Logistics

- See Canvas for additional information of potential help with Project 1
- Assignment 2 due Thursday, October 29, 5pm
- Answer sheets to Assignments 1 and 2 to be provided by COB Thursday
OS in the News!

Following users complaints that the Facebook app was sucking the battery from their iPhones, the company has released a software update and explained in detail what was causing the issue.

In a Facebook post, engineer manager Ari Grant began by pointing to an overworking issue related to processing. "The first issue we found was a 'CPU spin' in our network code," he wrote. "A CPU spin is like a child in a car asking, 'Are we there yet? Are we there yet? Are we there yet?' with the question not resulting in any progress to reaching the destination. This repeated processing causes our app to use more battery than intended. The version released today has some improvements that should start making this better."

He also mentioned audio sessions that continued to be active even after the user had closed the app.

"This is similar to when you close a music app and want to keep listening to the music while you do other things, except in this case it was unintentional and nothing kept playing. The app isn't actually doing anything while awake in the background, but it does use more battery simply by being awake," Grant wrote.

The updated iOS app is available in the app store now.
Outline

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
Objectives

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system
System Model

- System consists of resources
- 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 process utilizes a resource as follows:
  - Request
  - Use
  - Release
Deadlock Characterization

- Deadlock can arise if four conditions hold simultaneously:
  1. **Mutual exclusion**: only one process at a time can use a resource
  2. **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
  3. **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task using the resource
  4. **Circular wait**: there exists a set \( \{P_1, P_2, \ldots, P_n\} \) of waiting processes such that \( P_1 \) is waiting for a resource that is held by \( P_2 \), \( P_2 \) is waiting for a resource that is held by \( P_3 \), \ldots, \( P_{n-1} \) is waiting for a resource that is held by \( P_n \), and \( P_n \) is waiting for a resource that is held by \( P_1 \)
Resource Allocation Graph

- A set of vertices \( V \) and a set of edges \( E \)
- \( V \) is partitioned into two types:
  - \( P = \{P_1, P_2, \ldots, P_n\} \), the set consisting of all the processes in the system
  - \( R = \{R_1, R_2, \ldots, R_m\} \), the set consisting of all resource types in the system

- Request edge
  - Directed edge \( P_i \rightarrow R_j \)

- Assignment edge
  - Directed edge \( R_j \rightarrow P_i \)

- A resource allocation graph is used to determine “state” of the resource system
Resource-Allocation Graph Symbols

- **Process**

- **Resource type with 4 instances**

- $P_i$ requests instance of $R_j$

- $P_i$ is holding an instance of $R_j$
Example of a Resource Allocation Graph

- $P_1$ is requesting $R_1$ and holding an instance of $R_2$
- $P_2$ is holding an instance of $R_1$ and $R_2$ and is requesting $R_3$
- $P_3$ is holding an instance of $R_3$
- Resource allocation graph shows the "hold" and "wait" conditions
- Is there a deadlock?
Resource Allocation Graph With A Deadlock

- Look for hold and wait conditions that create a circular waiting state
  - Every process is waiting on a resource that is being held by another process in the cycle
- If there is a circular wait state, then no process can proceed because they are all blocked
- Every process is waiting indefinitely because no resource will be released
Graph With A Cycle But No Deadlock

- It is possible for there to be a cycle in the resource allocation graph, but the resource system is not deadlocked.
- Does this resource allocation graph have a deadlock?
- There exist a resource that is held by a process that is not in the cycle.
- A resource can be released by a process that is not in the cycle, allowing for a process in the cycle to proceed.
Basic Facts

- If resource allocation graph contains no cycles:
  - There can be no deadlock

- If resource allocation graph contains a cycle:
  - If there is only one instance per resource type
    ⇒ deadlock
  - If there are several instances per resource type
    ⇒ possibility of deadlock
Methods for Handling Deadlocks

- Basically, we need to ensure that the system will never enter a deadlock state
- This can be done in 2 ways:
  - Deadlock prevention
  - Deadlock avoidance
- However, you could also allow the system to enter a deadlock state and then recover
  - Need to guarantee that recovery is possible and valid
- Or, you could ignore the problem and pretend that deadlocks never occur in the system
  - Used by most operating systems, including UNIX
Deadlock Prevention – Requirements (1)

- Idea is to restrain the ways request can be made
- *Mutual exclusion*
  - Not required for sharable resources (e.g., read-only files)
  - Must hold for *non-sharable* resources
- *Hold and wait*
  - Must guarantee that whenever a process requests a resource, it does not hold any other resources (strict)
  - Require process to request and be allocated ALL of its resources before it begins execution
  - Allow process to request resources only when the process has none allocated to it
  - Low resource utilization and starvation possible
Deadlock Prevention– Requirements (2)

- **No preemption**
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held by that process are released.
  - Preempted resources are added to the list of resources for which the process is waiting.
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

- **Circular wait**
  - Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.
Example using Pthread Mutex Locks

