

Deadlock

Deadlock

- processes which share resources or communicate are permanently blocked -> deadlock
- if processes request resources without releasing the resources they hold, deadlock may occur

Deadlock

<u>P1</u>	<u>P2</u>
req(D);	req(T);
lock(D);	lock(T);
req(T);	req(D);
lock(T);	lock(D);
<.....>	<.....>
unlock(T);	unlock(D);
unlock(D);	unlock(T);

Deadlock potential !

Deadlock

Example: 200K memory is available for processes in a system. Assume no other processes use the memory, the following series of requests have a deadlock potential.

<u>P1</u>	<u>P2</u>
req(80K);	req(70K);
...	...
req(60K);	req(80K);
...	...

Deadlock

Example: If *receive_msg* works in blocking mode, then the following scenario has a deadlock potential.

<u>P1</u>	<u>P2</u>
receive_msg(P2);	receive_msg(P1);
...	...
send_msg(P2);	send_msg(P1);
...	...

Deadlock

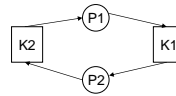
- if a resource is unavailable when requested:
 - process is blocked until resource becomes available
 - process receives an error message and tries later

Conditions for Deadlock

- mutual exclusion condition
 - only one process can use a shared resource at a time
- hold and wait condition
 - processes wait for a requested resource until it becomes available while holding onto its own resources
- no pre-emption condition
 - resources allocated to a process cannot be taken back without the process' consent
- circular wait condition
 - two or more processes wait for the other's resource while not releasing its own in a circular fashion

Conditions for Deadlock

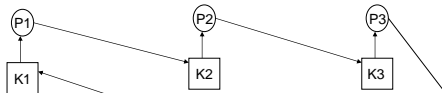
- a graph representation may be used
 - nodes in graph:
 - circle: process
 - square: resource
 - edges in graph:
 - process → resource : process requests resource
 - resource → process : resource allocated to process



Deadlock with circular wait.

Deadlock Example

P1 req(K1); lock(K1); req(K2); lock(K2); <.....> unlock(K2); unlock(K1);	P2 req(K2); lock(K2); req(K3); lock(K3); <.....> unlock(K3); unlock(K2);	P3 req(K3); lock(K3); req(K1); lock(K1); <.....> unlock(K1); unlock(K3);
--	--	--



Deadlock

- strategies used for dealing with deadlock
 - prevention: structure the system negating one of the deadlock conditions
 - detection and recovery: let deadlocks occur, detect them and take action
 - avoidance
 - don't start processes whose requests may cause a deadlock
 - don't grant requests which may cause a deadlock
 - ignore

Deadlock Avoidance

- the Banker's algorithm
 - Dijkstra, 1965
 - fixed no. of processes and resources in the system
 - system state: current allocation of resources to processes
 - state: *resource* and *free* vectors, *has* and *max_request* matrices

Banker's Algorithm

- *resource*: shows all resources in system
- *free*: shows all free resources in system
- *has*: shows the amount of each resource allocated to each process
- *max_request*: shows the maximum no. of requests a process will make during its lifetime for each type of resource

Banker's Algorithm

safe state: a state is **safe** if it is not deadlocked and there exists some scheduling order in which every process can run to completion even if all of them request their maximum no. of resources immediately.

unsafe state: such a scheduling order cannot be found

Banker's Algorithm

- when a process requests a resource, the request is granted if:
 - (resources process already has) + (resources it requests) \leq (max_request)
 - if **after** granting this request, some scheduling order in which every process can run to completion even if all of them request their maximum no. of resources immediately **still** exists

Banker's Algorithm – Example One Type of Resource

Example 1: Total no. of resources = 12

Safe state v

- give 2 resources to B
- B releases 6 resources when it is completed
- A and C can run to completion.

Process	Has	Max_Request	Remaining_Request
A	1	4	3
B	4	6	2
C	5	8	3

Allocated=10 Free=2

Banker's Algorithm – Example One Type of Resource

Example 2: Total no. of resources = 12

Unsafe state X

- potential deadlock

Process	Has	Max_Request	Remaining_Request
A	8	10	2
B	2	5	3
C	1	3	2

Allocated=11 Free=1

Banker's Algorithm – Example One Type of Resource

Example 3: Total no. of resources = 12

- system is in safe state of example 1.
- C requests one more resource

Q: Is this request granted?

- update the system state **as if** the request has been granted
- check if the new state is safe

Unsafe state X

- Request is NOT granted!

Process	Has	Max_Request	Remaining_Request
A	1	4	3
B	4	6	2
C	6	8	2

Allocated=11 Free=1

Banker's Algorithm – Example One Type of Resource

Example 4: Total no. of resources = 12

- system is in safe state of example 1.
- B requests one more resource

Q: Is this request granted?

