Operating Systems ECE344

Lecture 8: Deadlock

Ding Yuan

Deadlock

- Synchronization is a live gun – we can easily shoot ourselves in the foot
  - Incorrect use of synchronization can block all processes
  - We have talked about this problem already
- More generally, processes that allocate multiple resources generate dependencies on those resources
  - Locks, semaphores, monitors, etc., just represent the resources that they protect
- If one process tries to request for a resource that a second process holds, and vice-versa, they can never make progress
- We call this situation deadlock, and we'll look at:
  - Representation of deadlock conditions
  - Approaches to dealing with deadlock

Traffic Deadlock

Deadlock

- Deadlock is a problem that can arise:
  - When processes compete for access to limited resources
  - When processes are incorrectly synchronized
- Definition:
  - Deadlock exists among a set of processes if every process is waiting for the others to finish, and thus no one ever does (*deadly embrace*).

Process 1

```
lockA->Acquire();
…
lockB->Acquire();
```

Process 2

```
lockB->Acquire();
…
lockA->Acquire();
```

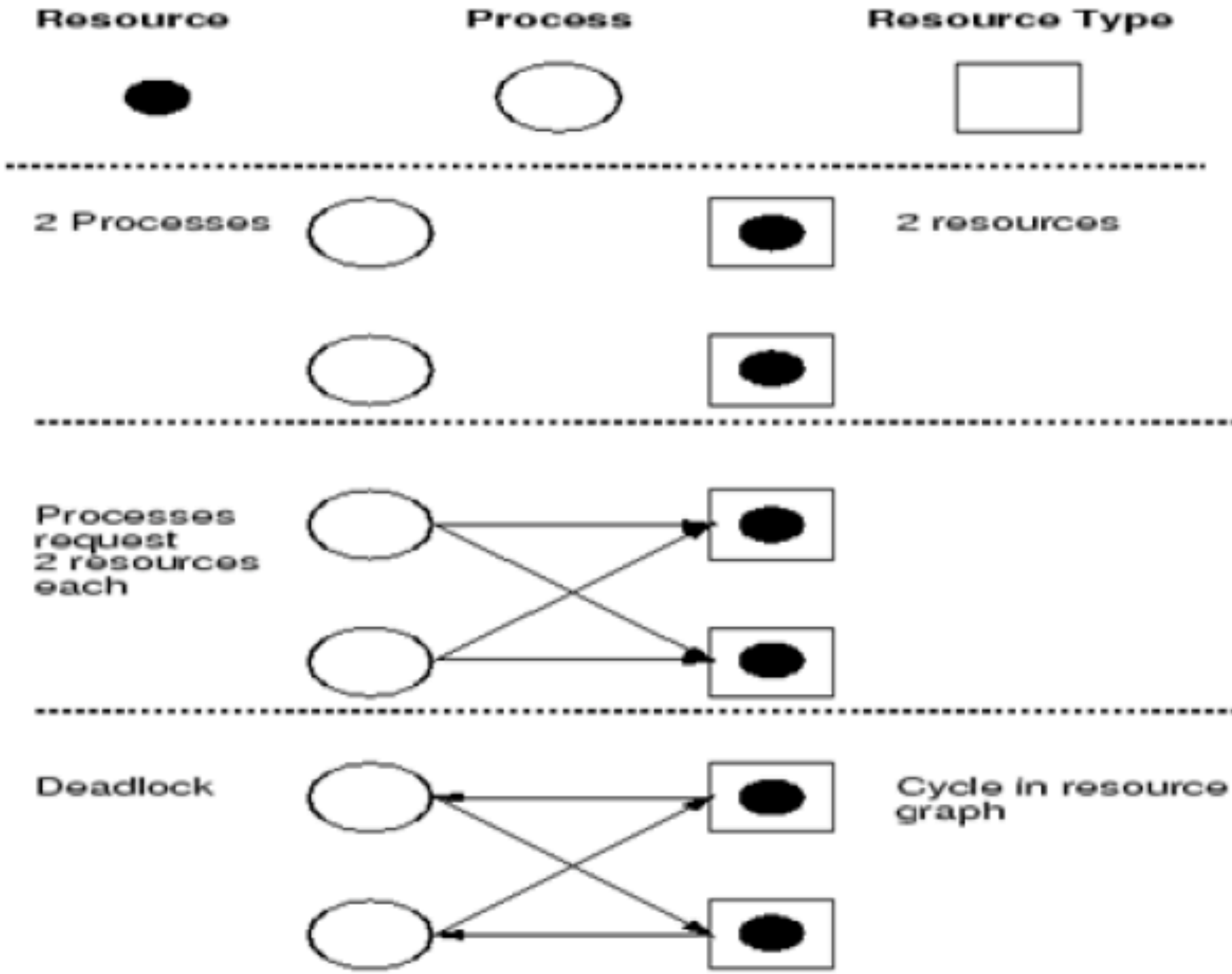
Conditions for Deadlock

- Deadlock can exist if and only if the following four conditions hold simultaneously:
  1. Mutual exclusion – Processes claim **exclusive** control of the resources they acquire
  2. Hold and wait – There must be one process holding one resource and waiting for another resource
  3. No preemption – Resources cannot be preempted (critical sections cannot be aborted externally)
