Deadlock Revisited

CS439: Principles of Computer Systems

April 27, 2015

Last Time

Distributed File Systems

- Consistency Models
- NFS
- GFS

Today's Agenda

Deadlocks

- What causes them (again)
- Deadlock Avoidance
- Deadlock Prevention
- Banker's algorithm

Deadlock Revisited

Deadlock, More Formally

- *Deadlock* occurs when two or more threads **or processes** are waiting for an event that can only be generated by these same threads **or processes**
- Deadlock is not starvation
  - Starvation can occur without deadlock
    - occurs when a thread **or process** waits indefinitely for some resources, but other threads **or processes** are actually using it
  - But deadlock does imply starvation

Necessary Conditions for Deadlock

Deadlock *can* happen if all of the following conditions hold:

1. **Bounded Resources**: a finite number of threads **or processes** can use a resource and resources are finite
   - relaxation of mutual exclusion condition
2. **Hold and Wait**: at least one thread **or process** holds a resources and is waiting for other resources to become available. A different thread holds the resource.
3. **No Pre-emption**: a thread **or process** only releases a resource voluntarily; another thread, **process**, or the OS cannot force the thread **or process** to release the resource
4. **Circular Wait**: A set of waiting **processes or** threads $\{t_1, \ldots, t_n\}$ where t_i is waiting on t_{i+1} (*i=1 to n*) and t_n is waiting on t_1

Managing Deadlocks

- **Deadlock prevention** adopts a policy that breaks one of the four conditions
- **Deadlock avoidance** algorithms check resource requests and possible availability to prevent deadlock
  - Guarantee that deadlock will never occur
  - Breaks one of the four necessary conditions
- **Deadlock detection** algorithms find instances of deadlock and try to recover
  - Admit the possibility of deadlock occurring and periodically check for it

Deadlock Prevention

Prevent deadlock by insuring that at least one of the necessary conditions doesn't hold

1. **Bounded Resources**: make resources sharable or provide more resources
2. **Hold and Wait**: guarantee a thread **or process** cannot hold one resource when it requests another (or must request all at once)
3. **No Pre-emption**: If a thread **or process** requests a resource that cannot be immediately allocated to it, then the OS pre-empts all the resources the thread **or process** is currently holding. Only when all the resources are available will the OS restart the thread **or process**
4. **Circular Wait**: Impose an ordering on the resources and request them in order

Deadlock Prevention: Resource Ordering

- Order all locks (or semaphores or resources)
- All code grabs locks in a predefined order
- Complications:
  - Maintaining global order is difficult in a large project
  - Global order can force a client to grab a lock earlier than it would like, tying up a resource for longer than necessary
- What happens when we apply this to system resources?

Avoiding Deadlock: The Banker's Algorithm

- Allows sum of maximum resource needs to exceed the total available resources

  as long as there exists a schedule of loan fulfillments such that all clients can:

  - Receive their maximal loan
  - Build their respective houses
  - Pay back all the loan
- More efficient than atomically acquiring all resources

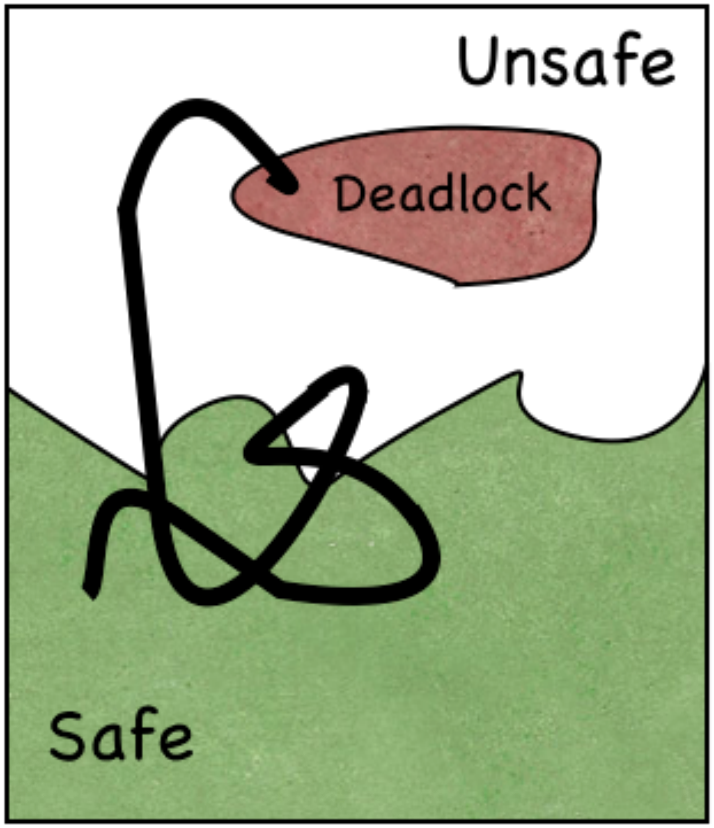
Avoiding Deadlock: The Banker's Algorithm Plain Text

