Unit 3- Synchronization Tools

J.Premalatha
Professor/IT
Kongu Engineering College
Perundurai

Need for Synchronization

- A **cooperating process** is one that can affect or be affected by other processes executing in the system.
- Cooperating processes can either **directly share a logical address space (that is, both code and data) or be allowed to share data** only through shared memory or message passing.
- **Concurrent access to shared data may result in data inconsistency**
- Various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that **data consistency is maintained.**

Need for Synchronization: Producer –Consumer Problem

The buffer is empty when in ==out;
the buffer is full when
((in + 1) % BUFFER SIZE) == out.

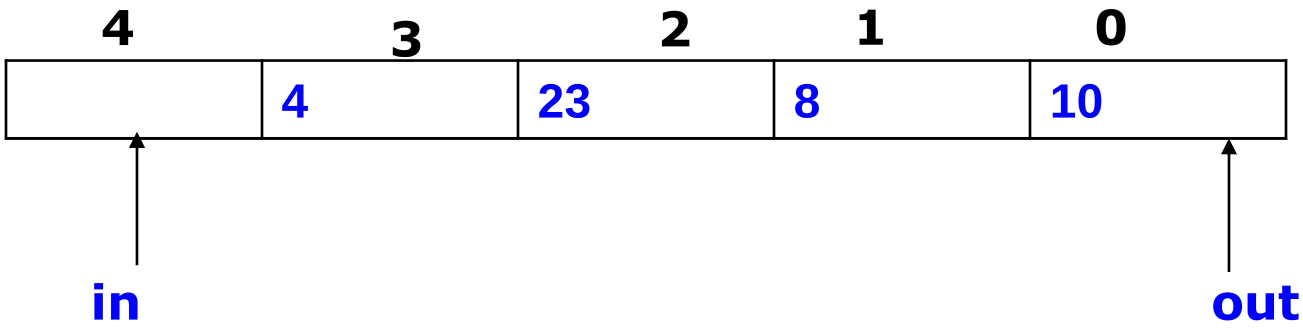
```
item next_produced;

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

The code for the producer process can be modified as follows

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

Original solution allowed at most BUFFER SIZE − 1 items in the buffer at the same time.

Buffer size=5



To modify the algorithm to remedy **[at most BUFFER SIZE − 1 items in the buffer at the same time]** this deficiency.
One possibility is **to add an integer variable, Count, initialized to 0.** Count is incremented every time when add a new item to the buffer and is decremented every time when remove one item from the buffer.

The code for the consumer process can be modified as follows:

```
while (true) {
while (count == 0) ;
/* do nothing */
 next consumed = buffer[out];
out = (out + 1) % BUFFER SIZE;
count--;
/* consume the item in next consumed */
}
```

```
item next_consumed;

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

- The producer and consumer routines shown are correct separately, they may not function correctly when executed concurrently.
- As an illustration, suppose that the value of the variable count is currently 5 and that the producer and consumer processes concurrently execute the statements “count++” and “count--”.
- Following the execution of these two statements, the value of the variable count may be 4, 5, or 6.
- The only correct result, though, is count == 5, which is generated correctly if the producer and consumer execute separately.
- The value of count may be incorrect as follows. **The statement “count++” may be implemented in machine language** as follows:

  ***register*1 = count**

  ***register*1 = *register*1 + 1**

  count = *register*1

where *register*1 is one of the local CPU registers.

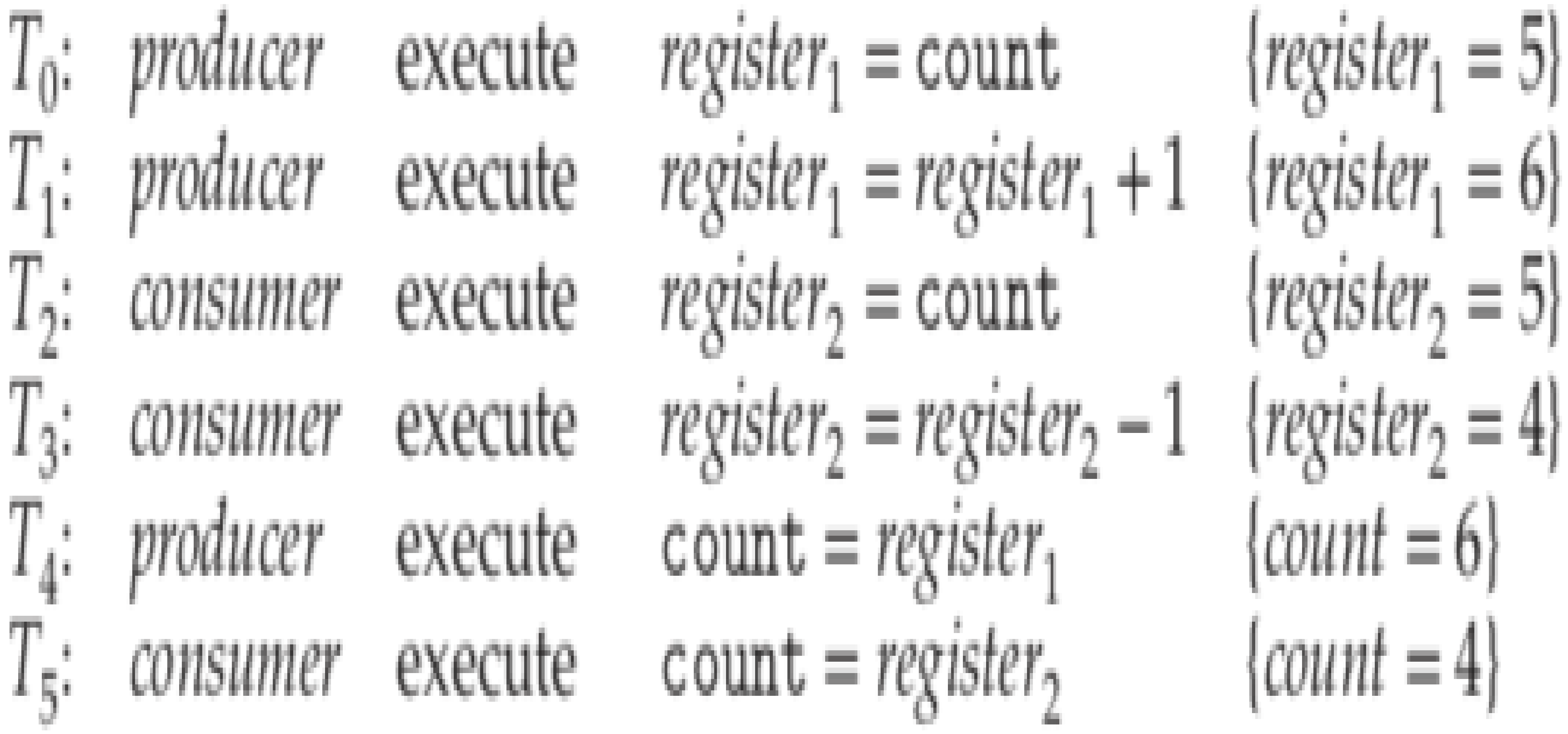
Similarly, the statement “count- -” is implemented as follows:

  ***register*2 = count**

  ***register*2 = *register*2 − 1**

  count = *register*2

The concurrent execution of “count++” and “count--” is equivalent to a sequential execution in which the lower-level statements presented are interleaved in some arbitrary order. One such interleaving is the following:



T_0:	*producer*	execute	$register_1 = \text{count}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \text{count}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\text{count} = register_1$	$\{count = 6\}$
T_5:	*consumer*	execute	$\text{count} = register_2$	$\{count = 4\}$

Race condition : Definition

Arrive the incorrect state because, both processes are allowed to manipulate the variable count concurrently.

A situation like this where,

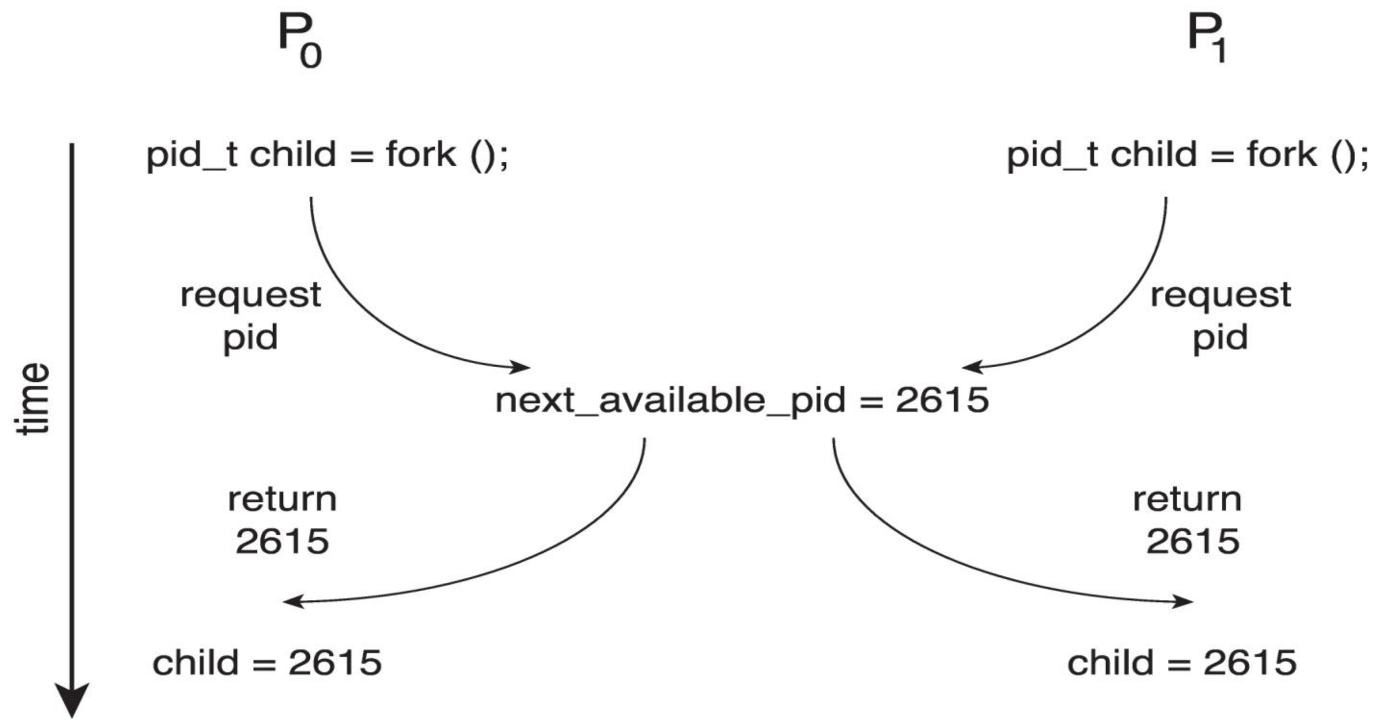
Several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition. Race conditions can result in corrupted values of shared data.

To **guard against the race condition** above, to **ensure that only one process at a time can be manipulating the variable** count.

To make such a guarantee, it **is require that the processes be synchronized** in some way.

Race Condition[Example]

- Processes P_0 and P_1 are creating child processes using the fork() system call
- Race condition on kernel variable next_available_pid which represents the next available process identifier (pid)



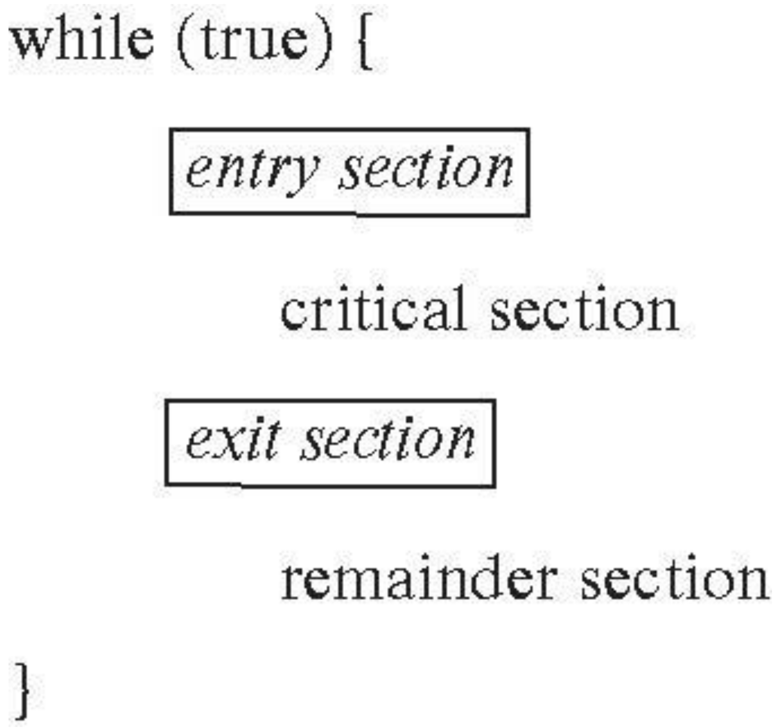
- Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable next_available_pid the same pid could be assigned to two different processes!

Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section
- The ***critical-section problem*** is **to design a protocol that the processes can use to synchronize their activity so as to cooperatively share data.**
- Each process must ask **permission to enter critical section in entry section[**The section of code implementing this request**]**, may follow critical section with **exit section**, then **remainder section** [remaining code]

Critical Section Problem (continued)

- General structure of process ***P***



```
while (true) {
        entry section
            critical section
        exit section
            remainder section
}
```

➢Each process must request to enter its **critical section.**

➢The section of code implementing this request is the **entry section.**

➢The critical section is followed by an **exit section.**

➢The remaining code is the **remainder section.**

Critical Section Problem (continued)

Requirements for solution to critical-section problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If **no process is executing in its critical section** and **some processes wish to enter their critical sections**, then only those **processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next**, and **this selection 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

Two general approaches are used to handle critical sections in operating systems: **preemptive kernels** and **nonpreemptive kernels**.

- ✓ A preemptive kernel allows a process to be preempted while it is running in kernel mode.
- ✓ A nonpreemptive kernel does not allow a process running in kernel mode to be preempted; a nonpreemptive kernel is essentially free from race conditions , as only one process is active in the kernel at a time.
- ✓ A preemptive kernel is more suitable for real-time programming, as it will allow a real-time process to preempt a process currently running in the kernel.
- ✓ Solutions are required for **preemptive kernels**

1. Interrupt-based Solution

- The **critical-section problem is solved simply in a single-core environment, if prevent interrupts while a shared variable was being modified.**
- So, the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable.

Entry section: disable interrupts

Exit section: enable interrupts

Will this solve the problem?

- What if the critical section is code that runs for an hour?
- Can some **processes starve** – never enter their critical section
- **Solution is not as feasible in a multiprocessor environment**. Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.

2. Peterson's Solution - Software Solution 1

- Peterson's solution is **restricted to two processes** that alternate execution between their critical sections and remainder sections. The processes are numbered *P*0 and *P*1. **For convenience, when presenting *Pi*, use *Pj* to denote the other process; that is, j equals 1 − i.**
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- Peterson's solution requires the **two processes to share two data items:**

int turn; [The variable **turn** indicates whose turn it is to enter the critical section]

boolean flag[2];

Boolean data type consists of True or false

- The **flag array** is used **to indicate if a process is ready to enter the critical section.**
  - **flag[i] = *true* implies that process P_i is ready**

Algorithm for Process P_i

```
while (true)
{
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j) ;
/* critical section */
    flag[i] = false;
/*remainder section */
}
```

Algorithm for Process P_j

```
while (true)
{
    flag[j] = true;
    turn = i;
  while (flag[i] && turn == i) ;
/* critical section */
  flag[j] = false;
/*remainder section */
}
```

It is a humble algorithm to give chance (ie)set turn value to other process

Provable that the three CS requirement are met:

1. Mutual exclusion is preserved, 2. Progress requirement is satisfied 3. Bounded-waiting requirement is met

Peterson’s Solution and Modern Architecture

- Although useful for demonstrating an algorithm, Peterson’s Solution is not guaranteed to work on modern architectures.
  - To **improve performance, processors and/or compilers may reorder operations that have no dependencies**
- Understanding why it will not work is useful for better understanding race conditions.
- For **single-threaded this is ok** as the **result will always be the same.**
- For **multithreaded the reordering may produce inconsistent or unexpected results.**

Modern Architecture Example

- **Two threads share the data:**

```
boolean flag = false;
 int x = 0;
```

- **Thread 1 performs**

```
while (!flag);
print x
```

- **Thread 2 performs**

```
x = 100;
flag = true
```

- **What is the expected output? 100**

➢ The expected behavior is, that Thread 1 outputs the value 100 for variable x.

➢ No data dependencies between the variables flag and x, it is possible that a processor may reorder the instructions for Thread 2 so that flag is assigned true before assignment of x = 100.

