P1 completes its execution with the resource for the particular time it releases the resource then P2 starts its execution

Circular Wait

- Try to make circular wait false.
- It is possible to make circular wait false.
- To remove circular wait give numbering to all the resource. OS has a mapping F:R→N(integer)
  CPU- 1
  Printer- 2
  Scanner- 3
- So that the process can request resource in increasing order.

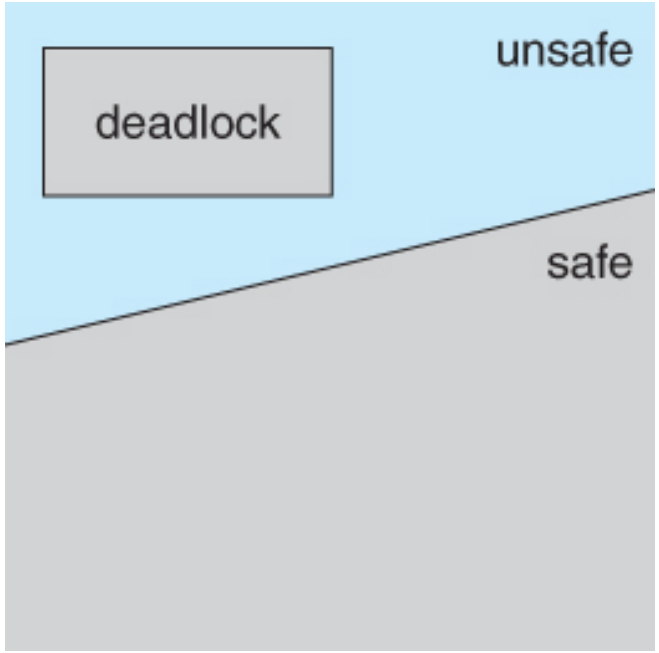


As circular wait is eliminated deadlock can be avoided

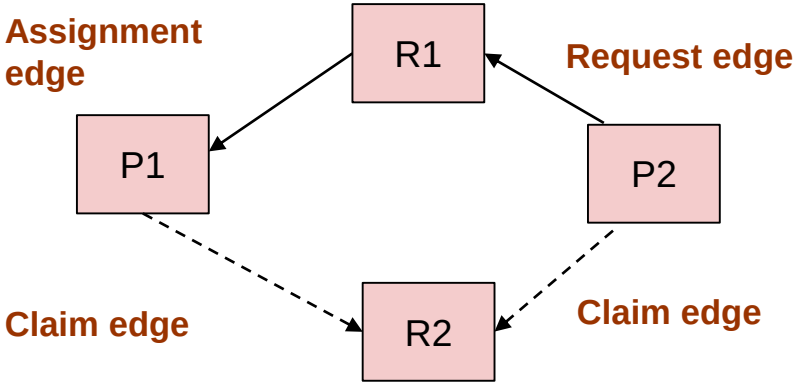
Deadlock Avoidance

- Simple and most useful model requires that each process declares maximum number of resource that it may need.
- Deadlock avoidance algorithm examines that the resource allocation can never be a circular wait condition.
- Basic facts- 2 states
- Safe state—>If a system in safe state→ If one process is allotted with one resource without any confusion or complex then its safe state—>no deadlock
- Unsafe state—>If a system in unsafe state—>possibility deadlock

Avoidance: Ensure that the system will never enter a unsafe


unsafe
deadlock
safe

Resource allocation graph algorithm

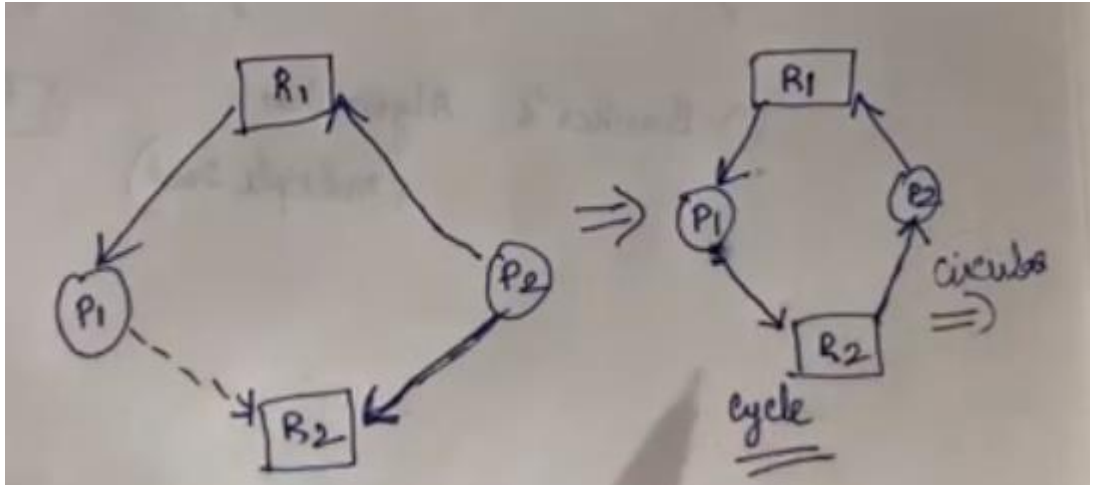


Claim edge: P_i - - - ->R_j (Pi requesting for the resource Rj in future) (going to request)

Request edge: P_i ⟶ R_j (Pi books a request for Resource Rj)

Assignment edge: P_i ⟵ R_j (Rj is assigned to the process Pi)

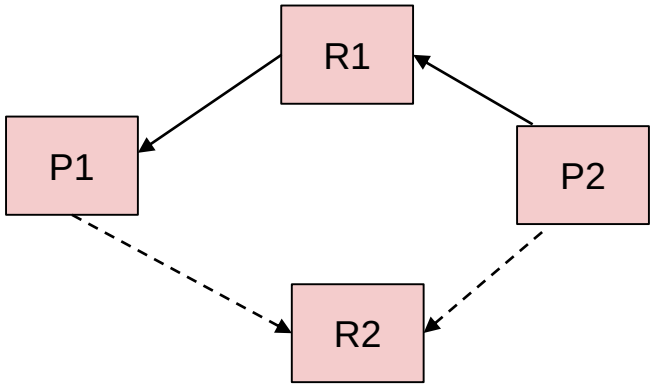
Allocated edge: P_i - - - ->R_j (once the process is done with the execution it releases the resource again and assignment converts to claim edge)