- Allows sum of maximum resource needs to exceed the total available resources as long as there exists a schedule of loan fulfillments such that:
  - All clients receive their maximal loan
  - Build their respective houses
  - pay back all the loan
- More efficient than atomically acquiring all resources
- Picture is from the movie *It's a Wonderful Life*, a classic starring Jimmy Stewart. In that movie, he is a banker.

The Banker's Algorithm: Details

- Banker has N units, but loans out many more
  - Okay as long as N+1 units are not needed at the same time
- Uses *safe* and *unsafe* states
  - *Safe* states are states where enough resources are potentially available such that at least one process can run to completion
  - *Unsafe* states may lead to deadlock
- If resource request leads to an unsafe state, request is denied even if resources are currently available

Living Dangerously: Safe, Unsafe, Deadlocked



A system's trajectory through its state space

- Safe: For any possible set of resource requests, there exists one safe schedule of processing requests that succeeds in granting all pending and future requests
  - no deadlock as long as system can enforce safe schedule
- Unsafe: There exists a set of (pending and future) resource requests that leads to a deadlock, for any schedule in which requests are processed
  - unlucky set of requests can force deadlock
- Deadlocked: The system has at least one deadlock

Living Dangerously: Safe, Unsafe, and Deadlocked
Plain Text

- Safe: for any possible set of resource requests, there exists one safe schedule of processing requests that succeeds in granting all pending and future requests
  - No deadlock as long as system can enforce safe schedule
- Unsafe: there exists a set of (pending and future) resource requests that leads to a deadlock, for any schedule in which requests are processed
  - Unlucky set of requests can force deadlock
- Deadlocked: the system has at least one deadlock
- A system's trajectory through its state space (as the states relate to deadlock)
  - Safe and deadlocked states are completely disjoint
  - Must go from unsafe to deadlocked state (cannot go straight to deadlock from safe)
  - Can move between safe and unsafe states without ever going into deadlock

Banker's Algorithm: Example

- 5 processes, 4 resources

Max

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	7	5	0
P_3	2	3	5	6
P_4	0	6	5	2
P_5	0	6	5	6

Allocated

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	0	0	0
P_3	1	3	5	3
P_4	0	6	3	2
P_5	0	0	1	4

Available (to be allocated)

R_1	R_2	R_3	R_4
1	5	2	0

- Is this a safe state?

Banker's Algorithm: Example Plain Text

- 5 processes (represented by P1-P5) competing for 4 resources (represented by R1-R4)
- First metric: maximum number of each kind of resource each process will need
  - P1: 0 R1s, 0 R2s, 1 R3, and 2 R4s
  - P2: 1 R1, 7 R2s, 5 R3s, and 0 R4s
  - P3: 2 R1s, 3 R2s, 5 R3s, and 6 R4s
  - P4: 0 R1s, 6 R2s, 5 R3s and 2 R4s
  - P5: 0 R1s, 6 R2s, 5 R3s and 6 R4s
- Second metric: number of each kind of resource that has already been allocated to each process
  - P1: 0 R1s, 0 R2s, 1 R3, and 2 R4s
  - P2: 1 R1, 0 R2s, 0 R3s, and 0 R4s
  - P3: 1 R1, 3 R2s, 5 R3s, and 3 R4s
  - P4: 0 R1s, 6 R2s, 3 R3s, and 2 R4s
  - P5: 0 R1s, 0 R2s, 1 R3, and 4 R4s
- Third metric: number of each kind of resource in the system that is still available to be allocated
  - R1: 1
  - R2: 5
  - R3: 2
  - R4: 0
- Big question: is this a safe state?

