Computer Organization and Software Systems

Webinar 4 – Semaphore & Deadlocks

BITS Pilani
Pilani Campus

Shamanth N

Semaphore

Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.

Semaphore ***S*** – integer variable

Can only be accessed via two indivisible (atomic) operations

- **`wait()`** and **`signal()`**

Definition of the **`wait() operation`**

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

Definition of the **`signal() operation`**

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

Problem 1 : Semaphores

Consider two processes A and B. Two semaphore variables S and T are used to synchronize the processes. S is initialized to 0 and T is initialized to 1. Processes are scheduled in the following order: A B A B B A A B A

What will be printed on the screen?

Process A:	Process B:

while (1)
{
wait (S) ;
print 'P';
print 'P';
signal(T);
}
	while (1) {
	wait (T) ;
	print 'I';
	print 'I';
	signal (S) ;
	}

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

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

```
Process A:
while (1)    {
        wait (S) ;
        print 'P';
        print 'P';
        signal(T);
}
```

```
Process B:
while (1)  {
        wait (T) ;
        print 'I';
        print 'I';
        signal (S) ;
}
```

Initially: S = 0, T = 1

Timeline	S	T	Print
A	0	1	
B	1	0	II
A	0	1	PP
B	1	0	II
B	1	0	
A	0	1	PP
A	0	1	
B	1	0	II
A	0	1	PP

Problem 2 : Semaphores

The following program consists of 3 concurrent processes and 3 binary semaphores. The semaphores are initialized as S0 = 1, S1 = 0, S2 = 0. Find out how many times Process P0 will print “0”. Assume the order of execution as P0, P1, P2, P0, P1.

Process P0:	Process P1:	Process P2:

while(true){
wait(S0);
printf("0");
signal(S1);
signal(S2);
}
	wait(S1);
	signal(S0);
		wait(S2);
		signal(S0);

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

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

Process P0:	Process P1:	Process P2:
while(true){ wait(S0); printf("0"); signal(S1); signal(S2); }	wait(S1); signal(S0);	wait(S2); signal(S0);

```
Initially: S0 = 1, S1 = 0, S2 = 0
```

Timeline	S0	S1	S2	Print
P0	0	1	1	0
P1	1	0	1	
P2	1	0	0	
P0	0	1	1	0
P1	1	0	1	

Problem 3: Resource Allocation Graph

A system has five processes P1 through P5 and four resource types R1 through R4.

There are 2 instances of each resource type. Given that:

P1 holds 1 instance of R1 and requests 1 instance of R4

P2 holds 1 instance of R3 and requests 1 instance of R2

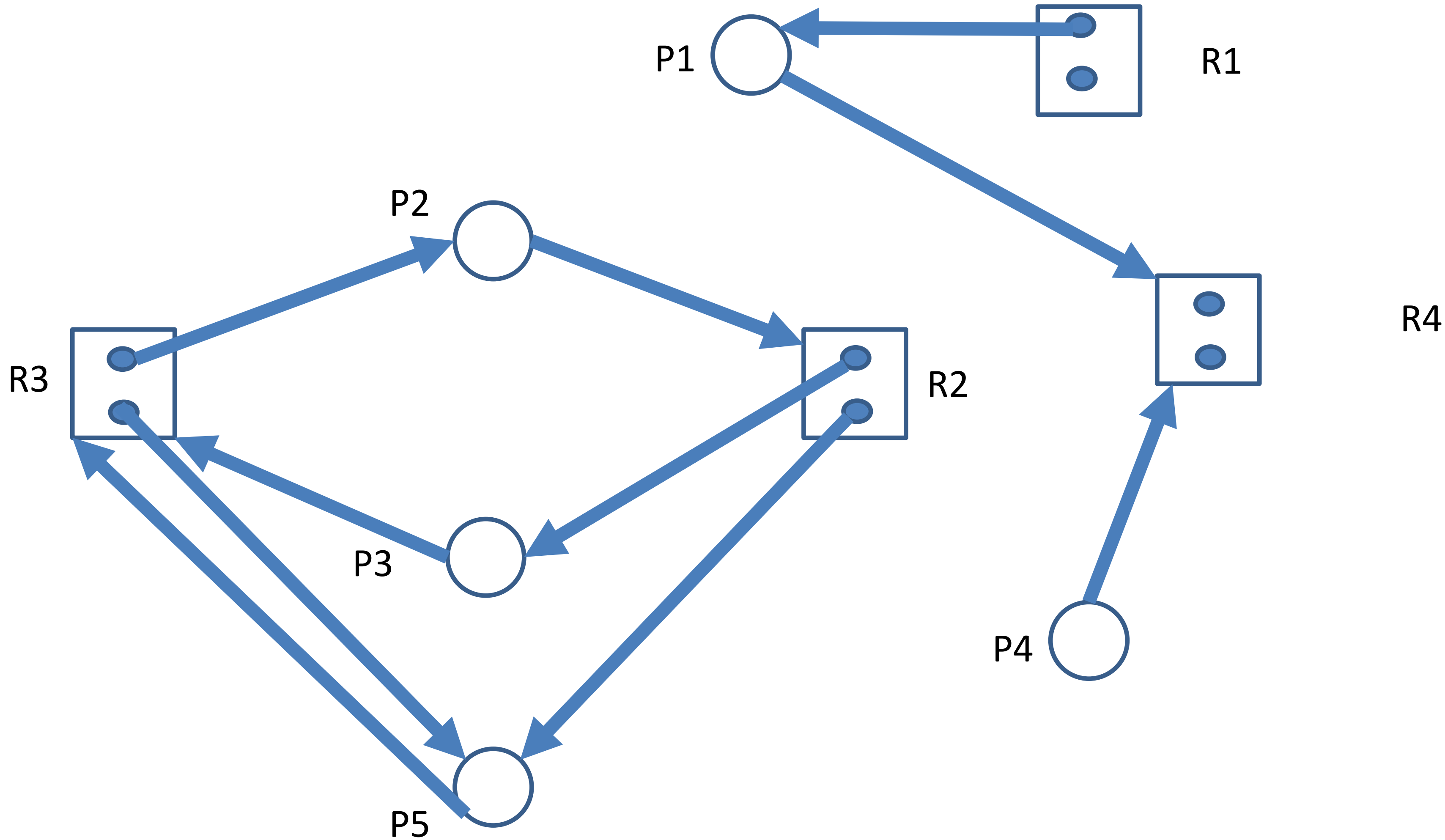
P3 holds 1 instance of R2 and requests 1 instance of R3

P4 requests 1 instance of R4

P5 holds 1 instance of R3 and 1 instance of R2, and requests 1 instance of R3

Show the resource graph for this state of the system. Is the system in deadlock, and if so, which processes are involved?

P1 holds 1 instance of R1 and requests 1instance of R4
P2 holds 1 instance of R3 and requests 1 instance of R2
P3 holds 1 instance of R2 and requests 1 instance of R3
P4 requests 1 instance of R4
P5 holds 1 instance of R3 and 1 instance of R2, and requests 1 instance of R3



Banker's Safety Algorithm

1. Let ***Available*** and ***Finish*** be vectors of length *m* and *n respectively*, where m and n represents number of processes and resources respectively. Initialize:

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

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

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

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

Problem 4: Banker's Algorithm

Apply Banker's algorithm for the following and find out whether the system is in safe state or not.

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

Need Matrix

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

Need = Max - Allocation

	A	B	C	D
P0	2	0	1	1
P1	1	6	5	0
P2	1	1	0	2
P3	1	0	2	0
P4	1	4	4	4

Need	A	B	C	D
P0	2	0	1	1
P1	1	6	5	0
P2	1	1	0	2
P3	1	0	2	0
P4	1	4	4	4

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

Step 1:
Work = Available = 3 2 1 1

	0	1	2	3	4
Finish =	F	F	F	F	F

Step 2:
For i=0
Finish[0] = F & Need[0] <= Work
2 0 1 1 <= 3 2 1 1 (T)
P0 -> Safe sequence

