Deadlock

CSE 2431: Introduction to Operating Systems
Instructor: Adam C. Champion, Ph.D.
Reading: Chap. 7, [OSC]
Outline

• Resources
• Deadlock
• Deadlock Prevention
• Deadlock Avoidance
• Deadlock Detection
• Deadlock Recovery
Review: Synchronization

• Data races
• Critical regions and mutual exclusions
• Solutions:
  – Peterson’s solution
  – TSL
  – Semaphores and Mutexes
  – Monitors
  – Message Passing
  – Barriers
• Classic Synchronization Problems
Resource (1)

• A resource is a commodity needed by a process.

• Resources can be either:
  
  – *Serially reusable*: e.g., CPU, memory, disk space, I/O devices, files.
    Acquire $\rightarrow$ use $\rightarrow$ release
  
  – *Consumable*: produced by a process, needed by a process; e.g., messages, buffers of information, interrupts.
    create $\rightarrow$ acquire $\rightarrow$ use
    Resource ceases to exist after it has been used, so it is not released.
Resource (2)

• Resources can also be either:
  – Preemptible: e.g., CPU, central memory or
  – Non-preemptible: e.g., tape drives.

• And resources can be either:
  – Shared among several processes or
  – Dedicated exclusively to a single process.
Outline

• Resources
• Deadlock
• Deadlock Prevention
• Deadlock Avoidance
• Deadlock Detection
• Deadlock Recovery
Using Semaphores to Share Resources

Process P();
{
    A.wait();
    B.wait();
    //use both resource
    B.signal();
    A.signal();
}

Process Q();
{
    A.wait();
    B.wait();
    //use both resource
    B.signal();
    A.signal();
}

External Semaphore A(1), B(1);
2 External Semaphore A(0), B(1);
3 External Semaphore A(0), B(0);
4 External Semaphore A(0), B(1);
5 External Semaphore A(1), B(1);
But Deadlock can Happen!

1. Process P();
   { A.wait();
     B.wait();
     //use both resources
     B.signal();
     A.signal();
   }

2. External Semaphore A(1), B(1);
3. External Semaphore A(0), B(1);
4. External Semaphore A(0), B(0);

DEADLOCK!
Deadlock

Mechanisms for Deadlock Control
Deadlock Definition

• What is a deadlock?
  – A process is *deadlocked* if it is waiting for an event that will never occur.
  – Typically, but not necessarily, more than one process will be involved together in a deadlock (the deadly embrace).

• Is deadlock the same as starvation (or infinitely postponed)?
  – A process is *infinitely postponed* if it is delayed repeatedly over a long period of time while the attention of the system is given to other processes (*i.e.*, logically the process may proceed but the system never gives it resources).
Conditions for Deadlock

• What conditions should exist in order to lead to a deadlock?

• Real life analogies such as
  – “You take the monitor, I grab the keyboard”
  – Cars in intersection

  – Your turn for real life analogies
Necessary Conditions for Deadlock

• **Mutual exclusion:** Processes claim exclusive control of the resources they require.

• **Wait-for condition:** Processes hold resources already allocated to them while waiting for additional resources.

• **No preemption condition:** Resources cannot be removed from the processes holding them until used to completion.

• **Circular wait condition:** A circular chain of processes exists in which each process holds one or more resources that are requested by the next process in the chain.
Resource Allocation Graph

- Resource
- Process
- Resource Type

1 Process, 2 Resources of same Type

Process requests resource

Process is assigned resource

Process releases resource
Deadlock Model

<table>
<thead>
<tr>
<th>Resource</th>
<th>Process</th>
<th>Resource Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2 resources</td>
</tr>
</tbody>
</table>

- **2 Processes**: Each process requests 2 resources.
- **Deadlock**: Cycle in resource graph.
- **Deadlock may not occur if there are enough resources**: Cycle in resource graph.
How To Deal With Deadlock?