Example: Determining Safety

- 5 processes, 4 resources

Max

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	7	5	0
P_3	2	3	5	6
P_4	0	6	5	2
P_5	0	6	5	6

−

Allocated

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	0	0	0
P_3	1	3	5	3
P_4	0	6	3	2
P_5	0	0	1	4

Available

R_1	R_2	R_3	R_4
1	5	2	0

=

Max Request

	R_1	R_2	R_3	R_4
P_1	0	0	0	0
P_2	0	7	5	0
P_3	1	0	0	3
P_4	0	0	2	0
P_5	0	6	4	2

- Determine MaxRequest by subtracting Allocated from Maximum

Example: Determining Safety

- 5 processes, 4 resources

Max

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	7	5	0
P_3	2	3	5	6
P_4	0	6	5	2
P_5	0	6	5	6

−

Allocated

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	0	0	0
P_3	1	3	5	3
P_4	0	6	3	2
P_5	0	0	1	4

Available

R_1	R_2	R_3	R_4
1	5	2	0

=

Max Request

	R_1	R_2	R_3	R_4
P_1	0	0	0	0
P_2	0	7	5	0
P_3	1	0	0	3
P_4	0	0	2	0
P_5	0	6	4	2

- While safe sequence does not include all processes:
  - Is there a P_i such that $MaxRequest_i$ <= Available?
    - if no, exit with unsafe
    - if yes, add P_i to the sequence and set Available = Available + Allocated

Example: Determining Safety

- 5 processes, 4 resources

Max

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	7	5	0
P_3	2	3	5	6
P_4	0	6	5	2
P_5	0	6	5	6

Allocated

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	0	0	0
P_3	1	3	5	3
P_4	0	6	3	2
P_5	0	0	1	4

Available

R_1	R_2	R_3	R_4
1	5	2	0

- Is this a safe state? YES!

Example: Determining Safety Plain Text

- Using the same metrics from slide 15
  - 5 processes competing for 4 resources
- Determine maximum number of resources that any process may still request by subtracting allocated from maximum
- Max requests by process:
  - P1:
    - Max needed: 0 R1s, 0 R2s, 1 R3, and 2 R4s
    - Allocated: 0 R1s, 0 R2s, 1 R3, and 2 R4s
    - Max request = max needed - allocated = 0 R1s, 0 R2s, 0 R3s, and 0 R4s
  - P2:
    - Max needed: 1 R1, 7 R2s, 5 R3s, and 0 R4s
    - Allocated: 1 R1, 0 R2s, 0 R3s, and 0 R4s
    - Max request = max needed - allocated = 0 R1s, 7 R2s, 5 R3s, and 0 R4s
  - P3:
    - Max needed: 2 R1s, 3 R2s, 5 R3s, and 6 R4s
    - Allocated: 1 R1, 3 R2s, 5 R3s, and 3 R4s
    - Max request = max needed - allocated = 1 R1, 0 R2s, 0 R3s, and 3 R4s
  - P4:
    - Max needed: 0 R1s, 6 R2s, 5 R3s, and 2 R4s
    - Allocated: 0 R1s, 6 R2s, 3 R3s, and 2 R4s
    - Max request = max needed - allocated = 0 R1s, 0 R2s, 2 R3s, and 0 R4s
  - P5:
    - Max needed: 0 R1s, 6 R2s, 5 R3s, and 6 R4s
    - Allocated: 0 R1s, 0 R2s, 1 R3, and 4 R4s
    - Max request = max needed - allocated = 0 R1s, 6 R2s, 4 R3s, and 2 R4s
- While safe sequence does not include all processes:
  - Is there P_i such that $MaxRequest_i$ <= available?
    - if no, exit with unsafe
  - Test this by creating a safe sequence that allows each process to complete.
- Is the state described before safe? YES!
  - We are able to create a sequence that allows each process to finish.

Updated Example: Determining Safety

- 5 processes, 4 resources

Max

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	7	5	0
P_3	2	3	5	6
P_4	0	6	5	2
P_5	0	6	5	6

Allocated

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	0	0	0
P_3	1	3	5	3
P_4	0	6	3	2
P_5	0	0	1	4

Available

R_1	R_2	R_3	R_4
1	5	2	0

- P2 wants to change its allocation to

0	4	2	0

- Safe?

Updated Example: Determining Safety

- 5 processes, 4 resources

Max

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	7	5	0
P_3	2	3	5	6
P_4	0	6	5	2
P_5	0	6	5	6

−

Allocated

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	0	4	2	0
P_3	1	3	5	3
P_4	0	6	3	2
P_5	0	0	1	4

Available

R_1	R_2	R_3	R_4
2	1	0	0

=

Max Request

	R_1	R_2	R_3	R_4
P_1	0	0	0	0
P_2	1	3	3	0
P_3	1	0	0	3
P_4	0	0	2	0
P_5	0	6	4	2

- P2 wants to change its allocation to

0	4	2	0

- Safe? No!

