Homework 3 is due Tuesday, 10/18.

We *will* catch cheating attempts! Tools are *very* smart. They will detect variable renaming, code refactoring. Don’t risk it!
Recall: Schedulers in Linux

O(n) scheduler
Linux 2.4 to Linux 2.6

O(1) scheduler
Linux 2.6 to 2.6.22

CFS scheduler
Linux 2.6.23 onwards

Did not scale with large number of processes
Heuristics too complex
Proportional Fair Sharing.
Throughput and Latency constraints
Gives all processes 1/N virtual time on CPU
I will if you will

I will if you will
Deadlock: A Deadly type of Starvation

Deadlock: cyclic waiting for resources

Thread A owns Res 1 and is waiting for Res 2
Thread B owns Res 2 and is waiting for Res 1
Deadlock: A Deadly type of Starvation

Starvation: thread waits indefinitely

Deadlock implies starvation
but starvation does not imply deadlock

Starvation can end (but doesn’t have to)
Deadlock can’t end without external intervention
Example: Single-Lane Bridge Crossing
Bridge Crossing Example

Each segment of road can be viewed as a resource

Rules:
- Car must own the segment under them
- Must acquire segment that they are moving into
- For bridge: traffic only in one direction at a time
Bridge Crossing Example

Car must own the segment under them
Must acquire segment that they are moving into
For bridge: traffic only in one direction at a time
Deadlock:
Circular waiting for resources
**Bridge Crossing Example**

**Deadlock:**
Circular waiting for resources

Could be resolved by “external” intervention:
- fork-lifting a car of the bridge (equivalent to killing a thread)
- Asking cars to backup (equivalent to removing the resource from the thread)
Stop sign: purple car must wait for cars to release resources.

Cars on highway never do!

Purple car is starved
**Deadlock with Locks**

Thread A:

- `x.Acquire();`
- `y.Acquire();`
- ...
- `y.Release();`
- `x.Release();`

Thread B:

- `y.Acquire();`
- `x.Acquire();`
- ...
- `x.Release();`
- `y.Release();`

Will threads deadlock:

- a) Always
- b) Never
- c) Sometimes
- d) I'm still trying to cross the road

This lock pattern exhibits *non-deterministic deadlock*

A system is subject to deadlock if deadlock can happen **in any execution**

**Crooks CS162 © UCB Fall 2022**
Deadlock with Locks: “Lucky” Case

Thread A:
- x.Acquire();
- y.Acquire();
- ...
- y.Release();
- x.Release();

Thread B:
- y.Acquire();
- ...
- x.Acquire();
- ...
- x.Release();
- y.Release();

Sometimes, schedule won’t trigger deadlock!
Other Types of Deadlock

Threads often block waiting for resources
- Locks
- Terminals
- Printers
- CD drives
- Memory

Threads often block waiting for other threads
- Pipes
- Sockets

You can deadlock on any of these!
Dining Computer Scientists Problem

Five chopsticks/Five computer scientists

Need two chopsticks to eat
Free for all leads to deadlock
Fixing deadlock needs external intervention!

How could we have prevented this?

- Give everyone two chopsticks
- Make everyone “give up” after a while
- Require everyone to pick up both chopsticks atomically
Four requirements for occurrence of deadlock

1) Mutual exclusion and bounded resources
   Only one thread at a time can use a resource.

2) Hold and wait
   Thread holding at least one resource is waiting to acquire additional resources held by other threads
Four requirements for occurrence of deadlock

3) No preemption
Resources are released only voluntarily by the thread holding the resource, after thread is finished with it

4) Circular wait
There exists a set \( \{T_1, \ldots, T_n\} \) of waiting threads
\( T_1 \) is waiting for a resource that is held by \( T_2 \)
\( T_2 \) is waiting for a resource that is held by \( T_3 \)
\( \ldots \)
\( T_n \) is waiting for a resource that is held by \( T_1 \)
Detecting Deadlock: Resource-Allocation Graph

System Model

A set of Threads $T_1, T_2, \ldots, T_n$

Resource types $R_1, R_2, \ldots, R_m$

CPU cycles, memory space, I/O devices

Each resource type $R_i$ has $W_i$ instances

Each thread

Request() / Use() / Release() a resource:
Detecting Deadlock: Resource-Allocation Graph

Resource-Allocation Graph

- V is partitioned into two types:

\[ T = \{ T_1, T_2, \ldots, T_n \}, \]
the set threads in the system.

\[ R = \{ R_1, R_2, \ldots, R_m \}, \]
the set of resource types in system

- request edge – directed edge \( T_1 \to R_j \)
- assignment edge – directed edge \( R_j \to T_i \)
Resource-Allocation Graph Examples

Simple Resource Allocation Graph

Allocation Graph With Deadlock

Allocation Graph With Cycle, but No Deadlock
Deadlock Detection Algorithm

Let \([X]\) represent an \(m\)-ary vector of non-negative integers (quantities of resources of each type)

\([\text{FreeResources}]\): Current free resources each type

\([\text{Request}_X]\): Current requests from thread \(X\)