Step 2:
For i=1
Finish[1] = F & Need[1] <= Work
1 6 5 0 <= 7 2 1 2 (F)
P1 -> Wait

Step 3:
Work = Work + Allocation[0]
Work = 3 2 1 1 + 4 0 0 1 = 7 2 1 2

	0	1	2	3	4
Finish =	T	F	F	F	F

Step 2:
For i=2
Finish[2] = F & Need[2] <= Work
1 1 0 2 <= 7 2 1 2 (T)
P2 -> Safe sequence

Need	A	B	C	D
P0	2	0	1	1
P1	1	6	5	0
P2	1	1	0	2
P3	1	0	2	0
P4	1	4	4	4

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

Step 3:
Work = Work + Allocation[2]
Work = 7 2 1 2 + 1 2 5 4 = 8 4 6 6

	0	1	2	3	4
Finish =	T	F	T	F	F

Step 2:
For i=3
Finish[3] = F & Need[3] <= Work
1 0 2 0 <= 8 4 6 6 (T)
P3 -> Safe sequence

Step 2:
For i=4
Finish[4] = F & Need[4] <= Work
1 4 4 4 <= 8 10 9 9 (T)
P4 -> Safe sequence

Step 3:
Work = Work + Allocation[3]
Work = 8 4 6 6 + 0 6 3 3 = 8 10 9 9

	0	1	2	3	4
Finish =	T	F	T	T	F

Step 3:
Work = Work + Allocation[4]
Work = 8 10 9 9 + 0 2 1 2 = 8 12 10 11

	0	1	2	3	4
Finish =	T	F	T	T	T

Step 2:
For i=1
Finish[1] = F & Need[1] <= Work
1 6 5 0 <= 8 12 10 11 (T)
P1-> Safe sequence

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

Need	A	B	C	D
P0	2	0	1	1
P1	1	6	5	0
P2	1	1	0	2
P3	1	0	2	0
P4	1	4	4	4

Step 3:
Work = Work + Allocation[1]
Work = 8 12 10 11 + 1 1 0 0 = 9 13 10 11

	0	1	2	3	4
Finish =	T	T	T	T	T

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

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:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- 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

Problem 5: Resource- Request Algorithm

Apply Banker's algorithm for the following and find out whether the system is in safe state or not. If process P1 requests for additional resources (1,2,0,0) will the system go to unsafe state or not. Check using Resource-Request algorithm.

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

Resource-Request

Step 1:
Request[1] <= Need[1]
1 2 0 0 <= 1 6 5 0 (T)

Step 2:
Request[1] <= Available
1 2 0 0 <= 3 2 1 1 (T)

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

Need	A	B	C	D
P0	2	0	1	1
P1	1	6	5	0
P2	1	1	0	2
P3	1	0	2	0
P4	1	4	4	4

Step 3:
Available = Available – Request[1] = 3 2 1 1 – 1 2 0 0 = 2 0 1 1
Allocation[1] = Allocation[1]+Request[1] = 1 1 0 0 + 1 2 0 0 = 2 3 0 0
Need[1] = Need[1] – Request[1] = 1 6 5 0 – 1 2 0 0 = 0 4 5 0

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

Need	A	B	C	D
P0	2	0	1	1
P1	0	4	5	0
P2	1	1	0	2
P3	1	0	2	0
P4	1	4	4	4

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

Step 1:

Work = Available = 2 0 1 1

Finish =

0	1	2	3	4
F	F	F	F	F

Step 2:

For i=0

Finish[0] = F & Need[0] <= Work

2 0 1 1 <= 2 0 1 1 (T)

P0 -> Safe sequence

Step 3:

Work = Work + Allocation[0]

Work = 2 0 1 1 + 4 0 0 1 = 6 0 1 2

Finish =

0	1	2	3	4
T	F	F	F	F

Step 2:

For i=1

Finish[1] = F & Need[1] <= Work

0 4 5 0 <= 6 0 1 2 (F)

P1 -> Wait

Step 2:

For i=2

Finish[2] = F & Need[2] <= Work

1 1 0 2 <= 6 0 1 2 (F)