Updated Example: Determining Safety Plain Text

- P2 wants to change its allocated to: 0 R1s, 4 R2s, 2 R3s and 0 R4s
- This would change the metrics to:
  - First metric: maximum number of each kind of resource each process will need
    - P1: 0 R1s, 0 R2s, 1 R3, and 2 R4s
    - P2: 1 R1, 7 R2s, 5 R3s, and 0 R4s
    - P3: 2 R1s, 3 R2s, 5 R3s, and 6 R4s
    - P4: 0 R1s, 6 R2s, 5 R3s, and 2 R4s
    - P5: 0 R1s, 6 R2s, 5 R3s, and 6 R4s
  - Second metric: number of each kind of resource that has already been allocated to each process
    - P1: 0 R1s, 0 R2s, 1 R3, and 2 R4s
    - P2: 0 R1s, 4 R2s, 2 R3s, and 0 R4s
    - P3: 1 R1, 3 R2s, 5 R3s, and 3 R4s
    - P4: 0 R1s, 6 R2s, 3 R3s, and 2 R4s
    - P5: 0 R1s, 0 R2s, 1 R3, and 4 R4s
  - Third metric: number of each kind of resource in the system that is still available to be allocated
    - R1: 2
    - R2: 1
    - R3: 0
    - R4: 0
- Updated max requests by process:
  - P1:
    - Max needed: 0 R1s, 0 R2s, 1 R3, and 2 R4s
    - Allocated: 0 R1s, 0 R2s, 1 R3, and 2 R4s
    - Max request = max needed - allocated = 0 R1s, 0 R2s, 0 R3, and 0 R4s
  - P2:
    - Max needed: 1 R1, 7 R2s, 5 R3s, and 0 R4s
    - Allocated: 0 R1, 4 R2s, 2 R3s, and 0 R4s
    - Max request = max needed - allocated = 1 R1, 3 R2s, 3 R3s, and 0 R4s
  - P3:
    - Max needed: 2 R1s, 3 R2s, 5 R3s, and 6 R4s
    - Allocated: 1 R1, 3 R2s, 5 R3s, and 3 R4s
    - Max request = max needed - allocated = 1 R1, 0 R2s, 0 R3s, and 3 R4s
  - P4:
    - Max needed: 0 R1s, 6 R2s, 5 R3s, and 2 R4s
    - Allocated: 0 R1s, 6 R2s, 3 R3s, and 2 R4s
    - Max request = max needed - allocated = 0 R1s, 0 R2s, 2 R3s, and 0 R4s
  - P5:
    - Max needed: 0 R1s, 6 R2s, 5 R3s, and 6 R4s
    - Allocated: 0 R1s, 0 R2s, 1 R3, and 4 R4s
    - Max request = max needed - allocated = 0 R1s, 6 R2s, 4 R3s, and 2 R4s
- Is this allocation safe? NO!
  - We cannot create a sequence of requests such that all processes can finish.

Detecting Deadlock: Work at Home Problem

- 5 processes, 3 resources

Allocated

	R_1	R_2	R_3
P_1	0	1	0
P_2	2	0	0
P_3	3	0	3
P_4	2	1	1
P_5	0	0	2

Available

R_1	R_2	R_3
0	0	0

Max Request

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	2
P_3	0	0	0
P_4	1	0	2
P_5	0	0	2

- Given the set of pending requests is there a safe sequence?
  - If no, then deadlock!

Detecting Deadlock: Work at Home Problem

- 5 processes, 3 resources

Allocated

	R_1	R_2	R_3
P_1	0	1	0
P_2	2	0	0
P_3	3	0	3
P_4	2	1	1
P_5	0	0	2

Available

R_1	R_2	R_3
0	0	0

Max Request

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	2
P_3	0	0	1
P_4	1	0	2
P_5	0	0	2

- Given the set of maximum requests is there a safe sequence?
  - If no, then deadlock!

