# CS 4410
# Operating Systems

# Deadlocks
# Prevention & Avoidance

Summer 2016

Cornell University

# Today

- Deadlock prevention
- Deadlock avoidance

# Deadlock Prevention

**Negate one of necessary conditions:**

- Mutual exclusion:
  - Make resources sharable
  - Not always possible (printers?)
- Hold and wait
  - Do not hold resources when waiting for another
    - Request all resources before beginning execution
    - Processes do not know what they will need
    - Starvation (if waiting on many popular resources)
    - Low utilization (Need resource only for a bit)
  - Alternative: Release all resources before requesting anything new
    - Still has the last two problems

# Deadlock Prevention

- No preemption:
  - Make resources preemptable (2 approaches)
    - Preempt requesting processes' resources if all not available
    - Preempt resources of waiting processes to satisfy request
  - Good when easy to save and restore state of resource
    - CPU registers, memory virtualization
- Circular wait: (2 approaches)
  - Single lock for entire system? (Problems)
  - Impose **partial ordering** on resources, request them in order

# Deadlock Prevention

- Prevention: Breaking circular wait
  - Order resources (lock1, lock2, ...)
  - Acquire resources in strictly increasing/decreasing order
  - Intuition: Cycle requires an edge from low to high, and from high to low numbered node, or to same node.
  - Ordering not always easy...

# Deadlock Avoidance

- If we have future information:
  - Max resource requirement of each process before they execute.
- Can we guarantee that deadlocks will never occur?
- Avoidance Approach:
  - Before granting resource to a process, check if resulting state is safe.
  - If the state is safe $\Rightarrow$ no deadlock!
    - Grant the resource.
  - Otherwise, wait.
    - Until some other process releases enough resources.

# Safe State

- A state is said to be **safe**, if it has a process sequence {P1, P2,…, Pn}, such that
  - for each Pi, the resources that Pi can still request can be satisfied by the currently available resources,

    plus the resources held by all Pj, where j < i.
- State is safe because OS can definitely avoid deadlock
  - by blocking any new requests until safe order is executed.
- This avoids **circular wait** condition.
  - Process waits until safe state is guaranteed.

# Safe State Example

- Suppose there are 12 tape drives

| | Max need | Current usage | Could ask for |
|---|---|---|---|
| p0 | 10 | 5 | 5 |
| p1 | 4 | 2 | 2 |
| p2 | 9 | 2 | 7 |

  – 3 drives are available.

- 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

# Safe State

To decide when a state is safe:

- Construct the resource allocation graph for that state.
- Apply the graph reduction algorithm.
- If the reduced graph is empty:
  - the state is safe,
  - the order with which processes were eliminated during the execution of the algorithm gives the safe sequence of processes.
- If the reduces graph is not empty:
  - The state is unsafe.

# Banker’s Algorithm

- Decides whether a resource request can be safely granted.
- Assumption: each process declares the maximum number of instances of each resource type that it may need.
  - This number may not exceed the total number of resources in the system.

# Banker's Algorithm

For a request R of additional resources issued by process P, which is the next process scheduled to run:

1. If R does not exceed P's maximum claim, go to 2. Otherwise, error.
2. If R does not exceed the available resources, go to 3. Otherwise, P should wait.
3. Pretend that R is granted to P.

   Update the state of the system.

   If the state is safe, then give requested resources to P.

   Otherwise, P should wait and the old state is restored.

# Banker's Algorithm

Data structures:

| | |
|---|---|
| n | number of processes |
| m | number of resource-types |
| available[1..m] | available[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 number of resource Rj instances that Pi may still request |
| | (need = max - allocation) |

# Banker's Algorithm : safety algorithm

```
free[1..m] = available              /* how many resources are available */
finish[1..n] = false (for all i)    /* none finished yet */
```

**Step 1:**

Find an i such that finish[i]=false and need[i] <= free

If no such i exists, go to step 3   /*we're done */

**Step 2:** 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 1
```

**Step 3:** If finish[i] = true for all i, the system is safe. Else the system is unsafe.

# Banker's Algorithm: resource-request 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.

# Banker's Algorithm: Example

| | Allocation | Max | Available |
|---|---|---|---|
| | A B C | A B C | A B C |
| P0 | 0 1 0 | 7 5 3 | 3 3 2 |
| P1 | 2 0 0 | 3 2 2 | |
| P2 | 3 0 2 | 9 0 2 | |
| P3 | 2 1 1 | 2 2 2 | |
| P4 | 0 0 2 | 4 3 3 | |

- This is a safe state: safe sequence <P1, P3, P4, P2, P0>

- Suppose that P1 requests (1,0,2)
- Add it to P1's Allocation and subtract it from Available.

# Banker’s Algorithm: Example

| | Allocation | Max | Available |
|---|---|---|---|
| | A B C | A B C | A B C |
| P0 | 0 1 0 | 7 5 3 | 2 3 0 |
| P1 | 3 0 2 | 3 2 2 | |
| P2 | 3 0 2 | 9 0 2 | |
| P3 | 2 1 1 | 2 2 2 | |
| P4 | 0 0 2 | 4 3 3 | |

- This is still safe: safe seq <P1, P3, P4, P0, P2>
- In this new state, P4 requests (3,3,0)
- Not enough available resources.
- P0 requests (0,2,0)
- Let’s check resulting state...

# Banker's Algorithm: Example

| | Allocation | Max | Available |
|---|---|---|---|
| | A B C | A B C | A B C |
| P0 | 0 3 0 | 7 5 3 | 2 1 0 |
| P1 | 3 0 2 | 3 2 2 | |
| P2 | 3 0 2 | 9 0 2 | |
| P3 | 2 1 1 | 2 2 2 | |
| P4 | 0 0 2 | 4 3 3 | |

- This is unsafe state (why?).
- So P0's request will be denied.

# Today

- Deadlock prevention
- Deadlock avoidance

# Coming up…

- Next lecture: memory management
- HW: Short concise answers!