R1
P1
P2
R2
R1
P1
P2
circular
R2
cycle

If a single instance is allocated to the process then it is known as **Resource allocation graph algorithm**

Eg: System is connected with only one printer one hard disk is single instance

More than one printer or hard disk is attached to the system then it is multiple instance

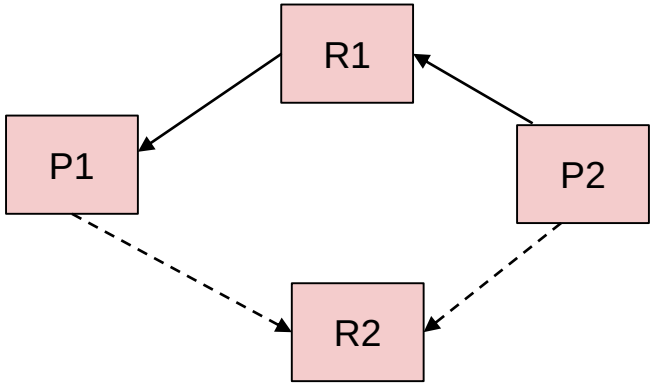


P1—R2 is claim edge
R1—P1 is assignment edge
P2—R1 is request edge
P2—R2 is claim edge

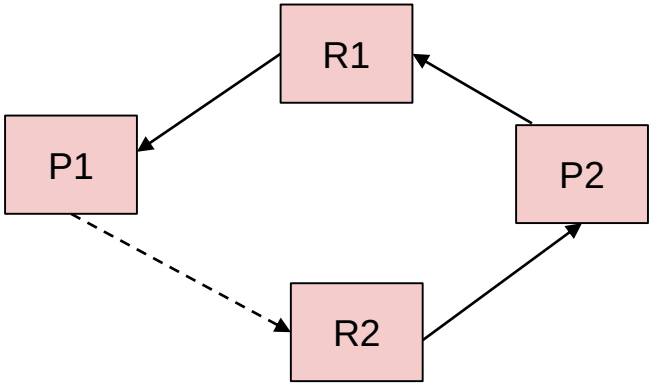
If the cycle is formed then deadlock is there so OS cannot allocate resource to the process. if the deadlock is not preset that is no cycle formed then OS can allocate resource to the process.

In this situation operating system checks whether by allocating resource to the process any deadlock is present or not if there is possibility of deadlock then it is unsafe state. Unsafe state means there is deadlock, safe state means no deadlock

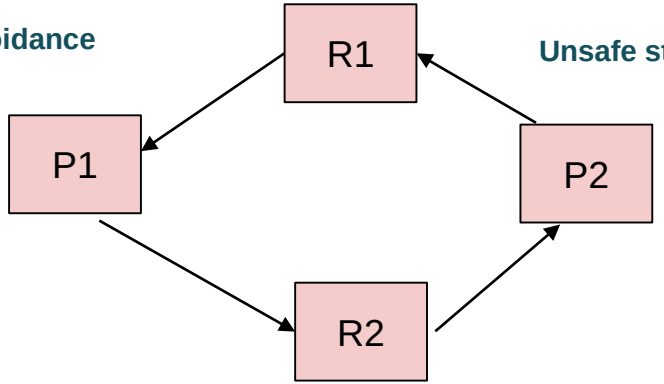
Unsafe state Resource allocation graph



Resource allocation for deadlock avoidance

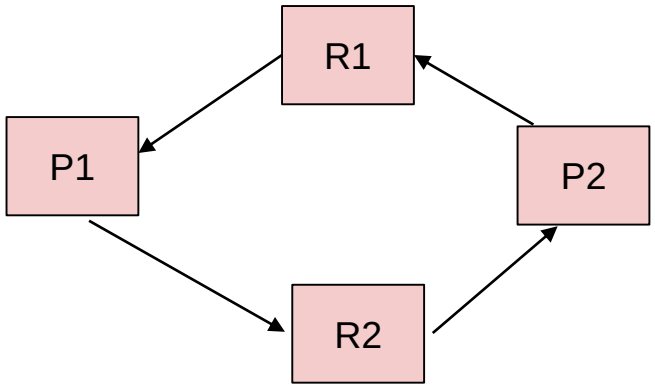


Unsafe state in Resource allocation



Cycle is formed here it means deadlock is there.

Here OS checks internally for the deadlock here R1 allocated to P1, P2 req to R1, P1 Req R2 and if R2 is allotted to P2 then cycle will form that is occurrence of deadlock so OS will wait for some time for the any other sufficient number of resource after that resource will be allocated. In this way deadlock can be avoided with the help of resource allocation graph. **This resource allocation graph is helpful only when the single instance of resource is there. When it comes to multiple instance of resource then we have to use bankers algorithm**



Banker's Algorithm

- Safety Algorithm
- Resource Request Algorithm

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

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

  Finish: Boolean value, either true or false.

  If finish[i]=true for all i return safe else unsafe

Safety Algorithm

1. Let ***Available*** and ***Finish*** be vectors of length m and n, respectively. Initialize:
   Finish $[i]$ = ***false*** for $i = 0, 1, \ldots, n-1$

2. Find an i such that both:
   (a) ***Finish*** $[i]$ = ***false***
   (b) $Need_i \leq Available$
   If no such i exists, go to step 4

3. *Available* = ***Available*** + $Allocation_i$
   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

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

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

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

Deadlock Avoidance-Banker's Algorithm

Example:

Considering a system with five processes P_0 through P_4 and three resources of type A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t_0 following snapshot of the system has been taken:

Consider the following example of a system. Check whether the system is safe or not using banker's algorithm. Determine the sequence if it safe.

Process	Allocation N0 of the allocated resources By a process A B C	Max Max resources that may be used by a process A B C	(Work) Available	**(Max - allocation) Needed**
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Step 1: To find need matrix =(Max - allocation)

Step 2: Need <=Available —------>Available=Available+Allocation

A has 10 instances, B has 5 instances and type C has 7 instances.