Detecting Deadlock: Work at Home Problem Plain Text

- 5 processes competing for 3 resources
- Allocated resources:
  - P1: 0 R1s, 1 R2, and 0 R3s
  - P2: 2 R1s, 0 R2s, and 0 R3s
  - P3: 3 R1s, 0 R2s, and 3 R3s
  - P4: 2 R1s, 1 R2, and 1 R3
  - P5: 0 R1s, 0 R2s, and 2 R3s
- Available resources:
  - R1: 0
  - R2: 0
  - R3: 0
- Max requests:
  - P1: 0 R1s, 0 R2s, and 0 R3s
  - P2: 2 R1s, 0 R2s, and 2 R3s
  - P3: 0 R1s, 0 R2s, and 0 R3s
  - P4: 1 R1, 0 R2s, and 2 R3
  - P5: 0 R1s, 0 R2s, and 2 R3s
- Given the set of maximum requests is there a safe sequence?
  - If no, the deadlock!
- Now consider a change in the system such that P3's max request becomes P3: 0 R1s, 0 R2s, and 1 R3
  - Is there a safe sequence?

iClicker Question

The Banker's Algorithm is a good choice for deadlock avoidance in a modern day OS

A. True

B. False

Weaknesses of Deadlock Avoidance Algorithms

- Must know resource requests of processes up front
- Processes may not enter the system
- Resources are assumed to always be working

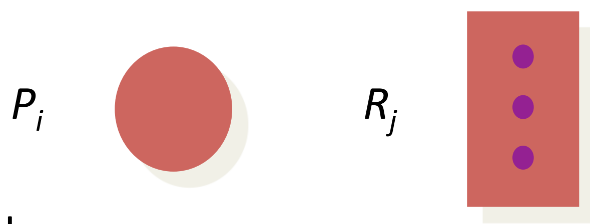
Deadlock Detection

Resource Allocation Graphs

- Loosely, graphs the state of resources in the system
- Used to detect deadlock
- Threads/processes are represented by circles
- Resources are represented by squares
- Arrows represent dependency
  - Arrows from a thread to a resource indicate “waiting for”
  - Arrows from resources to threads indicate “owned by”

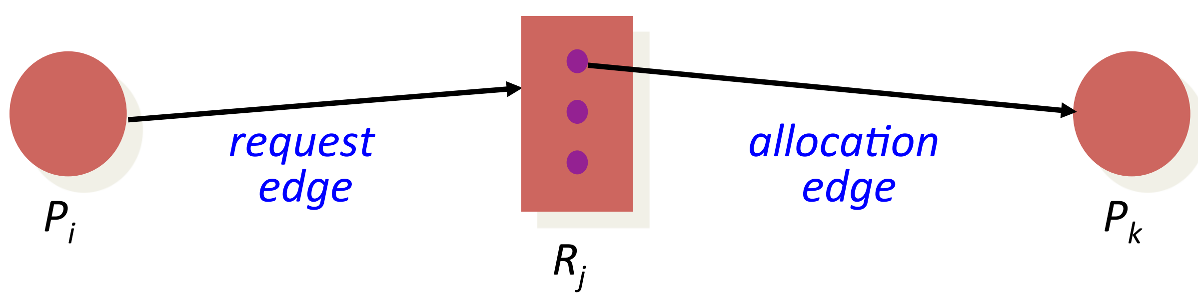
Resource Allocation Graphs, Formally

- Basic components of any resource allocation problem
  - Processes and resources
- Model the state of a computer system as a directed graph
  - $G = (V, E)$
  - V = the set of vertices = $\{P_1, \ldots, P_n\} \cup \{R_1, \ldots, R_m\}$



