Deadlocks

Today
- Resources & deadlocks
- Dealing with deadlocks
- Other issues

Next Time
- Midterm!
System model

- System – a collection of resources to be shared among a set of processes
- Resources partitioned in types, each with multiple instances (printers, files, memory,…)
- Resources can be
  - Preemptable - can be taken away from process w/o ill effects e.g. memory
  - Nonpreemptable - process will fail if resource was taken away e.g. CD recorder
- A request for resource type R can be satisfied by any instance of the type
System model

- A process must request a resource before using it & release it once done \((\text{open/close, malloc/free, ...})\)

- Sequence of events to use a resource
  1. request it – if not granted then block or return error \(\text{down}(\text{semaphore})\)
  2. use it
  3. release it \(\text{up}(\text{semaphore})\)

- Suppose
  - Process A holds resource R & ... requests S
  - Process B holds resources S & ... requests R
  - A & B are now blocked
Deadlock modeling

- Modeled with directed graphs
  - Process B is requesting/waiting for resource S
  - Resource R assigned to process A
  - Process C & D in deadlock over resources T & U

You can generalize it to multiple resource instances per class
Introduction to deadlocks

- **Formal definition**
  
  *A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.*

- **None of the processes can …**
  - run
  - release resources
  - be awakened

- **Assumptions**
  - Processes are single threaded
  - There are no interrupts possible to wake up a blocked process

- **A “cute” example**
  “When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up until the other has gone.” An actual law passed by the Kansas legislature …
Deadlock modeling

Clearly, the ordering of operations plays a role

Requests and releases of each process

A
- Request R
- Request S
- Release R
- Release S

B
- Request S
- Request T
- Release S
- Release T

C
- Request T
- Request R
- Release T
- Release R

1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R

....

deadlock

1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S

....

no deadlock
Conditions for deadlock

1. Mutual exclusion - Each resource assigned to 1 process or available
2. Hold and wait - A process holding resources can request others
3. No preemption - Previously granted resources cannot forcibly be taken away
4. Circular wait – A circular chain of 2+ processes, each waiting for resource held by next one

All conditions must hold for a deadlock to occur. Each of the 1-3 conditions is associated with a policy the system can or not have; break one condition → no deadlock
Dealing with deadlocks

Possible strategies

- Ignore the problem altogether – ostrich “algorithm”
- Detection and recovery – do not stop it; let it happen, detect it and recover from it
- Dynamic avoidance – careful resource allocation
- Prevention – negating one of the four necessary conditions
The ostrich algorithm

- Pretend there is no problem
- Reasonable if
  - deadlocks occur very rarely
  - cost of prevention is high
- UNIX’s & Windows’ approach
- A clear trade off between
  - convenience
  - correctness
Basic facts

- If graph contains no cycles $\Rightarrow$ no deadlock.
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, maybe a deadlock.
Deadlock detection – single instance

- How, when & what
- Simplest case

1. L ← empty
   all arcs set as unmarked
2. For each node N
   /* depth-first search */
   2.1. Add N to L & check
       if N in L twice there’s a deadlock; exit
   2.2. Pick one arc at random,
       mark it & follow it to next current node
3. At end, if no arc no deadlock

Arcs:

- A→S, A←R, B→T, C→S
- D→S, D←T, E→V, E←T
- F→S, F←W, G→V, G←V

L:[R], L:[R,A], L:[R,A,S]
L:[B], L:[B,T], L:[B,T,E], ...
Detection - multiple instances

\[ \text{n processes, m classes of resources} \]
\[ E \rightarrow \text{vector of existing resources} \]
\[ A \rightarrow \text{vector of available resources} \]
\[ C \rightarrow \text{matrix of currently allocated resources} \]
\[ R \rightarrow \text{request matrix} \]
\[ C_{ij} \rightarrow P_i \text{ holds } C_{ij} \text{ instances of resource class } j \]
\[ R_{ij} \rightarrow P_i \text{ wants } C_{ij} \text{ instances of resource class } j \]

\[ \text{Invariant} \quad \sum_i C_{ij} + A_j = E_j \]
\[ (\text{Currently allocated + available = existing}) \]
\[ \text{i.e. all resources are either allocated or available} \]

Algorithm:

All processes unmarked
1. Look for unmarked process \( P_i \) for which \( R_i \leq A \)
2. If found, add \( C_i \) to \( A \), mark the process and go to 1
3. If not, exit