- **Prevention:** Design a system such that deadlocks cannot occur, at least with respect to serially reusable resources.
- **Avoidance:** Impose less stringent conditions than for prevention, allowing the possibility of deadlock, but sidestepping it as it approaches.
- **Detection:** In a system that allows the possibility of deadlock, determine if deadlock has occurred, and which processes and resources are involved.
- **Recovery:** After a deadlock has been detected, clear the problem, allowing the deadlocked processes to complete and the resources to be reused. Usually involves destroying the affected processes and restarting them.
Outline

• Resources
• Deadlock
• **Deadlock Prevention**
• Deadlock Avoidance
• Deadlock Detection
• Deadlock Recovery
The Ostrich Algorithm

• Guess: what is implemented in Linux?
• Don’t do anything, simply restart the system (stick your head into the sand, pretend there is no problem at all).
• Rationale:
  – Make the common path faster and more reliable
  – Deadlock prevention, avoidance or detection/recovery algorithms are expensive
  – If deadlock occurs only rarely, it is not worth the overhead to implement any of these algorithms.
Deadlock Prevention: Havender’s Algorithms

• Break one of the deadlock conditions:
  – **Mutual exclusion**
    • **Solution:** exclusive use of resources is an important feature, but for some resources (virtual memory, CPU), it is possible.
  – **Hold-and-Wait condition**
    • **Solution:** Force each process to request all required resources at once. It cannot proceed until all resources have been acquired.
  – **No preemption condition**
    • **Solution:** Forcibly take away the resources assigned to the process due to lack of other requested resources.
  – **Circular wait condition**
    • **Solution:** All resource types are numbered. Processes must request resources in numerical order.
Locking with Two Phases

• 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, start Phase Two
  – Perform updates
  – Release locks

• Note similarity to requesting all resources at once

• Problems?
Break Circular Wait Condition

• Request one resource at a time.

• Global ordering of resources
  – Requests have to be made in increasing order
  – Req(resource1), req(resource2)..

• Why no circular wait?

• Problems?
## Summary: Deadlock Prevention

<table>
<thead>
<tr>
<th>Condition</th>
<th>How to Break It</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>
Review

• Resource: preemptible or non-preemptible
• Deadlock conditions
  – Mutual exclusion
  – Hold and wait
  – Non-preemption
  – Circular wait
• Deadlock prevention
  – Break the condition
  – Locking with two phases
Questions

• Design an algorithm to detect deadlocks?
  – Single resource per type
  – Multiple resources per type

• How to avoid deadlocks?

• How to recover from deadlocks?
Deadlock Avoidance

• The system needs to know the resource ahead of time
• Banker’s algorithm (Dijkstra, 1965)
  – Each customer tells banker the max. number of resources needed
  – Customer borrows resources from banker
  – Customer returns resources to banker
  – Customer eventually pays back loan
  – Banker only lends resources if the system will be in a safe state after the loan
• Safe state: there is a lending sequence such that all customers can take out a loan
• Unsafe state: there is a possibility of deadlock
Safe State and Unsafe State

• Safe State
  – there is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately
  – From a safe state, the system can guarantee that all processes will finish

• Unsafe state: no such guarantee
  – Not a deadlock state
  – Some process may be able to complete
Is Allocation (1 0 2) to P1 Safe?

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc A</th>
<th>Alloc B</th>
<th>Alloc C</th>
<th>Max A</th>
<th>Max B</th>
<th>Max C</th>
<th>Need A</th>
<th>Need B</th>
<th>Need C</th>
<th>Available A</th>
<th>Available B</th>
<th>Available C</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If P1 requests resources (1 0 2), can it complete
Run Safety Test

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A  B  C</td>
<td>A  B  C</td>
<td>A  B  C</td>
<td>A  B  C</td>
</tr>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td>7 4 3</td>
<td>3 3 2</td>
</tr>
<tr>
<td>P1</td>
<td>3 0 2</td>
<td>3 2 2</td>
<td>0 2 0</td>
<td>2 3 0</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 2</td>
<td>9 0 2</td>
<td>6 0 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>2 2 2</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

If P1 requests max resources, it can complete.
Allocate \((0 \ 2 \ 0)\) to P1

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>P0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>P1</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
Release Resources: P1 Finishes

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>P0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>P1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Now P3 can acquire max resources and release
Release Resources: P3 Finishes

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td>7 4 3</td>
<td>5 3 2</td>
</tr>
<tr>
<td>P1</td>
<td>0 0 0</td>
<td>3 2 2</td>
<td>3 2 2</td>
<td>7 4 3</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 0</td>
<td>9 0 2</td>
<td>6 0 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>0 0 0</td>
<td>2 2 2</td>
<td>2 2 2</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

Now P4 can acquire max resources and release
Release Resources: P4 Finishes

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc A</th>
<th>Alloc B</th>
<th>Alloc C</th>
<th>Max A</th>
<th>Max B</th>
<th>Max C</th>
<th>Need A</th>
<th>Need B</th>
<th>Need C</th>
<th>Available A</th>
<th>Available B</th>
<th>Available C</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>P1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now P2 can acquire max resources and release
## Release Resources: P2 Finishes

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>P0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>P1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>P3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