E = the set of edges =

{*edges from a resource to a process*} ∪ {*edges from a process to a resource*}



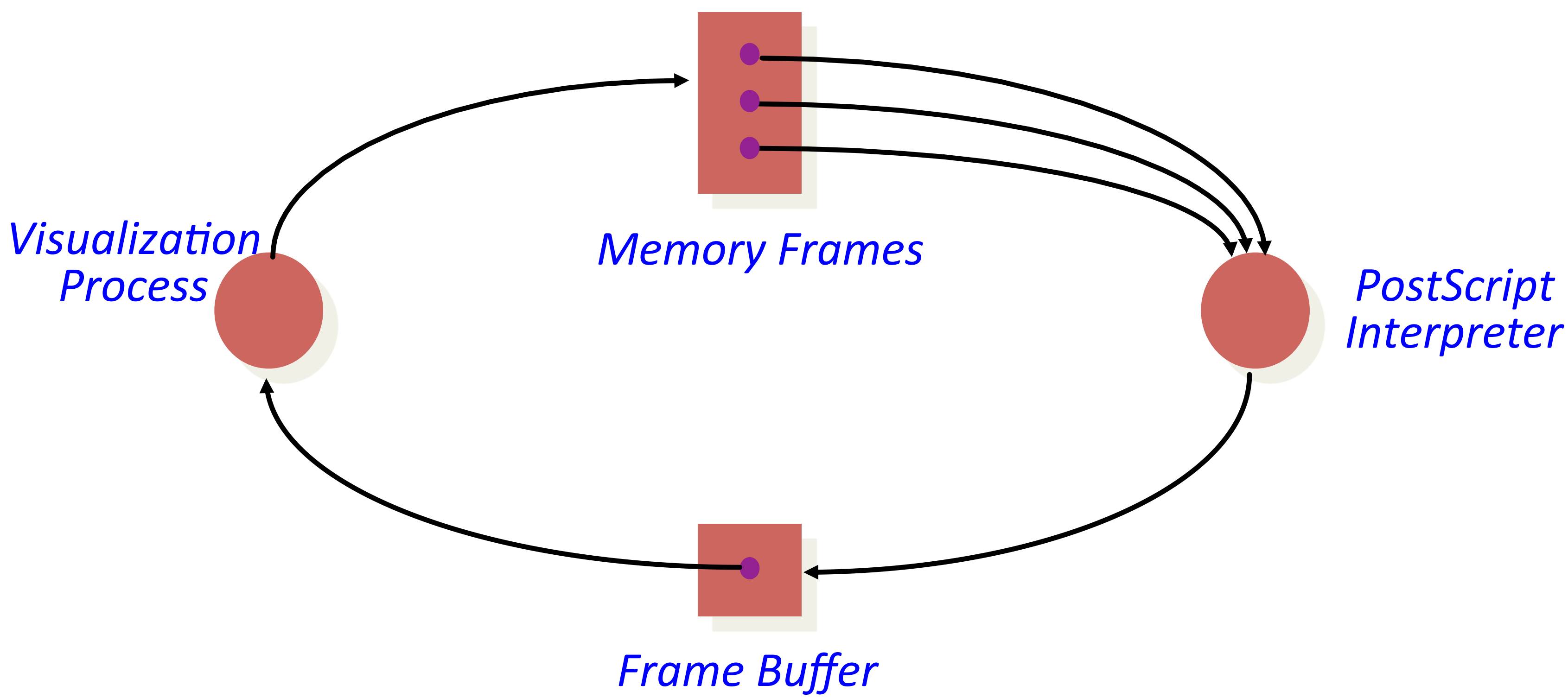
Resource Allocation Graphs, Formally: Plain Text

- Basic components of any resource allocation problem
  - Processes and resources
- Model the state of a computer system as a directed graph
  - G = (V, E)
  - V = set of vertices = $\{P_1, \ldots, P_n\}$ union $\{R_1, \ldots, R_m\}$
  - E = the set of edges = {edges from a resource to a process} union {edges from a process to a resource}
    - An edge from a process to a resource exists if the process is requesting that resource
    - An edge from a resource to a process exists if that resource is allocated to that process
- Resources modeled as rectangles that contain marks for every copy of said resources
  - ex: If there were multiple copiers there would just be 1 copier vertex with a mark for each copier that exists

Resource Allocation Graphs: An Example

A PostScript interpreter that is waiting for the frame buffer lock and a visualization process that is waiting for memory