➢ In this situation, it is possible that Thread 1 would output 0 for variable x.

However, since the variables flag and x are independent of each other, the instructions:

```
    boolean flag = false;
      int x = 0;
Thread 1 performs
     while (!flag);
  print x
Thread 2 performs
     flag = true  ⎫
      x = 100;    ⎭  ← Reordered
```

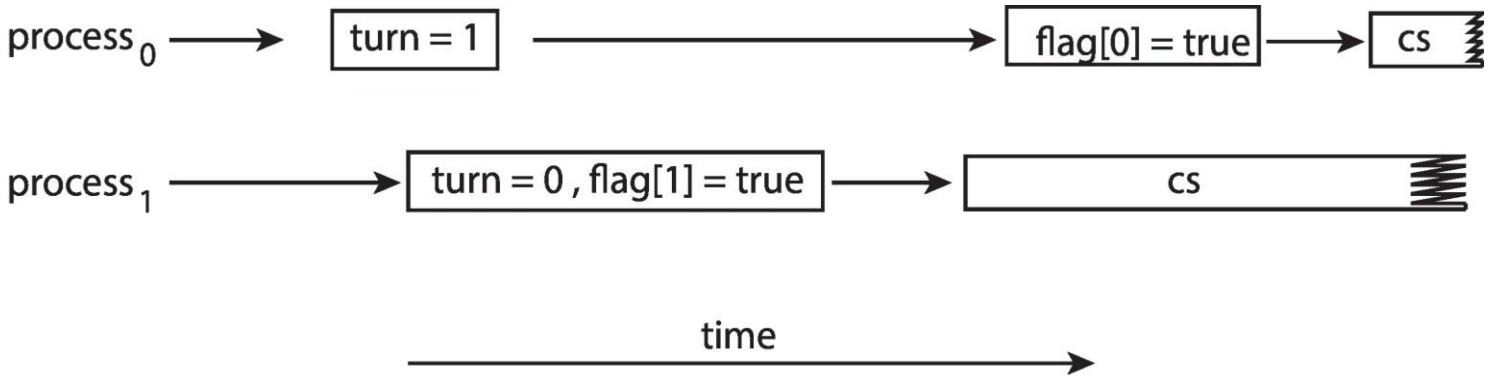
What is the expected output? 0

The Thread 2 may be reordered.
If this occurs, the output is 0

- Also, the processor may also reorder the statements issued by Thread 1 and load the variable x before loading the value of flag.
- If this were to occur, Thread 1 would output 0 for variable x even if the instructions issued by Thread 2 were not reordered.

Peterson's Solution Revisited

- The effects of instruction reordering in Peterson's Solution



- This allows both processes to be in their critical section at the same time!
- To ensure that Peterson's solution will work correctly on modern computer architecture then use **Memory Barrier**.

Algorithm for Process P_i

```
while (true)
{
turn = j;            1
flag[i] = true;  <-  After Pj
                     entering into CS
while (flag[j] && turn == j) ;
/* critical section */          False
   flag[i] = false;
/*remainder section */
}
```

Algorithm for Process P_j

```
while (true)
{
 turn = i;                 2
 flag[j] = true;           3
  while (flag[i] && turn == i) ;
/* critical section */          False
   flag[j] = false;
/*remainder section */
}
```

G1. Consider the methods used by processes P1 and P2 for accessing their critical sections whenever needed, as given below. The initial values of shared boolean variables S1 and S2 are randomly assigned.(Gate 2010)

Method Used by P1	**Method Used by P2**

while (S1 == S2) ;
Critica1 Section
S1 = S2;
	while (S1 != S2) ;
	Critica1 Section
	S2 = not (S1);

Which one of the following statements describes the properties achieved?

(A) Mutual exclusion but not progress

(B) Progress but not mutual exclusion

(C) Neither mutual exclusion nor progress

(D) Both mutual exclusion and progress

- ➢It can be easily observed that the Mutual Exclusion requirement is satisfied by the above solution, P1 can enter critical section only if S1 is not equal to S2, and P2 can enter critical section only if S1 is equal to S2.
- ➢But here **Progress Requirement is not satisfied.** Suppose when **s1=1 and s2=0** and process **p1 is not interested** to enter into critical section but p2 want to enter critical section. P2 is not able to enter critical section in this as only when p1 finishes execution, then only p2 can enter (then only s1 = s2 condition be satisfied).
- ➢Progress will not be satisfied when any process which is not interested to enter into the critical section will not allow other interested process to enter into the critical section.

G2. Two processes, P1 and P2, need to access a critical section of code. Consider the following synchronization construct used by the processes: Here, wants1 and wants2 are shared variables, which are initialized to false. Which one of the following statements is TRUE about the above construct?(Gate 2007)

Method Used by P1	**Method Used by P2**

while (true) {
wants1 = true;
while (wants2 == true);
/* Critical Section */
wants1=false; }
/* Remainder section */
}
	while (true) {
	wants2 = true;
	while (wants1==true);
	/* Critical Section */
	wants2 = false; }
	/* Remainder section */
	}

(A) It does not ensure mutual exclusion.
(B) It does not ensure bounded waiting.
(C) It requires that processes enter the critical section in strict alternation.
(D) It does not prevent deadlocks, but ensures mutual exclusion.

➢Two processes, P1 and P2, need to access a critical section of code. Here, wants1 and wants2 are shared variables, which are initialized to false.

➢Now, when both wants1 and wants2 become true, both process p1 and p2 enter in while loop and waiting for each other to finish. This while loop run indefinitely which leads to deadlock.

➢Now, Assume P1 is in critical section (it means wants1=true, wants2 can be anything, true or false). So this ensures that p2 won’t enter in critical section and vice versa. This satisfies the property of mutual exclusion.

G3. Consider three concurrent processes P1, P2 and P3 as shown below, which access a shared variable D that has been initialized to 100. (GATE 2019)

P1	P2	P3
.	.	.
.	.	.
.	.	.
.	.	.
D = D + 20	D = D - 50	D = D + 10
.	.	.
.	.	.
.		.

The process are executed on a uniprocessor system running a time-shared operating system. If the minimum and maximum possible values of D after the three processes have completed execution are X and Y respectively, then the value of Y–X is ___________.

(A) 80 **(B)** 130 **(C)** 50 **(D)** None of these

Minimum value (X) of D will possible when,
P2 reads D=100, preempted.
P1 executes D=D+20, D=120.
P3 executes D=D+10, D=130.
Now, P2 has D=100, executes, D = D-50 = 100-50 = 50. P2 writes D=50 final value.
So, minimum value (X) of D is 50.

Maximum value (Y) of D will possible when,
P1 reads D=100, preempted.
P2 reads D=100, executes, D = D-50 = 100-50 = 50.
Now, P1 executes, D = D+20 = 100+20 = 120.
And now, P3 reads D=120, executes D=D+10, D=130. P3 writes D=130 final value.
So, maximum value (Y) of D is 130.

Therefore, Y - X = 130 - 50 = 80

G4.Consider Peterson's algorithm for mutual exclusion between two concurrent processes i and j. The program executed by process is shown below. (GATE 2001)

```
Repeat
        flag[i]=true;
        turn=j;
        while (P) do no-op;
        Enter critical section,
        perform actions,
        then exit critical section
        Flag[i]=false;
        Perform other non-critical section actions.
Until false;
```

For the program to guarantee mutual exclusion, the predicate P in the while loop should be

(A) flag[j] = true and turn = i **(B) flag[j] = true and turn = j**

(C) flag[i] = true and turn = j (D) flag[i] = true and turn = i

Hardware Support for Synchronization

***Software-based* solution:** The algorithm[Peterson] have no special support from the operating system or specific hardware instructions to ensure mutual exclusion. So, it is called as software based solution. **Software-based solutions are not guaranteed to work on modern computer architectures.**

So, hardware support is required for synchronization. The three hardware instructions that provide support for solving the critical-section problem are

1. Memory Barriers
2. Hardware instructions [Test and Set , Compare And Swap(CAS)]
3. Atomic variables

1. Memory Barrier

- **Memory model** are the memory guarantees a computer architecture makes to application programs.
- Memory models may be either:
  - **Strongly ordered** – where a memory modification of one processor is immediately visible to all other processors.
  - **Weakly ordered** – where a memory modification of one processor may not be immediately visible to all other processors.
- A **memory barrier is an instruction** that forces any change in memory to be propagated (made visible) to all other processors.

Memory Barrier Instructions

- When a **memory barrier instruction is performed**, the **system ensures that all loads and stores are completed** before any subsequent load or store operations are performed.
- Therefore, **even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory** and visible to other processors before future load or store operations are performed.

Memory Barrier Example

- Add a memory barrier to the following instructions to ensure Thread 1 outputs 100:

```
boolean flag = false;
   int x = 0;
```

- Thread 1 now performs

```
    while (!flag);
      memory_barrier();
     print x
```

- Thread 2 now performs

```
    x = 100;
  memory_barrier();
   flag = true
```

Place a memory barrier between the first two assignment statements in the entry section to avoid the reordering of operations

- For Thread 1 , are guaranteed that that the value of flag is loaded before the value of x; For Thread 2 , ensure that the assignment to x occurs before the assignment flag.

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Three forms of hardware support:

1. Hardware instructions [Test and Set , Compare And Swap(CAS)]

2. Atomic variables

Hardware Instructions

- **Special hardware instructions** that allow us to **either *test-and-modify* the content of a word**, or to ***swap* the contents of two words atomically** (**uninterruptedly**)

  1. Test-and-Set instruction 2. **Compare-and-Swap** instruction

The test_and_set Instruction

- There is a **shared variable lock** which can take **either of the two values, 0 or 1.**
- Before entering into the critical section, a process **inquires about the lock**[ie **called as test**]
- If it is locked(lock=1) , it keeps on waiting till it becomes free.
- **If it is not locked(lock=0), it takes the lock and executes the critical section** (**At this moment, it set the lock as 1**)

Definition

```
boolean test_and_set (boolean *target)
        {
                boolean rv = *target;
                *target = true;
                return rv:
        }
```

Properties

- Executed atomically
- Returns the original value of passed parameter
- Set the new value of passed parameter to **true**

Solution Using test_and_set()

Shared boolean variable **lock**, initialized to **false**

Solution:

```
do {
       while (test_and_set(&lock)) ;
            /* do nothing */


                    /* critical section */


                lock = false;
            /* remainder section */
    } while (true);
```

locked(lock=1)
Not locked (lock=0)

locked(lock=1)
Not locked (lock=0)

Definition of Test and set

```
boolean test_and_set (boolean *target)
        {
                boolean rv = *target;
                *target = true;
                return rv:
        }
```

```
do {
    while (test_and_set(&lock)) ;
          /* do nothing */

                /* critical section */

             lock = false;
          /* remainder section */
       } while (true);
```

P2 →

P1 ←

```
do {
      while (test_and_set(&lock)) ;
          /* do nothing */

                /* critical section */

             lock = false;
          /* remainder section */
       } while (true);
```

Does it solve the critical-section problem properties?

1. Mutual exclusion is achieved   2. No bounded waiting

P3

P4

Compare_And_Swap (CAS) Instruction

- Definition

```
int compare_and_swap(int *value, int expected, int new_value)
 {
  int temp = *value;
   if (*value == expected)
      *value = new_value;
    return temp;
 }
```

swap

compare

- Properties
  - Executed atomically
  - Returns the original value of passed parameter **value**
  - Set the variable **value** the value of the passed parameter **new_value** but only if ***value == expected** is true. That is, the swap takes place only under this condition.

Solution using compare_and_swap

- Shared integer **lock** initialized to 0;
- Solution:

```
 while (true)
{
       while (compare_and_swap(&lock, 0, 1) != 0) ;
       /* do nothing */

       /* critical section */

       lock = 0;

    /* remainder section */
 }
```

- Does it solve the critical-section problem?

locked(lock=1)
Not locked (lock=0)

```
int compare_and_swap(int *value, int expected, int new_value)
    {
     int temp = *value;
      if (*value == expected)
          *value = new_value;
       return temp;
    }
```

P1

```
while (true)
{
 while (compare_and_swap(&lock, 0, 1) != 0) ;

          /* do nothing */

              /* critical section */

              lock = 0;

        /* remainder section */
 }
```

P2

```
while (true)
{
 while (compare_and_swap(&lock, 0, 1) != 0);

          /* do nothing */

              /* critical section */

              lock = 0;

        /* remainder section */
 }
```

Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools.
- One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and booleans.
- For example:
  - Let **sequence** be an atomic variable
  - Let **increment()** be operation on the atomic variable **sequence**
  - The Command:

    increment(&sequence);

    ensures **sequence** is incremented without interruption:

Atomic Variables

- The **increment()** function can be implemented as follows:

```
void increment(atomic_int *v)
{
      int temp;
      do {
            temp = *v;
      }
      while (temp != (compare_and_swap(v,temp,temp+1));
}
```

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**
  - **Boolean variable indicating if lock is available(1) or not(0)**
- Protect a critical section by
  - First **acquire()** a lock
  - Then **release()** the lock

The definition of acquire() is as follows:

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

The definition of release() is as follows:

```
release()
{
available = true;
}
```

Solution to CS Problem Using Mutex Locks

```
while (true) {

        acquire lock
          critical section

        release lock

remainder section
}
```

Calls to **acquire()** and **release()** must be **atomic**

Usually implemented via hardware atomic instructions such as compare-and-swap.

The main disadvantage is **busy waiting.**

- While a process is in critical section, any other process that tries to enter its critical section must loop continuously in the call to **acquire().**
- This continual looping is a problem in a multiprogramming system, where a single CPU core is shared among many processes.
- Busy waiting wastes CPU cycles that some other process might be able to use productively.

Semaphore

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

Semaphores were introduced by the Dutch computer scientist Dijkstra, such that , the **wait() operation was originally termed P (from the Dutch *proberen, "to test");***

signal() was originally called V (from verhogen, "to increment").

Definition of the **wait() operation**

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

In semaphore, 0- hold and 1 - free

Definition of the **signal() operation**

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

➢All modifications to the integer value of the semaphore in the wait() and signal() operations must be executed atomically.

➢That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

➢In addition, in the case of wait(S), the testing of the integer value of S ($S \leq 0$), as well as its possible modification (S--), must be executed without interruption.

Types of semaphore

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

Binary semaphore – integer value can range only between 0 and 1. On some systems, binary semaphores are known as **mutex lock,** as they are locks to provide mutual exclusion.

Semaphore Usage Example

- Solution to the CS Problem

Create a semaphore “**mutex**” initialized to 1

```
wait(mutex);
  CS
signal(mutex);
```

In semaphore, 0- hold and 1 - free

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

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

Semaphore Implementation

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

Advantage:

- implementation code is short

Dis advantage:

- **busy waiting** in critical section implementation. 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**. This type of semaphore is also called as **spinlock** because the process “spins” while wait for the lock.
- **Busy waiting wastes CPU cycles** that some other process might be able to use productively.

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** – It places the process invoking the operation into a waiting queue, and the state of the process is switched to the waiting state.
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue for execution

Counting semaphores

- **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 (thereby decrementing the count).**
- **When a process releases a resource, it performs a signal() operation (incrementing the count).**
- When the count for the semaphore goes to 0, all resources are being used.
- After that, processes that wish to use a resource will block until the count becomes greater than 0.

G5. A counting semaphore was initialized to 10. Then 6 P (wait) operations and 4 V (signal) operations were completed on this semaphore. The resulting value of the semaphore (GATE 1998)

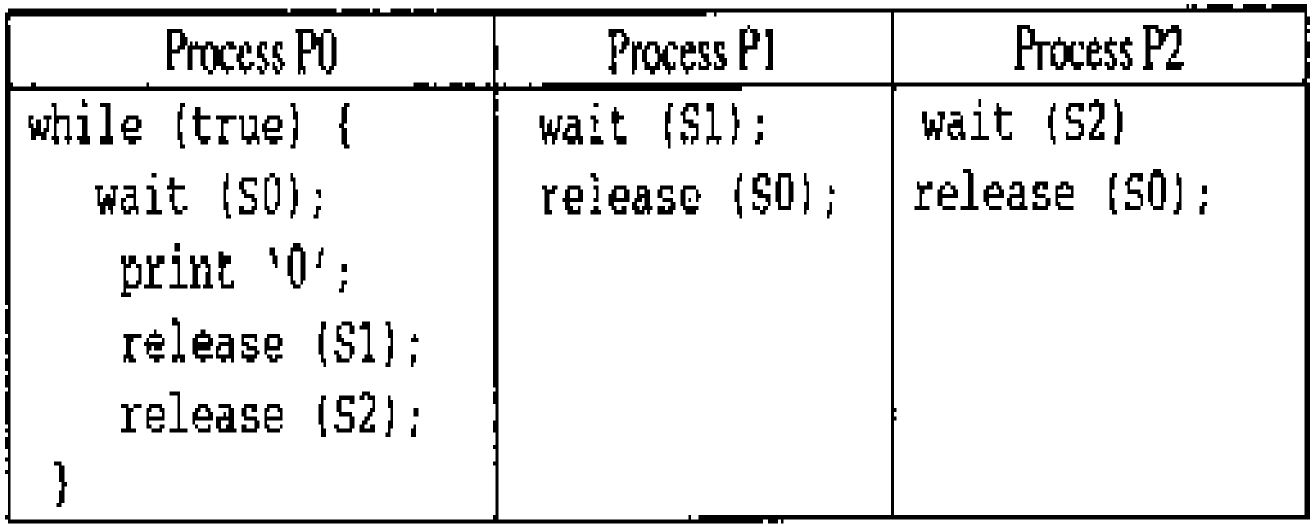
(A) 0 **(B)** 8 **(C)** 10 **(D)** 12

- Initially the semaphore value = 10
- Now , to perform 6 P operation means when perform one P operation it decreases the semaphore values to one.
- So after performing 6 P operation , semaphore values = 10 – 6 = 4
- Then perform 4 V operation means, when one V operation it increases the semaphore values to one. So after performing 4V operation , semaphore values = 4 + 4 = 8.