P2 -> Wait

Need	A	B	C	D
P0	2	0	1	1
P1	0	4	5	0
P2	1	1	0	2
P3	1	0	2	0
P4	1	4	4	4

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

Step 2:
For i=3
Finish[3] = F & Need[3] <= Work
1 0 2 0 <= 6 0 1 2 (F)
P3 -> Wait

Step 2:
For i=4
Finish[4] = F & Need[4] <= Work
1 4 4 4 <= 6 0 1 2 (F)
P4 -> Wait

The system is in unsafe state

Deadlock 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 $\boldsymbol{Allocation_i} \neq \mathbf{0}$, then ***Finish*[i] = *false***; otherwise, ***Finish*[i] = *true***

2. Find an index ***i*** such that both:
   (a) ***Finish*[*i*] == *false***
   (b) $\boldsymbol{Request_i} \leq \boldsymbol{Work}$

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

Problem 6: Deadlock Detection Algorithm

Consider the following snapshot of the system with four processes P0 to P3 and 3 resource types A(5 Instances), B(3Instances), and C(8 Instances).

Answer the following questions with reference to Deadlock Detection Algorithm.

a. Check whether the system is in deadlock or not.
b. If the system is in safe state then give safe sequence or else provide the process number(s) which is causing the deadlock.

Process	ALLOCATION				REQUEST			AVAILABLE		
	A	B	C		A	B	C	A	B	C
P0	1	0	2		0	0	1	0	0	0
P1	2	1	1		1	0	2			
P2	1	0	3		0	0	0			
P3	1	2	2		3	3	0			

Process	ALLOCATION			REQUEST		
	A	B	C	A	B	C
P0	1	0	2	0	0	1
P1	2	1	1	1	0	2
P2	1	0	3	0	0	0
P3	1	2	2	3	3	0

Initialization:

Work = Available = 0 0 0

	0	1	2	3
Finish =	F	F	F	F

Step 2:

For i=0

Finish[0] = F & Request[0] <= Work

0 0 1 <= 0 0 0 (F)

P0 -> Wait

Step 2:

For i=1

Finish[1] = F & Request[1] <= Work

1 0 2 <= 0 0 0 (F)

P1 -> Wait

Step 2:

For i=2

Finish[2] = F & Request[2] <= Work

0 0 0 <= 0 0 0 (T)

P0 -> Wait

Step 3:

Work = Work + Allocation[2]

Work = 0 0 0 + 1 0 3 = 1 0 3

	0	1	2	3
Finish =	F	F	T	F

Process	ALLOCATION			REQUEST		
	A	B	C	A	B	C
P0	1	0	2	0	0	1
P1	2	1	1	1	0	2
P2	1	0	3	0	0	0
P3	1	2	2	3	3	0

Step 2:
For i=3
Finish[3] = F & Request[3] <= Work
3 3 0 <= 1 0 3 (F)
P3 -> Wait

Step 2:
For i=0
Finish[0] = F & Request[0] <= Work
0 0 1 <= 1 0 3 (T)
P0 -> Safe sequence

Step 2:
For i=1
Finish[1] = F & Request[1] <= Work
1 0 2 <= 2 0 5 (T)
P1 -> Safe sequence

Step 3:
Work = Work + Allocation[0]
Work = 1 0 3 + 1 0 2 = 2 0 5

Finish =

0	1	2	3
T	F	T	F

Step 3:
Work = Work + Allocation[1]
Work = 2 0 5 + 2 1 1 = 4 1 6

Finish =

0	1	2	3
T	T	T	F

Step 2:

For i=3

Finish[3] = F & Request[3] <= Work

3 3 0 <= 4 1 6 (F)

P3 -> Wait

Process	ALLOCATION			REQUEST		
	A	B	C	A	B	C
P0	1	0	2	0	0	1
P1	2	1	1	1	0	2
P2	1	0	3	0	0	0
P3	1	2	2	3	3	0

The system is in unsafe state and Process P3 causes deadlock

Questions ?

Thank you.

BITS Pilani
Pilani Campus