Deadlock

Emin Gun Sirer

---

**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.

```
Proc1: Proc2:
request tape  request printer
...<use them>...<use them>
release printer release tape
```

---

**4 Conditions for Deadlock**

- Deadlock can exist if and only if 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 \([p_1, p_2, ..., p_n]\) such that \(p_1\) is waiting for \(p_2\), \(p_2\) for \(p_3\), and so on and \(p_n\) waits for \(p_1\)....

---

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

Dealing with Deadlocks

• **Deadlock Prevention & Avoidance.** Ensure that the system will never enter a deadlock state
  • **Deadlock Detection & Recovery.** Detect that a deadlock has occurred and recover
  • **Deadlock Ignorance.** Pretend that deadlocks will never occur

Deadlock Prevention

• Deadlock Prevention: ensure that at least one of the necessary conditions cannot exist.
  • Mutual exclusion: make resources shareable
  • Not 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
  • Low utilization, starvation
  • Preemption: take resources back if there is contention
  • Not always possible, hard model to write applications for
  • Circular wait: impose an ordering (numbering) on the resources and request them in order

Most real systems use deadlock prevention through resource ordering

• The resource order is a convention that the OS designers must know and follow
  • These conventions complicate system programming

E.g. must always acquire a buffer cache lock before acquiring the file system lock, before acquiring the disk lock

What happens when you get a page fault?
### Problems with Deadlock Prevention

- Prevention works by restraining how requests are made
- Might yield low utilization and low throughput
  - Certain resource request sequences are not allowed, limiting functionality
- With sufficient information about future behavior, we could allow any process to perform any set of resource accesses
- As long as their actions would not lead to a deadlock in the future

### Deadlock Avoidance

- **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 set of processes $<P_1, P_2, ..., P_n>$ as **safe** if for each $P_i$, the resources that $P_i$ can still request can be satisfied by the currently available resources plus the resources held by all $P_j$ where $j < i$.
  - this avoids circular waiting
  - when a process requests a resource, the system grants or forces it to wait, depending on whether this would be an unsafe state
- All deadlock states are unsafe. An unsafe state may lead to deadlock. By avoiding unsafe states, we avoid deadlock

### Example:

- Processes $p_0$, $p_1$, and $p_2$ compete for 12 tape drives

<table>
<thead>
<tr>
<th>Resource</th>
<th>$p_0$</th>
<th>$p_1$</th>
<th>$p_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current</td>
<td>10</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>Max Need</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

- current state is safe because a safe sequence exists: $<p_1, p_0, p_2>$
  - $p_1$ can complete with current resources
  - $p_0$ can complete with current+$p_1$
  - $p_2$ can complete with current+$p_1$+$p_0$

- if $p_2$ 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.

```plaintext
n: integer # of processes
m: integer # of resources
available[1..m]: avai[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 Ri to Pi
need[1..n,1..m]: max # of resource Ri that Pi may still request

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

- Example:

  $n$: integer # of processes
  $m$: integer # of resources
  available[1..m]: avai[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 Ri to Pi
  need[1..n,1..m]: max # of resource Ri that Pi may still request

  let request[i] be a vector of the # of instances of resource Ri 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 process to wait.

Safety Check

1. free[1..m] = available ; how many resources are available
   finish[1..n] = false (for all i) ; none finished yet
2. Find an i 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
   free = free + allocation[i] (assume this process were to finish, and 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 deadlock 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 Pi allocated to Pi
request[1..n,1..m] ; # of resources of each Pi requested by Pi

1. work = available
   for all i < n, if allocation[i] not 0
       then finish[i] = true else finish[i] = false
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 some 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

- Running the deadlock detection algorithm often will catch deadlock cycles early
  - Few processes will be affected
  - Note: there is no single process that caused the deadlock
  - May incur large overhead

Deadlock Recovery

- Once a deadlock is detected, there are two choices:
  1. abort all deadlocked processes (which will cause some computations to be repeated)
  2. abort one 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 resources allocated)
  - rollback (must rollback the victim to a previous state, may require a transactional programming model, or functional apps)
  - starvation (must not always pick same victim)