G6. At a particular time of computation the value of a counting semaphore is 7. Then 20 P operations and 15 V operations were completed on this semaphore. The resulting value of the semaphore is : (ISRO2015)
(A) 42 (B) 2 (C) 7 (D) 12

Value of a counting semaphore = 7
After 20 P operations value of semaphore = 7 – 20 = -13
After 15 V operations value of semaphore = -13 + 15 = 2

G7. The following program consists of 3 concurrent processes and 3 binary semaphores. The semaphores are initialized as S0 = 1, S1 = 0, S2 = 0. (GATE 2010, ISRO 2019)



Process P0	Process P1	Process P2

`while (true) {`
`   wait (S0);`
`    print '0';`
`    release (S1);`
`    release (S2);`
` }`
	`wait (S1);`
	`release (S0);`
		`wait (S2)`
		`release (S0);`

How many times will process P0 print '0'?

(A) At least twice

(B) Exactly twice

(C) Exactly thrice

(D) Exactly once

Initially only P0 can go inside the while loop as S0 = 1, S1 = 0, S2 = 0.

Minimum no. of time 0 printed is twice when execute in this order (p0 -> p1 -> p2 -> p0)

Maximum no. of time 0 printed is thrice when execute in this order (p0 -> p1 -> p0 -> p2 -> p0).

In semaphore, 0- hold and 1 - free

Process	**Sema phore**	**Initializ ation**	**Execution**	**Sema phore values**	**Comment**
P0	**S0**	**1**	**P0 executes, it print 0; Now S0 value is 1.It release S1 and S2 , so the values of S1 and S2 changed as 1**	**0**	**Now, both P1 and P2 have semaphore values as 1; If P1 is executing it change S0 as 1; And P1 complete its operation**
P1	**S1**	**0**	**1**	**1**	
P2	**S2**	**0**	**1**	**1**	

Process	Semaphore	Semaphore values	Execution	Semaphore values	Comment
P0	S0	1	Now, both P0 and P2 have semaphore values as 1; If P2 is executing it change S0 as 1(already S0 as 1 because of the Process P1); And P2 complete its operation	1	Now P0 is executes, it print 0; So, two times 0 is printed
P1	S1	0		0	
P2	S2	1		0	

The above are shown for the execution order p0 -> p1 -> p2 -> p0

Process	Semaphore	Semaphore values	Comment	Semaphore values	Comment	Semaphore values	Comment
P0	S0	1	Now, both P0 and P2 have semaphore values as 1; If P1 is executing it print 0; So far P2 is not executed	0	Now P2 is executes, it set S0 as 1 and P2 completes execution	1	Now P0 is executes, it print 0; So, three times 0 is printed
P1	S1	0		0		0	
P2	S2	1		1		0	

The above are shown for the execution order p0 -> p1 -> p0 -> p2 -> p0

G8.The following two functions P1 and P2 that share a variable B with an initial value of 2 execute concurrently. (GATE 2015)

```
P1()
{
   C = B – 1;
   B = 2*C;
}
P2()
{
   D = 2 * B;
   B = D - 1;
}
```

The number of distinct values that B can possibly take after the execution is (A)3 (B)4 (C)5 (D)6

There is possibility for executing in any one of the 4 ways.

1. execute P2 process after P1 process 2.execute P1 process after P2 process 3. Pre-empting P1 and executing P2 processes 4. Pre-empting P2 and executing P1 processes

1. Execute P2 process after P1 process, then B = 3
2. Execute P1 process after P2 process, then B = 4
3. Pre-empting P1 and executing P2 processes B=3
4. Pre-empting P2 and executing P1 processes B=2

   So, total no. of distinct values that B can possibly take after the execution is 3.

G9. Consider a non-negative counting semaphore S. The operation P(S) decrements S, and V(S) increments S. During an execution, 20 P(S) operations and 12 V(S) operations are issued in some order. The largest initial value of S for which at least one P(S) operation will remain blocked is _________. (GATE 2016)

(A) 7 **(B)** 8 **(C)** 9 **(D)**10

P(S) – V(S) = 1+X

20- 12 =1 +X $\rightarrow$ 7

Monitors

- A **high level abstraction that provides a convenient and effective mechanism for process synchronization.**
- A monitor type presents a set of **programmer defined operations that provide mutual exclusion within the monitor.**
- The monitor also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables.
- A procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.
- Similarly, the local variables of a monitor can be accessed by only the local procedures.

Pseudocode syntax of a monitor

```
monitor monitor-name
        {

 // shared variable declarations

           procedure P1 (…)
           {
               ….
           }
           procedure P2 (…)
           {
            ….
           }
           procedure Pn (…)
           {
           ……
           }
           initialization code (…)
           { … }
           }
```

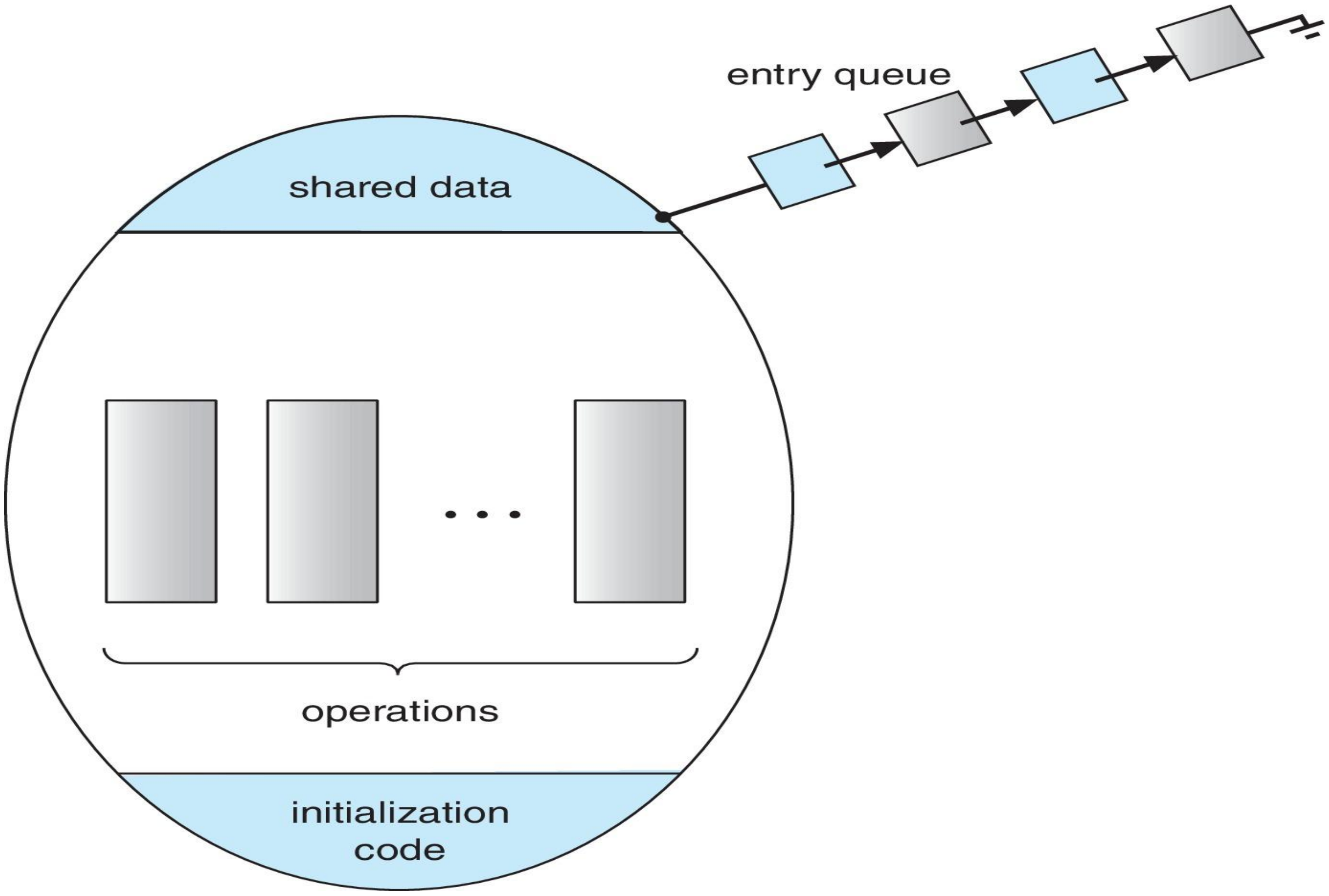
- The monitor construct ensures that only one process at a time can be active within the monitor

Condition Construct: Condition x,y;

- **The operation x.wait(); means that the process invoking this operation is suspended until another process invokes x.signal();**

- **The x.signal() operation resumes exactly one suspended process.**

Schematic view of a Monitor



Condition Variables[Shared Variables]

- **condition x, y;**
- Two operations are allowed on a condition variable:
  - **x.wait()** – a process that invokes the operation is suspended until **x.signal()**
  - **x.signal()** – resumes one of processes (if any) that invoked **x.wait()**

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

➢If the suspended process Q is allowed to resume its execution, the signaling process P must wait.

➢Otherwise, both P and Q would be active simultaneously within the monitor.

➢Conceptually both processes can continue with their execution.

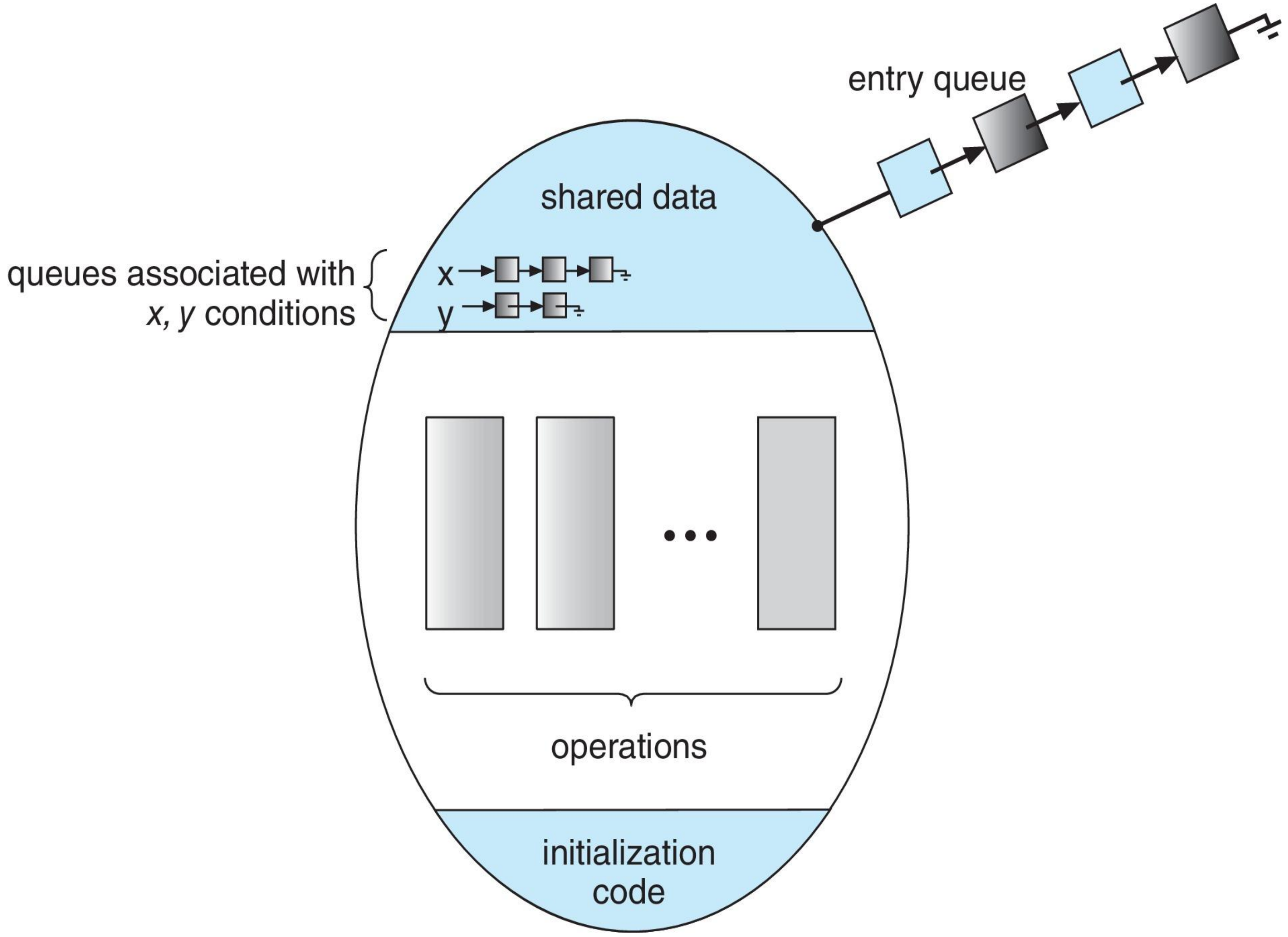
Two possibilities exist:

1.Signal and wait: P either waits until Q leaves the monitor

2. Signal and continue: Q either waits until P leaves the monitor

- **There are reasonable arguments in favor of adopting either option.**
- **On the one hand, since P was already executing in the monitor, the signal-and continue method seems more reasonable.**
- **On the other, if allow thread P to continue, then by the time Q is resumed, the logical condition for which Q was waiting may no longer hold.**
- **A compromise between these two choices exists as well: when thread P executes the signal operation, it immediately leaves the monitor. Hence, Q is immediately resumed.**
- **Many programming languages have incorporated the idea of the monitor , including Java and C#.**

Monitor with Condition Variables



Usage of Condition Variable Example

- Consider P_1 and P_2 that that need to execute two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
  - Create a monitor with two procedures F_1 and F_2 that are invoked by P_1 and P_2 respectively
  - One condition variable “x” and One Boolean variable “done”
  - **F1:**

```
    S₁;
    done = true;
    x.signal();
```

  - **F2:**

```
     S₂;
    done = false;
      x.wait();
```

Resuming Processes within a Monitor

- If several processes queued on condition variable **x**, and **x.signal()** is executed, which process should be resumed?
- FCFS frequently not adequate
- Use the **conditional-wait** construct of the form

  x.wait(c)

where:

- **c** is an integer (called the priority number)
- The process with lowest number (highest priority) is scheduled next

Deadlocks
System Model

- System consists of resources
- **Resource types R_1, R_2, . . ., R_m**
  - ***CPU cycles, memory space, I/O devices(Printers,DVD), Files***
  - Each resource type R_i has W_i instances
- Several processes may compete for a finite number of resources
- A process requests resources; and if the resources are not available at that time, the process enters a waiting state.
- Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called deadlock.

Each process utilizes a resource as follows:

1. **Request :** Process can request the resource. If the request cannot be granted immediately(if the resource is being used by another process), then the requesting process must wait until it can acquire the resource
2. **Use :** The process can operate on the resource
3. **Release:** The process release the resource

Deadlock with Semaphores

- Data:
  - A semaphore $\mathbf{S_1}$ initialized to 1
  - A semaphore $\mathbf{S_2}$ initialized to 1
- Two threads T_1 and T_2
- $\boldsymbol{T_1}$**:**

wait(s_1)

wait(s_2)

- $\boldsymbol{T_2}$**:**

wait(s_2)

wait(s_1)

Deadlock Characterization

Deadlock can arise if **four conditions hold simultaneously**.

