Concurrency: Deadlock and Starvation

Rashmi Pande TA, CSE

Deadlock

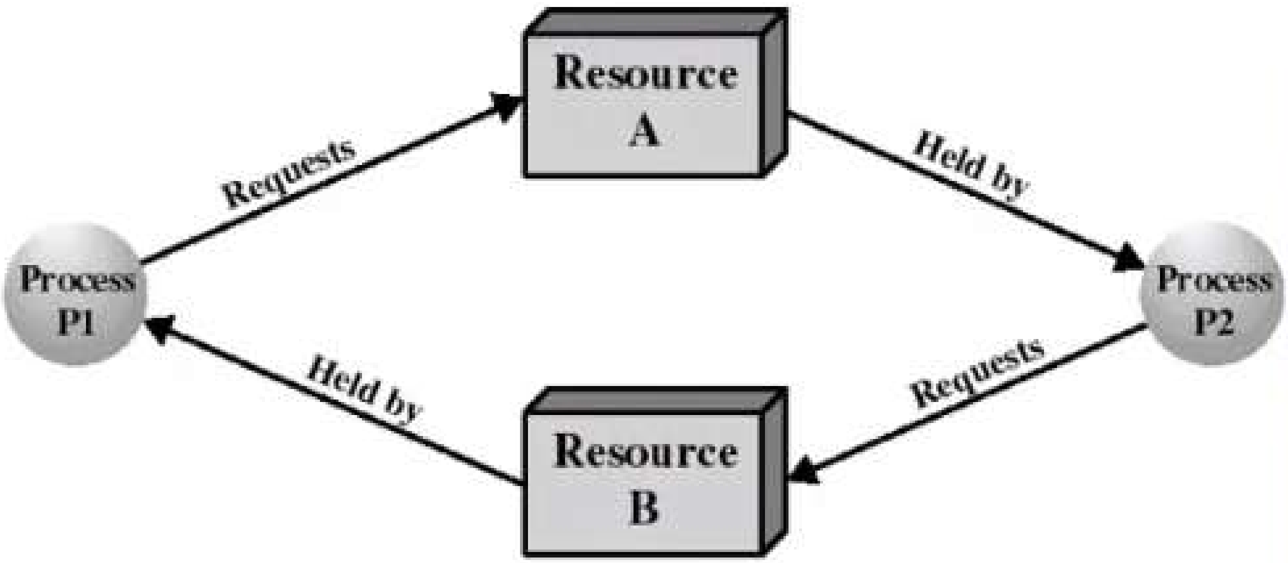
- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
- Involves conflicting needs for resources by two or more processes
- There is no satisfactory solution in the general case
- Some OS (ex: Unix SVR4) ignore the problem and pretend that deadlocks never occur...

The Conditions for Deadlock

- These 3 conditions of policy must be present for a deadlock to be possible:
  - 1: Mutual exclusion
    - only one process may use a resource at a time
  - 2: Hold-and-wait
    - a process may hold allocated resources while awaiting assignment of others
  - 3: No preemption
    - no resource can be forcibly removed from a process holding it

The Conditions for Deadlock

- We also need the occurrence of a particular sequence of events that result in:
  - 4: Circular wait
    - a closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain



The Conditions for Deadlock

- Deadlock occurs if and only if the circular wait condition is unresolvable
- The circular wait condition is unresolvable when the first 3 policy conditions hold

- Thus the 4 conditions taken together constitute necessary and sufficient conditions for deadlock

Methods for handling deadlocks

- Deadlock prevention
  - disallow 1 of the 4 necessary conditions of deadlock occurrence
- Deadlock avoidance
  - do not grant a resource request if this allocation might lead to deadlock
- Deadlock Ignore
- Deadlock detection
  - always grant resource request when possible. But periodically check for the presence of deadlock and then recover from it

Deadlock Prevention

- The OS is design in such a way as to exclude a priori the possibility of deadlock
- Indirect methods of deadlock prevention:
  - to disallow one of the 3 policy conditions
- Direct methods of deadlock prevention:
  - to prevent the occurrence of circular wait