Banker's Algorithm – One Type of Resource

```

free = total_resources;
for i= 1 to no_of_processes do
begin
    free = free - has[i];
    may_not_finish[i] = TRUE;
    remaining_request[i] = max_request[i] - has[i];
end;
flag = TRUE;
while (flag) do
begin
    flag = FALSE;
    for i=1 to no_of_processes do
    begin
        if ((may_not_finish[i]) AND
            (remaining_request[i] ≤ free)) then
        begin
            may_not_finish[i] = FALSE;
            free = free + has[i];
            flag = TRUE;
        end;
    end;
end;
if (free EQ total_resources) then
    ----- SAFE STATE -----
else
    ----- UNSAFE STATE -----

```

Banker's Algorithm – Multiple Type of Resources

Example

Max_Request Matrix

	K1	K2	K3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Has Matrix

	K1	K2	K3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Remaining_Request Matrix

	K1	K2	K3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

K1	K2	K3
9	3	6

Resource Vector

K1	K2	K3
0	1	1

Free Vector

if processes are executed in the order P2, P3, (P1 or P4) all may run to completion

⇒ SAFE STATE ✓

Banker's Algorithm – Multiple Type of Resources

Example

Q: When the system is in the safe state given in the previous slide, if P3 requests one more K3, will this request be granted?

A: Granting this request will cause an unsafe state, so it will not be granted.

Application of the Banker's Algorithm

- Are there any rows in the *remaining_request* matrix \leq *free* vector?
if not: unsafe state
- Assume that the process corresponding to the row chosen above, requests all the resources it needs and finishes.
- Mark the process as completed and add all its resources to the *free* vector
- Repeat steps 1 and 2 until either all processes are marked as "completed" (safe state) or until a deadlock occurs (unsafe state)

Banker's Algorithm

- to be able to apply the algorithm:
 - all processes must declare all their resource requests when they start execution
 - number of resources and processes must be fixed
 - order of process execution should not be important
 - any process holding a resource should not exit without releasing all its resources
- the algorithm grants or rejects requests based on the worst case scenario
 - not all rejected requests would cause a deadlock (inefficient use of resources)
- the algorithm is executed each time a request is made (high cost)

Deadlock Detection

- not as restrictive as avoidance strategies
- all requests are granted
- system is checked for deadlock periodically
 - if deadlock is detected:
 - terminate all deadlocked processes
 - or terminate processes one by one until deadlock is removed
 - or ...
- has lower cost since it is not executed on each request
- provides more efficient resource use
- period for checking for deadlock is set based on the frequency of deadlock on the system

Deadlock Detection

- *Has* matrix and *Free* vector used.
- *Q Request* matrix defined. q_{ij} shows the amount of j type resources process i requests
- algorithm determines processes which are not deadlocked and marks them
- initially all processes are unmarked

Deadlock Detection Steps

- **Step 1:** Mark all processes which correspond to rows with all 0's in the *Has* matrix
- **Step 2:** Create a temporary *W* vector to represent the *Free* vector
- **Step 3:** Find an i for which all corresponding values in the *Q* matrix are LE those in the *W* vector (P_i must be unmarked).

$$Q_{ik} \leq W_k, 1 \leq k \leq m$$

Deadlock Detection Steps

- **Step 4:** Terminate algorithm if no such row exists
- **Step 5:** If such a row exists, mark the i th process and add the corresponding row in the *Has* matrix to the *W* vector

$$W_k = W_k + A_{ik}, 1 \leq k \leq m$$
- **Step 6:** Return to step 3.

Deadlock Detection

- when algorithm terminates, if there are unmarked processes \Rightarrow Deadlock exists
 – unmarked processes are deadlocked
- algorithm only detects if a deadlock exists in the current state or not

Deadlock Detection

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request Matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation Matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource Vector

R1	R2	R3	R4	R5
0	0	0	0	1

Available Vector

- 1) Mark P4
- 2) $W = (0 \ 0 \ 0 \ 0 \ 1)$
- 3) P3's request LE $W \Rightarrow$ Mark P3
 $W = W + (0 \ 0 \ 0 \ 1 \ 0) = (0 \ 0 \ 0 \ 1 \ 1)$
- 4) No other such processes can be found \Rightarrow Terminate algorithm

P1 and P2 remain unmarked \Rightarrow deadlocked!

After Deadlock Detection

- terminate all deadlocked processes
- roll-back all deadlocked processes to a previous control point in time and resume from there
 - same deadlock may occur again
- terminate deadlocked processes one by one until deadlock no longer exists
- remove allocated resources from deadlocked processes one by one until deadlock no longer exists

Which Deadlocked Process to Select for Termination

- select the one which has used the least amount of CPU
- select the one which has the longest expected time to completion
- select the one which has the least no of allocated resources
- select the one with the lowest priority
-