Allocation of resource A is for P0 to P4 is 0+2+3+2+0=7
Total resource of A is 10 so 10-7=3
Likewise for B 1+1=2 so 5-2= 3
C 2+1+2=5 so 7-5=2
So available = 3 3 2

To complete each resource task maximum required is

Banker's Algorithm

$Need_i$ <=Available ——------>Available=Available+Al

P_0 : 7 4 3 <= 3 3 2 ✖

✓

Process	Allocation A B C	Max A B C	(Work) Available	(Max - allocation) Needed
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	3 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Process	Allocation A B C	Max A B C	(Work) Available	(Max - allocation Needed
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Safe sequence

< P1

P1 entered safe sequence

Banker's Algorithm

$Need_i$ <=Available --->Available=Available+Allocation

P0 : 7 4 3 <= 3 3 2 ✖

P1: 1 2 2 <= 3 3 2 ✓ 3 3 2 + 2 0 0=5 3 2

Process	Allocation A B C	Max A B C	(Work) Available	(Max - allocation) Needed
P0	0 1 0	7 5 3	3 3 2	7 4 3
P1	2 0 0	3 2 2		1 2 2
P2	3 0 2	9 0 2		6 0 0
P3	3 1 1	2 2 2		0 1 1
P4	0 0 2	4 3 3		4 3 1

Process	Allocation A B C	Max A B C	(Work) Available	(Max - allocation) Needed
P0	0 1 0	7 5 3	3 ~~3 2~~	7 4 3
P1	2 0 0	3 2 2	5 3 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

Safe sequence

< P1

P1 entered safe sequence

Banker's Algorithm

$Need_i$ <=Available --->Available=Available+Allocation

P0 : 7 4 3 <= 3 3 2 ✖

P1: 1 2 2 <= 3 3 2 ✓ 3 3 2 + 2 0 0=5 3 2

P2: 6 0 0 <= 5 3 2 ✖

Safe sequence

< P1

P1 entered safe sequence

Process	Allocation A B C	Max A B C	(Work) Available	(Max - allocation) Needed
P0	0 1 0	7 5 3	3 3 2	7 4 3
P1	2 0 0	3 2 2		1 2 2
P2	3 0 2	9 0 2		6 0 0
P3	3 1 1	2 2 2		0 1 1
P4	0 0 2	4 3 3		4 3 1

Process	Allocation A B C	Max A B C	(Work) Available	(Max - allocation Needed
P0	0 1 0	7 5 3	~~3 3 2~~	7 4 3
P1	2 0 0	3 2 2	5 3 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

Banker's Algorithm

$Need_i$ <=Available --->Available=Available+Allocation

P0 : 7 4 3 <= 3 3 2 ✗

P1: 1 2 2 <= 3 3 2 ✓ 3 3 2 + 2 0 0=5 3 2

P2: 6 0 0 <= 5 3 2 ✗

P3: 0 1 1 <= 5 3 2 ✓ 5 3 2 + 2 1 1= 7 4 3

Safe sequence

< P1, P3

P1, P3 entered safe sequence

Process	Allocation A B C	Max A B C	(Work) Available	(Max - allocation) Needed
P0	0 1 0	7 5 3	3 3 2	7 4 3
P1	2 0 0	3 2 2		1 2 2
P2	3 0 2	9 0 2		6 0 0
P3	3 1 1	2 2 2		0 1 1
P4	0 0 2	4 3 3		4 3 1

Process	Allocation A B C	Max A B C	(Work) Available	(Max - allocation Needed
P0	0 1 0	7 5 3	~~3 3 2~~	7 4 3
P1	2 0 0	3 2 2	~~5 3 2~~	1 2 2
P2	3 0 2	9 0 2	7 4 3	6 0 0
P3	2 1 1	2 2 2		0 1 1
P4	0 0 2	4 3 3		4 3 1

Banker's Algorithm

$Need_i$ <=Available --->Available=Available+Allocation

Process	Allocation A B C	Max A B C	(Work) Available	(Max - allocation) Needed
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	3 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

P_0 : 7 4 3 <= 3 3 2 ✗

P1: 1 2 2 <= 3 3 2 ✓ 3 3 2 + 2 0 0=5 3 2

P2: 6 0 0 <= 5 3 2 ✗

P3: 0 1 1 <= 5 3 2 ✓ 5 3 2 + 2 1 1= 7 4 3

P4: 4 3 1 <= 7 4 3 ✗

Process	Allocation A B C	Max A B C	(Work) Available	(Max - allocation Needed
P_0	0 1 0	7 5 3	~~3 3 2~~	7 4 3
P_1	2 0 0	3 2 2	~~5 3 2~~	1 2 2
P_2	3 0 2	9 0 2	7 4 3	6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Safe sequence

< P1, P3

P1, P3 entered safe sequence

Banker's Algorithm

$Need_i$ <=Available --->Available=Available+Allocation

P0 : 7 4 3 <= 3 3 2 ✗

P1: 1 2 2 <= 3 3 2 ✓ 3 3 2 + 2 0 0=5 3 2

P2: 6 0 0 <= 5 3 2 ✗

P3: 0 1 1 <= 5 3 2 ✓ 5 3 2 + 2 1 1= 7 4 3

P4: 4 3 1 <= 7 4 3 ✓ 7 4 3+ 0 0 2= 7 4 5

Safe sequence

< P1, P3, P4

P1, P3, P4 entered safe sequence

Process	Allocation A B C	Max A B C	(Work) Available	(Max - allocation) Needed
P0	0 1 0	7 5 3	3 3 2	7 4 3
P1	2 0 0	3 2 2		1 2 2
P2	3 0 2	9 0 2		6 0 0
P3	3 1 1	2 2 2		0 1 1
P4	0 0 2	4 3 3		4 3 1

Process	Allocation A B C	Max A B C	(Work) Available	(Max - allocation Needed
P0	0 1 0	7 5 3	~~3 3 2~~	7 4 3
P1	2 0 0	3 2 2	~~5 3 2~~	1 2 2
P2	3 0 2	9 0 2	~~7 4 3~~	6 0 0
P3	2 1 1	2 2 2	7 4 5	0 1 1
P4	0 0 2	4 3 3		4 3 1

Banker's Algorithm

$Need_i$ <=Available --->Available=Available+Allocation

Process	Allocation A B C	Max A B C	(Work) Available	(Max - allocation) Needed
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	3 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