Indirect methods of deadlock prevention

- Mutual Exclusion
  - cannot be disallowed
  - ex: only 1 process at a time can write to a file
- Hold-and-Wait
  - can be disallowed by requiring that a process request all its required resources at one time
  - block the process until all requests can be granted simultaneously
- process may be held up for a long time waiting for all its requests
- resources allocated to a process may remain unused for a long time. These resources could be used by other processes
- an application would need to be aware of all the resources that will be needed

Indirect methods of deadlock prevention

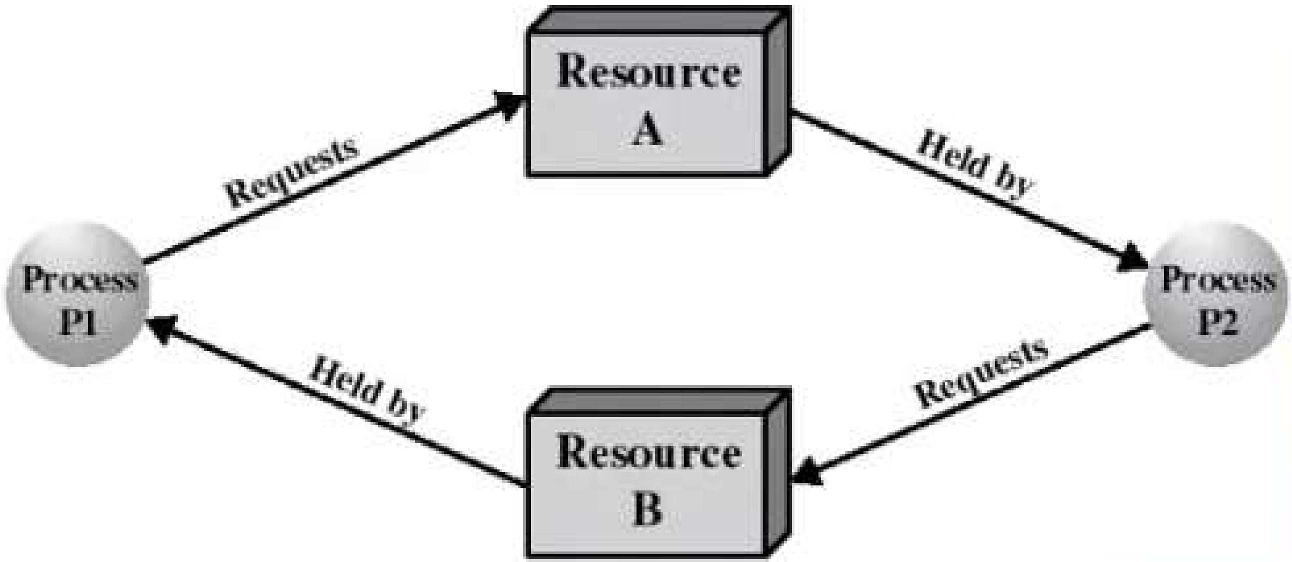
- No preemption
  - Can be prevented in several ways. But whenever a process must release a resource who's usage is in progress, the state of this resource must be saved for later resumption.
  - Hence: practical only when the state of a resource can be easily saved and restored later, such as the processor.

Direct methods of deadlock prevention

- A protocol to prevent circular wait:
  - define a strictly increasing linear ordering O() for resource types. Ex:
    - R1: tape drives: O(R1) = 2
    - R2: disk drives: O(R2) = 4
    - R3: printers: O(R3) = 7
  - A process initially request a number of instances of a resource type, say Ri. A single request must be issued to obtain several instances.
  - After that, the process can request instances for resource type Rj if and only if O(Rj) > O(Ri)

Prevention of circular wait

- Circular wait cannot hold under this protocol. Proof:
  - Processes {P0, P1..Pn} are involved in circular wait iff Pi is waiting for Ri which is held by Pi+1 and Pn is waiting for Rn held which is held by P0 (circular waiting)



Prevention of circular wait

- under this protocol, this means:
  - $O(R0) < O(R1) < .. < O(Rn) < O(R0)$ impossible!
- This protocol prevents deadlock but will often deny resources unnecessarily (inefficient) because of the ordering imposed on the requests

