CMSC 321: Operating Systems
Lecture 9
Deadlocks
Don’t Be Confused

deadlock
dreadlock
What is Deadlock?

- *Permanent blocking* of a set of processes that
  - compete for system resources, or
  - communicate with each other

- No efficient solution in general

- Involves conflicting needs for resources
  - hardware or software
  - in general, *nonpreemptable* resources
    - *preemptable*: taken away with no ill effect (e.g., RAM)
    - *nonpreemptable*: if taken away, causes failure (e.g., burning CD)
Illustration of Deadlock

(a) Deadlock possible

(b) Deadlock
Using Resources

- Events required to use a resource:
  - request
  - use
  - release

- Process must wait if request denied
  - may be blocked
  - request may fail with error code (try again)

- Deadlocks can occur when processes have been granted exclusive access to resources
More Formally

- **Deadlock**: set of processes in which each process is waiting for an event that only another process in the set can cause.

- Usually, event needed is releasing a resource.

- None of the processes can
  - run
  - release a resource
  - be awakened
Four Conditions for Deadlock

1. **Mutual exclusion:**
   only one process may use resource at a time

2. **Hold-and-wait:**
   may hold resources while awaiting others

3. **No preemption:**
   no resource can be forcibly removed from holding process

4. **Circular wait:**
   chain of two or more processes, each waiting for resource held by next in chain

First three are necessary, but not sufficient, for deadlock
Circular Wait

- Unresolvable circular wait is another definition of deadlock
Digraphs to Model Deadlocks

- Processes are circles, resources are squares
  - directed arc from circle to square:
    - process is requesting, but does not possess, resource
  - directed arc from square to circle:
    - process currently possesses that resource

- Deadlock: a cycle in the graph
An Example Using Digraph

Request R  
Request S  
Release R  
Release S

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

Request S  
Request T  
Release S  
Release T

Request T  
Release T  
Release R
An Example Using Digraph

Cycle in step (j) represents deadlock
An Example Using Digraph

- To avoid deadlock, could deny B’s request
Dealing with Deadlocks

• Four strategies:
  – ignore
  – detect and recover
    allow deadlocks to occur, detect, and then take action
  – avoid
    carefully allocate resources to avoid deadlock
  – prevent
    eliminate one of the four necessary conditions for deadlock
The Ostrich Algorithm

- Pretend there is no problem
- Reasonable if
  - deadlocks occur rarely
  - cost of prevention is high

- Unix and Windows take this approach
  - general solutions are hard to find

- Trade-off: convenience vs. correctness

- “Actually, this bit of folklore is nonsense. Ostriches can run at 60 km/hour and their kick is powerful enough to kill any lion with visions of a big chicken dinner.” (Tanenbaum)
Detect and Recover

- Allow deadlocks to occur
- Periodically check if circular wait exists

- Consider one resource of each type:
  - Process A holds R, wants S
  - Process B wants T
  - Process C wants S
  - Process D holds U, wants S and T
  - Process E holds T, wants V
  - Process F holds W, wants S
  - Process G holds V, wants U

- Construct the resource graph...
One Resource of Each Type

Cycle exists, denoting deadlock
Detection Algorithm

- Need a formal algorithm to detect cycle
- Ideas?
Detection Algorithm

• For each node $n$:
  – List $L = \emptyset$, all edges unmarked
  – Append $n$ to $L$, check if $n$ appears twice
    • if so, cycle is present; terminate
  – If unmarked outgoing edges from $n$:
    • pick one at random, mark, follow to new $n$, goto step 2
  – If none:
    • remove $n$, and return to previous $n$, goto step 2

• Take each node as root of (hopefully) tree
  – performs depth-first search

• There are better algorithms...
Multiple of Each Resource

<table>
<thead>
<tr>
<th>Tape drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD Roms</th>
</tr>
</thead>
<tbody>
<tr>
<td>E = ( 4 2 3 1 )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tape drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD Roms</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = ( 2 1 0 0 )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Current allocation matrix

\[
C = \begin{bmatrix}
0 & 0 & 1 & 0 \\
2 & 0 & 0 & 1 \\
0 & 1 & 2 & 0
\end{bmatrix}
\]

Request matrix

\[
R = \begin{bmatrix}
2 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0
\end{bmatrix}
\]
Matrix-based Algorithm

- Initially, each process is “unmarked”

- Find unmarked $P_i$ for which $i^{th}$ row of $R < A$
  - If found:
    - add $i^{th}$ row of $C$ to $A$
    - mark $P_i$
    - goto first step
  - If not found, terminate

- At finish, all unmarked processes are deadlocked

- Looking for process than can run to completion
Multiple of Each Resource

<table>
<thead>
<tr>
<th>Tape drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD Roms</th>
</tr>
</thead>
<tbody>
<tr>
<td>E = (4 2 3 1)</td>
<td>A = (2 1 0 0)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Current allocation matrix

\[
C = \begin{bmatrix}
0 & 0 & 1 & 0 \\
2 & 0 & 0 & 1 \\
0 & 1 & 2 & 0 \\
\end{bmatrix}
\]

Request matrix

\[
R = \begin{bmatrix}
2 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0 \\
\end{bmatrix}
\]

P₁, P₂, P₃
Multiple of Each Resource

<table>
<thead>
<tr>
<th>Tape drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD Roms</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tape drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD Roms</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-2</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Current allocation matrix

\[
C = \begin{bmatrix}
0 & 0 & 1 & 0 \\
2 & 0 & 0 & 1 \\
0 & 1 & 2 & 0
\end{bmatrix}
\]

