Deadlock

H. Levy

Deadlock

- Deadlock is a problem that can exist when a group of processes compete for access to fixed resources.
- Def: deadlock exists among a set of processes if every process is waiting for an event that can be caused only by another process in the set.
- Example: two processes share 2 resources that they must *request* (before using) and *release* (after using). Request either gives access or causes the proc. to block until the resource is available.

Proc 1:	Proc 2:
request tape	request printer
request printer	request tape
... <use them>	... <use them>
release printer	release tape
release tape	release printer

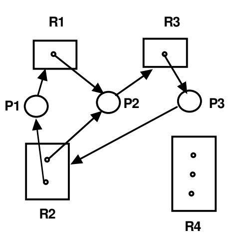
4 Conditions for Deadlock

- Deadlock can exist if and only iff 4 conditions hold simultaneously:
  1. mutual exclusion: at least one resource must be held in a non-sharable mode.
  2. hold and wait: there must be a process holding one resource and waiting for another
  3. no preemption: resources cannot be preempted
  4. circular wait: there must exist a set of processes [p1, p2, ..., pn] such that p1 is waiting for p2, p2 for p3, and so on...

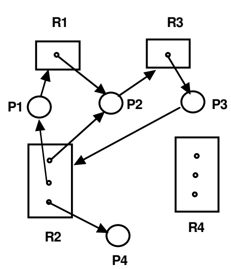
Resource Allocation Graph

- Deadlock can be described through a *resource allocation graph*.
- The RAG consists of a set of vertices $P=\{P_1,P_2,..,P_n\}$ of processes and $R=\{R_1,R_2,..,R_m\}$ of resources.
- A directed edge from a process to a resource, $P_i \rightarrow R_j$, implies that P_i has requested R_j.
- A directed edge from a resource to a process, $R_j \rightarrow P_i$, implies that R_j has been allocated by P_i.
- If the graph has no cycles, deadlock cannot exist. If the graph has a cycle, deadlock may exist.

Resource Allocation Graph Example



There are two cycles here: P1-R1-P2-R3-P3-R2-P1 and P2-R3-P3-R2-P2, and there is *deadlock.*



Same cycles, but no deadlock.

Possible Approaches

- Deadlock Prevention: ensure that at least 1 of the necessary conditions cannot exist.
  - mutual exclusion: make resources sharable (isn't really possible for some resources)
  - hold and wait: guarantee that a process cannot hold a resource when it requests another, or, make processes request all needed resources at once, or, make it release all resources before requesting a new set.
  - circular wait: impose an ordering (numbering) on the resources and request them in order

Possible Approaches

- Deadlock Avoidance
  - general idea: provide info in advance about what resources will be needed by processes to guarantee that deadlock will not exist.
- E.g., define a sequence of procs <P1,P2,..Pn> as *safe* if for each Pi, the resources that Pi can still request can be satisified by the currently available resources plus the resources held by all Pj, j<i.
  - this avoids circularities.
  - when a process requests a resource, the system grants or forces it to wait, depending on whether this would be an unsafe state.

Example:

- Processes p0, p1, and p2 compete for 12 tape drives

	max need	current usage	could ask for
p0	10	5	5
p1	4	2	2
p2	9	2	7

3 drives remain

- current state is safe because a safe sequence exists: <p1,p0,p2>

  p1 can complete with current resources
  p0 can complete with current+p1
  p2 can complete with current +p1+p0

- if p2 requests 1 drive, then it must wait, because that state would be unsafe.

The Banker's Algorithm

- Banker's algorithm decides whether to grant a resource request. Define data structures:

n: integer	# of processes
m: integer	# of resources
available[1..m]	avail[i] is # of avail resources of type i
max[1..n,1..m]	max demand of each Pi for each Ri
allocation[1..n,1..m]	current allocation of resource Rj to Pi
need[1..n,1..m]	max # of resource Rj that Pi may still request

let request[i] be a vector of the # of instances of resource Rj that Process Pi wants

The Basic Algorithm

1. If request[i] > need[i] then error (asked for too much)
2. If request[i] > available[i] then wait (can't supply it now)
3. Resources are available to satisfy the request:
   Let's *assume* that we satisfy the request. Then we would have:
   available = available - request[i]
   allocation[i] = allocation[i] + request[i]
   need[i] = need[i] - request[i]
   Now, check if this would leave us in a safe state; if yes, grant the request, if no, then leave the state as is and cause the process to wait.

Safety Check

1. new vars: work[1..m] = available ; to accumulate resources
   finish[1..n] = false (for all i) ; none finished yet
2. find an <u>i</u> s.t. finish[i]= false and need[i] <= work
   (find a proc that can complete its request now)
   if no such i exists, go to step 4 (we're done)
3. Found an i:
   finish[i] = true ; done with this process
   work = work + allocation[i] (assume this process were to finish, add its allocation back to the available list)
   go to step 2
4. If finish[i] = true for all i, the system is safe.

Deadlock Detection

- If there is neither deadlock prevention nor avoidance, then deadlocks may occur.
- In this case, we must have:
  - an algorithm that determines whether a deadlock has occurred
  - an algorithm to recover from the deadlock
- This is doable, but it's costly.

Deadlock Detection Algorithm

available[1..m] ; # of available resources
allocation[1..n,1..m] ; # of resource of each Ri allocated to Pj
request[1..n,1..m] ; # of resources of each Ri requested by Pj

1. work=available
for all i < n, if allocation[i] notequal 0
then finish[i]= false else finish[i]=true
2. find an index i such that:
finish[i]=false;
request[i] <= work
if no such i exists, go to 4.
3. work=work+allocation[i]
finish[i] = true, go to 2
4. if finish[i] = false for <u>some</u> i, then system is deadlocked with Pi in deadlock

Deadlock

- **Deadlock detection algorithm is expensive. How often we invoke it depends on:**
  - how often or likely is deadlock
  - how many processes are likely to be affected when deadlock occurs

Deadlock Recovery

- **Once a deadlock is detected, there are 2 choices:**
  1. abort all deadlocked processes (which will cost in the repeated computations necessary)
  2. abort 1 process at a time until cycle is eliminated (which requires re-running the detection algorithm after each abort)
- **Or, could do process preemption: release resources until system can continue. Issues:**
  - selecting the victim (could be clever based on R's allocated).
  - rollback (must rollback the victim to a previous state)
  - starvation (must not always pick same victim)