\([\text{Alloc}_X]\): Current resources held by thread \(X\)
Deadlock Detection Algorithm

See if tasks can eventually terminate on their own

\[ \text{[Avail]} = \text{[FreeResources]} \]
Add all threads to UNFINISHED

do {
    done = true
    Foreach thread in UNFINISHED {
        if (\text{[Request}_{\text{node}}] <= \text{[Avail]}) {
            remove thread from UNFINISHED
            \text{[Avail]} = \text{[Avail]} + \text{[Alloc}_{\text{node}]} \\
            done = false
        }
    }
} until(done)

Threads left in UNFINISHED \Rightarrow \text{deadlocked}
Deadlock Detection Algorithm

[Avail] = [FreeResources]
Add all threads to UNFINISHED
do {
    done = true
    Foreach thread in UNFINISHED {
        if ([Requestnode] <= [Avail]) {
            remove thread from UNFINISHED
            [Avail] = [Avail] + [Allocnode]
            done = false
        }
    }
} until(done)

Threads left in UNFINISHED ⇒ deadlocked

[Avail] = {0,0}
UNFINISHED = T1, T2, T3, T4

Looking at T1: [1,0] > [0,0]

Looking at T2: [0,0] <= [0,0]
Avail = [1,0]
UNFINISHED = T1,T3,T4

Looking at T3: [0,1] > [1,0]

Looking at T4
[0,0] <= [0,0]
Avail = [1,1]
UNFINISHED = T1, T3

Looking at T1: [1,0] <= [1,1]
Avail = [2,1]
UNFINISHED = T3

Looking at T3: [0,1] <= [2,1]
Avail = [2,2]
UNFINISHED = Empty!
How should a system deal with deadlock?

**Deadlock prevention**
Write your code in a way that it isn’t prone to deadlock

**Deadlock recovery**
Let deadlock happen, and figure out how to recover from it

**Deadlock avoidance**
Dynamically delay resource requests so deadlock doesn’t happen

**Deadlock denial**
Ignore the possibility of deadlock
Deadlock prevention

**Condition 1: Mutual exclusion and bounded resources**

⇒ Provide sufficient resources

**Condition 2: Hold and wait**

⇒ Abort request or acquire requests atomically

**Condition 3: No preemption**

⇒ Preempt threads

**Condition 4: Circular wait**

⇒ Order resources and always acquire resources in the same way
**Condition 1 Fix: (Virtually) Infinite Resources**

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllocateOrWait(1 MB)</td>
<td>AllocateOrWait(1 MB)</td>
</tr>
<tr>
<td>AllocateOrWait(1 MB)</td>
<td>AllocateOrWait(1 MB)</td>
</tr>
<tr>
<td>Free(1 MB)</td>
<td>Free(1 MB)</td>
</tr>
<tr>
<td>Free(1 MB)</td>
<td>Free(1 MB)</td>
</tr>
</tbody>
</table>

With virtual memory we have “infinite” space so everything will always succeed.
Condition 2 Fix: Request Resources Atomically

Rather than:

Thread A:
- x.Acquire();
- y.Acquire();
- ...
- y.Release();
- x.Release();

Thread B:
- y.Acquire();
- x.Acquire();
- ...
- x.Release();
- y.Release();

Consider instead:

Thread A:
- Acquire_both(x, y);
- ...
- y.Release();
- x.Release();

Thread B:
- Acquire_both(y, x);
- ...
- x.Release();
- y.Release();
Condition 3 Fix: Preemption

Force thread to give up resource

Common technique in databases using database aborts
- A transaction is “aborted”: all of its actions are undone, and the transaction must be retried

Common technique in wireless networks:
- Everyone speaks at once. When a resource collision is detected, retry at a new, random time
Condition 4 Fix: Circular Waiting

Force all threads to request resources in the same order

Thread A:
\[
x.\text{Acquire}();
y.\text{Acquire}();
...
y.\text{Release}();
x.\text{Release}();
\]

Thread B:
\[
y.\text{Acquire}();
x.\text{Acquire}();
...
x.\text{Release}();
y.\text{Release}();
\]
**Condition 4 Fix: Circular Waiting**

Garcia: first 1 then 5
Crooks: first 2 then 1
Turing: first 3 then 2
Johnson: first 4 then 3
Liskov: first 5 then 4

If ensure that Garcia graphs chopstick 5 followed by 1, no deadlock!
How should a system deal with deadlock?

**Deadlock prevention**
Write your code in a way that it isn’t prone to deadlock

**Deadlock recovery**
Let deadlock happen, and figure out how to recover from it

**Deadlock avoidance**
Dynamically delay resource requests so deadlock doesn’t happen

**Deadlock denial**
Ignore the possibility of deadlock
Techniques for Deadlock Avoidance

Attempt 1

When a thread requests a resource, OS checks if it would result in deadlock

If not, it grants the resource right away

If so, it waits for other threads to release resources
Techniques for Deadlock Avoidance

This does not work!

