CONCURRENCY CONTROL / PROCESS SYNCRONIZATION

SCHOOL OF COMPUTER ENGINEERING AND TECHNOLOGY

Syllabus – Unit III

Concurrency Control

Concurrent processes, Critical Section, Race Conditions, Mutual Exclusion, Hardware Solution,

Semaphores, Monitors, Message Passing.

Classical IPC Problems: Reader's & Writer Problem, The Producer Consumer Problem

Deadlocks: Definition, Necessary and sufficient conditions for Deadlock, Deadlock Prevention,

Deadlock Avoidance: Banker's algorithm, Deadlock detection and Recovery.

Concurrency

- Multiprogramming: Management of multiple processes within a uni-processor system
- Multiprocessing: Managing multiple processes within multiprocessor.

Design issues in concurrency

- Communication among processes
- Sharing & competition for resources
- Synchronization of activities of multiple processes
- Allocation of processor time to processes

Concurrency

- **Different contexts of concurrency**
  - Multiple applications
    - Multiprogramming
  - Structured applications
    - Some applications can be effectively programmed as a set of concurrent processes(Principles of modular design & structured programming)
  - OS structure
    - OS often implemented as a set of processes or threads

Structured applications

- **Example: Web Browser as a Structured Application**
- **How It's Structured**
- The browser is divided into **concurrent modules (processes or threads)**, each responsible for a specific task:
- **UI Process** – Handles buttons, menus, address bar, tab management.
- **Rendering Engine Process** – Displays the webpage content.
- **Network Process** – Downloads web pages, images, and scripts from the internet.
- **JavaScript Engine Thread** – Runs scripts in the page.
- **Plugin/Extension Processes** – Run add-ons like PDF viewers.

- **Why It’s Structured Concurrently**
- If the **network thread** is waiting for a slow server, the **UI thread** can still respond to clicks.
- If one tab crashes, other tabs (separate processes) can still work.
- Multiple pages can load in parallel.

- **OS Concept Mapping**
- **Structured Application**: Designed as multiple concurrent processes/threads.
- **Modular Design**: Each module focuses on one job.
- **Concurrency Control**: Synchronization ensures modules share data correctly (e.g., caching, cookies)

Key terms related to concurrency

- Atomic operation:

A sequence of one or more statements that appear to be indivisible, i.e., no other process can see an intermediate state or interrupt the operation

- Critical section:

A section of code within a process that requires access to shared resources & that must not be executed while another process is in a corresponding section of code

- Deadlock:

A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something

Atomic operation:

A sequence of one or more statements that appear to be indivisible, i.e., no other process can see an intermediate state or interrupt the operation

Without Atomic Operation

If the increment is not atomic, it may be broken into steps like:

1.Read count into register.

2.Add 1.

3.Write back to count.

If both processes do this at the same time:

- **Process A** reads 5.
- **Process B** reads 5.
- Both add 1 → both get 6.
- Both write 6 back → final result is **6** instead of **7**.

(Race condition occurs)

With Atomic Operation

If count++ is atomic:

- Only one process can read, add, and write back without interruption.
- If Process A increments to 6, Process B will see the updated value (6) and increment to 7.

Critical section:

Example: Printing to a Shared Printer Scenario

Two computers on a network share the same printer.

Both send print jobs to the printer's **print queue** stored in a shared memory location.

Critical Section

The **code segment** that:

1.Accesses the shared print queue.

2.Adds a new job to the queue.

3.Updates the queue status.

This part must be executed by **only one process at a time** to avoid corruption.

- **Without Critical Section Control**
- **Computer A** starts adding a job to the queue.
- **Computer B** also starts adding its job at the same time.
- The data in the print queue becomes **distorted**, and one or both jobs fail.

Critical section:

With Critical Section Control

- **Computer A** enters the critical section → locks the queue.
- **Computer B** must wait.
- A finishes adding its job → unlocks the queue.
- B can now safely add its job.

OS Mapping

- **Critical Section** → Code accessing the shared print queue.
- **Concurrency Control Tool** → Mutex, Semaphore, or Monitor.

Deadlock:

A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something

- **Example: Four-Way Traffic Deadlock**
- **Scenario**
- Four cars arrive at a four-way intersection at the same time.
- Each driver wants to go straight but the cars block each other because:
  - Each one has entered the intersection.
  - Each is waiting for the car in front to move.
  - No one can move forward because another car is blocking the way.

Key terms related to concurrency

- Mutual exclusion:

The requirement that when one process is in Critical Section that accesses shared resources no other process may be in critical section that accesses any of those shared resources.

- Race condition:

A situation in which multiple threads/processes read & write a shared data item and final result depends on relative timing of their execution.

- Starvation:

A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

Difficulties due to concurrency

- Sharing of global resources: Eg. Two processes both make use of global variable & both perform read & write on that variable. Order in which read & write are done is critical.
- Management of resources optimally: Eg. Process has gained the ownership of i/o device but is suspended before using it, thus locking i/o device and preventing its use by other processes.
- Error locating in program: Results are not deterministic and reproducible.

Example

```
void echo() { chin = getchar();
              chout = chin;
              putchar(chout);
            }
```

- Uniprocessor multiprogramming , single user environment
- Many applications can call this procedure repeatedly to accept user i/p & display on screen
- User can jump from one application to other.
- Each application needs/ uses procedure echo.
- Echo is made shared procedure and loaded into a portion of memory, global to all applications
- Single copy of echo procedure is used, saving space

Problem and solution

- Sequence of execution
  1. Process p1 invokes echo & gets a character i/p from keyboard say x. At this point it gets interrupted.
  2. Process P2 invokes echo accepts character as y and displays it on screen.
  3. Process p1 resumes, value of chin was x but now overwritten as y, thus chin has lost value which it had.

- Solution: When echo procedure is invoked by any process and if that process gets suspended for any reason before completing it, then no other process can invoke echo till process that was suspended is resumed & completes echo. Thus other processes are blocked from entering into echo.
- Same is applicable to multiprocessor systems

Race condition

- A race condition occurs when multiple competing processes or threads read and write data items so that final result depends on the order of execution of instructions in multiple processes.
- Two processes p1 & p2, share global variable ‘a’. P1 updates a to 1 and then p2 updates it to 2. Thus two tasks are in race to write variable ‘a’. Loser of race (the one who updates last) determines the value of ‘a’

Race condition

Another eg.

Initially, shared global variables have values b=1 & c=2

P1 executes b=b+c

P2 executes c=b+c

If p1 executes first, then b=? & c=?

If p2 executes first, then b=? & c=?

Race condition

Another eg.

Initially, shared global variables have values b=1 & c=2

P1 executes b=b+c

P2 executes c=b+c

If p1 executes first, then b=3 & c=5

If p2 executes first, then b=4 & c=3

How Processes interact with each other

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

How Processes interact with each other

- **Processes unaware of each other (Competition)**
  - E.g. Multiprogramming of multiple independent processes
  - OS need to know about competition for resources such as printer, disk, file, etc
  - Potential problems : mutual exclusion, deadlock, starvation

- **Processes indirectly aware of each other (Cooperation by sharing)**
  - Shared access to some object such as shared variable
  - Cooperation by sharing
  - Potential problem: mutual exclusion, deadlock, starvation, data coherence

How Processes interact with each other

- **Processes directly aware of each other (Cooperation by communication)**
  - Cooperation by communication, communication primitives available
  - Potential control Problems: Deadlock and starvation
  - Mutual exclusion not a problem
  - Deadlock possible
  - Starvation possible

Three Control Problems: Competition

- Need for mutual exclusion: Two or more processes require access to single non-sharable resource such as printer. Such a resource is called as Critical resource and portion of code using it is called as critical section(CS) of program.
- Mutual Exclusion - Only one process should be allowed in its critical section.
- Deadlock
- Starvation

Control problems: Competition

- Deadlock: Mutual exclusion creates a problem of deadlock.

  p1← printer and p2← file

  both p1 and p2 require printer and file to complete the task but printer is blocked from using by p1 and file is blocked from using by p2. Thus leading a deadlock

Control problems: Competition

Starvation:

- Three processes p1, p2, p3 require a periodic access to resource type R.
- Process p1 currently using R, so p2 and p3 must wait.
- When process p1 exits critical section, suppose p3 gains the control, and suppose p1 again needs resource R and OS assigns it to p1 when p3 exits the CS.
- This situation may continue and p2 never gets ownership of R.

