CS450/550 Operating Systems
Lecture 3 Deadlocks

Dr. Xiaobo Zhou
Department of Computer Science

Review: Summary of Chapter 2

° Sequential process model
° Multi-threading: user-space vs. kernel-space
° IPC: semaphores, monitors, messages
  • Race conditions
  • Mutual exclusion
  • Critical regions
  • Classic IPC problems
° Scheduling
  • Process scheduling
  • Thread scheduling
° More reading: textbook 2.1 - 2.7
Chapter 3: Deadlocks

3.1. Resources
3.2. Introduction to deadlocks
3.3. The ostrich algorithm
3.4. Deadlock detection and recovery
3.5. Deadlock avoidance
3.6. Deadlock prevention
3.7. Other issues

Deadlock Legends (by Tom Eggers)

Deadlock Legends

Texas Law:
“When two trains meet each other at a railroad crossing, each shall come to a full stop, and neither shall proceed until the other has gone.”

Tombstone, Arizona Territory, 1881:
Doc Holliday: “Drop your gun, Billy!”
Billy Clanton: “You first, Doc!”

Fall 2005
**Deadlock Definitions**

1. Two or more processes each blocked and waiting for resources they will never get without drastic actions
   - Something preempts a resource
   - A process is killed

2. 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, thus, no process can
   - run
   - release resources
   - be awakened

**Resources and Deadlocks (1)**

- Examples of computer resources
  - printers
  - tape drives
  - tables
  - software

- Processes need access to resources in reasonable order

- Suppose a process holds resource A and requests resource B
  - at same time another process holds B and requests A
  - both are blocked and remain so, *deadlocks*!

**Why is the key problem in the example?**
Both processes want to have *exclusive access* to A and B!
Deadlocks occur when ...  
• processes are granted *exclusive access* to hardware, e.g., I/O devices  
• processes are granted *exclusive access* to software, e.g., database records  
• we refer to these generally as *resources*  

Preemptable resources  
• can be taken away from a process with no ill effects, e.g., Mem  

Nonpreemptable resources  
• will cause the process to fail if taken away, e.g., CD burner  

In general, deadlocks involve *non-preemptive* and *exclusive* resources!

Sequence of events required to use a resource  
1. request the resource  
2. use the resource  
3. release the resource  

Must wait if request is denied  
• requesting process may be blocked  
• may fail with error code
Can using semaphores avoid deadlocks?

```c
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A (void) {
    down(&resource_1);
    use_resource_1 ();
    up(&resource_1);
    down(&resource_2);
    use_both_resources();
    up(&resource_2);
    up(&resource_1);
}

void process_B (void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources();
    up(&resource_2);
    up(&resource_1);
    up(&resource_2);
}
```

Using semaphore to protect resources. (a) One resource. (b) Two resources.

But using semaphores wisely!

---

(a) Deadlock-free code. (b) Code with a potential deadlock, why?
Four Conditions for Deadlock

Coffman (1971)

1. Mutual exclusion condition
   - each resource assigned to 1 process or is available

2. Hold and wait condition
   - process holding resources can request additional

3. No preemption condition
   - previously granted resources cannot forcibly taken away

4. Circular wait condition
   - must be a circular chain of 2 or more processes
   - each is waiting for resource held by next member of the chain

All four must be met for a deadlock to occur!

---

Deadlock Modeling (1)

- Modeled with directed graphs
  - A cycle means a deadlock involving the processes and resources

(a) A → R
(b) S → B
(c) T → U → C → D

- resource R assigned to process A
- process B is requesting/waiting for resource S
- process C and D are in deadlock over resources T and U
### Deadlock Modeling (2)

#### Sequential model
- No deadlock, no parallelism.

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)

(a) – (c)

#### How deadlock occurs

What if the OS knew the impending deadlock of granting B resource S at step (f)?

**How deadlock occurs**

### Deadlock Modeling (3)

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)

(k) – (n)

**How deadlock can be avoided by OS’ re-ordering**
Strategies for dealing with Deadlocks

1. just ignore the problem altogether
2. detection and recovery
3. dynamic avoidance
   • careful resource allocation
4. prevention
   • negating one of the four necessary conditions

The Ostrich Algorithm

° Pretend there is no problem

° Reasonable if
  • deadlocks occur very rarely, e.g., fork() -> process table
  • cost of prevention is high

° UNIX and Windows takes this approach

° It is a trade off between
  • convenience
  • correctness
Detection with One Resource of Each Type (1)

- Assumption: only one resource of each type exists

![Diagram showing resource ownership and requests]