  4. Circular wait – A circular chain of processes exists in which each process holds one or more resources that are requested by the next process in the chain

Resource Allocation Graph

- Deadlock can be described using a resource allocation graph (RAG)
- The RAG consists of a set of vertices $P=\{P_1, P_2, \ldots, P_n\}$ of processes and $R=\{R_1, R_2, \ldots, R_m\}$ of resources
  - A directed edge from a process to a resource, $P_i \rightarrow R_i$, means that P_i has requested R_j
  - A directed edge from a resource to a process, $R_i \rightarrow P_i$, means 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

Deadlock Model



Dealing With Deadlock

- Prevention
  - make it impossible for deadlock to happen
- Avoidance
  - impose less stringent conditions than for prevention, allowing the possibility of deadlock, but sidestepping it as it approaches
- Detection and Recovery
  - in a system that allows the possibility of deadlock, detect the occurrence and recover

The Ostrich Algorithm

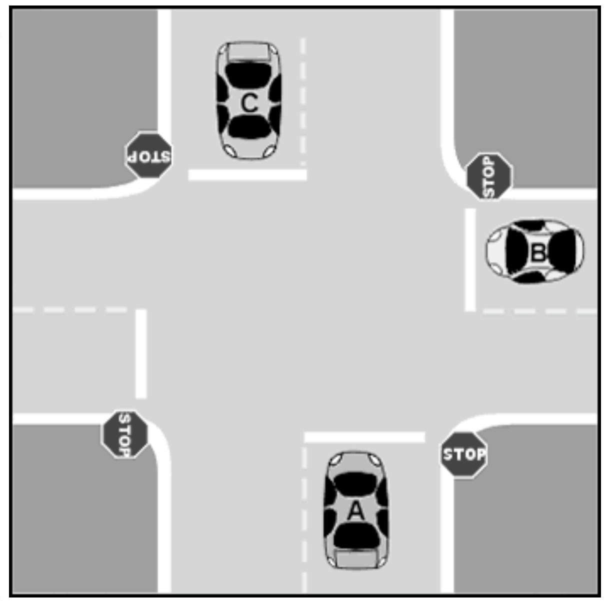
- Don't do anything, simply restart the system (stick your head into the sand, pretend there is no problem at all)
- Rationale: make the common path fast
  - Deadlock prevention, avoidance or detection/ recovery algorithms are expensive
  - If deadlock occurs only rarely, it is not worth the overhead

Then why do we still learn about deadlocks?

- How about aircraft control systems?
- How about the software running in your car?

How do we prevent deadlocks?