Deadlock Prevention: Summary

- We disallow one of the 3 policy conditions or use a protocol that prevents circular wait

- This leads to inefficient use of resources and inefficient execution of processes

Deadlock Avoidance

- We allow the 3 policy conditions but make judicious choices to assure that the deadlock point is never reached
- Allows more concurrency than prevention
- Two approaches:
  - do not start a process if it's demand might lead to deadlock
  - do not grant an incremental resource request if this allocation might lead to deadlock
- In both cases: maximum requirements of each resource must be stated in advance

Resource types

- Resources in a system are partitioned in resources types
- Each resource type in a system exists with a certain amount. Let R(i) be the total amount of resource type i present in the system. Ex:
  - R(main memory) = 128 MB
  - R(disk drives) = 8
  - R(printers) = 5
- The partition is system specific (ex: printers may be further partitioned...)

Process initiation denial

- Let C(k,i) be the amount of resource type i claimed by process k.
- To be admitted in the system, process k must show C(k,i) for all resource types i
- C(k,i) is the maximum value of resource type i permitted for process k.
- Let U(i) be the total amount of resource type i unclaimed in the system:
  - U(i) = R(i) - S_k C(k,i)

Process initiation denial

- A new process n is admitted in the system only if C(n,i) <= U(i) for all resource type i
- This policy ensures that deadlock is always avoided since a process is admitted only if all its requests can always be satisfied (no matter what will be their order)
- A sub optimal strategy since it assumes the worst: that all processes will make their maximum claims together at the same time

Resource allocation denial: the banker's algorithm

- Processes are like customers wanting to borrow money (resources) to a bank...
- A banker should not allocate cash when it cannot satisfy the needs of all its customers
- At any time the state of the system is defined by the values of R(i), C(j,i) for all resource type i and process j and the values of other vectors and matrices.

The banker's algorithm

- We also need the amount allocated A(j,i) of resource type i to process j for all (j,i)
- The total amount available of resource i is given by: V(i) = R(i) - S_k A(k,i)
- We also use the need N(j,i) of resource type i required by process j to complete its task: N(j,i) = C(j,i) - A(j,i)
- To decide if a resource request made by a process should be granted, the banker's algorithm test if granting the request will lead to a safe state:
  - If the resulting state is safe then grant request
  - Else do not grant the request

The banker's algorithm

- A state is safe iff there exist a sequence {P1..Pn} where each Pi is allocated all of its needed resources to be run to completion
  - ie: we can always run all the processes to completion from a safe state
- The safety algorithm is the part that determines if a state is safe
- Initialization:
  - all processes are said to be "unfinished"
  - set the work vector to the amount resources available: W(i) = V(i) for all i;

The banker's algorithm

- REPEAT: Find a unfinished process j such that N(j,i) <= W(i) for all i.
  - If no such j exists, goto EXIT
  - Else: "finish" this process and recover its resources: W(i) = W(i) + A(j,i) for all i. Then goto REPEAT
- EXIT: If all processes have "finished" then this state is safe. Else it is unsafe.

The banker's algorithm

- Let Q(j,i) be the amount of resource type i requested by process j.
- To determine if this request should be granted we use the banker's algorithm:
  - If Q(j,i) <= N(j,i) for all i then continue. Else raise error condition (claim exceeded).
  - If Q(j,i) <= V(i) for all i then continue. Else wait (resource not yet available)
  - Pretend that the request is granted and determine the new resource-allocation state:

The banker's algorithm

1. If $Request_i \le Need_i$
   Then, we have to go step 2. Otherwise, an error condition is raised because the process has crossed its maximum limit.
2. If $Request_i \le Available$
   Then, we have to go step 3. Otherwise, the process P_i should wait, because currently, the resources are not available.
3. Now, we assume that the resources are assigned to the process 'P_i'.
   And we performed the below steps:
   a.) $Available = Available - Request_i$
   b.) $Allocation_i = Allocation_i + Request_i$
   c.) $Need_i = Need_i - Request_i$

   - $V(i) = V(i) - Q(j,i)$ for all i
   - $A(j,i) = A(j,i) + Q(j,i)$ for all i
   - $N(j,i) = N(j,i) - Q(j,i)$ for all i

- If the resulting state is safe then allocate resource to process j. Else process j must wait for request Q(j,i) and restore old state.