- Note the resource ownership and requests
- A cycle can be found within the graph, denoting deadlock

Detect a Cycle in a Graph

- A data structure to find if a graph is a tree that is cycle-free
  - depth-first searching (P.170)
  - Left-right, top-to-bottom: R, A, B, C, S, D, T, E, F

![Diagram showing detection of cycles]
Detection with Multiple Resources of Each Type (1)

- Deadlock detection algorithm:
  - Two vectors and two matrixes
  - Vector comparison; $A \leq B$ means $A_i \leq B_i$ for $1 \leq i \leq m$
  - Observation: $\sum C_{ij} + A_j = E_j$

Resources in existence
$(E_1, E_2, E_3, \ldots, E_m)$

Resources available
$(A_1, A_2, A_3, \ldots, A_m)$

Current allocation matrix
\[
\begin{bmatrix}
C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\
C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm}
\end{bmatrix}
\]

Row $n$ is current allocation to process $n$

Request matrix
\[
\begin{bmatrix}
R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\
R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm}
\end{bmatrix}
\]

Row $2$ is what process $2$ needs

Data structures needed by deadlock detection algorithm

Detection with Multiple Resources of Each Type (2)

- Key: a completed process can release its resources so as to other processes chances to acquire resources and run
  - Look for a process $P_i$, if $R[i] \leq A$? if so, $A = R[i] + C[i]$

E = (4 2 3 1)

A = (2 1 0 0)

What if process 2 needs a CD-ROM drive and 2 tape drivers and the plotter?

When to run the deadlock detection algorithm? Why CPU utilization?

Current allocation matrix
\[
\begin{bmatrix}
0 & 0 & 1 & 0 \\
2 & 0 & 0 & 1 \\
0 & 1 & 2 & 0
\end{bmatrix}
\]

Request matrix
\[
\begin{bmatrix}
2 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0
\end{bmatrix}
\]

An example for the deadlock detection algorithm
**Recovery from Deadlock**

- Recovery through preemption
  - take a resource from some other process
  - depends on nature of the resource

- Recovery through rollback
  - *checkpoint* a process periodically, resulting a sequence of checkpoint files
  - use this saved state
  - restart the process if it is found deadlocked
  - database applications; network services; *recoverable?*

- Recovery through killing processes
  - crudest but simplest way to break a deadlock
  - kill one of the processes in the deadlock cycle
  - the other processes get its resources
  - choose process that can be rerun from the beginning, *not easy!*

**Deadlock Avoidance**

- Allocate resources wisely to avoid deadlocks
  - But certain information should be available in advance
  - Base: concept of safe states

---

**Joint Progress Diagram**

Given: two processes named P1 and P2

- P1 starts at left and ends at right
- P2 starts at bottom and ends at top

What “trajectories” can they follow from lower left to upper right?
Deadlock Avoidance: Resource Trajectories

- Trajectories move only right and up!
  - Programs do not run backwards
  - All paths must be horizontal or vertical, never diagonal (if 1 CPU)

Two process resource trajectories

Safe States (w/ one resource type)

- Safe state
  - if it is not deadlocked, and, there is some scheduling order in which every process can run to completion even if all of them request their maximum number of resources immediately

<table>
<thead>
<tr>
<th>Has A</th>
<th>Max</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>9</td>
<td>2</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

Why state (a) is safe?
Unsafe States (w/ one resource type)

- **Unsafe state**
  - there is *no guarantee* of having some scheduling order in which every process can run to completion even if all of them request their maximum number of resources immediately
  - Not the same as a deadlocked state, why? What is the difference?

<table>
<thead>
<tr>
<th></th>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

Free: 3

(a) Why state (b) is NOT safe?

The Banker's Algorithm for a Single Resource

- **The algorithm models on the way of a banker might deal with a group of customers to whom he has granted lines of credit**
  - Not all customers need their maximum credit line simultaneously
  - To see if a state is safe, the banker checks to see if he has enough resources to satisfy some customer

<table>
<thead>
<tr>
<th></th>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

Free: 10

(a)

<table>
<thead>
<tr>
<th></th>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>

Free: 2

(b)

<table>
<thead>
<tr>
<th></th>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>

Free: 1

(c)

Three resource allocation states: (a) safe; (b) safe; (c) unsafe
The Banker’s Algorithm for Multiple Resources

- The algorithm looks for a process \( P_i \), if \( R[i] \leq A \) if so, \( A = R[i] + C[i] \)
  - How \( R \) is achieved? \( R = M \) (Maximum) - \( C \)
  - What is the underlying assumption? \( M \) info available in advance

<table>
<thead>
<tr>
<th>Process</th>
<th>Tape drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD ROMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\( R \) achieved:

<table>
<thead>
<tr>
<th>Process</th>
<th>Tape drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD ROMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\( E = M - C \)

If process B requests a scanner, can it be granted? Why?