V = {*PS interpret*, *visualization*} ∪ {*memory frames*, *frame buffer lock*}

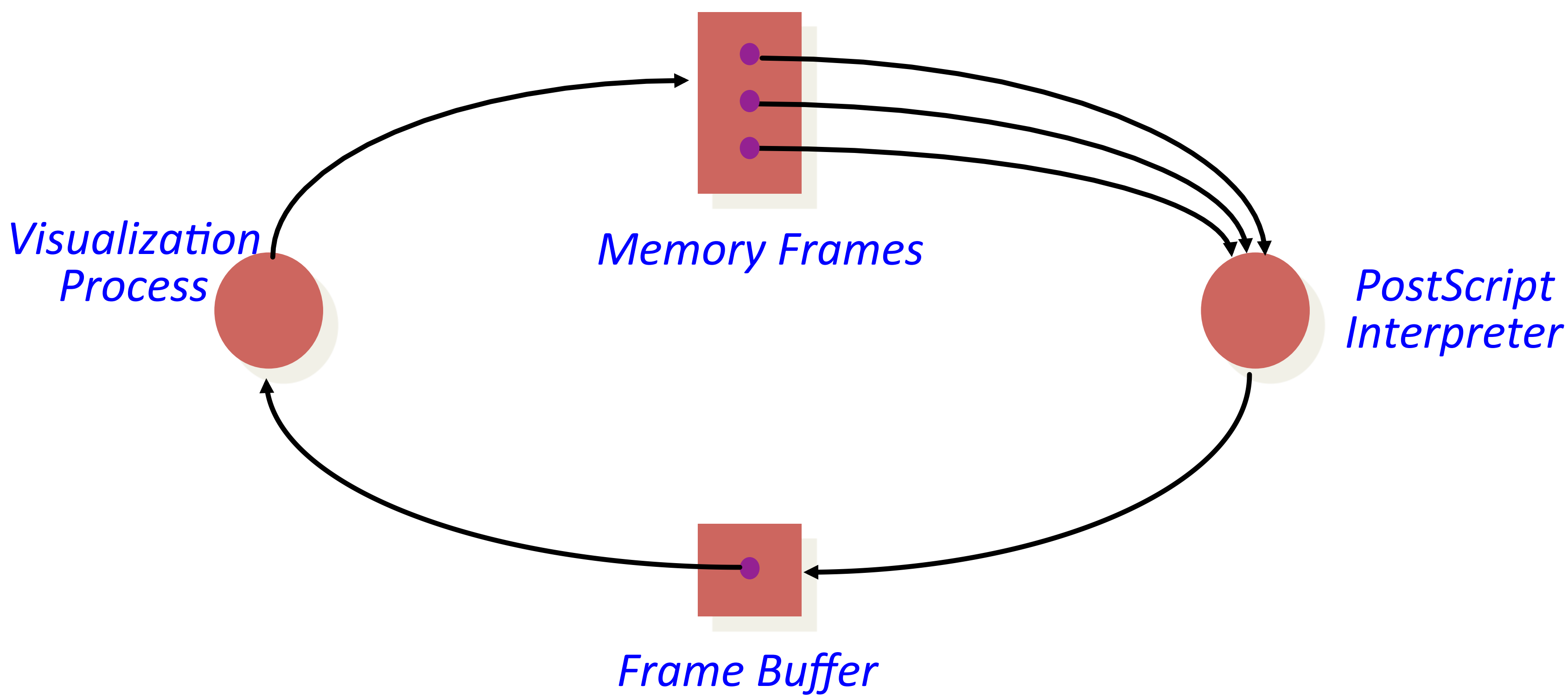


Resource Allocation Graphs: An Example Plain Text

- V = {Processes: PS interpreter, visualization} union {Resources: memory frames, frame buffer lock}
- Frame buffer has been allocated to the visualization process
- All memory frames have been allocated to the PostScript interpreter
- A PostScript interpreter that is waiting for the frame buffer lock and a visualization process that is waiting for memory
- The graph:
  – Nodes: memory frames (R1), PostScript Interpreter (P), Frame Buffer (R2) and Visualization Process (P)
  – Edges:
    - All 3 memory frames to PostScript Interpreter
    - PostScript Interpreter to frame buffer
    - Frame buffer to Visualization Process
    - Visualization Process to Memory Frames
  – There are cycles in the graph! What does this mean?