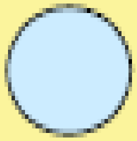
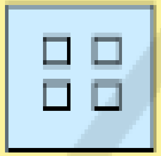
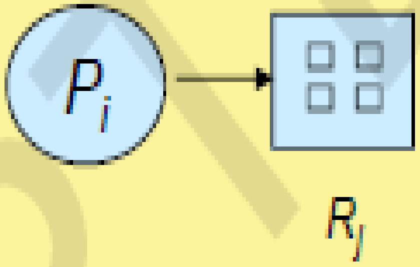
- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{T_0, T_1, \ldots, T_n\}$ of waiting processes such that T_0 is waiting for a resource that is held by T_1, T_1 is waiting for a resource that is held by T_2, …, T_{n-1} is waiting for a resource that is held by T_n, and T_n is waiting for a resource that is held by T_0.

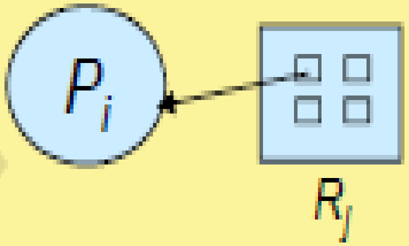
Resource-Allocation Graph

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

- V is partitioned into two types:
  - $T = \{T_1, T_2, \ldots, T_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $T_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow T_i$

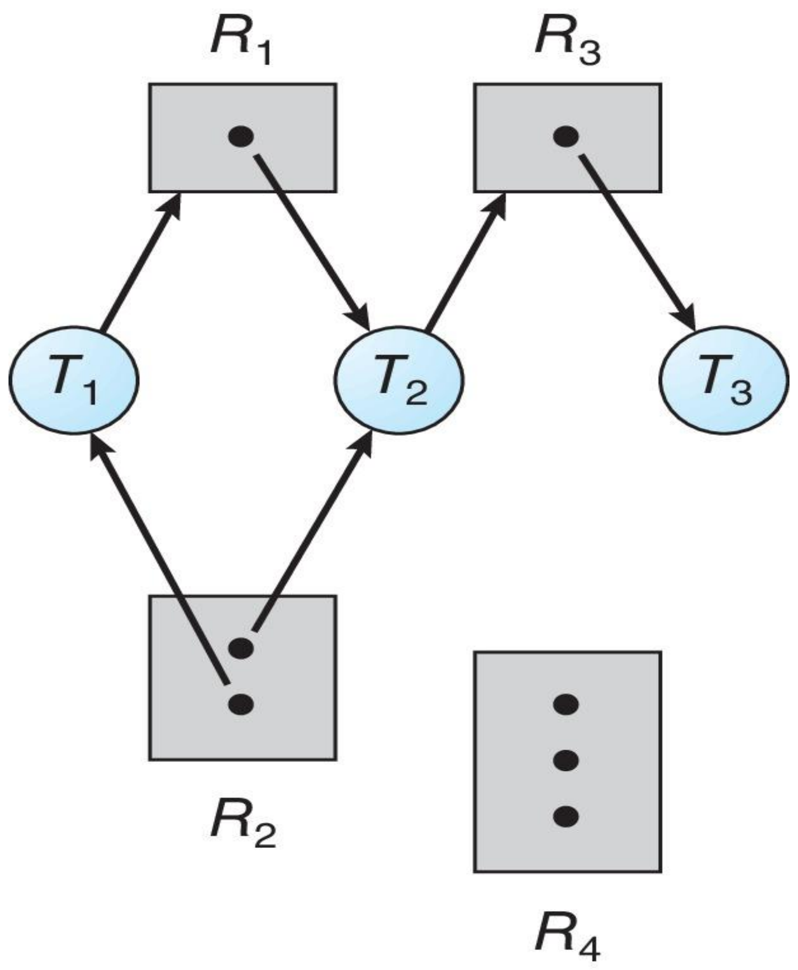
Resource-Allocation Graph (Cont.)

- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j 

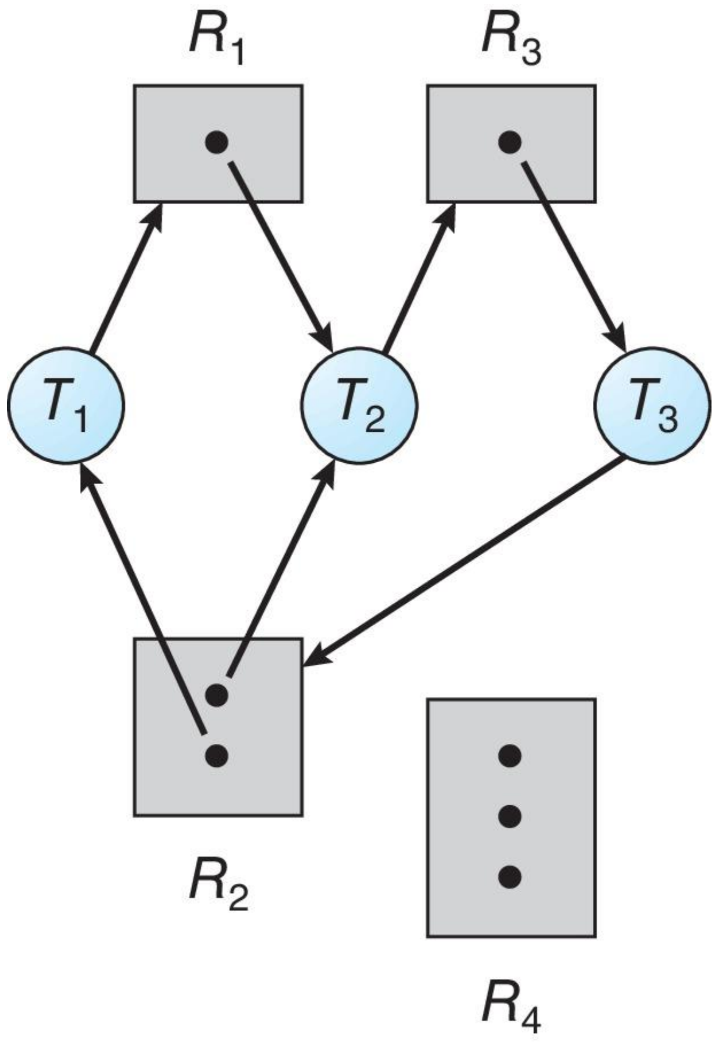
- P_i is holding an instance of R_j 

Resource Allocation Graph Example

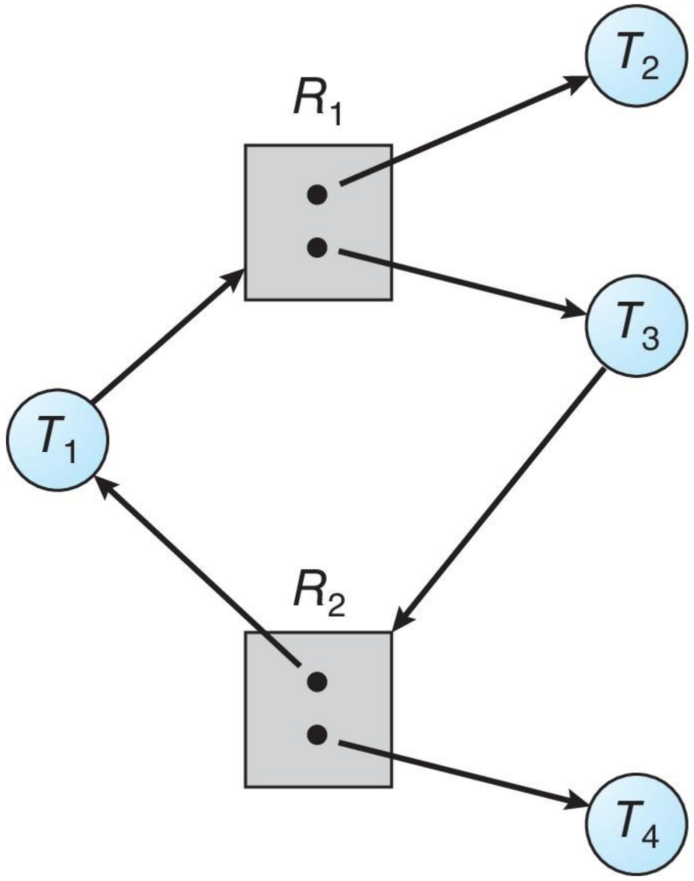
- One instance of R_1
- Two instances of R_2
- One instance of R_3
- Three instance of R_4
- T_1 holds one instance of R_2 and is waiting for an instance of R_1
- T_2 holds one instance of R_1, one instance of R_2, and is waiting for an instance of R_3
- T_3 is holds one instance of R_3



Resource Allocation Graph with a Deadlock



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

1. Ensure that the system will **never** enter a deadlock state:
   i. Deadlock prevention
   ii. Deadlock avoidance
2. Allow the system to enter a deadlock state and then recover
3. Ignore the problem and pretend(To give a false appearance of) that deadlocks never occur in the system.

Deadlock Prevention

Ensure that at least one of the necessary conditions for deadlocks does not hold.

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a thread requests a resource, it does not hold any other resources – So, hold and wait is not possible.

Two protocols:

1. **Require threads to request and be allocated all its resources before it begins execution**
2. allow **thread to request resources** only **when the thread has none allocated to it.**

- Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

- **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 thread is waiting
  - **Thread 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 thread requests resources in an increasing order** of enumeration

Circular Wait

- Invalidating the circular wait condition is most common.
- Simply assign each resource a unique number.
- Resources must be acquired in order.
- Thread can request an instance of resource *Rj if and only if*

    F(Rj) > F(Ri).

- [Ex]:A **thread that wants to use both first mutex and second mutex** at the same time must first request first mutex and then second mutex.

first_mutex = 1

second_mutex = 5

Deadlock Avoidance

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

- Simplest and most useful model requires that each thread declare the ***maximum number*** of resources of each type that it may need. Given this a **priori information**, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State

- A **state is *safe if the system can allocate resources to each thread (up to its*** **maximum) in some order** and **still avoid a deadlock.** A **system is in a safe state only if there exists a safe sequence.**
- System is in **safe state** if there exists a sequence $<T_1, T_2, \ldots, T_n>$ of all the threads in the systems **such that for each T_i, the resources that T_i can still request can be satisfied by currently available resources plus resources held by all the T_j** .
- That is:
  - If T_i resource needs are not immediately available, then T_i can wait until all T_j have finished
  - When T_j is finished, T_i can obtain needed resources, execute, return allocated resources, and terminate
  - When T_i terminates, T_{i+1} can obtain its needed resources, and so on

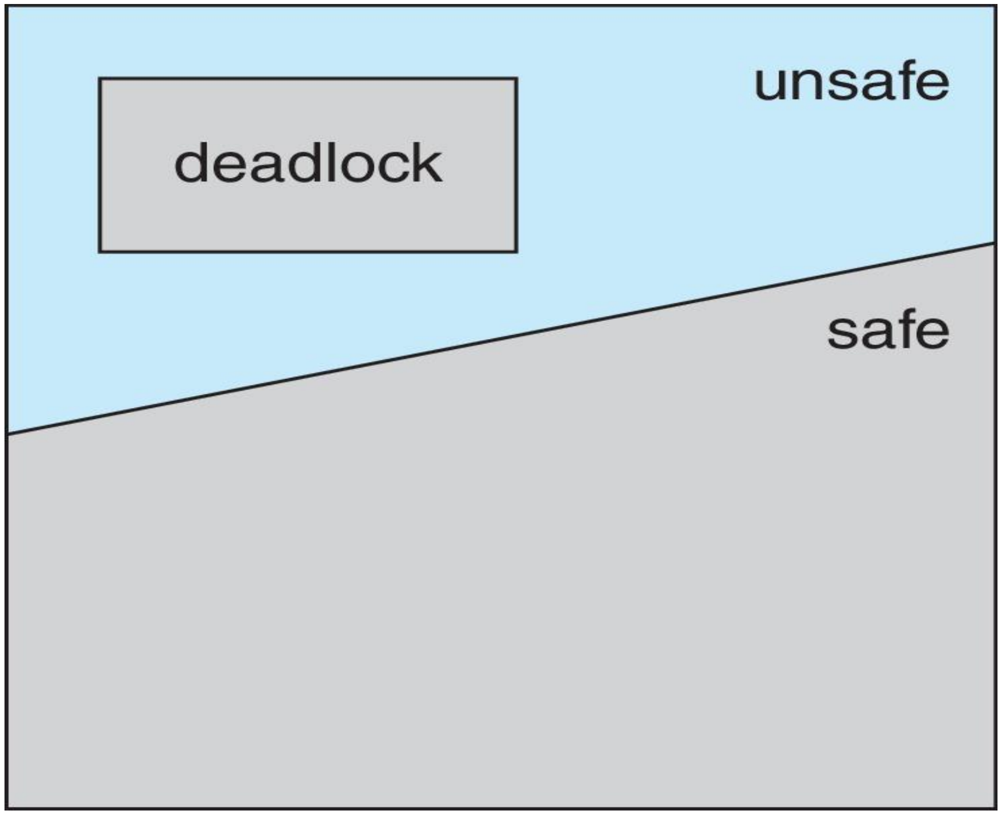
Example: In a system with one tape drive and one printer , the system might need to know **that process P will request first the tape drive and then the printer** before releasing both resources, whereas **process Q will request first the printer and then the tape drive.** With this knowledge of the complete sequence of requests and releases for each process, the **system can decide for each request** whether or not the process should wait in order to **avoid a possible future deadlock.**

Each request requires that in making this decision by the system,

1. **consider the resources currently available**
2. **the resources currently allocated to each process and**
3. **the future requests and releases of each process**

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.



[Ex]:Consider a system with **twelve resources** and three threads:T0, T1 andT2. Thread T0 requires ten resources, thread T1 requires four, and thread T2 requires nine resources.

Suppose that, at time t0, thread T0 is holding five resources, thread T1 is holding two resources, and thread T2 is holding two resources. (**Thus, there are three free resources.)** **Available:3**

T ID	Maximum Needs	Allotted	Current Needs
T0	10	5	5
T1	4	2	2
T2	9	2	7

At time t0, the system is in a safe state. The sequence < T1, T0, T2> satisfies the safety condition.

Thread T1 can immediately be allocated all its resources and then return them (the system will then have five available resources); then thread T0 can get all its resources and return them (the system will then have ten available resources); and finally thread T2 can get all its resources and return them (the system will then have all twelve resources available).

A system can go from a safe state to an unsafe state. Suppose that**, at time *t1*, thread T2 requests and is allocated one more resource.** *The system is no longer* in a safe state. At this point, only thread *T1 can be allocated all its resources.* When **it returns them, the system will have only four available resources.** Since thread *T0 is allocated five resources but has a maximum of ten, it may* request five more resources. If it does so, it will have to wait, because they are unavailable. Similarly, thread *T2 may request six additional resources and have* to wait, resulting in a deadlock.

T ID	**Max Need**	**Allotted**	**Current Needs**
T0	10	5	5
T1	4	2	2
T2	9	3	6

Available = 2 Resources

At time t1, the system is in a unsafe state. The mistake was in granting the request from thread *T2 for one more resource. If T2 had made wait until either of the other* threads had finished and released its resources, then the deadlock is avoided.

Avoidance Algorithms

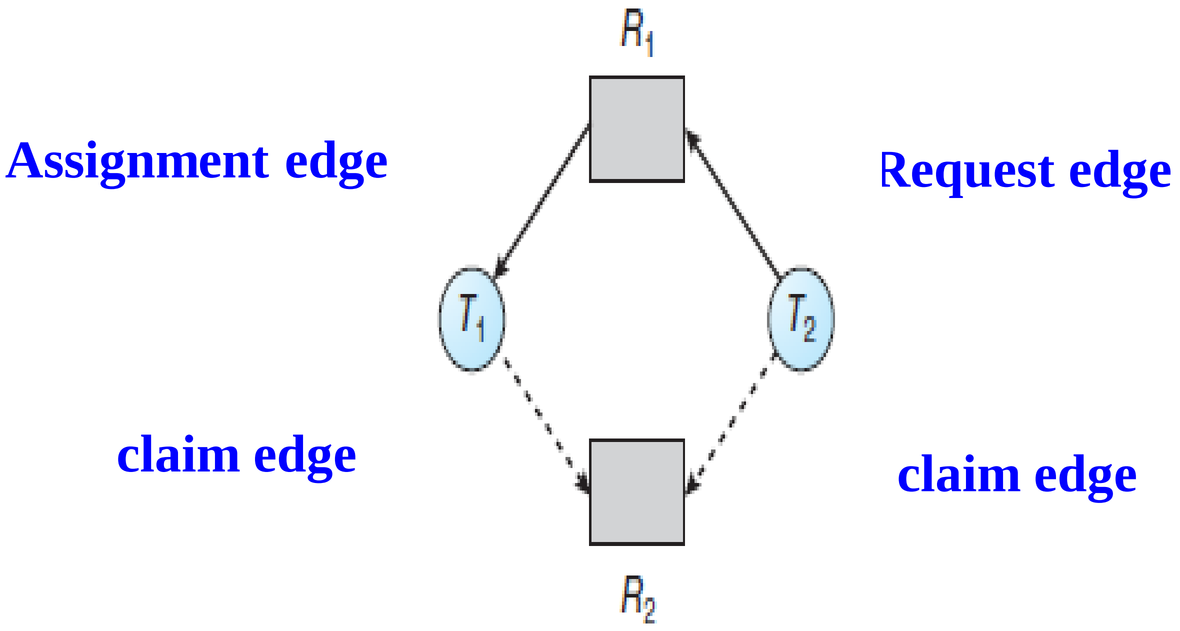
- Single instance of a resource type
  - Use a resource-allocation graph

- Multiple instances of a resource type
  - Use the Banker’s Algorithm

Resource-Allocation Graph Scheme

- **Claim edge** $T_i \rightarrow R_j$ indicated that process T_j may request resource R_j; represented by a dashed line
- **Claim edge converts to request edge when a thread requests a resource; Request edge converted to an assignment edge when the resource is allocated to the thread; When a resource is released by a thread, assignment edge reconverts to a claim edge**

Resource-Allocation Graph



- ➢Resources must be claimed a priori in the system. That is, before thread Ti starts executing, all its claim edges must already appear in the resource-allocation graph.