Co-operation among processes by sharing

- Processes that interact with other processes without being explicitly aware of them. Access to shared variables/files/databases
- Processes may use & update shared data without reference to other processes but know that other processes may have access to same data
- Processes must co-operate to ensure that the data that they share are properly managed.
- Control problems of mutual exclusion, deadlock & starvation are again present.
- Data items are accessed in 2 modes: reading & writing
- Writing operations must be mutually exclusive.

Co-operation among processes by sharing

- New requirement introduced i.e. Data coherence
- Suppose 2 data items a & b are to be maintained in the relationship a=b i.e. any program that updates one value must update other to maintain the relationship
- Consider 2 processes

p1: a = a+1

b = b+1;

p2: b = 2*b

a = 2*a

Requirement is always a=b

Initially a=b=1

Concurrent execution

a = a + 1

b = 2 * b

b= b + 1;

a = 2 * a

This leads to a=4 & b=3

Co-operation among processes by sharing

- New requirement introduced i.e. Data coherence
- Suppose 2 data items a & b are to be maintained in the relationship a=b i.e. any program that updates one value must update other to maintain the relationship
- Consider 2 processes

p1: a = a+1

b = b+1;

p2: b = 2*b

a = 2*a

Requirement is always a=b

Initially a=b=1

Concurrent execution

a = a + 1

b = 2 * b

b= b + 1;

a = 2 * a

This leads to a=4 & b=3

Solution : Declare entire sequence in each process as CS

Co-operation among processes by communication

- Processes co-operate by communication, since they participate in common effort that links all of the processes.
- Communication provides a way to synchronize / co-ordinate various activities
- Communication is in form of sending and receiving messages
- Mutual exclusion is not a problem, since there is no sharing.
- Deadlock possible: Two processes may be blocked, waiting for communication from each other.
- Starvation possible: (p1,p2,p3) p1 repeatedly attempts to communicate with p2 and p3. p2 & p3 attempt to communicate with p1. A sequence may arise in which p1 & p2 continuously communicate with each other while p3 is blocked & waiting for communication

Requirements for Mutual exclusion

Any facility/capability providing support for mutual exclusion should meet following requirements:

- Mutual exclusion must be **enforced**: Only one process at a time in the CS, among all processes that have CS for same resource or shared object.
- A process that **halts in its non CS** must do without interfering with other processes.
- A process requiring access to CS must not be **denied/delayed indefinitely** (no deadlock or starvation)
- When **no process in its CS**, any process that requests a entry to its CS, must be permitted to enter without delay.
- No assumptions are made about the **relative speed of processes or total no. of processes**.
- Process remains in its **CS for a finite time** only.

Approaches to satisfy the requirements of Mutual Exclusion

1. Hardware Approach
2. Support from OS or programming language
3. Software approach (No Support from OS or programming language)

Hardware Approach for Mutual Exclusion

1. Disabling Interrupts
   - A process runs until it invokes an operating system service or until it is interrupted
   - Disabling interrupts guarantees mutual exclusion
   - Because CS cannot be interrupted, mutual exclusion is guaranteed
   - Will not work in multiprocessor architecture, since computer includes more than one processor & possible for more than one process to be executing at a time
   - The efficiency of execution degraded as processor is limited in its ability to interleave programs

Pseudo Code

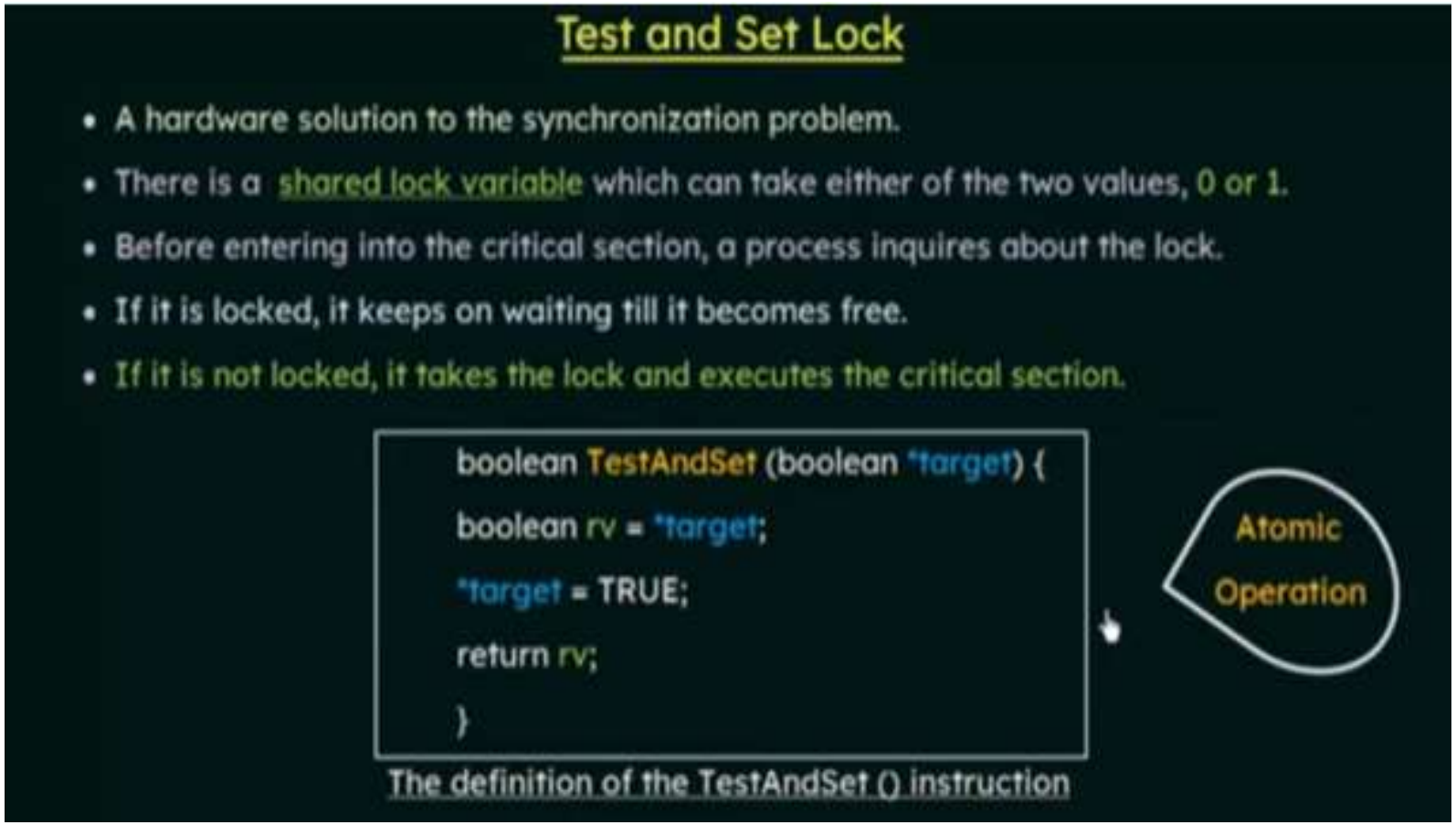
```
while (true) {
   /* disable interrupts */;
   /* critical section */;
   /* enable interrupts */;
   /* remainder */;
}
```

Hardware Approach : Special Machine Instructions

2. Special Machine Instructions

- Processor designers proposed machine instructions that carry out two actions atomically
- Compare and exchange /compare & swap instruction or testset

Test and Set instruction



Test and Set instruction

```
boolean testset(int i)//done
 atomically
{
  if (i==0)
     { i= 1;
     return true;
     }
  else
     return false;
}
```

```
const int n = /* no of processes */;
int bolt;
void P(int i)
{
   while(1)
    {

      while(testset(bolt)!=true)
            {
        /* do nothing */ ;
            }
     /* critical section */ ;
       bolt = 0;                    lock=false
    /* remainder * /;

   }
```

```
 main()
{
  bolt =0;

 Parbegin(P(1),P(2),….P(n));



}
```

```
boolean testset(int i)//done atomically
{
  if (i==0)
     { i= 1;
     return true;
     }
  else
     return false;
}
```

Mutual Exclusion: Hardware Support Advantages

- Applicable to any number of processes on either a single processor or multiple processors sharing main memory
- It is simple and therefore easy to verify
- It can be used to support multiple critical sections