P0 : 7 4 3 <= 3 3 2 ✗

P1: 1 2 2 <= 3 3 2 ✓ 3 3 2 + 2 0 0=5 3 2

P2: 6 0 0 <= 5 3 2 ✗

P3: 0 1 1 <= 5 3 2 ✓ 5 3 2 + 2 1 1= 7 4 3

P4: 4 3 1 <= 7 4 3 ✓ 7 4 3 + 0 0 2= 7 4 5

P0: 7 4 3 <= 7 4 5 ✓ 7 4 5 + 0 1 0= 7 5 5

Process	Allocation A B C	Max A B C	(Work) Available	(Max - allocation Needed
P_0	0 1 0	7 5 3	~~3 3 2~~	7 4 3
P_1	2 0 0	3 2 2	~~5 3 2~~	1 2 2
P_2	3 0 2	9 0 2	~~7 4 3~~	6 0 0
P_3	2 1 1	2 2 2	~~7 4 5~~	0 1 1
P_4	0 0 2	4 3 3	7 5 5	4 3 1

Safe sequence

< P1, P3, P4, P0

P1, P3, P4, P0 entered safe sequence

Banker's Algorithm

$Need_i$ <=Available --->Available=Available+Allocation

Process	Allocation A B C	Max A B C	(Work) Available	(Max - allocation) Needed
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	3 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

P_0 : 7 4 3 <= 3 3 2 ✗

P1: 1 2 2 <= 3 3 2 ✓ 3 3 2 + 2 0 0=5 3 2

P2: 6 0 0 <= 5 3 2 ✗

P3: 0 1 1 <= 5 3 2 ✓ 5 3 2 + 2 1 1= 7 4 3

P4: 4 3 1 <= 7 4 3 ✓ 7 4 3 + 0 0 2= 7 4 5

P0: 7 4 3 <= 7 4 5 ✓ 7 4 5 + 0 1 0= 7 5 5

P2: 6 0 0 <= 7 5 5 ✓ 7 5 5 + 3 0 2= 1 0 5 7

Process	Allocation A B C	Max A B C	(Work) Available	(Max - allocati Needed
P_0	0 1 0	7 5 3	~~3 3~~ 2	7 4 3
P_1	2 0 0	3 2 2	~~5 3~~ 2	1 2 2
P_2	3 0 2	9 0 2	~~7 4~~ 3	6 0 0
P_3	2 1 1	2 2 2	~~7 4~~ 5	0 1 1
P_4	0 0 2	4 3 3	~~7 5 5~~ 1 0 5 7	4 3 1

Safe sequence
< P1, P3, P4, P0, P2

P1, P3, P4, P0, P2 entered safe sequence

Therefore, The system is in safe state

Resource request algorithm.

1. $Request_i \leq need_i$ goto (2)
2. $Request_i \leq available$ goto (3)
3. $available = available - request_i$

   $allocation = allocation + request_i$

   $need_i = need_i - request_i$
4. Check if new state is safe or not (using Banker's algorithm)

Example:

Considering a system with five processes P_0 through P_4 and three resources of type A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t_0 following snapshot of the system has been taken:

Process	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	

Find the safe sequence.

Three resources R1, R2 ,R3 are printer, scanners, buses, etc Each resource can provide service. For example. R1 has 10 instance it means it can provides 10services.

safe sequence: Without deadlock which order will be safe is safe sequence.

Allocation means resource allocated to a process for B 1 resource is allocated from 5 resource.

Process	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	

A has 10 instances, B has 5 instances and type C has 7 instances.

Allocation of resource A is for P0 to P4 is 0+2+3+2+0=7
Total resource of A is 10 so 10-7=3
Likewise for B 1+1=2 so 5-2= 3
C 2+1+2=5 so 7-5=2
So available = 3 3 2

To complete each resource task maximum required is

Available
A B C
3 3 2

Q.1: What will be the content of the Need matrix?

Need [i, j] = Max [i, j] – Allocation [i, j]
So, the content of Need Matrix is:

Process	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	

Process	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

Q.2: Is the system in a safe state? If Yes, then what is the safe sequence? Applying the Safety algorithm on the given system,

m=3, n=5 — Step 1 of Safety Algo

Work = Available

Work = | 3 | 3 | 2 |

	0	1	2	3	4
Finish =	false	false	false	false	false

Step 2

For i = 0

$Need_0$ = 7, 4, 3 — 7,4,3 — 3,3,2

Finish [0] is false and $Need_0$ > Work

So P_0 must wait — But Need ≤ Work

Step 2

For i = 1

$Need_1$ = 1, 2, 2 — 1,2,2 — 3,3,2

Finish [1] is false and $Need_1$ < Work

So P_1 must be kept in safe sequence

Step 3

3, 3, 2 — 2, 0, 0

Work = Work + $Allocation_1$

	A	B	C
Work =	5	3	2

	0	1	2	3	4
Finish =	false	true	false	false	false

Step 2

For i = 2

$Need_2$ = 6 , 0, 0 — 6, 0, 0 — 5,3, 2

Finish [2] is false and $Need_2$ > Work

So P_2 must wait

Step 2

For i=3

$Need_3$ = 0, 1, 1 — 0, 1, 1 — 5, 3, 2

Finish [3] = false and $Need_3$ < Work

So P_3 must be kept in safe sequence

Step 3

5, 3, 2 — 2, 1, 1

Work = Work + $Allocation_3$

	A	B	C
Work =	7	4	3

	0	1	2	3	4
Finish =	false	true	false	true	false

Step 2

For i = 4

$Need_4$ = 4, 3, 1 — 4, 3, 1 — 7, 4, 3

Finish [4] = false and $Need_4$ < Work

So P_4 must be kept in safe sequence

Step 3

7, 4, 3 — 0, 0, 2

Work = Work + $Allocation_4$

	A	B	C
Work =	7	4	5

	0	1	2	3	4
Finish =	false	true	false	true	true

Step 2

For i = 0

$Need_0$ = 7, 4, 3 — 7, 4, 3 — 7, 4, 5

Finish [0] is false and Need < Work

So P_0 must be kept in safe sequence

Step 3

7, 4, 5 — 0, 1, 0

Work = Work + $Allocation_0$

	A	B	C
Work =	7	5	5

	0	1	2	3	4
Finish =	true	true	false	true	true

Step 2

For i = 2

$Need_2$ = 6 , 0, 0 — 6, 0, 0 — 7, 5, 5

Finish [2] is false and $Need_2$ < Work

So P_2 must be kept in safe sequence

Step 3

7, 5, 5 — 3, 0, 2

Work = Work + $Allocation_2$

	A	B	C
Work =	10	5	7

	0	1	2	3	4
Finish =	true	true	true	true	true

Step 4

Finish [i] = true for 0 ≤ i ≤ n

Hence the system is in Safe state

The safe sequence is P_1, P_3 , P_4 , P_0, P_2

Q.3: What will happen if process P_1 requests one additional instance of resource type A and two instances of resource ty