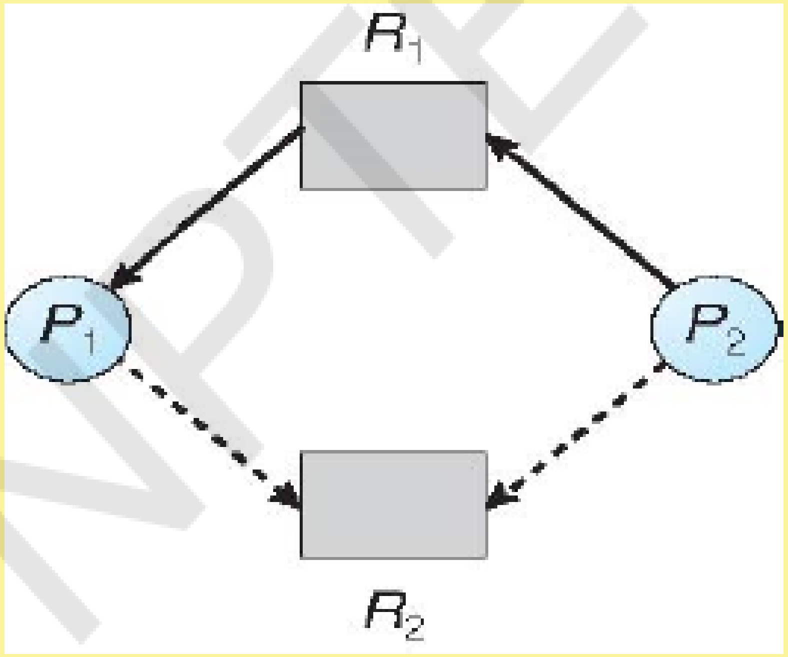
- ➢suppose that thread Ti requests resource Rj. The request can be granted only if converting the request edge Ti → Rj to an assignment edge Rj → Ti does not result in the formation of a cycle in the resource-allocation graph.

- ➢**If no cycle exists, then the allocation of the resource will leave the system in a safe state**. **If a cycle is found, then the allocation will put the system in an unsafe state.** In that case, thread Ti will have to wait for its requests to be satisfied.

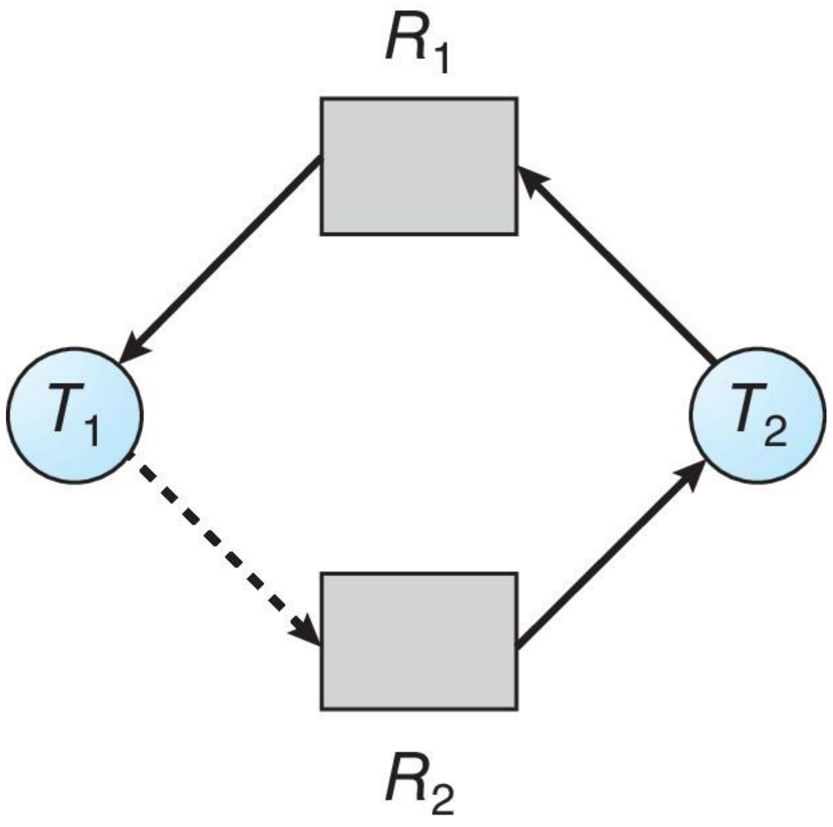
Safe State In Resource-Allocation Graph

Resource-Allocation Graph with claim edges

- P1 is holding resource R1 and has a claim on R2
- P2 is requesting R1 and has a claim on R2
- No cycle. So system is in a safe state.



Unsafe State In Resource-Allocation Graph



- Suppose that T1 requests R2, it cannot allocate to T1, since this action will create a cycle in the graph .
- A cycle indicates that the system is in an unsafe state. If T1 requests R2, and T2 requests R1, then a deadlock will occur.
- The **request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph**

Banker’s Algorithm

- Multiple instances of resources
- Each thread must a priori claim maximum use
- When a thread requests a resource, it may have to wait
- When a thread gets all its resources it must return them in a finite amount of time
- **Banker’s algorithm is a combination of safety algorithm and Resource request algorithm**.
- Safety algorithm for finding out whether or not a system is in a safe state.
- Resource-Request algorithm for determining whether requests can be safely granted.

Data Structures for the Banker's Algorithm

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

- **Available**: A one dimensional array of size m. If available $[j] = k$, there are k instances of resource type R_j available
- **Max**: *A two* dimensional array of size $n \times m$ matrix that specifies the maximum demand of each process in a system. If *Max* $[i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation**: *A two* dimensional array of size $n \times m$ that specifies the number of resources of each type currently allocated to each process. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need**: *A two* dimensional array of size $n \times m$ that specifies the remaining resource needs of each process. 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 where ***m* = number of resources and *n* = number of processes**.

   Initialize:

   Work = Available [Number of resources are available]

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

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

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

   (b) $\boldsymbol{Need_i \leq Work}$

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

3. $\boldsymbol{Work = Work + Allocation_i}$

   Finish*[*i*] = *true

   go to step 2

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

Resource-Request Algorithm for Process P_i

Request$_i$ = request vector for process $\boldsymbol{P_i}$. If ***Request$_i$* [*j*] = *k*** then process $\boldsymbol{T_i}$ wants ***k*** instances of resource type $\boldsymbol{R_j}$

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

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If the resulting resource-allocation state is safe, the resources are allocated to $\mathbf{P}_i$. If unsafe $\Rightarrow$ $\boldsymbol{P_i}$ must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm- Safety 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:

Process	**Allocation**			**Max**			**Available**			**Need**		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
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	**2**	**1**	**1**	**2**	**2**	**2**				**0**	**1**	**1**
P4	**0**	**0**	**2**	**4**	**3**	**3**				**4**	**3**	**1**

Step 1	m=3 , n=5				
Work = Available					
Work =	3		3		2
Process	0	1	2	3	4
Finish=	false	false	false	false	false

Step 2	P0 for i=0		
Need0=	7	4	3
7,4,3 > 3, 3, 2			
Finish [0] = false and Need0 > Work			
P0 must wait			

Step 2	P1 for i=1		
Need1=	1	2	2
1,2,2 < 3, 3, 2			
Finish [1] = false and Need1 <Work			
P1 can be kept in safe sequence			

Step 3	P1				
Work=	Work + Allocation				
	3,3,2 + 2, 0, 0				
	5		3		2
Process	0	1	2	3	4
Finish=	false	True	false	false	false

Step 2	P2 for i=2		
Need2=	6	0	0
6,0,0 > 5, 3, 2			
Finish [2] = false and Need2 > Work			
P2 must wait			

Step 2	P3 for i=3		
Need3=	0	1	1
0,1,1 < 5, 3, 2			
Finish [3] = false and Need3 <Work			
P3 can be kept in safe sequence			

Step 3	P3				
Work=	Work + Allocation				
	5,3,2 + 2, 1, 1				
	7		4		3
Process	0	1	2	3	4
Finish=	false	True	false	True	false

Step 2	P4 for i=4		
Need4=	4	3	1
4,3,1< 7,4,3			
Finish [4] = false and Need4 <Work			
P4 can be kept in safe sequence			

Step 3	P4				
Work=	Work + Allocation				
	7,4,3 + 0,0,2				
	7		4		5
Process	0	1	2	3	4
Finish=	false	True	false	True	True

Step 2	P0 for i=0		
Need0=	7	4	3
7,4,3 < 7,4,5			
Finish [0] = false and Need0 < Work			
P0 can be kept in safe sequence			

Step 3	P0				
Work=	Work + Allocation				
	7,4,5 + 0,1,0				
	7	5	5		
Process	0	1	2	3	4
Finish=	True	True	false	True	True

Step 2	P2 for i=2		
Need2=	6	0	0
6,0,0 < 7,5,5			
Finish [0] = false and Need2 < Work			
P2 can be kept in safe sequence			

Step 3	P2				
Work=	Work + Allocation				
	7,5,5 + 3,0,2				
	10		5		7
Process	0	1	2	3	4
Finish=	True	True	True	True	True

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

Example of Banker's Algorithm- Resource-Request Algorithm

Suppose now that Process P1 requests one additional instance of resource type A and two instances of resource type C, so Request1 = (1,0,2). Can this request be immediately granted?

Process	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
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	2	1	1	2	2	2				0	1	1
P4	0	0	2	4	3	3				4	3	1

Step 1	Check condition
1,0,2 <= 1,2,2	
Request1 <= Need1	
Condition satisfied	

Step 2	Check condition
1,0,2 <= 3,3,2	
Request1 <= Available	
Condition satisfied	

$$\textit{Available} = \textit{Available} - \textit{Request}_1;$$
$$\textit{Allocation}_1 = \textit{Allocation}_1 + \textit{Request}_1;$$
$$\textit{Need}_1 = \textit{Need}_1 - \textit{Request}_1;$$

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

Determine whether this new system state is safe or not. For that, execute safety algorithm again and find the system state is safe or not. If it is safe, then that sequence is safe sequence.

Step 1	m=3 , n=5				
Work = Available					
Work =	2		3		0
Process	0	1	2	3	4
Finish=	false	false	false	false	false

Step 2	P0 for i=0		
Need0=	7	4	3
7,4,3 > 2, 3, 0			
Finish [0] = false and Need0 > Work			
P0 must wait			

Step 2	P1 for i=1		
Need1=	0	2	0
0,2,2 < 3, 3, 2			
Finish [1] = false and Need1 <Work			
P1 can be kept in safe sequence			

Step 3	P1				
Work=	Work + Allocation				
	2,3,0 + 3, 0, 2				
	5		3		2
Process	0	1	2	3	4
Finish=	false	True	false	false	false

Step 2	P2 for i=2		
Need2=	6	0	0
6,0,0 > 5, 3, 2			
Finish [2] = false and Need2 > Work			
P2 must wait			

Step 2	P3 for i=3		
Need3=	0	1	1
0,1,1 < 5, 3, 2			
Finish [3] = false and Need3 <Work			
P3 can be kept in safe sequence			

Step 3	P3				
Work=	Work + Allocation				
	5,3,2 + 2, 1, 1				
	7		4		3
Process	0	1	2	3	4
Finish=	false	True	false	True	false

Step 2	P4	for i=4	
Need4=	4	3	1
4,3,1< 7,4,3			
Finish [4] = false and Need4 <Work			
P4 can be kept in safe sequence			

Step 3	P4				
Work=	Work + Allocation				
	7,4,3 + 0,0,2				
	7		4		5
Process	0	1	2	3	4
Finish=	false	True	false	True	True

Step 2	P0 for i=0		
Need0=	7	4	3
7,4,3 < 7,4,5			
Finish [0] = false and Need0 < Work			
P0 can be kept in safe sequence			

Step 3	P0				
Work=	Work + Allocation				
	7,4,5 + 0,1,0				
	7	5	5		
Process	0	1	2	3	4
Finish=	True	True	false	True	True

Step 2	P2	for i=2	
Need2=	6	0	0
6,0,0 < 7,5,5			
Finish [0] = false and Need2 < Work			
P2 can be kept in safe sequence			

So by applying the Resource- Request algorithm and by checking the state of the system using the safety algorithm, find that granting the request of process P1 still keeps the system in a safe state and hence will not lead to a deadlock.

Step 3	P2				
Work=	Work + Allocation				
	7,5,5 + 3,0,2				
	10	5	7		
Process	0	1	2	3	4
Finish=	True	True	True	True	True

Now, Finish[i] = true for all i. So the system is in a safe state since the sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety criteria.

Book Exercise Problems

1. Consider the following snapshot of a system:

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
T0	*0 0 1 2*	*0 0 1 2*	*1 5 2 0*
T1	*1 0 0 0*	*1 7 5 0*	
T2	*1 3 5 4*	*2 3 5 6*	
T3	*0 6 3 2*	*0 6 5 2*	
T4	*0 0 1 4*	*0 6 5 6*	

Answer the following questions using the banker's algorithm:

a. What is the content of the matrix *Need?*

b. Is the system in a safe state?

c. If a request from thread *T1 arrives for (0,4,2,0), can the request be* granted immediately?

a. The values of *Need for processes P0 through P4, respectively, are (0,*0, 0, 0), (0, 7, 5, 0), (1, 0, 0, 2), (0, 0, 2, 0), and (0, 6, 4, 2).

b. The system is in a safe state. With *Available equal to (1, 5, 2, 0),*

either process *P0 or P3 could run*. *Once process P3 runs, it releases*

its resources, which allows all other existing processes to run.

c. The request can be granted immediately. The value of *Available is*

then (1, 1, 0, 0). One ordering of processes that can finish is *P0*, *P2,*

P3*, *P1*, *and P4.

2. Consider the following snapshot of a system:

	Allocation	*Max*
	A B C D	*A B C D*
T0	*3 0 1 4*	*5 1 1 7*
T1	*2 2 1 0*	*3 2 1 1*
T2	*3 1 2 1*	*3 3 2 1*
T3	*0 5 1 0*	*4 6 1 2*
T4	*4 2 1 2*	*6 3 2 5*

Using the banker's algorithm, determine whether or not each of the following states is unsafe. If the state is safe, illustrate the order in which the threads may complete. Otherwise, illustrate why the state is unsafe.

a. Available = (0, 3, 0, 1)

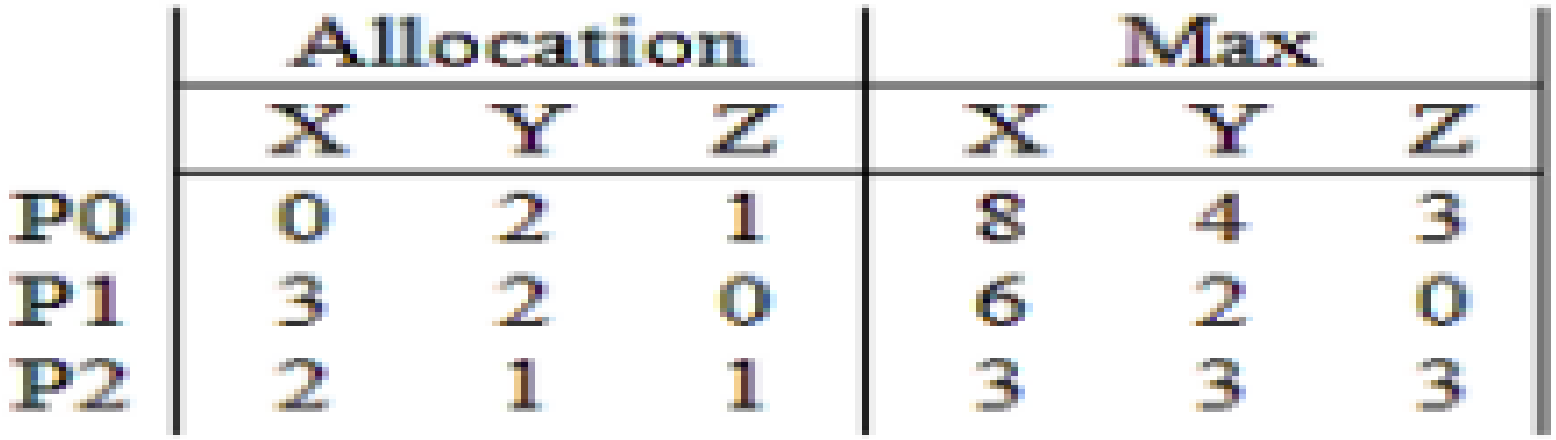
b. Available = (1, 0, 0, 2)

a. Not safe. Processes *P2, P1, and P3 are able to finish, but no remaining* processes can finish.

b. Safe. Processes *P1, P2, and P3 are able to finish. Following this*, processes *P0 and P4 are also able to finish.*

G10. An operating system uses the Banker's algorithm for deadlock avoidance when managing the allocation of three resource types X, Y, and Z to three processes P0, P1, and P2. The table given below presents the current system state. Here, the Allocation matrix shows the current number of resources of each type allocated to each process and the Max matrix shows the maximum number of resources of each type required by each process during its execution.

