COS 140: Foundations of Computer Science

Handling Deadlocks: Banker's Algorithm

Fall 2018

Homework, reminder

- ☐ Chapter 22 (online)
- ☐ Homework at the end of chapter
- ☐ Homework due 11/16 (later than usual!)
- ☐ Prelim II: Wednesday, 11/14

Deadlocks

Operating systems as resource managers

- ☐ Example of resources
- ☐ Sharable vs non-sharable resources
- ☐ *Preemptible* vs non-preemptible resources
- ☐ Potential problem: deadlocks

What are deadlocks?

- ☐ A deadlock occurs when each process in a set of processes is waiting for some event that only another process in the set can cause. [after Tannenbaum]
- ☐ Example:
  - P1: needs CD-ROM and sound card
  - P1: asks for CD-ROM and receives it
  - P2: needs CD-ROM and sound card
  - P2: asks for sound card and gets it
  - P1: asks for sound card ⇒ blocks
  - P2: asks for CD-ROM ⇒ blocks

Conditions for deadlocks

Mutual exclusion: Resource is either available or assigned to at most one process
Hold-and-wait: Process can hold one resource and then ask for others
No preemption: Can't take a resource away from a process once assigned
Circular wait: ≥ 2 processes in circle in which each is waiting for resource held by next in circle

Digression: Directed graphs

☐ Many areas of CS require us to think of objects and relationships between them; e.g., paths between locations, data dependencies, constraints in logic puzzles

☐ Can represent this formally as a *graph*:

  - *Vertices* (or nodes) represent the objects
  - *Edges* (or arcs, or links) represent the relationships

☐ Sometimes, relationship is directional

  - Think “one-way streets”
  - Now the edges have direction, and the graph is called a *directed graph* or *digraph*

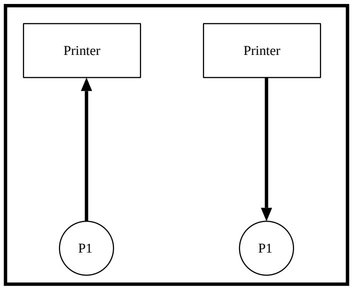
Modeling deadlocks as digraphs

☐ Circles: processes

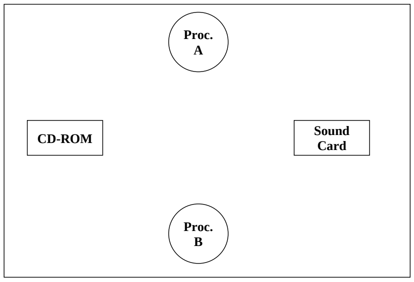
☐ Squares: resources

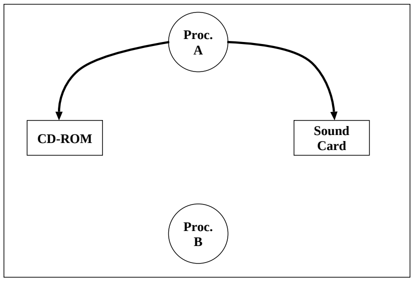
☐ Link from process → resource: process requests resource

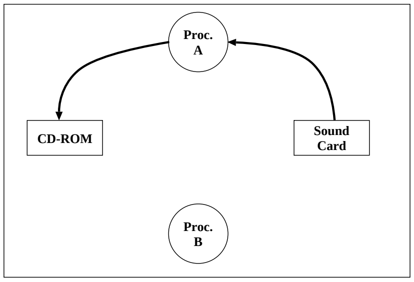
☐ Link from resource → process: process has control of resource

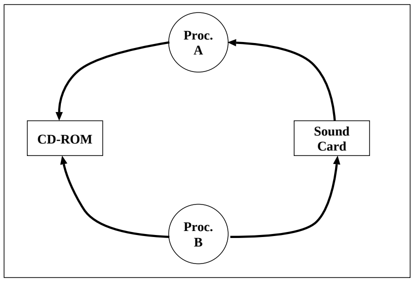


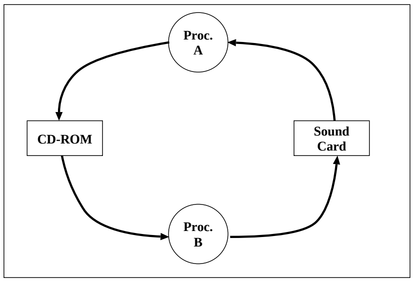
Modeling deadlocks as digraphs











Handling deadlocks

What do we do about deadlocks?

☐ Ignore them
☐ Detect them and (try to) recover
☐ Prevent them altogether
☐ Predict and avoid them

Ignoring deadlocks: The Ostrich Algorithm

- ☐ Sounds stupid, but...
- ☐ Consider:
  - How often will a deadlock happen?
  - How severe will it be if it does happen?
  - How hard would it be to avoid/prevent/detect?