Request matrix

\[
R = \begin{bmatrix}
2 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0
\end{bmatrix}
\]

P₃
Multiple of Each Resource

<table>
<thead>
<tr>
<th>Tape drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD Roms</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Current allocation matrix

\[
C = \begin{bmatrix}
0 & 0 & 1 & 0 \\
2 & 0 & 0 & 1 \\
0 & 1 & 2 & 0
\end{bmatrix}
\]

Request matrix

\[
R = \begin{bmatrix}
2 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0
\end{bmatrix}
\]

P₂

P₃
Multiple of Each Resource

\[
E = (4 \quad 2 \quad 3 \quad 1)
\]

\[
A = (4 \quad 2 \quad 3 \quad 1)
\]

\[
C = \begin{bmatrix}
0 & 0 & 1 & 0 \\
2 & 0 & 0 & 1 \\
0 & 1 & 2 & 0
\end{bmatrix}
\]

\[
R = \begin{bmatrix}
2 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0
\end{bmatrix}
\]
When to Check for Deadlock?

- Every time a resource request is made
  - expensive

- Every $k$ time units

- If CPU utilization drops below threshold
  - likely that processes are deadlocked
Recovering from Deadlock

- Preemption: take a resource from a process
  - depends on nature of resource

- Rollback: use periodic checkpoints
  - restart process at checkpoint if deadlocked

- Kill process:
  - can kill process in the cycle
  - can kill select process outside the cycle
  - should kill a process that can be rerun from start
Deadlock Avoidance

- Dynamically decide if current request can potentially lead to deadlock

- Requires knowledge of future process requests
- Based on concept of *safe state*:
  - not deadlocked
  - some sequence allows all processes to run to completion

- Note: unsafe state is *not* deadlocked state
  - just no guarantee that all processes will finish
Visual Depiction of Deadlock

At $t$, must run $A$ until $I_4$
Banker’s Algorithm

• Uses vectors and matrices from before
  – $E, A, C, R$

• When process makes request, assume the request is granted

• Update the system accordingly

• Determine if safe state
  – if so, grant the request
  – if not, block until safe to grant the request
Example: Safe State?

(a) Initial state

(b) P2 runs to completion
Example: Safe State?

Claim Matrix | Allocation Matrix
---|---
| P1 | P2 | P3 | P4 |
| R1 | R2 | R3 | R1 | R2 | R3 | R1 | R2 | R3 |
| 3 | 2 | 2 | 1 | 0 | 0 | 6 | 2 | 3 |
| 0 | 0 | 0 | 0 | 0 | 0 |   |   |   |
| 3 | 1 | 4 | 2 | 1 | 1 |   |   |   |
| 4 | 2 | 2 | 0 | 0 | 2 |   |   |   |

Available Vector

(b) P2 runs to completion

Claim Matrix | Allocation Matrix
---|---
| P1 | P2 | P3 | P4 |
| R1 | R2 | R3 | R1 | R2 | R3 | R1 | R2 | R3 |
| 0 | 0 | 0 | 0 | 0 | 0 | 7 | 2 | 3 |
| 0 | 0 | 0 | 0 | 0 | 0 |   |   |   |
| 3 | 1 | 4 | 2 | 1 | 1 |   |   |   |
| 4 | 2 | 2 | 0 | 0 | 2 |   |   |   |

Available Vector

(c) P1 runs to completion
Example: Safe State?

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Claim Matrix  Allocation Matrix

(c) P1 runs to completion

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Claim Matrix  Allocation Matrix

(d) P3 runs to completion
Another Example: P1 wants R1, R3

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>6</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Claim Matrix

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Allocation Matrix

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Resource Vector

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Available Vector

Not a deadlocked state --- but has potential for one

(a) Initial state

(b) P1 requests one unit each of R1 and R3
Banker’s Algorithm: Caveat

• Essentially useless in practice
  - rarely know resource needs a priori
  - # processes varies across time
  - resources can appear and vanish

• Few, if any, systems use the algorithm

• Still, important to understand concept
Deadlock Prevention

• Attempt to eliminate one of the four necessary conditions for deadlock

• *Mutual Exclusion*:
  – if no exclusive access, no deadlocks!
  – in general, unrealistic

  – still, avoid assigning a resource when not absolutely necessary
Deadlock Prevention

• *Hold and Wait*:  
  - require all requests before executing  
    • unrealistic  
    • less than optimal use of resources  
  
  - when requesting, require process to temp. release all held resources  
    • then try to get everything all @ once  

• *No preemption*:  
  - most difficult  
  - applicable only for resources w/ state easily saved/restored
Deadlock Prevention

- **Circular Wait:**
  - provide global numbering to resources
  - requests must be made in numerical order
  - will never have cycles, even with multiple resources!

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD Rom drive

- variation: no process requests resource lower than any already held
Dining Philosophers

```c
semaphore fork[5] = {1};

philosopher(int myPos) {
    while (true) {
        think();
        wait(fork[i]);
        wait(fork[(myPos+1) % N]);
        chowDown();
        signal(fork[i]);
        signal(fork[(myPos+1) % N]);
    }
}
```

Deadlock: what if all five pick up first fork @ same time?
Dining Philosophers

Solution: allow only four into the room at once!

```c
semaphore fork[5] = {1};
semaphore room = {4};

philosopher(int myPos) {
    while (true) {
        think();
        wait(room);
        wait(fork[i]);
        wait(fork[(myPos+1) % N]);
        chowDown();
        signal(fork[i]);
        signal(fork[(myPos+1) % N]);
        signal(room);
    }
}
```
Deadlock: Recap

- Good general solutions are difficult, if not impossible

- In practice, often just hope for the best