There are 3 units of type X, 2 units of type Y and 2 units of type Z still available. The system is currently in a safe state. Consider the following independent requests for additional resources in the current state: (GATE 2014)

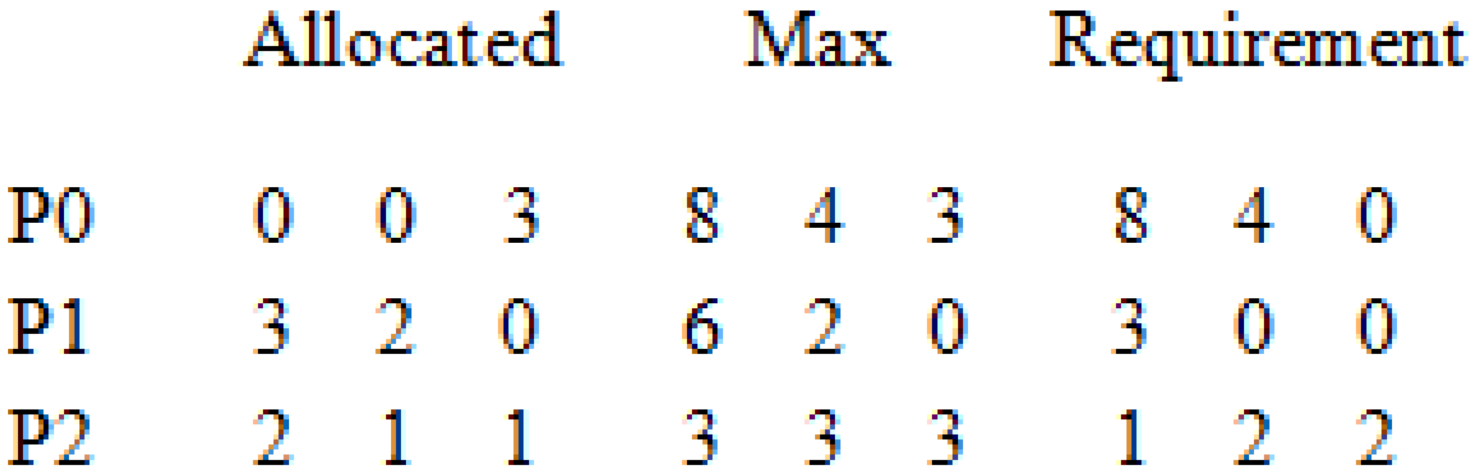


	Allocation			Max		
	X	Y	Z	X	Y	Z
P0	0	2	1	8	4	3
P1	3	2	0	6	2	0
P2	2	1	1	3	3	3

REQ1: P0 requests 0 units of X, 0 units of Y and 2 units of Z
REQ2: P1 requests 2 units of X, 0 units of Y and 0 units of Z
Select the suitable statement.

A	Only REQ1 can be permitted.
B	Only REQ2 can be permitted.
C	Both REQ1 and REQ2 can be permitted.
D	Neither REQ1 nor REQ2 can be permitted.

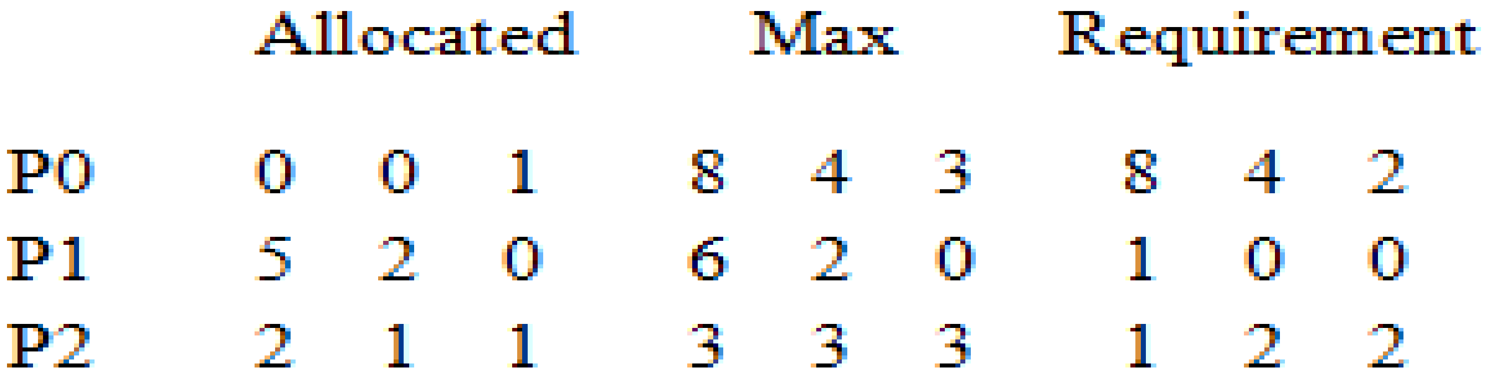
Lets take 1st case, After allowing Req 1,



	Allocated			Max			Requirement		
P0	0	0	3	8	4	3	8	4	0
P1	3	2	0	6	2	0	3	0	0
P2	2	1	1	3	3	3	1	2	2

Available: X=3, Y=2, Z=0
With this we can satisfy P1’s requirement. So available becomes: X=6, Y=4, Z=0.
Since Z is not available, neither P0’s nor P2’s requirement can be satisfied. So, it is an unsafe state.

Lets take 2nd case, After allowing Req 2,



	Allocated			Max			Requirement		
P0	0	0	1	8	4	3	8	4	2
P1	5	2	0	6	2	0	1	0	0
P2	2	1	1	3	3	3	1	2	2

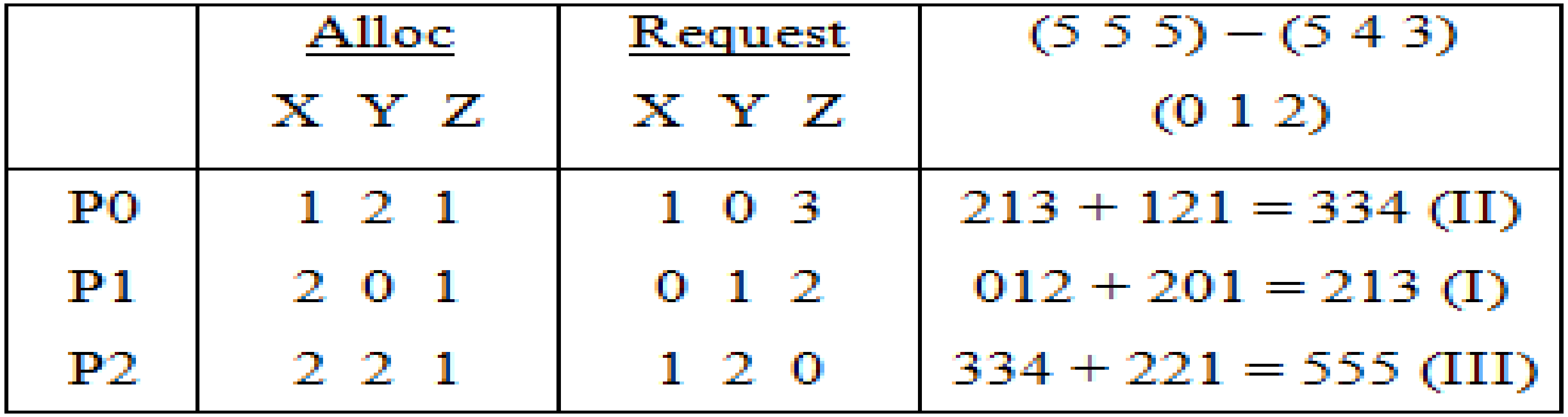
Available: X=1, Y=2, Z=2
With this we can satisfy any one of P1's or P2's requirement.
Lets first satisfy P1's requirement. So Available now becomes:
X=6, Y=4, Z=2
Now with the availability we can satisfy P2's requirement. So Available now becomes,
X=8, Y=5, Z=3
With this availability P0 can also be satisfied. So, hence it is in safe state.
So from above two cases Req 1 cannot be permitted but Req 2 can be permitted.

G11. A single processor system has three resource types X, Y and Z, which are shared by three processes. There are 5 units of each resource type. Consider the following scenario, where the column alloc denotes the number of units of each resource type allocated to each process, and the column request denotes the number of units of each resource type requested by a process in order to complete execution. Which of these processes will finish LAST? (GATE 2007)

	Alloc	request
	X Y Z	X Y Z
P0	1 2 1	1 0 3
P1	2 0 1	0 1 2
P2	2 2 1	1 2 0

A	P0
B	P1
C	P2
D	None of the above, since the system is in a deadlock.

Given that there are 5 units of each resource type.

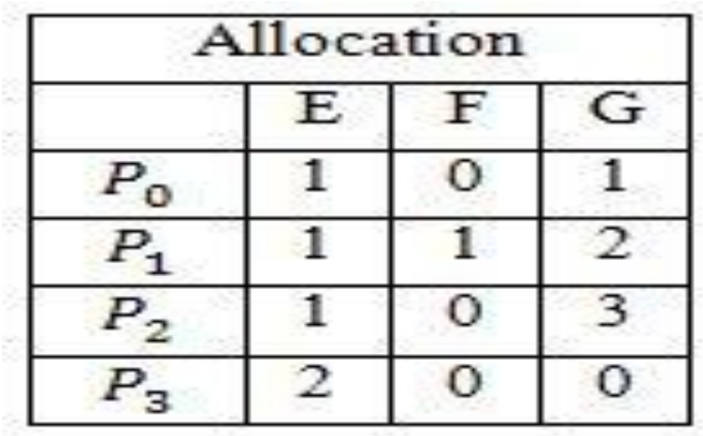


	Alloc	Request	(5 5 5) − (5 4 3)
	X Y Z	X Y Z	(0 1 2)
P0	1 2 1	1 0 3	213 + 121 = 334 (II)
P1	2 0 1	0 1 2	012 + 201 = 213 (I)
P2	2 2 1	1 2 0	334 + 221 = 555 (III)

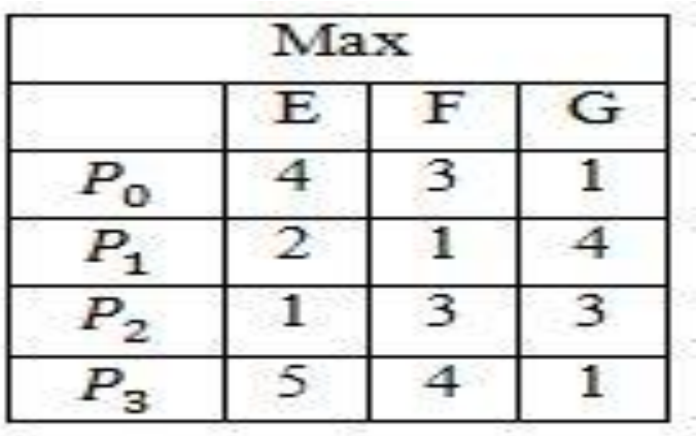
System is in safe state
Order of Execution = P2 will execute last.

G12. In a system, there are three types of resources: E, F and G. Four processes P_0, P_1, P_2 and P_3 execute concurrently. At the outset, the processes have declared their maximum resource requirements using a matrix named Max as given below. For example, Max[P_2, F] is the maximum number of instances of F that P_2 would require. The number of instances of the resources allocated to the various processes at any given state is given by a matrix named Allocation.

Consider a state of the system with the Allocation matrix as shown below, and in which 3 instances of E and 3 instances of F are the only resources available. (GATE 2018)



Allocation			
	E	F	G
P_0	1	0	1
P_1	1	1	2
P_2	1	0	3
P_3	2	0	0



Max			
	E	F	G
P_0	4	3	1
P_1	2	1	4
P_2	1	3	3
P_3	5	4	1

From the perspective of deadlock avoidance, which one of the following is true?

A The system is in safe state.

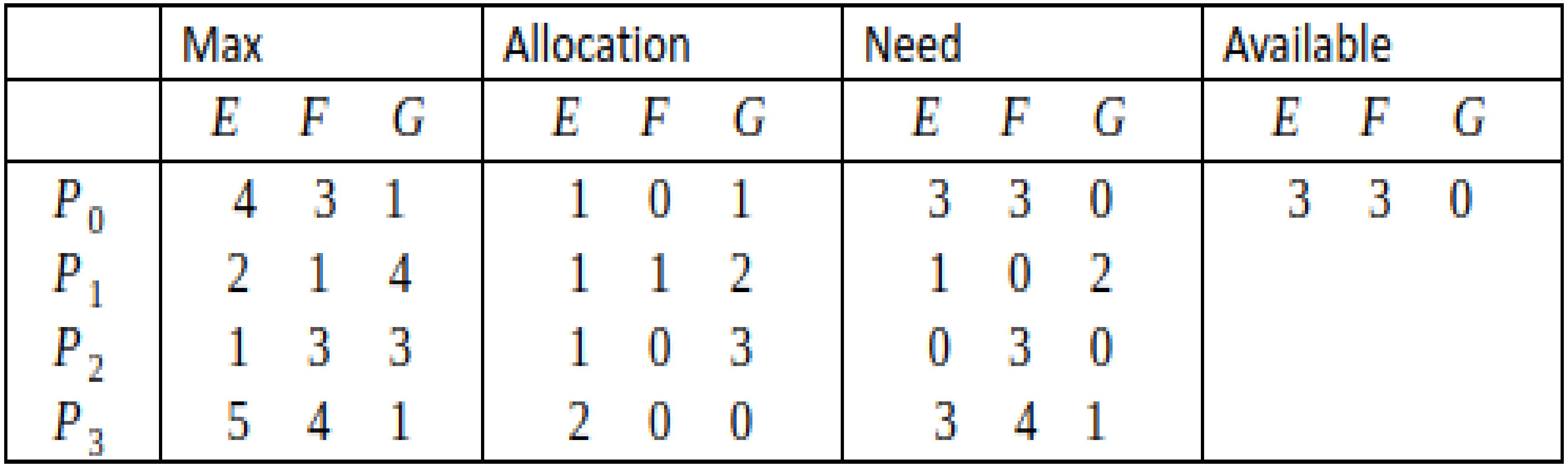
B The system is not in safe state, but would be safe if one more instance of E were available.

C The system is not in safe state, but would be safe if one more instance of F were available.

D The system is not in safe state, but would be safe if one more instance of G were available.

Solution:

Now, ⟨E, F, G⟩ = ⟨3, 3, 0⟩



	Max			Allocation			Need			Available		
	E	*F*	*G*	*E*	*F*	*G*	*E*	*F*	*G*	*E*	*F*	*G*
P_0	4	3	1	1	0	1	3	3	0	3	3	0
P_1	2	1	4	1	1	2	1	0	2			
P_2	1	3	3	1	0	3	0	3	0			
P_3	5	4	1	2	0	0	3	4	1			

Safe sequence: $\langle P_0, P_2, P_1, P_3\rangle$

P_0: P_0 can be allotted $\langle 3,3,0\rangle$.

After completion Available = $\langle 4,3,1\rangle$

P_2: P_2 can be allotted $\langle 0,3,0\rangle$.