After process B was granted a scanner, now process E wants the last scanner, can it be granted? Why?

Why in practice the algorithm is essentially useless?
Deadlock Prevention (1)

Attack the mutual exclusion condition of Coffman Rules

° Some devices (such as printer) can be spooled
  • only the printer daemon uses printer resource
  • thus deadlock for printer eliminated
  • But the disk could be deadlocked, though more unlikely

° Not all devices can be spooled, e.g., process table

° Principle:
  • avoid assigning resource when not absolutely necessary
  • as few processes as possible actually claim the resource

Deadlock Prevention (2)

Attack the Hold-and-Wait condition of Coffman Rules

° Require processes to request all resources before starting
  • a process never has to wait for what it needs

° Problems
  • may not know required resources at start of run
  • also ties up resources other processes could be using
    - Less concurrency!

° Variation:
  • process must temporarily give up all resources
  • then request all immediately needed
Deadlock Prevention (3)

Attack the No-Preemption Condition of Coffman Rules
° This is not a viable option
° Consider a process given the printer
  • halfway through its job
  • now forcibly take away printer
  • !??

Deadlock Prevention (4)

Attack the Circular Wait Condition of Coffman Rules
° A process is entitled only to a single resource at any moment
° Provide a global numbering of all the resources
  • A process can request resources whenever they want to, but all requests must be made in numerical order (or no process requests a resource lower than what it is already holding)
  • Why no deadlock?
  • Is it feasible in implementation? – what is a good ordering?

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

(a) Normally ordered resources

(b) a resource graph
Deadlock Prevention Summary

<table>
<thead>
<tr>
<th>Condition</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutual exclusion</td>
<td>Spool everything</td>
</tr>
<tr>
<td>Hold and wait</td>
<td>Request all resources initially</td>
</tr>
<tr>
<td>No preemption</td>
<td>Take resources away</td>
</tr>
<tr>
<td>Circular wait</td>
<td>Order resources numerically</td>
</tr>
</tbody>
</table>

Summary of approaches to deadlock prevention

What is the difference between deadlock avoidance and deadlock prevention?

dynamic scheduling vs. static ruling

Two-Phase Locking

- **Phase One**
  - process tries to lock all records it needs, one at a time
  - if needed record found locked, start over
  - (no real work done in phase one)

- If phase one succeeds, it starts second phase,
  - performing updates
  - releasing locks

- Note similarity to requesting all resources at once
  - **Attacking the hold-and-wait condition**

- Algorithm works where programmer can arrange
  - program can be stopped and restarted in the first phase, instead of blocking!
Nonresource Deadlocks

- Possible for two processes to deadlock
  - Each is waiting for the other to do some task

- Can happen with semaphores
  - Each process required to do a `down()` on two semaphores
    (mutex and another)
  - If done in wrong order, deadlock results

Re: The Producer-consumer Problem w/ Semaphores

```c
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void) { 
  int item;
  
  while (TRUE) { /* TRUE is the constant 1 */
    item = produce_item(); /* generate something to put in buffer */
    down(&empty); /* decrement empty count */
    down(&mutex); /* enter critical region */
    count_item(item); /* put new item in buffer */
    up(mutex()); /* leave critical region */
    up(&full); /* increment count of full slots */
  }
  
  what if the two downs in the producer’s code were reversed in order, so mutex was decremented before empty instead of after it?
}

void consumer(void) { 
  int item;
  
  while (TRUE) { /* infinite loop */
    down(&mutex); /* decrement full count */
    down(&empty); /* enter critical region */
    item = remove_item(); /* take item from buffer */
    up(mutex()); /* leave critical region */
    up(&empty); /* increment count of empty slots */
    consume_item(item); /* do something with the item */
  }
}
```
Starvation

- Algorithm to allocate a resource
  - may be to give to shortest job first
  - works great for multiple short jobs in a system
- It may cause long job to be postponed indefinitely
  - even though not blocked
  - Strict priority may give trouble!
- Solution:
  - First-come, first-serve resource allocation policy

What is the key difference between deadlock and starvation?

Summary of Lecture 3

- Deadlocks and its modeling
- Deadlock detection
- Deadlock recovery
- Deadlock avoidance
  - Resource trajectories
  - Safe and unsafe states
  - The banker's algorithm
- Two-phase locking
- More reading: Textbook 3.1 - 3.9