Resource Allocation Graphs: Cycles

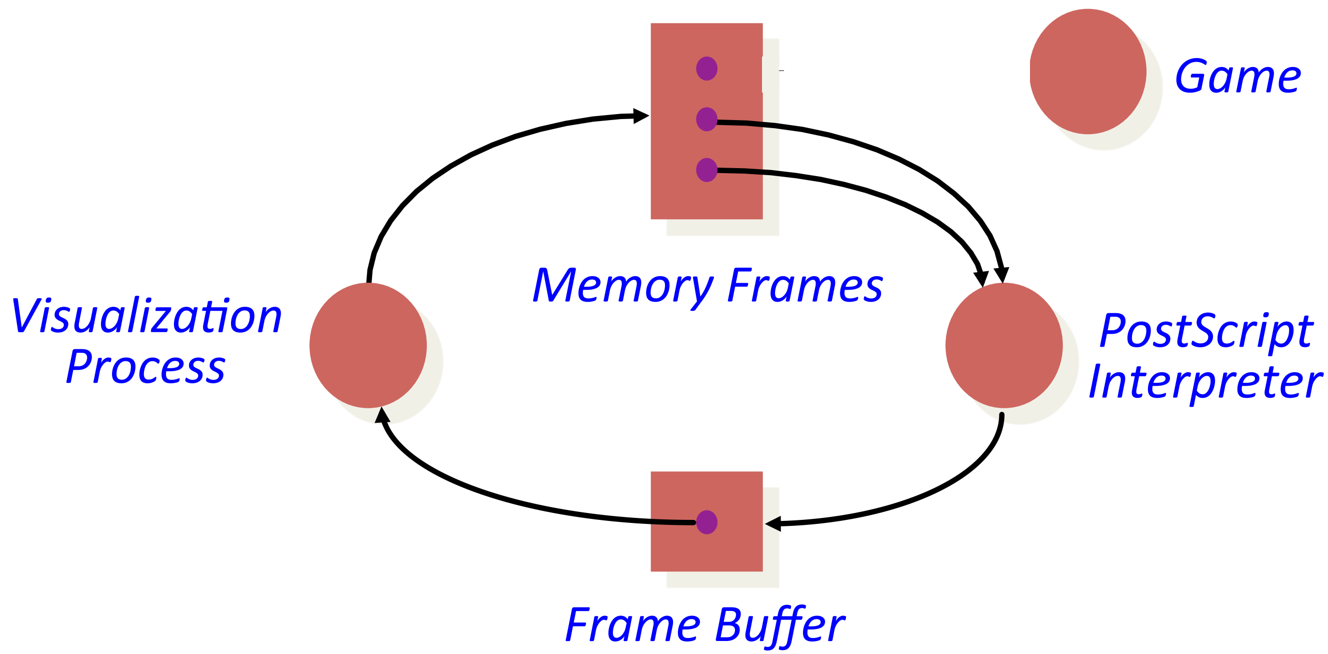
Theorem: If a resource allocation graph does not contain a cycle, then no processes are deadlocked.



Resource Allocation Graphs: Cycles

A cycle in a *RAG is* a necessary condition for deadlock

Is the existence of a cycle a sufficient condition?



Resource Allocation Graphs: Cycles Plain Text

- Theorem: if a resource allocation graph does not contain a cycle, then no processes are deadlocked
- A cycle in a RAG is necessary condition for deadlock
- Is the existence of a cycle a sufficient condition?
  - NO!
- Example in update of RAG from previous slide
  - Nodes: memory frames (R1), PostScript Interpreter (P), Frame Buffer (R2) and Visualization Process (P), Game (P)
  - Edges:
    - 2 memory frames allocated to PostScript Interpreter (means there is still 1 available)
    - 1 memory frame allocated to the game
    - PostScript Interpreter requesting frame buffer
    - Frame buffer allocated to Visualization Process
    - Visualization Process requesting memory frames
  - Over the course of time, the game process releases its memory frame
- How does this update change things?
  - Now the visualization process could get a memory frame and make progress

Cycles, In Words

- If the graph has no cycles, no deadlock exists
- If the graph has a cycle, deadlock *might* exist
  - If there is only a single unit of all resources then a set of processes are deadlocked if and only if there is a cycle in the resource allocation graph
  - If there are multiple instances of a resource(s) and any instance of a resource involved in the cycle is held by a thread/process that is not in the cycle, progress might be made when that thread/process releases the resource

Deadlock Detection

Using a resource allocation graph, scan the graph for cycles, then break the cycles

- Kill threads or processes in the cycle
- Kill threads or processes one at a time, forcing each to give up its resources
- Pre-empt resources one at a time, and rollback the state of the thread/process holding the resource to its state prior to acquiring the resource
  - Common to database transactions

Deadlock Detection

- Detecting cycles in the graph requires $O(n^2)$, where n is the number of vertices (processes and resources)
- When should we execute the algorithm?
  - Just before granting a resource?
  - When a request is denied?
  - On a regular schedule?
  - When CPU utilization drops below some threshold?

Deadlock Handling: Real Life

- Ostrich Algorithm

Summary

Deadlock is a situation in which a set of threads/processes cannot proceed because each requires resources held by another member of the set. Approaches to handling deadlock are:

- Prevention: design resource allocation strategies that guarantee that one of the necessary conditions never holds
- Avoidance: don't allocate a resource if it would introduce a cycle
- Detection and recovery: recognize deadlock after it has occurred and break it
- In real life:
  - For resources managed by the program, code carefully! (Does not work for OS managed resources)
  - Ignore the possibility!

Announcements

- Homework 11 (Last one!) due Friday, 5/1, 8:45a
- Project 4 (Last one!) due Friday, 5/8, 11:59p
  - No slip days!
- If you have a conflict for the final, you should have already contacted me (email, please!)
  - Thursday, May 14, 7p-10p in UTC 2.102A