Thread A:
- x.Acquire();
- Blocks...
- y.Acquire();
- ...
- y.Release();
- x.Release();

Thread B:
- y.Acquire();
- x.Acquire();
- Wait?
- ...
- x.Release();
- y.Release();
- But it’s already too late...
**Deadlock Avoidance: Three States**

**Safe state**
System can delay resource acquisition to prevent deadlock

**Unsafe state**
No deadlock yet...
But threads can request resources in a pattern that *unavoidably* leads to deadlock

**Deadlocked state**
There exists a deadlock in the system

**Deadlock avoidance: prevent system from reaching an unsafe state**
Deadlock Avoidance: Three States

Thread A:
- x.Acquire();
- y.Acquire();
- ...
- y.Release();
- x.Release();

Thread B:
- y.Acquire();
- x.Acquire();
- ...
- x.Release();
- y.Release();

A acquires x.
There exists a sequence A-A(y), A-R(y), A-R(x), B-A(y), B-A(x), B-R(x), B-R(y) => safe state

B acquires y.
No sequence that won’t lead to deadlock. => unsafe state
**Banker’s Algorithm for Avoiding Deadlock**

Banker’s algorithm ensures never enter an unsafe state.

Evaluate each request and grant if some ordering of threads is still deadlock free afterward.

Technique: pretend each request is granted, then run deadlock detection algorithm.
Banker’s Algorithm for Avoiding Deadlock

\[ \text{[Avail]} = \text{[FreeResources]} \]
Add all threads to UNFINISHED
do {
    done = true
    Foreach thread in UNFINISHED {
        if ([Request\_thread] <= [Avail]) {
            remove thread from UNFINISHED
            [Avail] = [Avail] + [Alloc\_thread]
            done = false
        }
    }
} until(done)

\[ \text{[Avail]} = \text{[FreeResources]} \]
Add all threads to UNFINISHED
do {
    done = true
    Foreach threads in UNFINISHED {
        if ([Max\_threads]-[Alloc\_thread] <= [Avail]) {
            remove thread from UNFINISHED
            [Avail] = [Avail] + [Alloc\_thread]
            done = false
        }
    }
} until(done)
Banker's Algorithm for Avoiding Deadlock

\[
\text{[Avail]} = \text{[FreeResources]}
\]
Add all threads to UNFINISHED
\[
do \{
\]
done = true
\[
\text{Foreach threads in UNFINISHED} \{
\]
if \((\text{[Max}\_\text{threads}]-\text{[Alloc}\_\text{thread}] \leq \text{[Avail]})\) {
\]
remove thread from UNFINISHED
\[
\text{[Avail]} = \text{[Avail]} + \text{[Alloc}\_\text{thread]}
\]
done = false
\}
\}
until(done)

**Step 1:** “Assume” request is made

**Step 2:** If request is made, is system still in SAFE state? There exists a sequence \(\{T_1, T_2, \ldots, T_n\}\) such that all transactions finish

**Step 3:** If SAFE, grant resources. If UNSAFE, delay
**Banker's Algorithm for Avoiding Deadlock**

\[ \text{[Avail]} = \text{[FreeResources]} \]

Add all threads to UNFINISHED

\[
\text{do} \quad \text{done} = \text{true} \\
\quad \text{Foreach threads in UNFINISHED} \{ \\
\quad \quad \text{if } (\text{[Max\_thread]} - \text{[Alloc\_thread]} \leq \text{[Avail]}) \{ \\
\quad \quad \quad \text{remove thread from UNFINISHED} \\
\quad \quad \quad \text{[Avail]} = \text{[Avail]} + \text{[Alloc\_thread]} \\
\quad \quad \quad \text{done} = \text{false} \\
\quad \quad \} \\
\} \text{ until(done)}
\]

**When Thread A acquires x:**

Avail = \([0,1]\)

For A: \([1,1] - [1,0] \leq [0,1]\)

Update Avail to = \([1,1]\).

Remove A from UNFINISHED

For B:

\([1,1] - [0,0] \leq [1,1]\)

Update Avail to = \([1,1]\).

Remove B from UNFINISHED

Safe state!

**When Thread B acquires y:**

Avail = \([0,0]\)

For A: \([1,1] - [1,0] \leq [0,0]\)

For B: \([1,1] - [0,1] \leq [0,0]\)

UNFINISHED not empty

Unsafe state! Must delay acquiring y!

Thread A:

\( x.\text{Acquire}(); \quad y.\text{Acquire}(); \quad \ldots \quad y.\text{Release}(); \quad x.\text{Release}(); \)

Thread B:

\( y.\text{Acquire}(); \quad x.\text{Acquire}(); \quad \ldots \quad x.\text{Release}(); \quad y.\text{Release}(); \)
**Summary**

Deadlock $\Rightarrow$ Starvation, Starvation does not imply deadlock

Four conditions for deadlocks:
- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

Techniques for addressing deadlock: prevention, recovery, avoidance, or denial

Banker’s algorithm for avoiding deadlock