Example of the banker's algorithm

1. Calculate the content of the need matrix?
2. Is the system in a safe state?
3. Determine the total amount of resources of each type?

Processes	Allocation A B C			Max A B C			Available A B C		
P_0	1	1	2	4	3	3	2	1	0
P_1	2	1	2	3	2	2			
P_2	4	0	1	9	0	2			
P_3	0	2	0	7	5	3			
P_4	1	1	2	1	1	2			

Example of the banker's algorithm

Process	Need		
	A	B	C
P_0	3	2	1
P_1	1	1	0
P_2	5	0	1
P_3	7	3	3
P_4	0	0	0

Content of the need matrix can be calculated by using the below formula

Need = Max - Allocation

Now, we check for a safe state

Safe sequence:

1. For process P_0, Need = (3, 2, 1) and

Available = (2, 1, 0)

Need ? Available = False

So, the system will move for the next process.

2. For Process $P_{1,}$ Need = (1, 1, 0)

Available = (2, 1, 0)

Need ? Available = True

Request of P_1 is granted.

Available = Available +Allocation

= (2, 1, 0) + (2, 1, 2)

= (4, 2, 2) (New Available)

3. For Process P_2, Need = (5, 0, 1)

Available = (4, 2, 2)

Need ? Available = False

So, the system will move to the next process.

4. For Process P_3, Need = (7, 3, 3)

Available = (4, 2, 2)

Need ? Available = False

So, the system will move to the next process.

5. For Process P_4, Need = (0, 0, 0)

Available = (4, 2, 2)

Need ? Available = True

Request of P_4 is granted.

Available = Available + Allocation

= (4, 2, 2) + (1, 1, 2)

= (5, 3, 4) now, (New Available)

6. Now again check for Process $P_{2,}$ Need = (5, 0, 1)

Available = (5, 3, 4)

Need ? Available = True

Request of P_2 is granted.

Available = Available + Allocation

= (5, 3, 4) + (4, 0, 1)

= (9, 3, 5) now, (New Available)

7. Now again check for Process $P_{3,}$ Need = (7, 3, 3)

Available = (9, 3, 5)

Need ? Available = True

Request of P_3 is granted.

Available = Available +Allocation

= (9, 3, 5) + (0, 2, 0) = (9, 5, 5)

8. Now again check for Process P_0, = Need (3, 2, 1)

= Available (9, 5, 5)

Need ? Available = True

So, the request will be granted to P_0.

Safe sequence: < $P_{1,}$ $P_{4,}$ $P_{2,}$ $P_{3,}$ $P_{0>}$

The system allocates all the needed resources to each process. So, we can say that system is in a safe state.

The total amount of resources = sum of columns of allocation + Available

= [8 5 7] + [2 1 0] = [10 6 7]

Example of the banker's algorithm

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	3	2
P2	2	0	0	3	2	2			
P3	3	0	2	9	0	2			
P4	2	1	1	2	2	2			
P5	0	0	2	4	3	3			

Example of the banker's algorithm

- What is the reference of the need matrix?
- Determine if the system is safe or not.
- What will happen if the resource request (1, 0, 0) for process P1 can the system accept this request immediately?