A B C

$Request_1 = 1, 0, 2$

To decide whether the request is granted we use Resource Request algorithm

Step 1

$Request_1 \; (1, 0, 2) < Need_1 \; (1, 2, 2)$ ✓

Step 2

$Request_1 \; (1, 0, 2) < Available \; (3, 3, 2)$ ✓

Step 3

$Available = Available - Request_1$

$Allocation_1 = Allocation_1 + Request_1$

$Need_1 = Need_1 - Request_1$

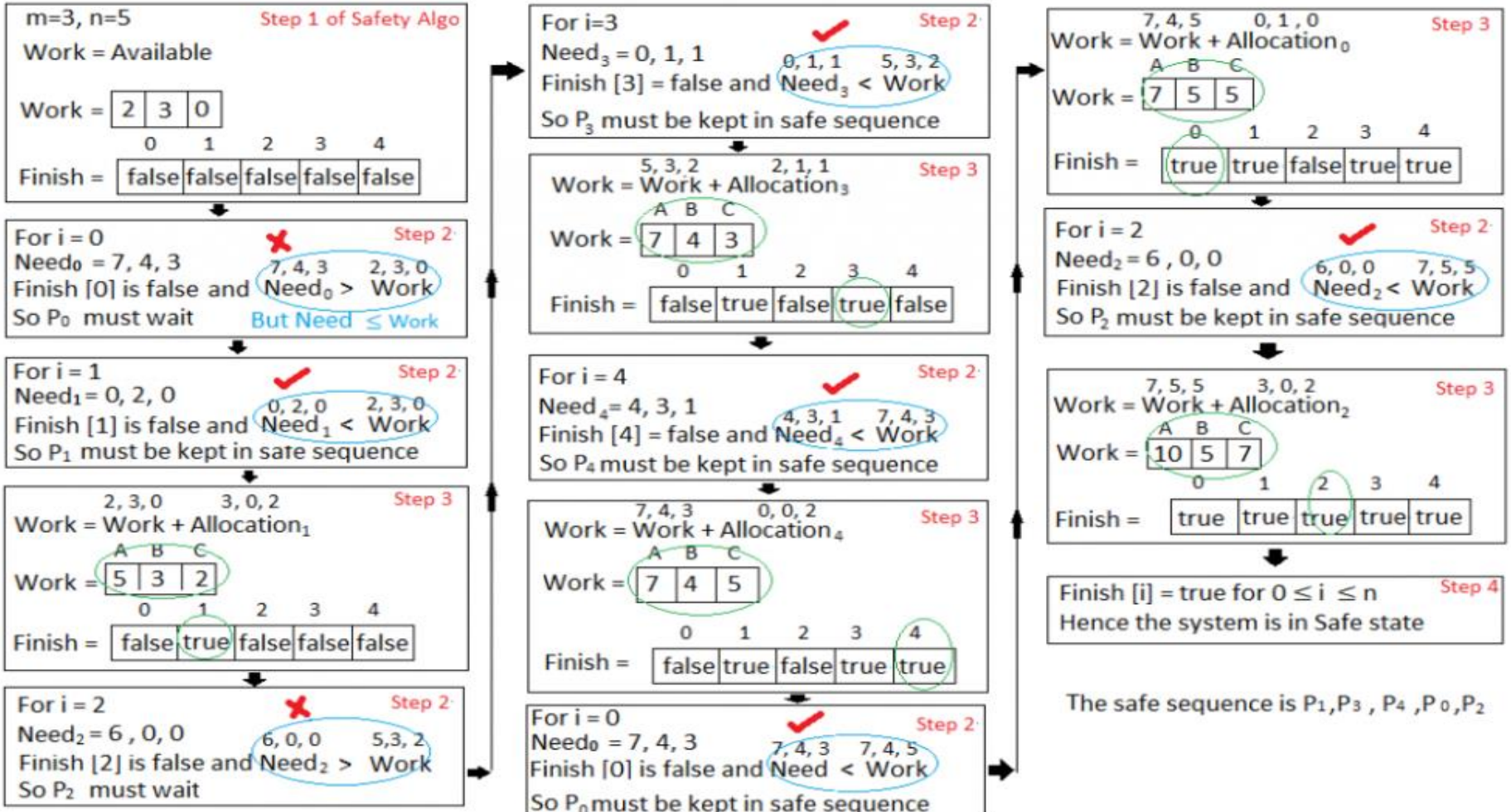
Apply the formula to find need, allocation and available for the requested process