- Anyone?
  - You can use real-life analogies (hint: consider road intersection)



Deadlock Prevention

- Break one of the deadlock conditions
  - Mutual exclusion
    - Make resources sharable (printer spool)
  - Hold and wait condition
    - Force each process to request all required resources at once. It cannot proceed until all resources have been acquired (intersection with stop signs)
  - No Preemption condition
    - If a process holding some resources and is further waiting for additional resources, it must release the resources it is currently holding and request them again later
    - Remember "wait()" in monitor?
  - Circular wait condition
    - Impose an ordering (numbering) on the resources and request them in order (popular implementation technique)

Deadlock Avoidance

- The system needs to know the resource requirement ahead of time
- Banker's Algorithm (Dijkstra, 1965)

Banker's Algorithm

- The Banker's Algorithm is the classic approach to deadlock avoidance for resources with multiple units

1. Assign a credit limit to each customer (process)
    - Maximum resources each process needs
        - Max resource requests must be known in advance
2. Reject any request that leads to a dangerous state
    - A dangerous state is one where a sudden request by any customer for the full credit limit could lead to deadlock
    - A recursive reduction procedure recognizes dangerous states

Safe State and Unsafe State

- Safe State
  - there is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately
  - From safe state, the system can guarantee that all processes will finish
- Unsafe state: no such guarantee
  - Not a deadlock state (may lead to deadlock)
  - Some processes may be able to complete

Example: single resource

- One resource with 10 units (total asset in the bank)

Process	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

Safe

If all processes request MAX resources, here is a *Safe* schedule:

1. give A 6 units, A completes
2. give B 5 units, B completes
3. give C 4 units, C completes
4. give D 7 units, D completes

Note: it is safe as long as there exists a safe schedule (OS is the banker, controls the scheduler)

Example: single resource

- One resource with 10 units (total asset in the bank)

Process	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

Safe

→

Process	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

Safe

Is it a safe state?

Here is a *Safe* schedule:
1 give C 2 units, C completes (4 available)
2 give B 4 units, B completes (5 available)
3 give A 5 units, A completes (6 available)
4 give D 7 units, D completes

Example: single resource

- One resource with 10 units (total asset in the bank)

Process	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

Safe

→

Process	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

Safe

→

Process	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

Unsafe

Banker's algorithm Implementation

- Whenever the OS receives a resource request, assume it is granted, and do the following:
  1. Look for a process (row) whose unmet resource needs are all smaller than or equal to *Free* resources. If no such row exists, *unsafe*. OS does not grant the request (put the requesting process to sleep and let others to run).
  2. Assume this process requests all the resources it needs and finishes. Mark that process as terminated and add all its resources to *Free* resources.
  3. Repeat steps 1 and 2 until either all processes are marked terminated (in which case the request will lead to a ***safe*** state and OS grant the request), or none of the remaining processes' resource needs can be met (*unsafe* state, do not grant the request)

Detection and Recovery

- Detection and recovery
  - If we don't have deadlock prevention or avoidance, then deadlock may occur
  - In this case, we need to detect deadlock and recover from it
- To do this, we need two algorithms
  - One to determine whether a deadlock has occurred
  - Another to recover from the deadlock
- Possible, but expensive (time consuming)
  - Implemented in VMS
  - Run detection algorithm when resource request times out

Deadlock Detection

- Detection
  - Traverse the Resource Allocation Graph looking for cycles
  - If a cycle is found, deadlock!
- Expensive
  - Many processes and resources to traverse
- Only invoke detection algorithm depending on
  - How often or likely deadlock is
  - How many processes are likely to be affected when it occurs

Deadlock Recovery

Once a deadlock is detected, we have two options…

1. Abort processes
   - Abort all deadlocked processes
     - Processes need start over again
   - Abort one process at a time until cycle is eliminated
     - System needs to rerun detection after each abort
2. Preempt resources (force their release)
   - Need to select process and resource to preempt
   - Need to rollback process to previous state
   - Need to prevent starvation

Deadlock Summary

- Deadlock occurs when processes are waiting on each other and cannot make progress
  - Cycles in Resource Allocation Graph (RAG)
- Deadlock requires four conditions
  - Mutual exclusion, hold and wait, no resource preemption, circular wait
- Four approaches to dealing with deadlock:
  - Ignore it – Living life on the edge
  - Prevention – Make one of the four conditions impossible
  - Avoidance – Banker's Algorithm (control allocation)
  - Detection and Recovery – Look for a cycle, preempt or abort