Mutual Exclusion: Hardware Support Disadvantages

- **Busy waiting or spin waiting is employed** – Thus a process waiting to access a CS continuously consumes processor time
- **Starvation is possible –** when a process leaves CS, selection of a waiting process is arbitrary & hence some process could be indefinitely denied access
- **Deadlock is possible –** Eg. P1 executes special instruction & enters its CS. P1 is interrupted to give processor to P2 (having higher priority). If P2 attempts to use same resource as P1, it will be denied access because of mutual exclusion mechanism. Thus it will go into busy waiting loop. However P1 will never be dispatched because it is of lower priority than P2

Mutual Exclusion: OS or programming language support

- Semaphore
- Mutex
- Monitors

Semaphore

- Provides multiple process solution for mutual exclusion
- Uses a variable which is an integer
- It is accessed only through two standard atomic(indivisible) operations: wait & signal
- Semaphores are of 2 types…
    1. Binary : Variable takes values 0 or 1
    2. General/Counting : Variable takes any integer value

Semaphore

- **Wait** : Is used for acquiring a shared resource represented by the semaphore Hence wait decrements the value of semaphore variable
- Denoted by P
- wait(semaphore s)

```
{
        while(s<=0);
                // no operation or busy waiting
        s--;
}
```

Semaphore

- **Signal:** releases the shared resource represented by the semaphore
  Therefore signal increments the value of semaphore variable

Denoted by V

```
signal(semaphore s)
  {
        s++;
  }
```

Semaphore : Removing busy waiting

- Semaphores suffer from busy waiting
- To overcome, modify the working of wait & signal
- When a process executes waits operation & finds semaphore value not greater than 0, it must wait
- Instead of process doing a busy wait, process is placed into a waiting queue associated with the semaphore & CPU selects another process to execute
- Waiting process is restarted, when some other process executes a signal operation. Process moves from waiting state to ready state & process is placed in ready queue

Semaphore contd.

For both counting semaphores and binary semaphores, a queue is used to hold processes waiting on the semaphore. The fairest removal policy is

- first-in-first-out (FIFO): The process that has been blocked the longest is released from the queue first;
- A semaphore whose definition includes this policy is called a **strong semaphore**.
- A semaphore that does not specify the order in which processes are removed from the queue is a **weak semaphore**

- initially $m = 1$

Process P_i

```
{
    wait(m);
       critical section
    signal(m);
       remainder section
}
```

Semaphore example

The following program consists of 3 concurrent processes and 3 binary semaphores. The semaphores are initialized as S0 = 1, S1 = 0 and S2 = 0.

P0	P1	P2

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

How many times will process P0 print '0'?

1.At least twice
2.Exactly twice
3.Exactly thrice
4.Exactly once

The semaphores are initialized as S0=1, S1=0, S2=0.

Because S0 =1 then P0 enter into the critical section and other processes will wait until either S1=1 or S2 =1

The minimum number of times 0 printed:

- S0 =1 then P0 enter into the critical section
- **print '0'**
- then release S1 and S2 means S1 =1 and s2 =1
- now either P1 or P2 can enter into the critical section
- if P1 enter into the critical section
- release S0
- then P2 enter into the critical section
- release S0
- P1 enter into the critical section
- **print '0'**

The minimum number of time **0 printed** is **twice** when executing in this order **(p0 -> p1 -> p2 -> p0)**

Semaphore example

The following program consists of 3 concurrent processes and 3 binary semaphores. The semaphores are initialized as S0 = 1, S1 = 0 and S2 = 0.

P0	P1	P2

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

How many times will process P0 print '0'?

1.At least twice

2.Exactly twice

3.Exactly thrice

4.Exactly once

Semaphore example

Suppose we want to synchronize two concurrent processes P1 and P2 using binary semaphores s and t. The code for the processes P1 and P2 is shown below-

Process P1:

```
while(1)
{
w:
_____
   print '0';
   print '0';
x:_____
___
}
```

Process P2:

```
while(1)
{
y: _____
  print '1';
  print '1';
z:_____
}
```

The required output is "001100110011 …….". Processes are synchronized using P & V operations on semaphores s & t. Choose the correct options from the following at points w, x, y & z. Which of the following option is correct?

1. P(s) at w, V(s) at x, P(t) at y, V(t) at z, s and t initially 0
2. P(s) at w, V(t) at x, P(t) at y, V(s) at z, s initially 1 and t initially 0
3. P(s) at w, V(t) at x, P(t) at y, V(s) at z, s and t initially 0
4. P(s) at w, V(s) at x, P(t) at y, V(t) at z, s initially 1 and t initially 1

Semaphore example

Suppose we want to synchronize two concurrent processes P1 and P2 using binary semaphores s and t. The code for the processes P1 and P2 is shown below-

Process P1:

```
while(1)
{
w:
_____
   print '0';
   print '0';
x:_____
__
}
```

Process P2:

```
while(1)
{
y: _____
  print '1';
  print '1';
z:_____
}
```

The required output is "001100110011 …….". Processes are synchronized using P & V operations on semaphores s & t. Choose the correct options from the following at points w, x, y & z. Which of the following option is correct?

1. P(s) at w, V(s) at x, P(t) at y, V(t) at z, s and t initially 0
2. **P(s) at w, V(t) at x, P(t) at y, V(s) at z, s initially 1 and t initially 0**
3. P(s) at w, V(t) at x, P(t) at y, V(s) at z, s and t initially 0
4. P(s) at w, V(s) at x, P(t) at y, V(t) at z, s initially 1 and t initially 1

Semaphore example

Suppose we want to synchronize two concurrent processes P and Q using binary semaphores S1 and S2. Initial value of S1 and S2 is 1.The code for the processes P and Q is shown below-

P	Q

while(1)
{
P(S1);
P(S2);
Critical Section
V(S1);
V(S2);
}
	while(1)
	{
	P(S2);
	P(S1);
	Critical Section
	V(S1);
	V(S2);
	}

This leads to -

1.Mutual Exclusion
2.Deadlock
3.Both (1) and (2)
4.None of these

Semaphore example

Suppose we want to synchronize two concurrent processes P and Q using binary semaphores S1 and S2. Initial value of S1 and S2 is 1.The code for the processes P and Q is shown below-

P	Q

while(1)
{
P(S1);
P(S2);
Critical Section
V(S1);
V(S2);
}
	while(1)
	{
	P(S2);
	P(S1);
	Critical Section
	V(S1);
	V(S2);
	}

This leads to -

1.Mutual Exclusion

2.Deadlock

3.Both (1) and (2)

4.None of these

a possible interleaving is:

1.P does wait(S1) → now S1 = 0.

2.Q does wait(S2) → now S2 = 0.

3.P tries wait(S2) → blocks (since S2 = 0).

4.Q tries wait(S1) → blocks (since S1 = 0).

Now both are blocked forever → **deadlock**.

Semaphore example

Suppose we want to synchronize two concurrent processes P1 and P2 using binary semaphores s and t. The code for the processes P1 and P2 is shown below-

- **Process P1:**
- **while**(1)
- {
- w: _____ print '0'; print '0';
- x:_____
- }

Process P2:

while(1)

{

y: _____
print '0';
print '0';

z:_____

}

The required output is “001100110011 …….”. Processes are synchronized using P & V operations on semaphores s & t. Choose the correct options from the following at points w, x, y & z. Which of the following option is correct?

1. P(s) at w, V(s) at x, P(t) at y, V(t) at z, s and t initially 0
2. P(s) at w, V(t) at x, P(t) at y, V(s) at z, s initially 1 and t initially 0
3. P(s) at w, V(t) at x, P(t) at y, V(s) at z, s and t initially 0
4. P(s) at w, V(s) at x, P(t) at y, V(t) at z, s initially 1 and t initially 1

Problems with semaphores

semaphores provide a powerful tool for enforcing mutual exclusion and coordinate processes

But wait(S) and signal(S) are scattered among several processes. Hence, difficult to understand their effects

Usage must be correct in all the processes

One bad (or malicious) process can fail the entire collection of processes

Producer-Consumer problem

- It is one of the classic problems of synchronization
- Producer produces an item and adds to a buffer of limited size(bounded buffer)
- Consumer takes out an item from buffer and consumes it.
- Buffer is a shared resource and must be used in a mutual exclusion manner by both processes.
- Producers to be prevented from adding into a full buffer.
- Consumers to be stopped from taking out an item from an empty buffer

Producer-Consumer problem

There is a buffer of n slots and each slot is capable of storing one unit of data.