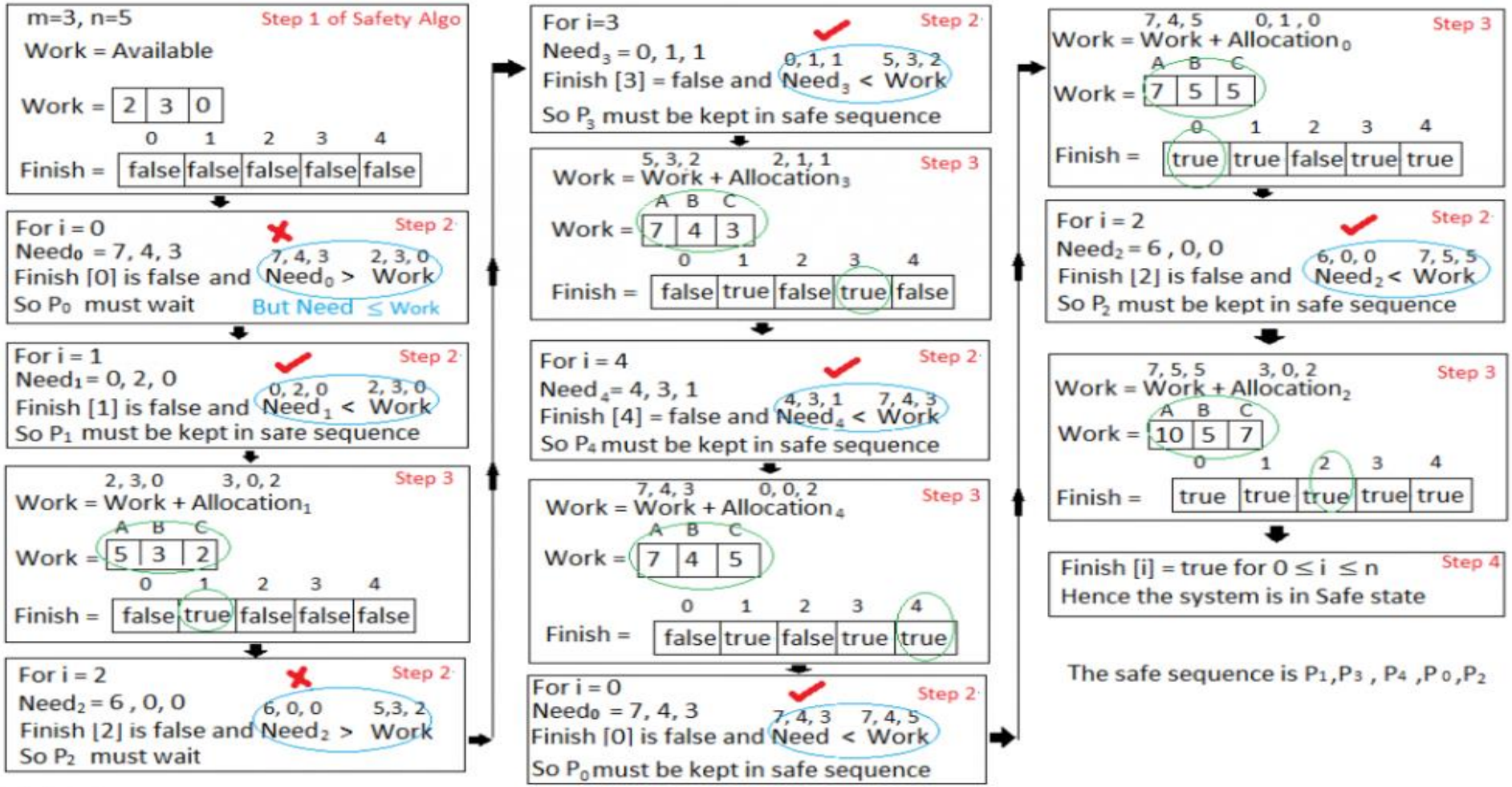
Process	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Process	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	

Process	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

We must determine whether this new system state is safe. To do so, we again execute Safety algorithm on the above data structures.





Hence the new system state is safe, so we can immediately grant the request for process P_1.

Example 2: Consider the snapshot of the process:

Process	allocation A B C D	maximum	available
P0	0 0 1 2	0 0 1 2	1 5 2 0
P1	1 0 0 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	

Answer the following questions using the banker's algorithm.

I. What is the content of the matrix needed?
II. Is the system in a safe state?
III. If a request from process Pi arrives for(0, 4, 2 ,0) can the request be granted immediately?

Need matrix = 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

Process P_0 :- Need $\leq$ available

$0000 \leq 1520$

Process P_0 execute

New available = Available + allocation

= 1520 + 0012

= 1532

Process P_1 : $0750 \leq 1532$

Process P_1 does not execute

Process P_2 : $1002 \leq 1532$

Process P_2 execute

New available = Available + allocation

= 1532 + 1354

= 2886

Process P_3 : $0020 \leq 2886$

Process P_3 execute

New available = 2886 + 0631

= 2 8 14 11 8

Process P_4 : $0642 \leq 2141118$

Process P_4 execute

New available = 2 14 1118 + 0014

= 2 14 12 12

Process P_1 : 0 7 5 0 ≤ 2 14 12 12

New available = 2 14 12 12 + 1 0 0 0

= 3 14 12 12

Order of execution

$< P_0, P_2, P_3, P_4, P_1 >$

Yes the system is in safe state

(iii) for process P_1

If Request ≤ need

(0 4 2 0) ≤ (0 7 5 0)

then Need for process 1 = 0 7 5 0

this condition is satisfied

If Request ≤ available

(0 4 2 0) ≤ (1 5 2 0)

this condition is also satisfied

New available = available - Request

= 1 5 2 0 - 0 4 2 0

= 1 1 0 0

Allocation = 1 0 0 0 + 0 4 2 0

= 1 4 2 0

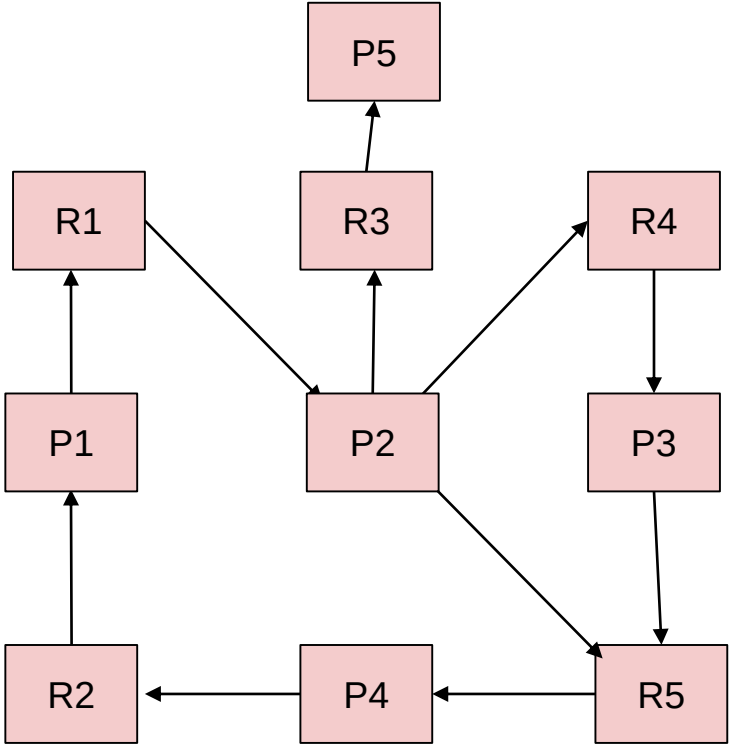
Hence Process P_1 can be granted immediately

