COMPUTER ORGANIZATION AND SOFTWARE SYSTEMS

WEBINAR 4 – BUDDY SYSTEMS , DEADLOCKS & SEMAPHORES

Vinayaka S P

BITS Pilani
Pilani Campus

Buddy System

- The **buddy system** is a memory allocation and management algorithm.
- It manages memory in **power of two increments**.
- Splitting memory into halves and to try to give a best fit.

- Provides two operations:
  - Allocate(2^k) : Allocates a block of 2^k and marks it as allocated
  - Free(A): Marks the previously allocated block A as free and merge it with other blocks to form a larger block.
- Algorithm: Assume that a process A of size "X" needs to be allocated
  - **If $2^{K-1} < X <= 2^K$:** Allocate the entire block
  - **Else:** Recursively divide the block equally and test the condition at each time, when it satisfies, allocate the block.

Problem

Consider a memory block of 16K. Perform the following:

Allocate (A: 3.5K)

Allocate (B: 1.2K)

Allocate (C: 1.3K)

Allocate (D: 1.9K)

Allocate (E: 3.2K)

Free (C)

Free (B)

Allocate (F: 1.6K)

Allocate (G: 1.8K)

Consider a memory block of 16K. Perform the following:

Allocate (A: 3.5K)
Allocate (B: 1.2K)
Allocate (C: 1.3K)
Allocate (D: 1.9K)
Allocate (E: 3.2K)
Free (C)
Free (B)
Allocate (F: 1.6K)
Allocate (G: 1.8K)

- 16K Memory Block

16K

- Allocate (A: 3.5K)

4k (A)	4K	8K

- Allocate (B: 1.2K)

4k (A)	2K(B)	2K	8K

- Allocate (C: 1.3K)

4k (A)	2K(B)	2K(C)	8K

- Allocate (D: 1.9K)

4k (A)	2K(B)	2K(C)	2K(D)	2K	4K

Consider a memory block of 16K. Perform the following:

Allocate (A: 3.5K)
Allocate (B: 1.2K)
Allocate (C: 1.3K)
Allocate (D: 1.9K)
Allocate (E: 3.2K)
Free (C)
Free (B)
Allocate (F: 1.6K)
Allocate (G: 1.8K)

- Allocate (E: 3.2K)

4k (A)	2K(B)	2K(C)	2K(D)	2K	4K(E)

- Free (C)

4k (A)	2K(B)	2K	2K(D)	2K	4K(E)

- Free (B)

4k (A)	2K	2K	2K(D)	2K	4K(E)

4k (A)	4K	2K(D)	2K	4K(E)

- Allocate (F: 1.6K)

4k (A)	4k	2K(D)	2K(F)	4K(E)

- Allocate (G: 1.8K)

4k (A)	2K(G)	2K	2K(D)	2K(F)	4K(E)

Problem 2: Buddy System

Consider a memory of 16 KB.
Following is the snapshot of the memory using Buddy system after allocating processes A and B.

Show the memory allocation of each of the following (provide diagram for each step).
Allocate (C: 3.3K)
Free (B)
Allocate (D : 3K)
Allocate (E : 512 B)
Allocate (F: 1K)

Advantages and Disadvantages

Advantage –

- Easy to implement a buddy system (Linux)
- Allocates block of correct size
- It is easy to merge adjacent holes
- Fast to allocate memory and de-allocating memory

Disadvantage –

- It requires all allocation unit to be powers of two
- It leads to internal fragmentation

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*** $_i$ $\leq$ ***Available***
   If no such ***i*** exists, go to step 4

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

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

Problem 3: 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				
P1				
P2				
P3				
P4				

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 1:

Work = Available = 3 2 1 1

	0	1	2	3	4
Finish[i] =	**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 3:

Work = Work + Allocation[0]

Work = 3 2 1 1 + 4 0 0 1 = 7 2 1 2

	0	1	2	3	4
Finish[i] =	**T**	**F**	**F**	**F**	**F**

Step 2:

For i=1

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

1 6 5 0 <= 7 2 1 2 (F)

P1 -> Wait

Step 2:

For i=2

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

1 1 0 2 <= 7 2 1 2 (T)

P2 -> 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 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=3
Finish[3] = F & Need[3] <= Work
1 0 2 0 <= 8 4 6 6 (T)
P3 -> Safe sequence

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

Finish[i] =

0	1	2	3	4
T	**F**	**T**	**T**	**F**

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[4]
Work = 8 10 9 9 + 0 2 1 2 = 8 12 10 11

Finish[i] =

0	1	2	3	4
T	**F**	**T**	**T**	**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 2:
For i=1
Finish[1] = F & Need[1] <= Work
1 6 5 0 <= 8 12 10 11 (T)
P1-> Safe sequence

Step 3:
Work = Work + Allocation[3]
Work = 8 12 10 11 + 1 1 0 9 = 8 13 10 20

	0	1	2	3	4
Finish[i] =	**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 4: 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

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 1:
Request[1] <= Need[1]
1 2 0 0 <= 1 6 5 0

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

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				

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

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

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 2:

For i=1

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

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

P1 -> Wait

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=2

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

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

P2 -> Wait

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

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

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 ***Allocation**$_i$* $\neq$ **0**, then
   Finish*[i] = *false; otherwise, ***Finish*[i] = *true***

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

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

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

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

	0	1	2	3
Finish =	T	F	T	F

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[1]
Work = 2 0 5 + 2 1 1 = 4 1 6

	0	1	2	3
Finish =	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

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

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);

$S0 = 1, S1 = 0, S2 = 0$

Timeline	S0	S1	S2	Print
P0				
P1				
P2				
P0				
P1				

Problem 2 : 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			
B			
A			
B			
B			
A			
A			
B			
A			

Questions ?

BITS Pilani
Pilani Campus

Thank you.