There are two processes running, namely, Producer and Consumer, which are operating on the buffer.

- The producer tries to insert data into an empty slot of the buffer.
- The consumer tries to remove data from a filled slot in the buffer.
- The Producer must not insert data when the buffer is full.
- The Consumer must not remove data when the buffer is empty.
- The Producer and Consumer should not insert and remove data simultaneously.

Producer-Consumer problem

General Situation:

- One or more producers are generating data and placing these in a buffer
- One or more consumers are taking items out of the buffer

The Problem:

- Only one producer or consumer may access the buffer at any one time
- Ensure that the producer can't add data into full buffer and consumer can't remove data from an empty buffer

Producer-Consumer problem

- To solve this problem, need two counting semaphores – Full and Empty.
- “Full” keeps track of number of items in the buffer at any given time and
- “Empty” keeps track of number of unoccupied slots.
- **Initialization of semaphores –**
  mutex = 1
  Full = 0 // Initially, all slots are empty. Thus full slots are 0
  Empty = n // All slots are empty initially

Solution to producer-consumer problem using semaphore

With a bounded buffer

The bounded buffer producer problem assumes that there is a fixed buffer size.

In this case, the consumer must wait if the buffer is empty and the producer must wait if the buffer is full.

Initialization

```
char item;              //could be any data type
char    buffer[n];
semaphore full = 0;     //counting semaphore for full slots
semaphore empty = n;    //counting semaphore for empty slots
semaphore mutex = 1;    //binary semaphore  for mutual exclusion of buffer
char  nextp, nextc;
```

Solution to producer-consumer problem using semaphore

Producer Process

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

Consumer Process

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

Producer-Consumer problem

Producer

```
do {
  wait (empty); // wait until empty>0
                and then decrement 'empty'
  wait (mutex); // acquire lock
  /* add data to buffer */
  signal (mutex); // release lock
  signal (full);  // increment 'full'
} while(TRUE)
```

Consumer

```
do {
  wait (full); // wait until full>0 and
                then decrement 'full'
  wait (mutex); // acquire lock
  /* remove data from buffer */
  signal (mutex); // release lock
  signal (empty);  // increment 'empty'
} while(TRUE)
```

Producer-Consumer problem

- When producer produces an item then the value of “empty” is reduced by 1 because one slot will be filled now.
- The value of mutex is also reduced to prevent consumer to access the buffer.
- Now, the producer has placed the item and thus the value of “full” is increased by 1.
- The value of mutex is also increased by 1 because the task of producer has been completed and consumer can access the buffer.

Producer-Consumer problem

- As the consumer is removing an item from buffer, therefore the value of “full” is reduced by 1 and the value is mutex is also reduced so that the producer cannot access the buffer at this moment.
- Now, the consumer has consumed the item, thus increasing the value of “empty” by 1.
- The value of mutex is also increased so that producer can access the buffer now.

Reader Writer Problem

- There is a data area shared among a number of processes.
- The data area could be a file or record
- There are number of processes that only read the data area(readers) and a number of processes that only write the data area (writers).
- Conditions that must be satisfied are as follows:
  - Any number of readers may simultaneously read the file.
  - Only one writer at a time may write to the file.
  - If a writer is writing to the file, no reader may read it.

Reader Writer Problem

- Three variables are used: **mutex, wrt, readcnt** to implement solution

- **semaphore** mutex, wrt; // semaphore **mutex** is used to ensure mutual exclusion when **readcnt** is updated i.e. when any reader enters or exit from the critical section and semaphore **wrt** is used by both readers and writers

- **int** readcnt; // **readcnt** tells the number of processes performing read in the critical section, initially 0

Reader Writer Problem

- **Writer process:**

- Writer requests the entry to critical section.
- If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
- It exits the critical section.

```
void writer()
{
while(true) // writer requests for critical section
{
   wait(wrt);
      ……
    writing is performed
            ……..
  signal(wrt);
 }
}
```

Pseudo Code readers-writers

```
void reader()
{while(true)
{
   wait(mutex);
          readcount++;
        if(readcount == 1)
wait(wrt);
     signal(mutex);
              ……
          reading is performed
            ……..
    wait(mutex);
          readcount--;
        if (readcount == 0)
          signal(wrt);
     signal(mutex);
   }
 }
```

- **Reader process:**
  Reader requests the entry to critical section.
- If allowed:
  - it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the **wrt** semaphore to restrict the entry of writers if any reader is inside.
  - It then, signals mutex as any other reader is allowed to enter while others are already reading.
  - After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore “wrt” as now, writer can enter the critical section.
- If not allowed, it keeps on waiting.

Synchronization with Mutex

- Mutex allows the programmer to “lock” an object so that only one thread can access it.
- To control access to a critical section of the code, programmer is required to lock a mutex before entering into a CS and then unlock the mutex while leaving the CS.
- Mutex is like a binary semaphore, but the thread which locks the mutex, can only unlock the mutex.

Synchronization with Mutex

- Strictly speaking, a mutex is **locking mechanism** used to synchronize access to a resource. Only one task (can be a thread or process) can acquire the mutex.
- It means there is ownership associated with mutex, and only the owner can release the lock (mutex)
- Semaphore is **signaling mechanism** (“I am done, you can carry on” kind of signal).

Synchronization with Monitors

- Monitors are a synchronization construct
- Monitors contain data variables and procedures.
- Data variables cannot be directly accessed by a process
- Monitors allow only a single process to access the variables at a time.

Synchronization with Mutex

- Strictly speaking, a mutex is **locking mechanism** used to synchronize access to a resource. Only one task (can be a thread or process) can acquire the mutex.
- It means there is ownership associated with mutex, and only the owner can release the lock (mutex)
- Semaphore is **signaling mechanism** (“I am done, you can carry on” kind of signal).

Synchronization with Monitors

- Monitors are a synchronization construct
- Monitors contain data variables and procedures.
- Data variables cannot be directly accessed by a process
- Monitors allow only a single process to access the variables at a time.

Difference between Mutex and Monitor

parameters	Mutex	Monitor
Purpose	Provide mutual exclusion and ensure thread safety.	Provide higher-level synchronization and communication functionality.
Mechanism	Uses a lock to provide mutual exclusion.	Uses lock and condition variables to provide higher-level synchronization.
Notification	Mutexes do not provide notification when a resource becomes available.	Monitors can notify waiting threads when a condition becomes true.
Wait/Signal	Mutexes do not have wait/signal operations.	Monitors have wait/signal operations for waiting on and signaling condition variables.
Scope	Mutexes can be used across processes.	Monitors are typically used within a single process.
Complexity	Mutexes are simpler to use and implement.	Monitors are more complex to use and implement due to the use of condition variables.

Difference between Mutex and Semaphore

Mutex	Semaphore
A mutex is an object.	A semaphore is an integer.
Mutex works upon the locking mechanism.	Semaphore uses signaling mechanism
Operations on mutex: •Lock •Unlock	Operation on semaphore: •Wait •Signal
Mutex doesn't have any subtypes.	Semaphore is of two types: •Counting Semaphore •Binary Semaphore

Difference between Semaphore and Monitor

Features	Semaphore	Monitor
Definition	A semaphore is an integer variable that allows many processes in a parallel system to manage access to a common resource like a multitasking OS.	It is a synchronization process that enables threads to have mutual exclusion and the wait() for a given condition to become true.

Syntax	// Wait Operation
	wait(Semaphore S) {
	while (S<=0);
	S--;
	}
	// Signal Operation
	signal(Semaphore S) {
	S++;
	}
		monitor {
		//shared variable declarations
		data variables;
		Procedure P1() { ... }
		Procedure P2() { ... }
		.
		.
		.
		Procedure Pn() { ... }
		}

Basic	Integer variable	Abstract data type
Access	When a process uses shared resources, it calls the wait() method on S, and when it releases them, it uses the signal() method on S.	When a process uses shared resources in the monitor, it has to access them via procedures.
Action	The semaphore's value shows the number of shared resources available in the system.	The Monitor type includes shared variables as well as a set of procedures that operate on them.
Condition Variable	No condition variables.	It has condition variables.