Now P0 can acquire max resources and release
So P1 Allocation (1 0 2) Is Safe

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A  B  C</td>
<td>A  B  C</td>
<td>A  B  C</td>
<td>A  B  C</td>
</tr>
<tr>
<td>P0</td>
<td>0  1  0</td>
<td>7  5  3</td>
<td>7  4  3</td>
<td>3  3  2</td>
</tr>
<tr>
<td>P1</td>
<td>3  0  2</td>
<td>3  2  2</td>
<td>0  2  0</td>
<td>2  3  0</td>
</tr>
<tr>
<td>P2</td>
<td>3  0  0</td>
<td>9  0  2</td>
<td>6  0  2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2  1  1</td>
<td>2  2  2</td>
<td>0  1  1</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0  0  2</td>
<td>4  3  3</td>
<td>4  3  1</td>
<td></td>
</tr>
</tbody>
</table>
Is allocation (0 2 0) to P0 Safe?

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>P0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>P1</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Try to allocate 2 of resource B to P0…
Run Safety Test

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A  B  C</td>
<td>A  B  C</td>
<td>A  B  C</td>
<td>A  B  C</td>
</tr>
<tr>
<td>P0</td>
<td>0  3  0</td>
<td>7  5  3</td>
<td>7  2  3</td>
<td>2  3  0</td>
</tr>
<tr>
<td>P1</td>
<td>3  0  2</td>
<td>3  2  2</td>
<td>0  2  0</td>
<td>2  1  0</td>
</tr>
<tr>
<td>P2</td>
<td>3  0  0</td>
<td>9  0  2</td>
<td>6  0  2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2  1  1</td>
<td>2  2  2</td>
<td>0  1  1</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0  0  2</td>
<td>4  3  3</td>
<td>4  3  1</td>
<td></td>
</tr>
</tbody>
</table>

No process can get its max resources and release them
So Unsafe State ⇒ **Do Not Enter**

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P0</td>
<td>0 3 0</td>
<td>7 5 3</td>
<td>7 2 3</td>
<td>2 3 0</td>
</tr>
<tr>
<td>P1</td>
<td>3 0 2</td>
<td>3 2 2</td>
<td>0 2 0</td>
<td>2 1 0</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 0</td>
<td>9 0 2</td>
<td>6 0 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>2 2 2</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

Return to safe state; *do not* allocate resource
When enough resources become available, P0 can awaken.
Results

• P0’s request for 2 Bs cannot be granted because that would prevent any other process from completing if they need their maximum claim
How to Compute Safety?

Given:
n kinds of resources
p processes
Set P of processes
struct {resource needs[n], owns[n]} ToDo[p]
resource_available[n]
......

while there exists p in P such that
for all i (ToDo[p].needs[i] < available[i])
{
    for all i
    {
        available[i] += ToDo[p].owns[i];
        P = P - {p};
    }
}

If (P is empty) then system is safe;
Just Because It’s Unsafe —

• P0 could have been allocated 2 Bs and a deadlock might not have occurred if:
  – P1 didn’t use its maximum resources but finished using the resources it had
If P1 Doesn’t Need Max—

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>P0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>P1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Then P0 would have finished…
Tradeoff

• What is the tradeoff?
• The banker’s algorithm is conservative: it reduces parallelism for safety’s sake
Banker Solution Issues

• Process needs to indicate its resource needs in the beginning
• Process may not terminate
• Process may request more than claimed
• A process may suffer infinite postponement (starvation problem)
Deadlock Detection

• Check to see if a deadlock has occurred!

• Single resource per type
  – Can use wait-for graph
  – Check for cycles
    • How?

• How about multiple resources per type?
Wait-For Graphs

Resource allocation graph

Corresponding wait-for graph
Detection: Multiple Resources per Type

- Run variant of banker’s algorithm to see if processes can finish
  - Time: $O(mn^2)$, $m$: # resources, $n$: # processes

- Optimizations?
  - Optimistic version: check only that at least one process can finish
  - The deadlock detection is delayed

- How often should the detection algorithm be run?
Recovery From Deadlock

• Options:
  – Kill deadlocked processes and release resources
  – Kill one deadlocked process at a time and release its resources
  – Rollback all or one of the processes to a checkpoint that occurred before they requested any resources

• Note: may have starvation problems
Deadlock Summary

• In general, deadlock detection or avoidance is expensive
• Must evaluate cost of deadlock against detection or avoidance costs
• Deadlock avoidance and recovery may cause starvation problems
• UNIX and Windows use Ostrich Algorithm