Deadlock Detection

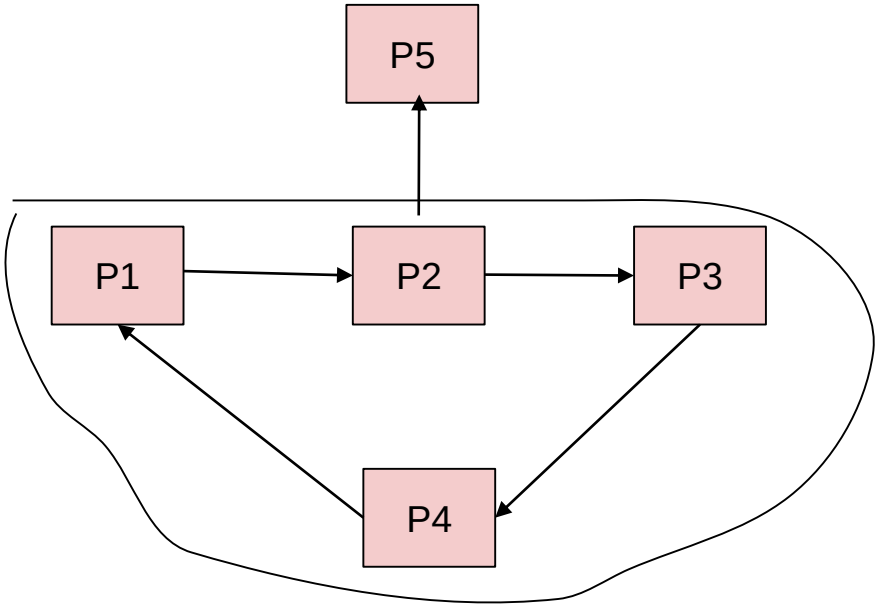
After allocating resources to the process, the OS checks for the presence of deadlock. In order to detect deadlock, two approaches are there :

1. **Single instance of each resource type**
2. **Several instance of resource type**

Single instance of each resource type

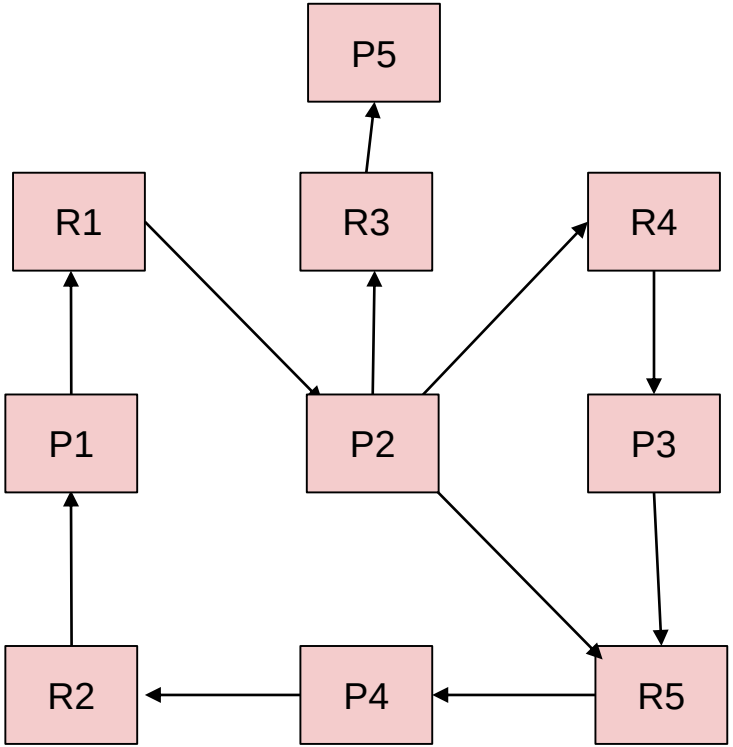


Corresponding wait for graph

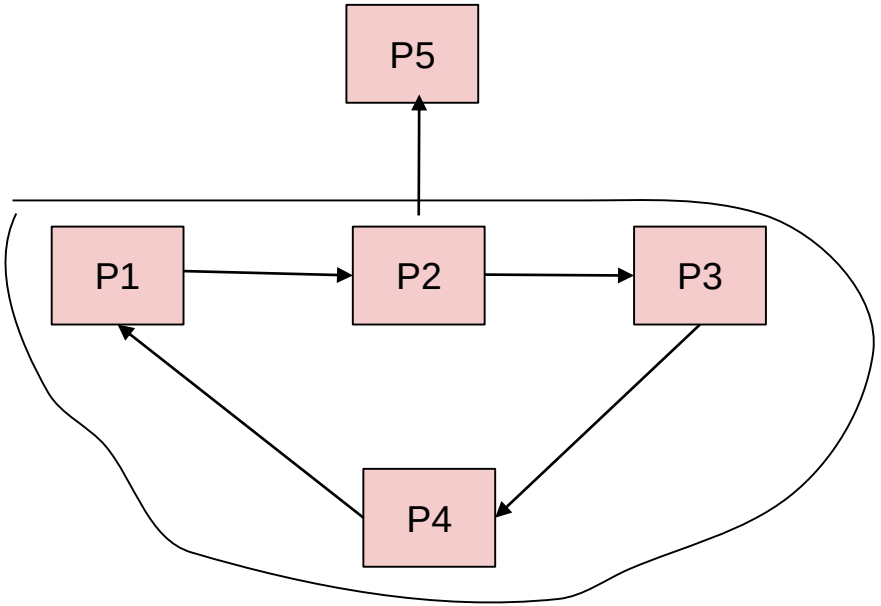


Check for deadlock

Single instance of each resource type



Corresponding wait for graph



Check for deadlock: P1-P2-P3-P4 cycle is formed so deadlock exist from P1to P4. P5 is out of deadlock. This is how u can find deadlock. Here the resource type is single instance if multiple instance then u have to go with bankers algm.