Deadlocks

Finite number of resources to be distributed among a number of competing processes

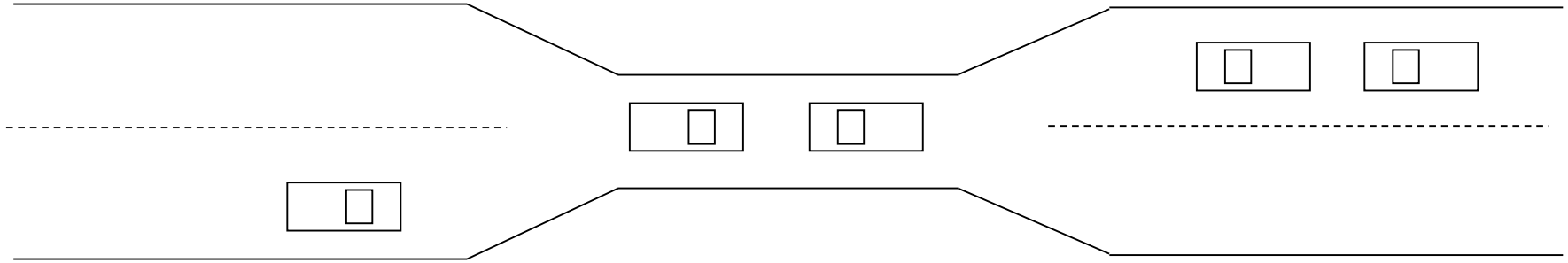
A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

The Deadlock Problem

Example

- System has 2 tape drives.
- P_1 and P_2 each hold one tape drive and each needs another one.

Bridge Crossing Example



Traffic only in one direction.

Each section of a bridge can be viewed as a resource.

If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).

Several cars may have to be backed up if a deadlock occurs.

Starvation is possible.

System Model

Resource types $R_1, R_2, \ldots, R_m$

CPU , memory space, I/O devices

Each resource type R_i has W_i instances.

Each process utilizes a resource as follows:

- request
- use
- release

Deadlock Characterization

Deadlock can arise iff four conditions hold simultaneously.

Mutual exclusion: resource must be held in a nonshareable mode

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 $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Resource-Allocation Graph

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

V is partitioned into two types:

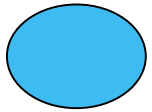
- $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
- $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.

request edge – directed edge $P_i \rightarrow R_j$

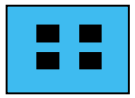
assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