All unmarked processes, if any, are deadlock

Idea: See if there’s any process that can be run to completion with available resources, mark it and free its resources …
Detection

(existing) (available)
E = (4 2 3 1) A = (2 1 0 0)

C =
| 0 0 1 0 |
| 2 0 0 1 |
| 0 1 2 0 |

R =
| 2 0 0 1 |
| 1 0 1 0 |
| 2 1 0 0 |

What process 1 needs

Algorithm:

All processes unmarked
1. Look for unmarked process \( P_i \) for which \( R_i \leq A \)
2. If found, add \( C_i \) to \( A \), mark the process and go to 1
3. If not, exit
All unmarked processes, if any, are deadlock

Idea: See if there’s any process that can be run to completion with available resources, mark it and free its resources …

Three processes and 4 resource types

After running process 3
A = (2 2 2 0)

Now you can run process 2
A = (4 2 2 1)
When to check & what to do

- **When to try**
  - Every time a resource is requested
  - Every fixed period of times or when CPU utilization drops

- **What to do then - recovery**
  - Through preemption
    - depends on nature of the resource
  - Through rollback
    - Need to checkpoint processes periodically
  - By killing a process
    - Crudest but simplest way to break a deadlock
    - Kill one in or not in the deadlock cycle
Deadlock avoidance

- Dynamically make sure not to get into a deadlock
- Two process resource trajectories

Your only option here is to run A up to $I_4$

* u (Both processes done)
Safe and unsafe states

- **Safe if**
  - There is no deadlock
  - There is some scheduling order by which all processes can run to completion

- **Un-safe is not deadlock – just no guarantee**

Example with one resource (10 instances of it)

<table>
<thead>
<tr>
<th>Safe</th>
<th>Has</th>
<th>Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free: 3</td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unsafe</th>
<th>Has</th>
<th>Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free: 3</td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

A requests and is granted another instance

In retrospect, A’s request should not have been granted
Banker's algorithm

- **Considers**
  - Each request as it occurs
  - Sees if granting it leads to a safe state i.e. there are enough resources to satisfy one customer

- **With multiple resources**
  1. Look for a row $R_i \leq A$, if none the system will eventually deadlock
  2. If found, mark $P_i$ and add $C_i$ to $A$
  3. Repeat until processes are terminated or a deadlock occurs

- **Very cute, but mostly useless**
  - Most processes don’t know in advance what they need
  - The lists of processes and resources are not static
  - Processes may depend on each other
Deadlock prevention

- Avoidance is pretty hard or impossible
- Can we break one of the condition?
  - Mutual exclusion
  - Hold & wait
  - No preemption
    - Not a viable option
    - How can you preempt a printer?
  - Circular wait
Attacking mutual exclusion

- Some devices can be spooled (printer)
  - Only the printer daemon uses printer resource
  - Thus deadlock for printer eliminated
- But not all devices can be spooled – process table?
- Principle:
  - Assigning resource only when absolutely necessary
  - Reduce number of processes that may claim the resource
Attacking hold & wait

- Processes request all resources at start (wait)
  - Process never has to wait for what it needs
- But
  - May not know required resources at start
  - It ties up resources others could be using
- Variation (hold)
  - Process must release all resources to request a new one
Attacking circular wait

- Impose total order on resources
- Processes request resources in order
- If all processes follow order, no circular wait occurs

Deadlock if $i \rightarrow A \rightarrow j$ & $j \rightarrow B \rightarrow i$
If $i < j$ then $A \rightarrow j$ ...

- Process cannot request resource lower than what it’s holding
- Advantage - Simple
- Disadvantage - Arbitrary ordering
Related issues

- **Two-phase locking** – gather all locks, work & free all
  - If you cannot get all, drop all you have and start again

- **Non-resource deadlocks**
  - Each is waiting for the other to do some task
  - E.g. communication deadlocks:
    - A sends a request and blocks until B replies, message gets lost!
    - Timeout!

- **Starvation**
  - Algorithm to allocate a resource
  - SJF – consider allocation of a printer
    - Great for multiple short jobs in a system
    - May cause long job to be postponed indefinitely
      - even though not blocked
  - Solution: FIFO
Next time

- We have discussed sharing CPU to improve utilization and turnaround time
- For that to happen we also need to share memory
- We’ll start with memory organization and basic management techniques (e.g. paging)
- Before moving to memory virtualization …
- … of course, all this after the midterm!