After completion: Available = $\langle 5,3,4\rangle$

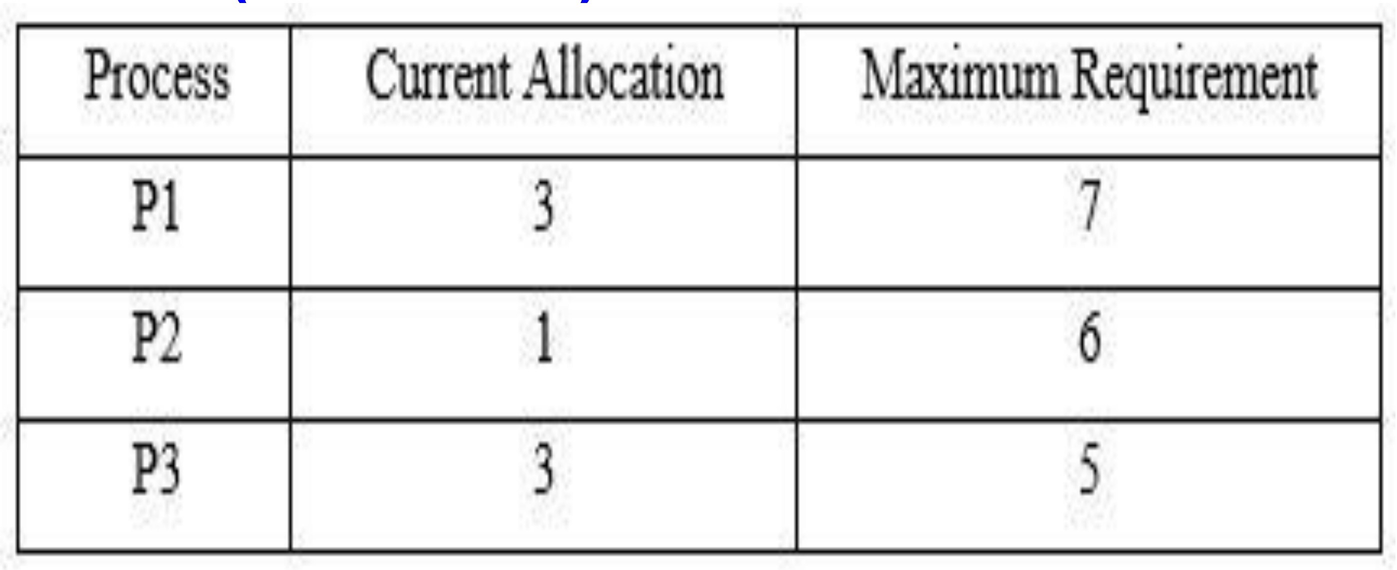
P_1: P_1 can be allotted $\langle 1,0,2\rangle$.

After completion: Available = $\langle 6,4,6\rangle$

P_3: P_3 can be allotted $\langle 3,4,1\rangle$.

After completion: Available = $\langle 8,4,6\rangle$

G13. A system shares 9 tape drives. The current allocation and maximum requirement of tape drives for three processes are shown below: (GATE 2017)



Process	Current Allocation	Maximum Requirement
P1	3	7
P2	1	6
P3	3	5

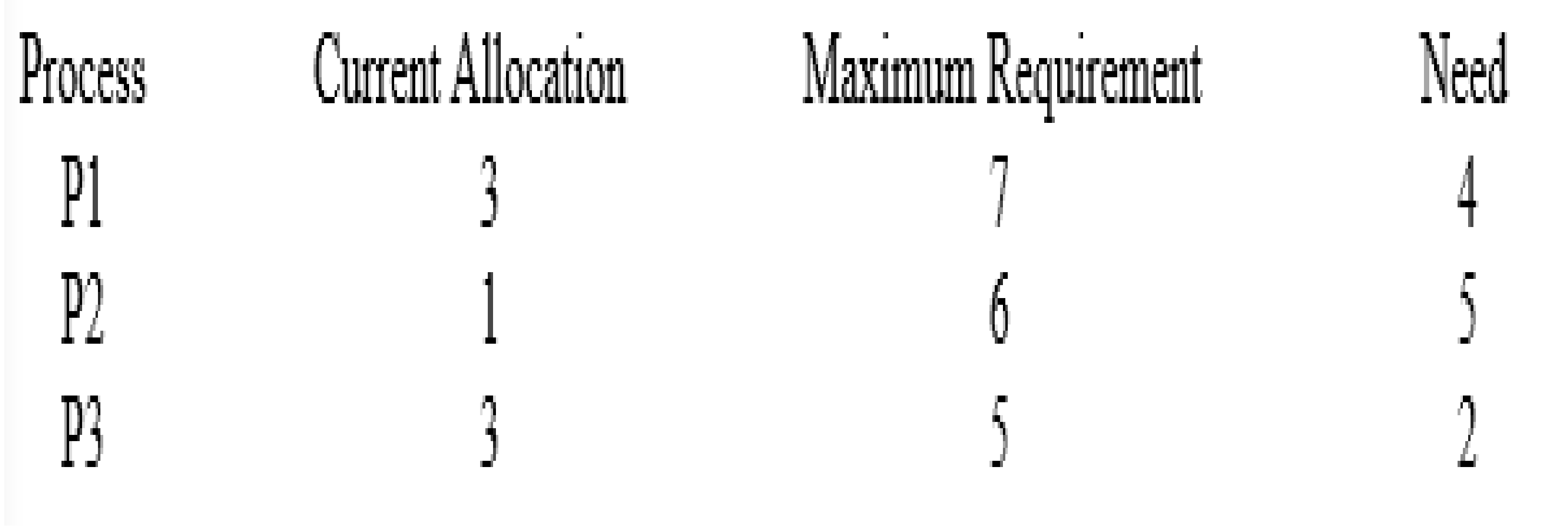
Which of the following best describe current state of the system?

A Safe, Deadlocked

[✓ B Safe, Not Deadlocked]

C Not Safe, Deadlocked

D Not Safe, Not Deadlocked



Process	Current Allocation	Maximum Requirement	Need
P1	3	7	4
P2	1	6	5
P3	3	5	2

Available: (9 - (3 + 1 + 3)) = 2, P3 can be satisfied.
After P3, Now available = 3 + 2 = 5
Now, P2 can be satisfied.
After P2, Now available: 5 + 1 = 6
Now, P1 can be satisfied. Thus safe sequence: P3 → P2 → P1
That is not deadlocked.

G14. The number of all possible safe sequences are

	R1	R2	R3
Total Resources	10	5	7

Process	Allocation			Max		
	R1	R2	R3	R1	R2	R3
P1	0	1	0	7	5	3
P2	2	0	0	3	2	2
P3	3	0	2	9	0	2
P4	2	1	1	2	2	2

A. 24　　B. 4　　C. 12　　D. 8 (Foreign University)

Output: Safe sequences are:

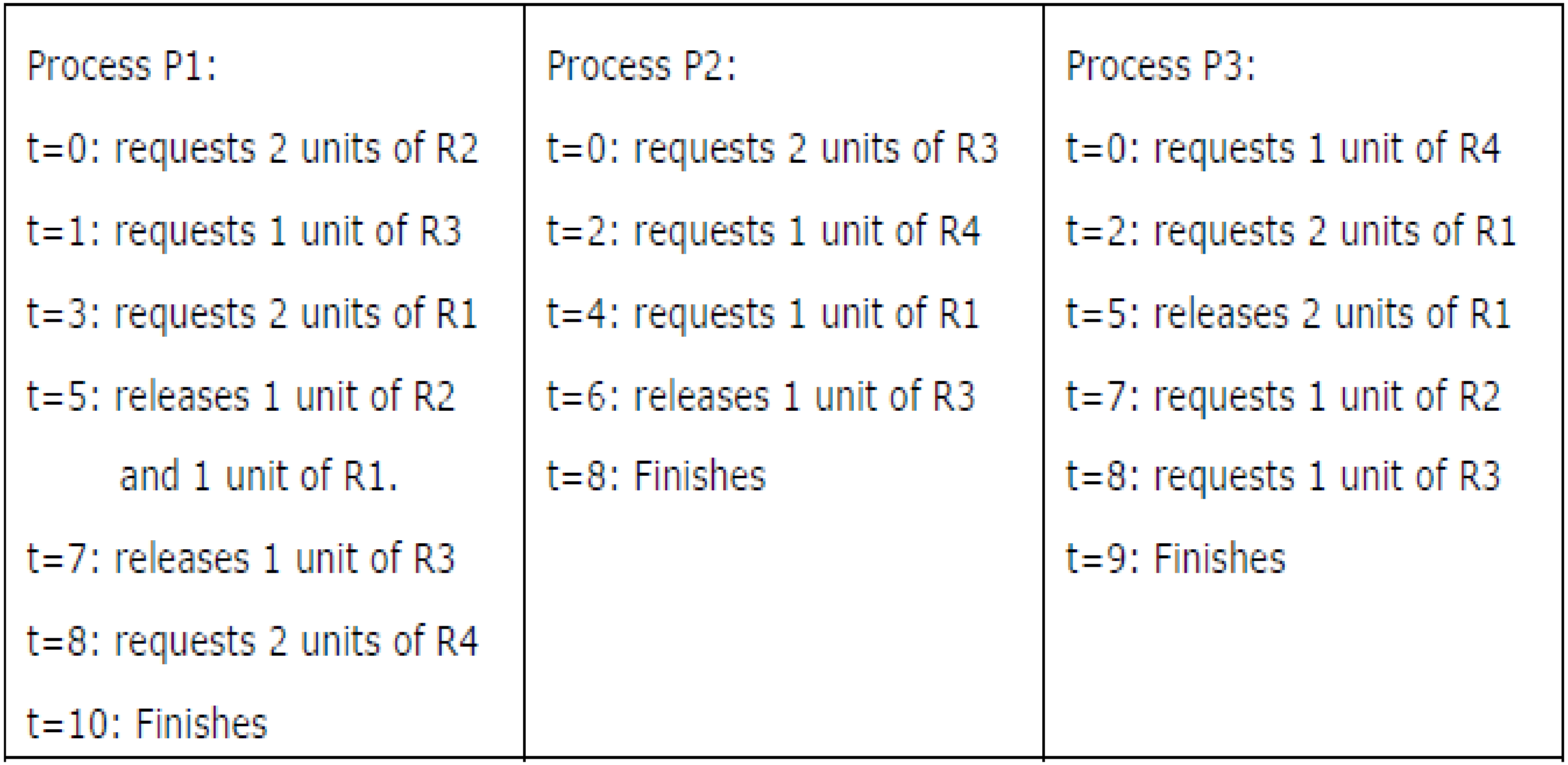
P2--> P4--> P1--> P3

P2--> P4--> P3--> P1

P4--> P2--> P1--> P3

P4--> P2--> P3--> P1 There are total 4 safe-sequences

G15. Consider a system with 4 types of resources R1 (3 units), R2 (2 units), R3 (3 units), R4 (2 units). A non-preemptive resource allocation policy is used. At any given instance, a request is not entertained if it cannot be completely satisfied. Three processes P1, P2, P3 request the sources as follows if executed independently. (GATE 2009)



Process P1:	Process P2:	Process P3:
t=0: requests 2 units of R2	t=0: requests 2 units of R3	t=0: requests 1 unit of R4
t=1: requests 1 unit of R3	t=2: requests 1 unit of R4	t=2: requests 2 units of R1
t=3: requests 2 units of R1	t=4: requests 1 unit of R1	t=5: releases 2 units of R1
t=5: releases 1 unit of R2	t=6: releases 1 unit of R3	t=7: requests 1 unit of R2
and 1 unit of R1.	t=8: Finishes	t=8: requests 1 unit of R3
t=7: releases 1 unit of R3		t=9: Finishes
t=8: requests 2 units of R4		
t=10: Finishes		

Which one of the following statements is TRUE if all three processes run concurrently starting at time t=0?

(A) All processes will finish without any deadlock

(B) Only P1 and P2 will be in deadlock.

(C) Only P1 and P3 will be in a deadlock.

(D) All three processes will be in deadlock.

Resource	Available	t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=9	t=10
R1	3	**3**	**3**	**1**	**1**	**0**	**1**	**1**	**1**	**2**	**2**	3
R2	2	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	2
R3	3	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**2**	**2**	**3**	3
R4	2	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**2**	2

Process P1:	Process P2:	Process P3:
t=0: requests 2 units of R2	t=0: requests 2 units of R3	t=0: requests 1unit of R4
t=1: requests 1 unit of R3		
	t=2: requests 1unit of R4	t=2: requests 2 units of R1
t=3: requests 2 units of R1		
	t=4: requests 1unit of R1	
t=5: releases 1 unit of R2 and 1 unit of R1		t=5: releases 2 units of R1
	t=6: releases 1 unit of R3	
t=7: releases 1unit of R3		t=7: requests 1unit of R2
t=8: requests 2 units of R4	t=8: Finishes	t=8: requests 1unit of R3
		t=9: Finishes
t=10: Finishes		

Resource	Available
R1	3
R2	2
R3	3
R4	2

Process P1:	Process P2:	Process P3:
t=0: requests 2 units of R2	t=0: requests 2 units of R3	t=0: requests 1unit of R4
t=1: requests 1 unit of R3		
	t=2: requests 1unit of R4	t=2: requests 2 units of R1
t=3: requests 2 units of R1		
	t=4: requests 1unit of R1	
t=5: releases 1 unit of R2 and 1 unit of R1		t=5: releases 2 units of R1
	t=6: releases 1 unit of R3	
t=7: releases 1unit of R3		t=7: requests 1unit of R2
t=8: requests 2 units of R4	t=8: Finishes	t=8: requests 1unit of R3
		t=9: Finishes
t=10: Finishes		

Resource	Available	t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=9	t=10
R1	3	3	3	1	1	0	2					
R2	2	0	0	0	0	0	0					
R3	3	1	0	0	0	0	0					
R4	2	1	1	0	0	0	0					

Process P1:	Process P2:	Process P3:
t=0: requests 2 units of R2	t=0: requests 2 units of R3	t=0: requests 1unit of R4
t=1: requests 1 unit of R3		
	t=2: requests 1unit of R4	t=2: requests 2 units of R1
t=3: requests 2 units of R1		
	t=4: requests 1unit of R1	
t=5: releases 1 unit of R2 and 1 unit of R1		t=5: releases 2 units of R1
	t=6: releases 1 unit of R3	
t=7: releases 1unit of R3		t=7: requests 1unit of R2
t=8: requests 2 units of R4	t=8: Finishes	t=8: requests 1unit of R3
		t=9: Finishes
t=10: Finishes		

Resource	Available	t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=9	t=10
R1	3	3	3	1	1	0	0					
R2	2	0	0	0	0	0	0					
R3	3	1	0	0	0	0	0					
R4	2	1	1	0	0	0	0					

Process P1:	Process P2:	Process P3:
t=0: requests 2 units of R2	t=0: requests 2 units of R3	t=0: requests 1unit of R4
t=1: requests 1 unit of R3		
	t=2: requests 1unit of R4	t=2: requests 2 units of R1
t=3: requests 2 units of R1		
	t=4: requests 1unit of R1	
t=5: releases 1 unit of R2 and 1 unit of R1		t=5: releases 2 units of R1
	t=6: releases 1 unit of R3	
t=7: releases 1unit of R3		t=7: requests 1unit of R2
t=8: requests 2 units of R4	t=8: Finishes	t=8: requests 1unit of R3
		t=9: Finishes
t=10: Finishes		

Resource	Available	t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=9	t=10
R1	3	3	3	1	1	0	1					
R2	2	0	0	0	0	0	1					
R3	3	1	0	0	0	0	0					
R4	2	1	1	0	0	0	0					

Process P1:	Process P2:	Process P3:
t=0: requests 2 units of R2	t=0: requests 2 units of R3	t=0: requests 1unit of R4
t=1: requests 1 unit of R3		
	t=2: requests 1unit of R4	t=2: requests 2 units of R1
t=3: requests 2 units of R1		
	t=4: requests 1unit of R1	
t=5: releases 1 unit of R2 and 1 unit of R1		t=5: releases 2 units of R1
	t=6: releases 1 unit of R3	
t=7: releases 1unit of R3		t=7: requests 1unit of R2
t=8: requests 2 units of R4	t=8: Finishes	t=8: requests 1unit of R3
		t=9: Finishes
t=10: Finishes		

Resource	Available	t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=9	t=10
R1	3	3	3	1	1	0	1	1				
R2	2	0	0	0	0	0	1	1				
R3	3	1	0	0	0	0	0	1				
R4	2	1	1	0	0	0	0	0				

Process P1:	Process P2:	Process P3:
t=0: requests 2 units of R2	t=0: requests 2 units of R3	t=0: requests 1unit of R4
t=1: requests 1 unit of R3		
	t=2: requests 1unit of R4	t=2: requests 2 units of R1
t=3: requests 2 units of R1		
	t=4: requests 1unit of R1	
t=5: releases 1 unit of R2 and 1 unit of R1		t=5: releases 2 units of R1
	t=6: releases 1 unit of R3	
t=7: releases 1unit of R3		t=7: requests 1unit of R2
t=8: requests 2 units of R4	t=8: Finishes	t=8: requests 1unit of R3
		t=9: Finishes
t=10: Finishes		