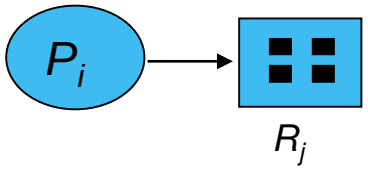
Process



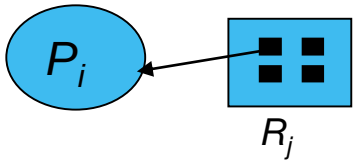
Resource Type with 4 instances



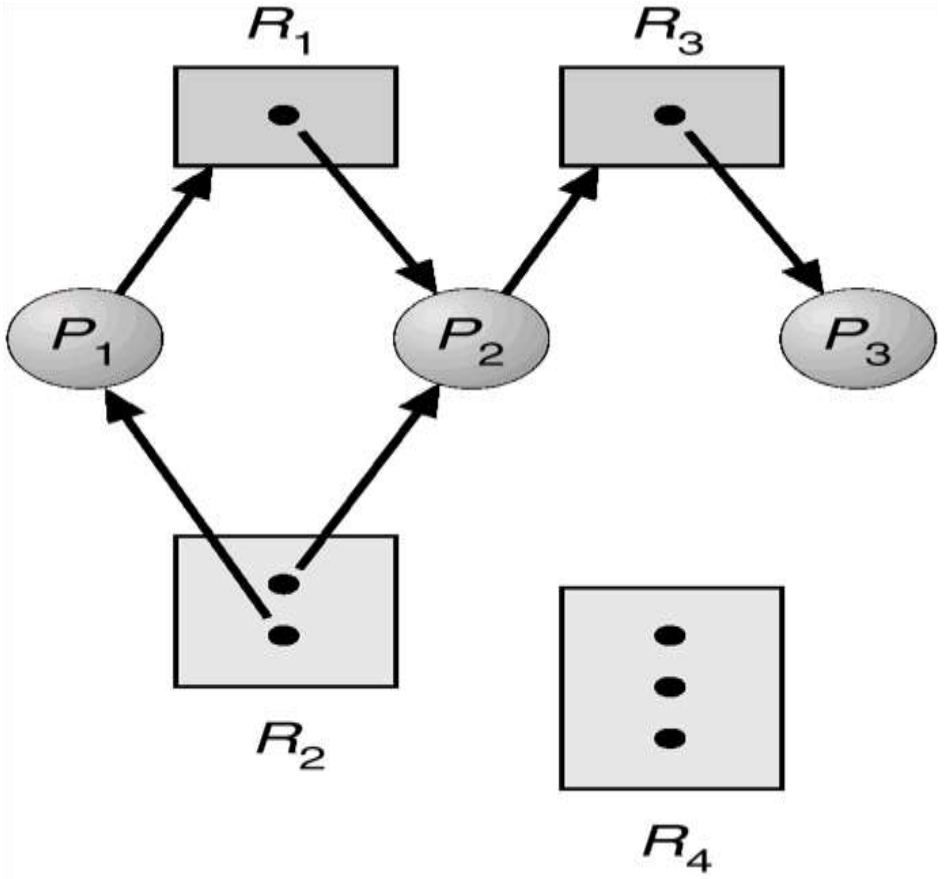
P_i requests instance of R_j



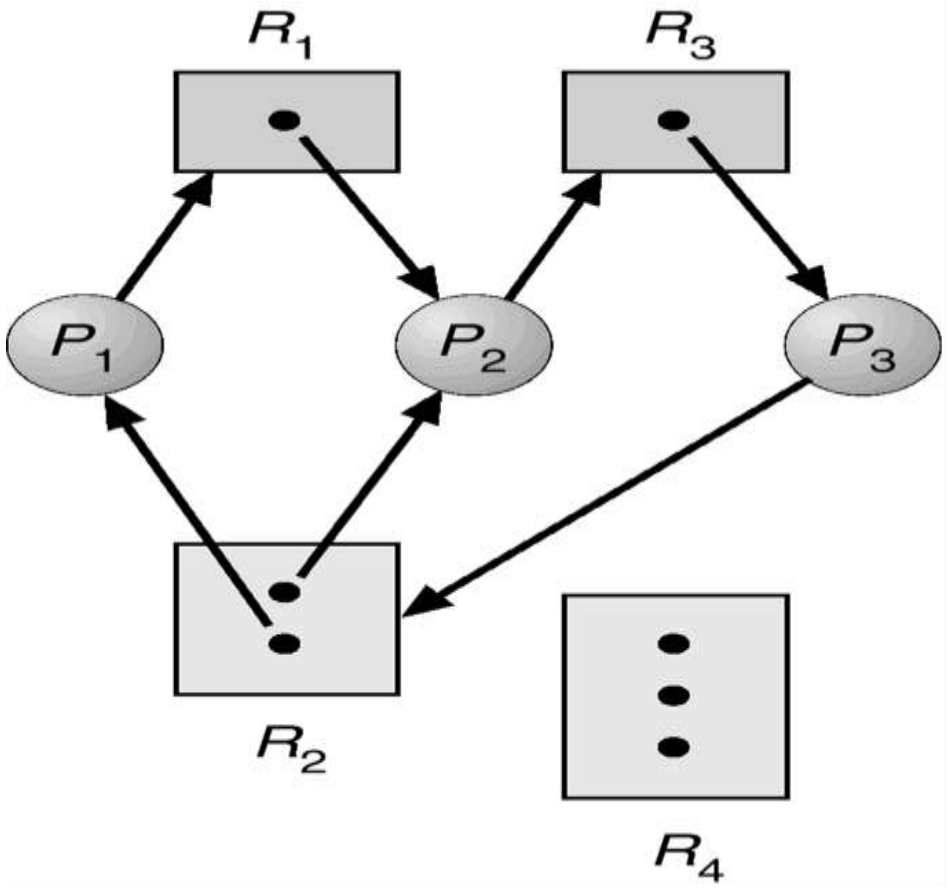
P_i is holding an instance of R_j



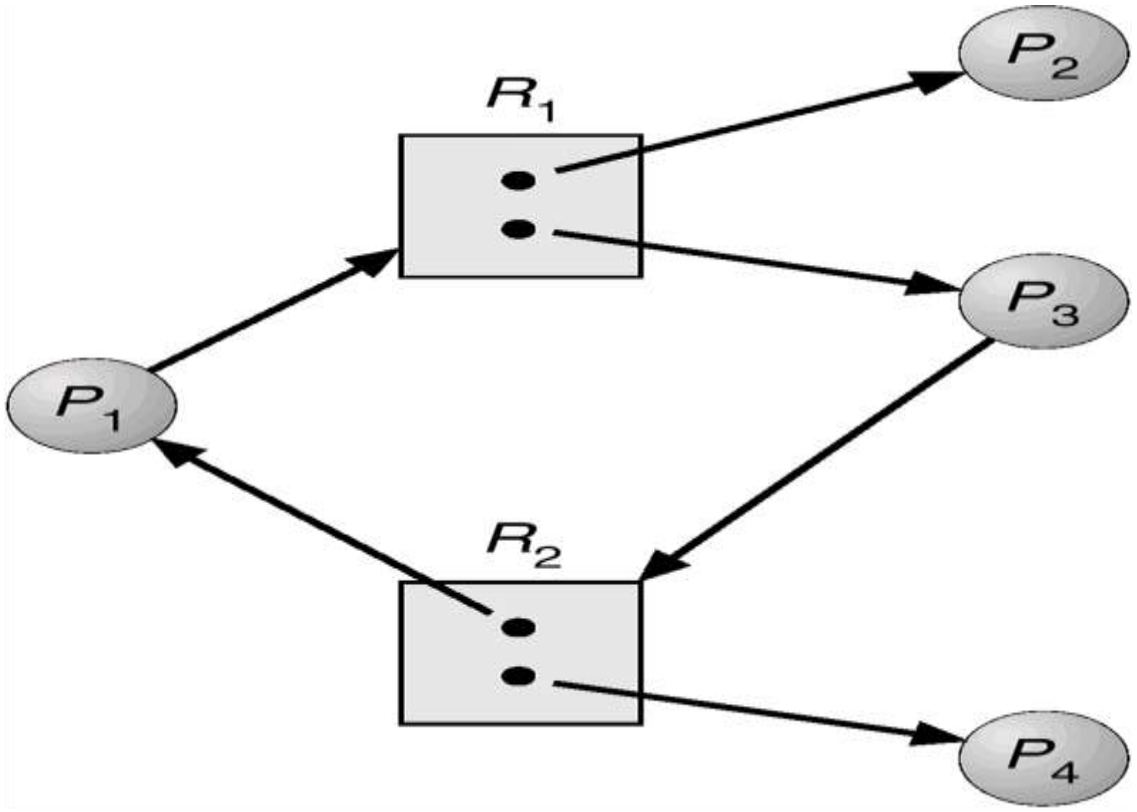
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Resource Allocation Graph With A Cycle But No Deadlock



Basic Facts

If graph contains no cycles ⇒ no deadlock.

If graph contains a cycle ⇒

- if only one instance per resource type, then deadlock.
- if several instances per resource type, possibility of deadlock.

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.
- Allow the system to enter a deadlock state and then recover
- Ignore

Deadlock Prevention

By ensuring that atleast one of the conditions cannot hold, we can prevent deadlock

1. Mutual Exclusion – some resources are inherently nonsharable eg. printer.

2. Hold and Wait –
   - Require process to request and be allocated all its resources before it begins execution
   - Low resource utilization; starvation possible.

Deadlock Prevention (Cont.)

3. No Preemption –
   - If a process that is holding some resources & requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
   - Preempted resources are added to the list of resources for which the process is waiting.
   - Process will be resumed only when it can regain its old resources, as well as the new ones that it is requesting.

4. Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Deadlock Prevention (Cont.)

Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration (eg. Disk needed before printer, so *Number assigned to printer* > *Number assigned to disk*)

$R = \{R_1, R_2, \ldots, R_m\}$ the set consisting of all resource types in the system

Assign a unique integer number to each resource type

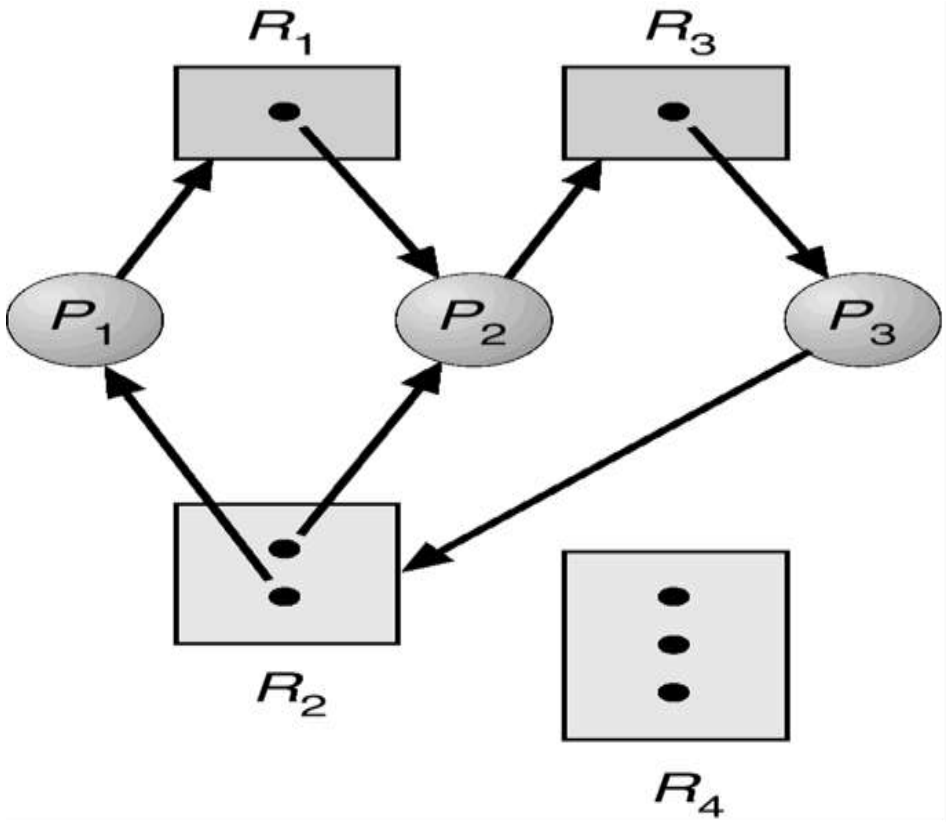
A process can initially request any number of instances of a resource type R_i

After that process can request instances of a resource type R_j iff

Number of R_j > *Number of* R_i

If several instances of some resource type are needed, a single request for all must be issued

Deadlock Prevention (Cont.)



Deadlock Avoidance

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

Simplest and most useful model requires that each process declare the ***maximum number*** **of resources** of each type that it may need.