Deadlock detection/recovery

- ☐ Detection:
  - Monitor resource allocation using (e.g.) a digraph
  - If detect a cycle ⇒ deadlock has occurred
- ☐ Recovery:
  - Kill one of the processes
  - If that doesn't work: kill another, etc.
- ☐ Another alternative: just look for processes that have been idle for a long time and kill them
- ☐ May be okay when aborting and restarting is okay (e.g., batch jobs)

Deadlock prevention

- Set things up so that deadlocks cannot occur at all
- Done by attacking one of the deadlock conditions
- Attacking mutual exclusion condition:
  - Don’t let non-sharable resources be assigned to anyone
  - E.g., spooling

Deadlock prevention

- Attacking hold-and-wait condition:
  - Process can’t request a resource if holding any
  - One way: processes request all resources up front
  - Problem: may not know ahead of time what you need!
  - Problem: hold resources too long in general
  - Another approach: release all you’re holding momentarily to request another
- Attacking no preemption condition: not realistic

Deadlock prevention

- ☐ Attacking the circular wait condition:
  - – Stupid way: processes can only hold a single resource at a time
  - – Better way:
    - ▹ Number the resources
    - ▹ Process can request whatever it wants, whenever it wants...as long as the requests are in numerical order

Deadlock prevention

- ☐ Attacking the circular wait condition (cont'd):
  - – Resource allocation graph can't have cycles in this scheme – why not?
    - ▹ Consider the case where process A holds resource i and B holds j – deadlock only possible if A requests j and B requests i
    - ▹ If $i > j$, then A can't request j
    - ▹ If $j > i$, then B can't request i
  - – Problem – may not be able to find an ordering that satisfies everyone!

Deadlock avoidance

- ☐ Idea: predict when some action → deadlock, avoid it
- ☐ Dijkstra's Banker's Algorithm (single resource version)
  - Modeled on the way a banker might deal with lines of credit to customers
  - Deadlock if there is no way to guarantee that all customers can borrow up to their maximum resource limit at some point in time

Dijkstra's Banker's Algorithm

- ☐ Safety:

  A state is *safe* if some sequence of other possible states exists that allows all customers (processes) to get up to their maximum resource limit at some time

- ☐ Keep track of maxmimum and current allocation for each customer
- ☐ Start in a safe state
- ☐ When process requests additional amount of resource, make sure that next state will also be safe
- ☐ If so, allow request, else disallow it

Banker's Algorithm Example

☐ Initial state:

Process	Current	Maximum
A	0	7
B	0	3
C	0	2
D	0	4

Remaining: 8

Banker's Algorithm Example 2

☐ Initial state:

Process	Current	Maximum
A	3	7
B	2	3
C	0	2
D	2	4

Remaining: 1

☐ Safe or not?

			Process Finishing							
Proc	Max	Init	B		D		C		A	
A	7	3	3	3	3	3	3	3	7	–
B	3	2	3	–	–	–	–	–	–	–
C	2	0	0	0	0	0	2	–	–	–
D	4	2	2	2	4	–	–	–	–	–
Rem		1	0	3	1	5	3	5	1	8

Banker's Algorithm Example 3

Proc	Curr	Max
A	3	7
B	2	3
C	0	2
D	2	4

Remaining: 1

$\xrightarrow{B \text{ wants } 1}$

Proc	Curr	Max
A	3	7
B	3	3
C	0	2
D	2	4

Remaining: 0

☐ Allow the request?

- Yes.
- Possible sequence of processes running to completion: B → D → C → A

Banker's Algorithm Example 4

Proc	Curr	Max
A	6	7
B	0	3
C	0	2
D	1	4

Remaining: 1

$\xrightarrow{A \text{ wants } 1}$

Proc	Curr	Max
A	7	7
B	0	3
C	0	2
D	1	4

Remaining: 0

☐ Allow the request?

- Yes.
- Possible sequence of processes running to completion: A → B → C → D

Banker's Algorithm Example 5

Proc	Curr	Max
A	4	7
B	1	3
C	0	2
D	1	4

Remaining: 2

$\xrightarrow{D \text{ wants } 1}$

Proc	Curr	Max
A	4	7
B	1	3
C	0	2
D	2	4

Remaining: 1

- Allow the request?
  - NO!
  - No sequence possible where they all can finish

Banker's Algorithm: Critique

- Is it too strong?
- After all – no guarantee that in the previous example:

Proc	Curr	Max
A	4	7
B	1	3
C	0	2
D	1	4

Remaining: 2

$\xrightarrow{D \text{ wants } 1}$

Proc	Curr	Max
A	4	7
B	1	3
C	0	2
D	2	4

Remaining: 1

D might not give back 1 immediately, moving back into safe state

- But we're interested in *guarantee* that there will be no deadlock, so this is what we need.

Banker's Algorithm: Critique

- ☐ Scales to multiple processes/resources
- ☐ Problems:
  - Need to know maximum resources needed per process – often (usually?) impossible for multiprocess system
  - Number of processes constantly changes
  - Resources can disappear
- ☐ But: Really no better general-purpose algorithm exists for this

Summary

- ☐ Handling deadlocks is difficult
- ☐ No best general solution
- ☐ How you choose to handle it depends on your situation: *trade-offs*