Example of the banker's algorithm

Process	Need		
	A	B	C
P1	7	4	3
P2	1	2	2
P3	6	0	0
P4	0	1	1
P5	4	3	1

Example of the banker's algorithm

1. For Process P1:

Need <= Available

7, 4, 3 <= 3, 3, 2 condition is **false**

2. For Process P2:

Need <= Available

1, 2, 2 <= 3, 3, 2 condition **true**

New available = available + Allocation

(3, 3, 2) + (2, 0, 0) => 5, 3, 2

3. For Process P1:

Need <= Available

6, 0, 0 < = 5, 3, 2 condition is **false**

Example of the banker's algorithm

4. P4 Need <= Available

    0, 1, 1 <= 5, 3, 2 condition is **true**

    New Available resource = Available + Allocation

    5, 3, 2 + 2, 1, 1 => 7, 4, 3

5. P5 Need <= Available

    4, 3, 1 <= 7, 4, 3 condition is **true**

    New available resource = Available + Allocation

    7, 4, 3 + 0, 0, 2 => 7, 4, 5

Example of the banker's algorithm

6\. P1 Need <= Available

7, 4, 3 <= 7, 4, 5 condition is **true**

New Available Resource = Available + Allocation

7, 4, 5 + 0, 1, 0 => 7, 5, 5

7\. P3 Need <= Available

6, 0, 0 <= 7, 5, 5 condition is true

New Available Resource = Available + Allocation

7, 5, 5 + 3, 0, 2 => 10, 5, 7

Example of the banker's algorithm

- **Hence, we execute the banker's algorithm to find the safe state and the safe sequence like P2, P4, P5, P1 and P3.**
- **Ans. 3:** For granting the Request (1, 0, 2), first we have to check that **Request <= Available**, that is (1, 0, 2) <= (3, 3, 2), since the condition is true. So the process P1 gets the request immediately.

banker's algorithm: comments

- A safe state cannot be deadlocked. But an unsafe state is not necessarily deadlocked.
  - Ex: P1 from the previous (unsafe) state could release temporarily a unit of R1 and R3 (returning to a safe state)
  - some process may need to wait unnecessarily
  - sub optimal use of resources
- All deadlock avoidance algorithms assume that processes are independent: free from any synchronization constraint

Deadlock Detection

- Resource access are granted to processes whenever possible. The OS needs:
  - an algorithm to check if deadlock is present
  - an algorithm to recover from deadlock
- The deadlock check can be performed at every resource request
- Such frequent checks consume CPU time

A deadlock detection algorithm

- Makes use of previous resource-allocation matrices and vectors
- Marks each process not deadlocked. Initially all processes are unmarked. Then

BATTERY 95% CHARGED! TURN OFF POWER SOURCE TO AVOID BATTERY DAMAGE!

OK

esource type i. (since

j,i) <= W(i) for all i.

  - If such j exists: mark process j and set W(i) = W(i) + A(j,i) for all i. Goto REPEAT
  - At the end: each unmarked process is deadlocked

Deadlock detection: comments

BATTERY 95% CHARGED! TURN OFF POWER SOURCE TO AVOID BATTERY DAMAGE!

OK

for all i.

j will require no more

ces. Thus: W(i) = W(i)

- If this assumption is incorrect, a deadlock may occur later
- This deadlock will be detected the next time the deadlock detection algorithm is invoked

Deadlock detection: example

Request					Allocated					Available				
R1	R2	R3	R4	R5	R1	R2	R3	R4	R5	R1	R2	R3	R4	R5

BATTERY 95% CHARGED! TURN OFF POWER SOURCE TO AVOID BATTERY DAMAGE!

OK

0 0 0 0 0 1
0
0
0

- Mark P4 since it has no allocated resources
- Set W = (0,0,0,0,1)
- P3's request <= W. So mark P3 and set W = W + (0,0,0,1,0) = (0,0,0,1,1)
- Algorithm terminates. P1 and P2 are deadlocked

Deadlock Recovery

BATTERY 95% CHARGED! TURN OFF POWER SOURCE TO AVOID BATTERY DAMAGE!

OK

...st

previously defined checkpoint and restart them (original deadlock may reoccur)

- Successively abort deadlock processes until deadlock no longer exists (each time we need to invoke the deadlock detection algorithm)

Deadlock Recovery (cont.)

BATTERY 95% CHARGED! TURN OFF POWER SOURCE TO AVOID BATTERY DAMAGE!

OK

cesses

no

st be

- For the 2 last approaches: a victim process needs to be selected according to:
  - least amount of CPU time consumed so far
  - least total resources allocated so far
  - least amount of "work" produced so far...

An integrated deadlock strategy

BATTERY 95% CHARGED! TURN OFF POWER SOURCE TO AVOID BATTERY DAMAGE!

OK

the following way:

classes and order them.

- Swappable space (secondary memory)
- Process resources (I/O devices, files...)
- Main memory...

- Use prevention of circular wait to prevent deadlock between resource classes
- Use the most appropriate approach for each class for deadlocks within each class