Uses concept of **safe state**

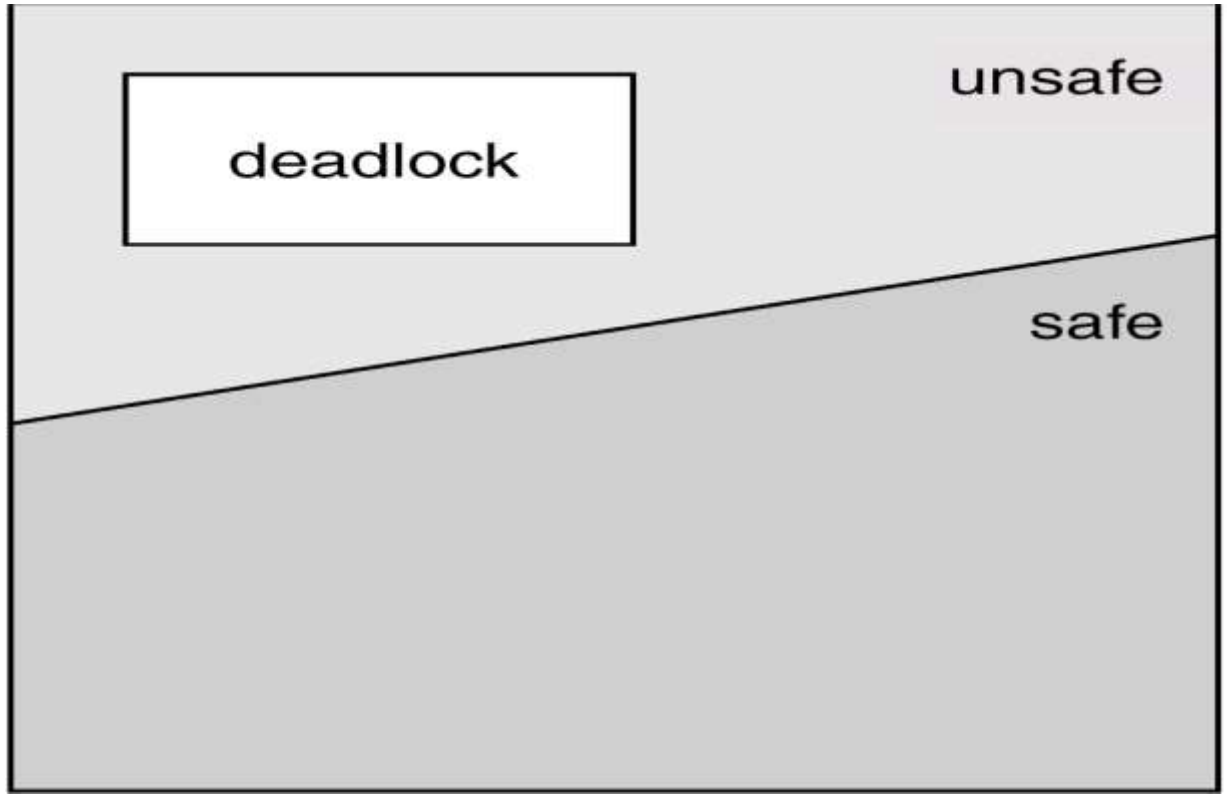
Safe State

- System is in safe state if there exists a safe sequence of all processes.
- Sequence <P_1, P_2, P_3…, P_n> is safe if for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j<i$.
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Safe State

- If no such sequence exists, then system is in unsafe state
- If a system is in safe state ⇒ no deadlocks.
- If a system is in unsafe state ⇒ possibility of deadlock.

Safe, unsafe , deadlock state spaces



Data Structures for the Banker's Algorithm

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

Available: Vector of length m. If available [j] = k, there are k instances of resource type R_j available.

Max: n x m matrix. If Max [i,j] = k, then process P_i may request at most k instances of resource type R_j.

Allocation: n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j.

Need: n x 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].

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:

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

- *Call safety algorithm*
- *If safe $\Rightarrow$ the resources are allocated to P_i.*
- *If unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is restored*

Safety Algorithm

To determine whether system is in safe state

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

   Work := *Available*

   Finish [*i*] = *false* for *i* = 1,2, …, *n.*

2. Find an *i* such that both:

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

   (b) $Need_i \leq Work$

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

3. *Work* := *Work* + $Allocation_i$

   Finish[*i*] := *true*

   go to step 2.

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

Example of Banker's Algorithm

5 processes P_0 through P_4; 3 resource types *A* (10 instances), *B* (5 instances), and *C* (7 instances).

	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	

Example (Cont.)

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

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

Example (Cont.)

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

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

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

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

For i = 0 — Step 2

$Need_0$ = 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

For i = 1 — Step 2

$Need_1$ = 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

Work = Work + $Allocation_1$ (3, 3, 2 + 2, 0, 0)

Work =	A	B	C
	5	3	2

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

For i = 2 — Step 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

For i=3 — Step 2

$Need_3$ = 0, 1, 1

Finish [3] = false and $Need_3$ < Work (0, 1, 1 < 5, 3, 2)

So P_3 must be kept in safe sequence

Step 3

Work = Work + $Allocation_3$ (5, 3, 2 + 2, 1, 1)

Work =	A	B	C
	7	4	3

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

For i = 4 — Step 2

$Need_4$ = 4, 3, 1

Finish [4] = false and $Need_4$ < Work (4, 3, 1 < 7, 4, 3)

So P_4 must be kept in safe sequence

Step 3

Work = Work + $Allocation_4$ (7, 4, 3 + 0, 0, 2)

Work =	A	B	C
	7	4	5

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

For i = 0 — Step 2

$Need_0$ = 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

Work = Work + $Allocation_0$ (7, 4, 5 + 0, 1, 0)

Work =	A	B	C
	7	5	5

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

For i = 2 — Step 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

Work = Work + $Allocation_2$ (7, 5, 5 + 3, 0, 2)

Work =	A	B	C
	10	5	7

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

Finish [i] = true for 0 ≤ i ≤ n — Step 4

Hence the system is in Safe state

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

Example (Cont.)……

At time T_1:

P_1 request (1,0,2)