Resource	Available	t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=9	t=10
R1	3	3	3	1	1	0	1	1	1			
R2	2	0	0	0	0	0	1	1	0			
R3	3	1	0	0	0	0	0	1	2			
R4	2	1	1	0	0	0	0	0	0			

Process P1:	Process P2:	Process P3:
t=0: requests 2 units of R2	t=0: requests 2 units of R3	t=0: requests 1unit of R4
t=1: requests 1 unit of R3		
	t=2: requests 1unit of R4	t=2: requests 2 units of R1
t=3: requests 2 units of R1		
	t=4: requests 1unit of R1	
t=5: releases 1 unit of R2 and 1 unit of R1		t=5: releases 2 units of R1
	t=6: releases 1 unit of R3	
t=7: releases 1unit of R3		t=7: requests 1unit of R2
t=8: requests 2 units of R4	t=8: Finishes	t=8: requests 1unit of R3
		t=9: Finishes
t=10: Finishes		

Resource	Available	t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=9	t=10
R1	3	3	3	1	1	0	1	1	1	2		
R2	2	0	0	0	0	0	1	1	0	0		
R3	3	1	0	0	0	0	0	1	2	3		
R4	2	1	1	0	0	0	0	0	0	1		

Process P1:	Process P2:	Process P3:
t=0: requests 2 units of R2	t=0: requests 2 units of R3	t=0: requests 1unit of R4
t=1: requests 1 unit of R3		
	t=2: requests 1unit of R4	t=2: requests 2 units of R1
t=3: requests 2 units of R1		
	t=4: requests 1unit of R1	
t=5: releases 1 unit of R2 and 1 unit of R1		t=5: releases 2 units of R1
	t=6: releases 1 unit of R3	
t=7: releases 1unit of R3		t=7: requests 1unit of R2
t=8: requests 2 units of R4	t=8: Finishes	t=8: requests 1unit of R3
		t=9: Finishes
t=10: Finishes		

Resource	Available	t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=9	t=10
R1	3	3	3	1	1	0	1	1	1	2		
R2	2	0	0	0	0	0	1	1	0	0		
R3	3	1	0	0	0	0	0	1	2	2		
R4	2	1	1	0	0	0	0	0	0	1		

Process P1:	Process P2:	Process P3:
t=0: requests 2 units of R2	t=0: requests 2 units of R3	t=0: requests 1unit of R4
t=1: requests 1 unit of R3		
	t=2: requests 1unit of R4	t=2: requests 2 units of R1
t=3: requests 2 units of R1		
	t=4: requests 1unit of R1	
t=5: releases 1 unit of R2 and 1 unit of R1		t=5: releases 2 units of R1
	t=6: releases 1 unit of R3	
t=7: releases 1unit of R3		t=7: requests 1unit of R2
t=8: requests 2 units of R4	t=8: Finishes	t=8: requests 1unit of R3
		t=9: Finishes
t=10: Finishes		

Resource	Available	t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=9	t=10
R1	3	3	3	1	1	0	1	1	1	2	2	
R2	2	0	0	0	0	0	1	1	0	0	1	
R3	3	1	0	0	0	0	0	1	2	2	3	
R4	2	1	1	0	0	0	0	0	0	1	2	

Process P1:	Process P2:	Process P3:
t=0: requests 2 units of R2	t=0: requests 2 units of R3	t=0: requests 1unit of R4
t=1: requests 1 unit of R3		
	t=2: requests 1unit of R4	t=2: requests 2 units of R1
t=3: requests 2 units of R1		
	t=4: requests 1unit of R1	
t=5: releases 1 unit of R2 and 1 unit of R1		t=5: releases 2 units of R1
	t=6: releases 1 unit of R3	
t=7: releases 1unit of R3		t=7: requests 1unit of R2
t=8: requests 2 units of R4	t=8: Finishes	t=8: requests 1unit of R3
		t=9: Finishes
t=10: Finishes		

Resource	Available	t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=9	t=10
R1	3	3	3	1	1	0	1	1	1	2	2	
R2	2	0	0	0	0	0	1	1	0	0	1	
R3	3	1	0	0	0	0	0	1	2	2	3	
R4	2	1	1	0	0	0	0	0	0	1	0	

Process P1:	Process P2:	Process P3:
t=0: requests 2 units of R2	t=0: requests 2 units of R3	t=0: requests 1unit of R4
t=1: requests 1 unit of R3		
	t=2: requests 1unit of R4	t=2: requests 2 units of R1
t=3: requests 2 units of R1		
	t=4: requests 1unit of R1	
t=5: releases 1 unit of R2 and 1 unit of R1		t=5: releases 2 units of R1
	t=6: releases 1 unit of R3	
t=7: releases 1unit of R3		t=7: requests 1unit of R2
t=8: requests 2 units of R4	t=8: Finishes	t=8: requests 1unit of R3
		t=9: Finishes
t=10: Finishes		

Resource	Available	t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=9	t=10
R1	3	3	3	1	1	0	1	1	1	2	2	3
R2	2	0	0	0	0	0	1	1	0	0	1	2
R3	3	1	0	0	0	0	0	1	2	2	3	3
R4	2	1	1	0	0	0	0	0	0	1	0	2

G16. A system has n resources $R_0,\ldots,R_{n-1}$,and k processes $P_0,\ldots.P_{k-1}$. The implementation of the resource request logic of each process P_i is as follows: (GATE 2010)

```
if (i % 2 == 0)
        {
                if (i < n) request $R_i$
                if (i+2 < n) request $R_{i+2}$
        }
else
        {
                if (i < n) request $R_{n-i}$
                if (i+2 < n) request $R_{n-i-2}$
        }
```

In which one of the following situations is a deadlock possible?

(A) n=40, k=26

(B) n=21, k=12

(C) n=20, k=10

(D) n=41, k=19

No. of resources, n = 21

No. of processes, k = 12

Processes {P0, P1....P11} make the following Resource requests: {R0, R20, R2, R18, R4, R16, R6, R14, R8, R12, R10, R10}

For example P0 will request R0 (0%2 is = 0 and 0< n=21).

Similarly, P10 will request R10.

P11 will request R10 as n - i = 21 - 11 = 10.

As different processes are requesting the same resource, deadlock may occur.

Deadlock Detection

- Allow system to enter deadlock state

- Detection algorithm

  1. Resource with single instance – wait for graph

  2. Resource with multiple instances – Detection algorithm similar to Banker’s algorithm

- Recovery scheme

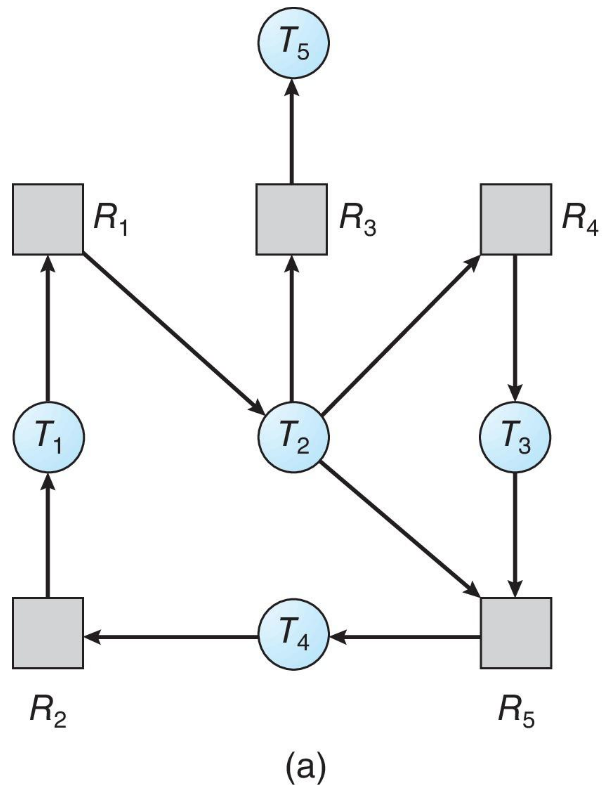
Single Instance of Each Resource Type

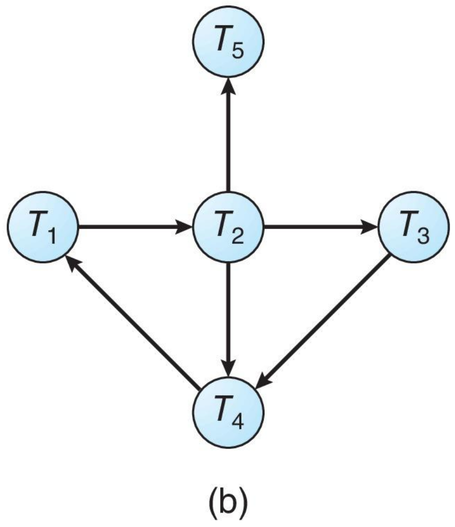
- Maintain **wait-for** graph and is a variant of the **Resource allocation graph**
  - Nodes are threads , to obtain this graph from the resource allocation graph **by removing the resource nodes** and **collapsing the appropriate edges**
  - $T_i \rightarrow T_j$ if T_i is waiting for T_j to release a resource that is needed by T_i . In a Resource allocation graph the same would be denoted as $T_i \rightarrow R_k$ and $R_k \rightarrow T_j$

- **Periodically invoke an algorithm that searches for a cycle in the graph**. If there is a cycle, there exists a deadlock

- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph





Resource-Allocation Graph   Corresponding 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 thread.
- **Request**: An ***n* x *m*** matrix indicates the current request of each thread. If ***Request* [*i*][*j*] = *k***, then thread T_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 $\boldsymbol{i = 1,2, \ldots, n}$, if $\boldsymbol{Allocation_i \neq 0}$, then
      Finish*[i] = *false; otherwise, ***Finish*[i] = *true***

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

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

   b) $\boldsymbol{Request_i \leq Work}$

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

3. $\boldsymbol{Work = Work + Allocation_i}$
   Finish*[*i*] = *true
   go to step 2

4. If ***Finish[i] == false***, for some ***i***, $1 \leq \boldsymbol{i} \leq \boldsymbol{n}$, then the system is in deadlock state. Moreover, if ***Finish*[*i*] == *false***, then $\boldsymbol{T_i}$ is deadlocked

Example of Detection Algorithm

- Five Processes $\boldsymbol{P_0}$ through $\boldsymbol{P_4}$; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)
- Snapshot at time T_0:

Process	**Allocation**			**Request**			**Available**		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P0	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
P1	**2**	**0**	**0**	**2**	**0**	**2**			
P2	**3**	**0**	**3**	**0**	**0**	**0**			
P3	**2**	**1**	**1**	**1**	**0**	**0**			
P4	**0**	**0**	**2**	**0**	**0**	**2**			

Safe Sequence order is $<\boldsymbol{P_0, P_2, P_3, P_4, P_1}>$ or $<\boldsymbol{P_0, P_2, P_3, P_1, P_4}>$

Example of Detection Algorithm

If follow this order< P_0, P_2, P_3, P_1, P_4 >

After executing the Process	Currently available		
	A	B	C
Initially	0	0	0
P0	0	1	0
P2	3	1	3
P3	5	2	4
P1	7	2	4
P4	7	2	6

If follow this order< P_0, P_2, P_3, P_4, P_1 >

After executing the Process	Currently available		
	A	B	C
Initially	0	0	0
P0	0	1	0
P2	3	1	3
P3	5	2	4
P1	7	2	4
P4	7	2	6

Example (Cont.)

- Suppose P_2 requests an additional instance of type C, then the request matrix is modified as

Process	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	1			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

After executing the Process	Currently available		
	A	B	C
Initially	0	0	0
P0	0	1	0
Now the system is deadlocked. After reclaim the resources from P0, the number of resources available is not sufficient to fulfill the requests by other processes			

Deadlock exists, consisting of processes $\boldsymbol{P_1}$, $\boldsymbol{P_2}$, $\boldsymbol{P_3}$, and $\boldsymbol{P_4}$

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?

- 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 threads “caused” the deadlock.

Recovery from Deadlock: Process Termination

- Abort all deadlocked threads
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  1. Priority of the process
  2. How long has the process computed, and how much longer to completion
  3. Resources that the process has used
  4. Resources that the process needs to complete
  5. How many process will need to be terminated
  6. Is the process interactive or batch?

Recovery from Deadlock: Resource Preemption

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

G17. Which of the following statements is/are TRUE with respect to deadlocks? (GATE 2022) [More than One correct answer]

A. Circular wait is a necessary condition for the formation of deadlock.

B. In a system where each resource has more than one instance, a cycle in its wait-for graph indicates the presence of a deadlock.

C. If the current allocation of resources to processes leads the system to unsafe state, then deadlock will necessarily occur.

D. In the resource-allocation graph of a system, if every edge is an assignment edge, then the system is not in deadlock state.

G18. Consider the following threads, T1, T2, and T3 executing on a single processor, synchronized using three binary semaphore variables, S1, S2, and S3, operated upon using standard wait() and signal(). The threads can be context switched in any order and at any time. (GATE 2021)

T_1	T_2	T_3
`while(true){ wait(S3); print("C"); signal(S2); }`	`while(true){ wait(S1); print("B"); signal(S3); }`	`while(true){ wait(S2); print("A"); signal(S1); }`

Which initialization of the semaphores would print the sequence BCABCABCA….?

(A) $S_1 = 1; S_2 = 1; S_3 = 1$

(B) $S_1 = 1; S_2 = 1; S_3 = 0$

(C) $S_1 = 1; S_2 = 0; S_3 = 0$

(D) $S_1 = 0; S_2 = 1; S_3 = 1$

Initially if S1 = 1, S2 = 0, S3 = 0 ,

Process T2 can successfully execute wait(S1); while T1 and T3 remain stuck at wait(S3); and wait(S2); respectively.

After process T2 **prints B** it executes signal(S3), and gets stuck at wait(S1);

(B gets printed in this process.)

After this Process T1 can successfully execute wait(S3); and then it executes **print("C"),** after which it executes signal(S2); and then gets stuck at wait(S3);

(C gets printed in this process.)

After this Process T3 can successfully execute wait(S2); and then it executes **print("A"),** after which it executes signal(S1); and then gets stuck at wait(S2);

(A gets printed in this process.)

After this Process T2 can execute wait(S1); successfully.

The process thus keeps repeating and the pattern printed is BCABCABCA…

G19. Consider a system with 3 processes that share 4 instances of the same resource type. Each process can request a maximum of K instances. Resource instances can be requested and released only one at a time. The largest value of K that will always avoid deadlock is ________ . (GATE 2018)

(A) 1 **(B)** 2 **(C)** 3 **(D)** 4

Since deadlock-free condition is:

$R \geq P(N - 1) + 1$ Where R is total number of resources,

P is the number of processes, and

N is the max need for each resource.

Given P= 3 , R =4

$4 \geq 3(N - 1) + 1$

$3 \geq 3(N - 1)$

$1 \geq (N - 1)$

$N \leq 2$

G20. A computer has six tape drives, with n processes competing for them. Each process may need two drives. What is the maximum value of n for the system to be deadlock free?
(A) 6 (B) 5 (C) 4 (D) 3 (GATE 1998)

Given tape drive = 6 and each process may need 2 drive.

When give 1 drive to 1 process then total process will be 6 but in this case there will definitely deadlock occur because every process contain 1 drive and waiting for another drive which is hold by other process.

Therefore when reduce 1 process then system to be deadlock free.

Hence maximum value of n = 6 – 1 = 5.

G21. Which of the following is NOT true of deadlock prevention and deadlock avoidance schemes? (GATE 2008)

A. In deadlock prevention, the request for resources is always granted if the resulting state is safe

B. In deadlock avoidance, the request for resources is always granted if the resulting state is safe

C. Deadlock avoidance is less restrictive than deadlock prevention

D. Deadlock avoidance requires knowledge of resource requirements *apriori*.

Answer: Deadlock Prevention scheme depends on the four techniques does not happen simultaneously.

G22. **Consider the following policies for preventing deadlock in a system with mutually exclusive resources. (GATE 2015)**

I. Processes should acquire all their resources at the beginning of execution. If any resource is not available, all resources acquired so far are released

II. The resources are numbered uniquely, and processes are allowed to request for resources only in increasing resource numbers

III. The resources are numbered uniquely, and processes are allowed to request for resources only in decreasing resource numbers

IV. The resources are numbered uniquely. A process is allowed to request only for a resource with resource number larger than its currently held resources

Which of the above policies can be used for preventing deadlock?

(A) Any one of I and III but not II or IV

(B) Any one of I, III and IV but not II

(C) Any one of II and III but not I or IV

(D) Any one of I, II, III and IV

Policy I : will avoid hold and wait
Policy II : will avoid circular wait
Policy III : will avoid circular wait
Policy IV : will avoid circular wait