Deadlock Detection Algorithm

Steps of Algorithm:

1. Let *Work* and *Finish* be vectors of length m and n respectively. Initialize *Work*= *Available*. For *i=0, 1, …., n-1*, if *Request*$_i$ = 0, then *Finish[i]* = true; otherwise, *Finish[i]*= false.
2. Find an index i such that both

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

   b) *Request*$_i$ <= *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, 0<=i<n, then the system is in a deadlocked state. Moreover, if *Finish[i]==false* the process P_i is deadlocked.

Example of Detection Algorithm

Five processes P_0 through P_4; three resource types
A (7 instances), *B* (2 instances), and *C* (6 instances)

A—> 7 instance (printer
B—>2 instance (scanner)
C—>6 instance (RAM)

Process	(Resource being allocated to the process) **Allocation A B C**	(REquest made by the process) **Request**	Resource in the system **(Work) Available**
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Multiple instances of each resource type

Several instance of each resource type

A—> 7 instance (printer
B—>2 instance (scanner)
C—>6 instance (RAM)

Process	(Resource being allocated to the process) **Allocation A B C**	(REquest made by the process) **Request**	Resource in the system **(Work) Available**
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

How to choose the process to enter into the system? Select the process whose request is lesser than the available.

work=available

P0: Request<=work(Available)
0 0 0<= 0 0 0 ✓
work=work +allocation
work= 0 0 0+ 0 1 0= 0 1 0

Safe sequence
< P0 entered safe sequence

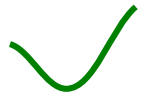
Several instance of each resource type

Process	(Resource being allocated to the process) **Allocation A B C**	(REquest made by the process) **Request**	Resource in the system **(Work) Available**
P_0	0 1 0	0 0 0	~~0 0 0~~ 0 1 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Select the process whose request is lesser than the available (0 1 0) here P2 request is< available

work=available

P2: Request<=work



0 0 0<= 0 1 0

work=work +allocation

work= 0 1 0+ 3 0 3= 3 1 3

Safe sequence

< P0, P2 entered safe sequence

Several instance of each resource type

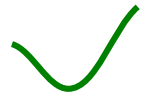
Process	(Resource being allocated to the process) **Allocation A B C**	(REquest made by the process) **Request**	Resource in the system **(Work) Available**
P_0	0 1 0	0 0 0	~~0 0 0~~ ~~0 1 0~~ 3 1 3
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Select the process whose request is lesser than the available (3 1 3)

work=available

P2: Request<=work

0 0 0<= 0 1 0



work=work +allocation

work= 0 1 0+ 3 0 3= 3 1 3

Safe sequence

< P0, P2 entered safe sequence

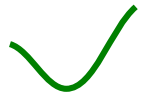
Several instance of each resource type

Process	(Resource being allocated to the process) **Allocation A B C**	(REquest made by the process) **Request**	Resource in the system **(Work) Available**
P_0	0 1 0	0 0 0	~~0 0 0~~ ~~0 1 0~~ 3 1 3
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

A—> 7 instance (printer
B—>2 instance (scanner)
C—>6 instance (RAM)

work=available

P3: Request<=work
1 0 0<= 3 1 3
work=work +allocation
work= 3 1 3+ 2 1 1= 5 2 4



Safe sequence
< P0, P2, P3 entered safe sequence

Several instance of each resource type

Process	(Resource being allocated to the process) **Allocation A B C**	(REquest made by the process) **Request**	Resource in the system **(Work) Available**
P_0	0 1 0	0 0 0	~~0 0 0~~ ~~0 1 0~~ ~~3 1 3~~ ~~5 2 4~~ 5 2 6
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

A—> 7 instance (printer
B—>2 instance (scanner)
C—>6 instance (RAM)

work=available

P1: Request<=work
2 0 2 <= 5 2 6
work=work +allocation
work= 5 2 6+2 0 0= 7 2 6

Safe sequence
< P0, P2,P3, P4, P1 entered safe sequence
In text book they have followed the Safe sequence in the given order better to follow this
< P0, P2,P3, P1, P4 entered safe sequence

If the resource is allotted in this sequence by the OS then there won't be deadlock.

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

Once deadlock is detected, how to recover from deadlock:

Approaches for recovery

1. Process Termination:
   Abort all deadlock processes.
   Abort one process at a time until the deadlock cycle is eliminated

1. Resource Preemption
   Preemption—>resource to be released and allocate resource to the process
   3 issues need to be considered while preempting:
   1. Selecting a victim
   2. Rollback
   3. Starvation

1. Abort all deadlock processes.

   If any of the process is suffered by deadlock then abort that process.
   Ex: 3 process is suffering from deadlock P1 completes 86% of execution
   P2 completes 80% of execution likewise P3 completes 90% of execution if u abort now CPU, memory, IO device and usage of resource is wasted that is the disadvantage of this.

1. Abort one process at a time until the deadlock cycle is eliminated

   Abort one process and apply deadlock detection algorithm to be applied on the cycle and check deadlock is removed if you continue with the abortion of the next process.
   From which process to abort it depends on the priority of the process that is the lowest execution % of time P2 can be aborted first

Advantages of Process Termination

- It is a simple method for breaking a deadlock.
- It ensures that the deadlock will be resolved quickly, as all processes involved in the deadlock are terminated simultaneously.
- It frees up resources that were being used by the deadlocked processes, making those resources available for other processes.

Disadvantages of Process Termination

- It can result in the loss of data and other resources that were being used by the terminated processes.
- It may cause further problems in the system if the terminated processes were critical to the system's operation.
- It may result in a waste of resources, as the terminated processes may have already completed a significant amount of work before being terminated.

2. Resource Preemption

Preemption—>resource to be released and allocate resource to the process

3 issues need to be considered while preempting:

1. Selecting a victim
2. Rollback (start execution from the beginning)
3. Starvation (releasing the resource repeatedly based on the lowest priority where others wait for long time is starvation)

Selecting a victim:

For example if 10 process are suffering from deadlock. Have to find which process to be released. Each process are allocated with the resource. Find the lowest priority resource with less execution time.

Rollback:

We must determine what should be done with the process from which resources are preempted. One simple idea is total rollback. That means aborting the process and restarting it.

Starvation

In a system, it may happen that the same process is always picked as a victim. As a result, that process will never complete its designated task. This situation is called Starvation and must be avoided. One solution is that a process must be picked as a victim only a finite number of times.