Resource-Request algorithm

A B C

$Request_1$ = 1, 0, 2

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

Step 1

1, 0, 2 < 1, 2, 2

$Request_1 < Need_1$

Step 2

1, 0, 2 < 3, 3, 2

$Request_1 < Available$

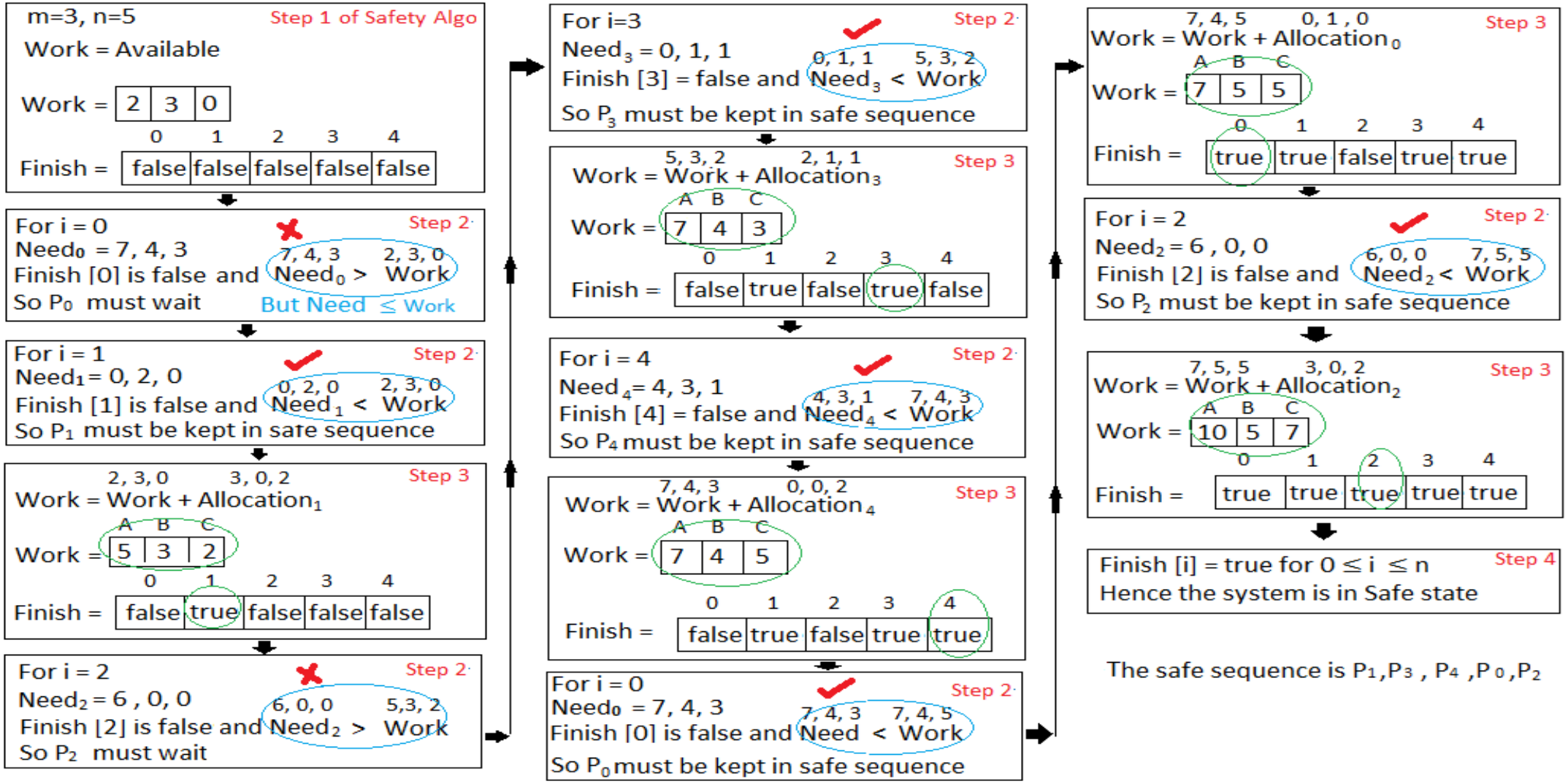
Step 3

$Available = Available - Request_1$

$Allocation_1 = Allocation_1 + Request_1$

$Need_1 = Need_1 - Request_1$

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	


m=3, n=5
Step 1 of Safety Algo
Work = Available
Work = 2 3 0
0 1 2 3 4
Finish = false false false false false
For i = 0
Need0 = 7, 4, 3
Finish [0] is false and 7, 4, 3 Need0 > 2, 3, 0 Work
So P0 must wait
But Need ≤ Work
Step 2
For i = 1
Need1 = 0, 2, 0
Finish [1] is false and 0, 2, 0 Need1 < 2, 3, 0 Work
So P1 must be kept in safe sequence
Step 2
2, 3, 0 3, 0, 2
Work = Work + Allocation1
A B C
Work = 5 3 2
0 1 2 3 4
Finish = false true false false false
Step 3
For i = 2
Need2 = 6 , 0, 0
Finish [2] is false and 6, 0, 0 Need2 > 5,3, 2 Work
So P2 must wait
Step 2
For i=3
Need3 = 0, 1, 1
Finish [3] = false and 0, 1, 1 Need3 < 5, 3, 2 Work
So P3 must be kept in safe sequence
Step 2
5, 3, 2 2, 1, 1
Work = Work + Allocation3
A B C
Work = 7 4 3
0 1 2 3 4
Finish = false true false true false
Step 3
For i = 4
Need4 = 4, 3, 1
Finish [4] = false and 4, 3, 1 Need4 < 7, 4, 3 Work
So P4 must be kept in safe sequence
Step 2
7, 4, 3 0, 0, 2
Work = Work + Allocation4
A B C
Work = 7 4 5
0 1 2 3 4
Finish = false true false true true
Step 3
For i = 0
Need0 = 7, 4, 3
Finish [0] is false and 7, 4, 3 Need < 7, 4, 5 Work
So P0 must be kept in safe sequence
Step 2
7, 4, 5 0, 1 , 0
Work = Work + Allocation0
A B C
Work = 7 5 5
0 1 2 3 4
Finish = true true false true true
Step 3
For i = 2
Need2 = 6 , 0, 0
Finish [2] is false and 6, 0, 0 Need2 < 7, 5, 5 Work
So P2 must be kept in safe sequence
Step 2
7, 5, 5 3, 0, 2
Work = Work + Allocation2
A B C
Work = 10 5 7
0 1 2 3 4
Finish = true true true true true
Step 3
Finish [i] = true for 0 ≤ i ≤ n
Hence the system is in Safe state
Step 4
The safe sequence is P1,P3 , P4 ,P0,P2

Example (Cont.): P_1 request (1,0,2)

Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) → *true*

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

Executing safety algorithm shows that sequence <P_1, P_3, P_4, P_0, P_2> satisfies safety requirement.

Example (Cont.)

At time T_2: Can request for (3,3,0) by P_4 be granted?

At time T_3: Can request for (0,2,0) by P_0 be granted?

2. Example of Banker's Algorithm

Consider the following snapshot of a system:

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

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 process P_1 arrives for (0,4,2,0), can the request be granted immediately?

Deadlock Detection

Allow system to enter deadlock state

- Detection algorithm
- Recovery scheme

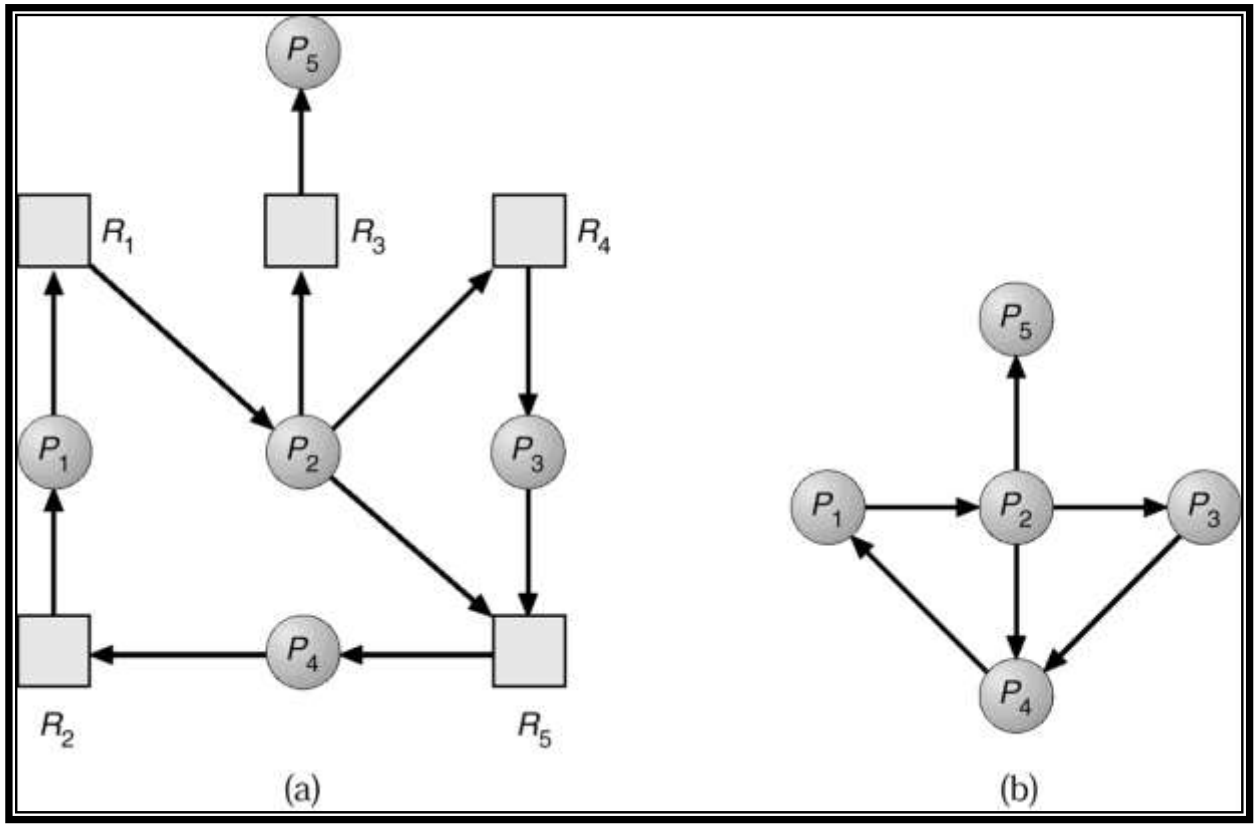
Single Instance of Each Resource Type

Maintain *wait-for* graph

- Nodes are processes.
- $P_i \rightarrow P_j$ if P_i is waiting for P_j.

Periodically invoke an algorithm that searches for a cycle 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 process.

Request: An *n* x *m* matrix indicates the current request of each process. If *Request* $[i_j] = k$, then process P_i is requesting *k* more instances of resource type. R_j.

Detection Algorithm

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

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

Detection Algorithm (Cont.)

3. *Work* = *Work* + *Allocation*$_i$
   Finish[*i*] = *true*
   go to step 2.

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

   Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked.

Example of Detection Algorithm

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

Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
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	

Sequence $<P_0, P_2, P_3, P_1, P_4>$ will result in *Finish*[*i*] = true for all *i*.

Example (Cont.)

P_2 requests an additional instance of type *C*.

	Request
	A B C
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

State of system?

- Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests.
- Deadlock exists, consisting of processes P_1, P_2, P_3, and 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 processes “caused” the deadlock.

Recovery from Deadlock: Process Termination

Abort all deadlocked processes.

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

In which order should we choose to abort?

- Priority of the process.
- How long process has computed, and how much longer to completion.
- Resources the process has used.
- Resources process needs to complete.
- How many processes will need to be terminated.
- etc…

Recovery from Deadlock: Resource Preemption

- Selecting a victim
- Rollback
- Starvation – same process may always be picked as victim, include number of rollbacks

Combined Approach to Deadlock Handling

Combine the three basic approaches

- prevention
- avoidance
- detection