- Think of the mutex locks as representing a resource being requested
- Does this code generate a deadlock?
- Why or why not?
- What is wrong?
- Hint: order matters

```c
/* thread one runs in this function */
void *do_work_one(void *param) {
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /** * Do some work */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    pthread_exit(0);
}

/* thread two runs in this function */
void *do_work_two(void *param) {
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /** * Do some work */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);
    pthread_exit(0);
}
```
An Account Transaction Example

- Transactions 1 and 2 execute concurrently
  - Transaction 1 transfers $25 from account A to account B
  - Transaction 2 transfers $50 from account B to account A

- Locks are locked in an order, unlocked in reverse

```java
void transaction(Account from, Account to, double amount) {
    mutex lock1, lock2;
    lock1 = get_lock(from);
    lock2 = get_lock(to);
    acquire(lock1);
    acquire(lock2);
    acquire(lock2);
    withdraw(from, amount);
    deposit(to, amount);
    release(lock2);
    release(lock1);
}
```

Does it work?
Deadlock Avoidance

- Requires that the system has some additional *a priori* information available
- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource allocation state to ensure that there can never be a circular-wait condition
- Resource allocation state is defined by the number of available and allocated resources, and the *maximum* possible demands of the processes
Safe State

- When a process requests an available resource, it must be decided if immediate allocation of the resource to the process leaves the system in a *safe state*.

- System is in safe state if there exists a sequence 
  \[ <P_1, P_2, \ldots, P_n> \]
  of ALL the processes in the system such that for each \( P_i \), the resources that \( P_i \) can still request can be satisfied by currently available resources + resources held by all the \( P_j \), with \( j < i \):

  - If \( P_i \) resource needs are not immediately available, then \( P_i \) can wait until all \( P_j \) have finished.
  - When \( P_j \) is finished, \( P_i \) can obtain needed resources, execute, return allocated resources, and terminate.
  - When \( P_i \) terminates, \( P_i + 1 \) can obtain its needed resources.
Basic Facts

- If a system is in safe state ⇒ no deadlocks

- If a system is in unsafe state ⇒ possibility of deadlock

- Avoidance achieved by ensuring that a system will never enter an unsafe state
State Relationships

- A resource allocation graph can be in 2 mutually exclusive states: safe, unsafe
- In the unsafe state, a resource allocation graph can be vulnerable to deadlock or in a deadlocked condition
Deadlock Avoidance Algorithms

- Avoidance algorithms prevent deadlocks from ever happening
  - Never allowing an unsafe state to be entered
- Approaches depend on assumptions about the resource allocation graph
- Single instance of a resource type
  - Use a resource allocation graph to evaluate
- Multiple instances of a resource type
  - Must run an algorithm on the resource allocation graph
  - Use the Banker’s algorithm
**Resource Allocation Graph Scheme**

- **Claim edge** $P_i \rightarrow R_j$ indicates that process $P_j$ may request resource $R_j$
  - Represented by a dashed line
- Claim edge converts to a *request edge* when a process requests a resource
- Request edge converted to an *assignment edge* when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system
Is this in a safe state?
Why is this unsafe?
Resource Allocation Graph Algorithm

- Suppose that process $P_i$ requests a resource $R_j$.
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph.
- Cycles are evaluated using all types of edges, including claim edges.
Banker’s Algorithm

- Suppose we have multiple instances
- Requirements:
  - Each process must *a priori* claim maximum use
  - When a process requests a resource it may have to wait
  - When a process gets all its resources it must return them in a finite amount of time
- Banker’s algorithms is a bookkeeping method for tracking and assigning resources
Data Structures for Banker’s Algorithm

- Let \( n \) = number of processes, and \( m \) = number of resources types
- **Available**: Vector of length \( m \). If \( \text{Available}[j] = k \), there are \( k \) instances of resource type \( R_j \) available.
- **Max**: \( n \times m \) matrix. If \( \text{Max}[i,j] = k \), then process \( P_i \) may request at most \( k \) instances of resource type \( R_j \)
- **Allocation**: \( n \times m \) matrix. If \( \text{Allocation}[i,j] = k \) then \( P_i \) is currently allocated \( k \) instances of \( R_j \)
- **Need**: \( n \times m \) matrix. If \( \text{Need}[i,j] = k \), then \( P_i \) may need \( k \) more instances of \( R_j \) to complete its task

\[
\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]
\]
Safety Algorithm

1. Let $Work$ and $Finish$ be vectors of length $m$ and $n$, respectively.
   Initialize:
   $Work = Available$
   $Finish[i] = false$ for $i = 0, 1, \ldots, n-1$

2. Find an $i$ such that both:
   $Finish[i] = false$
   $Need_i \leq Work$
   If no such $i$ exists, go to step 4

3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   go to step 2

4. If $Finish [i] == true$ for all $i$, then the system is in a safe state
Resource-Request Algorithm for Process $P_i$

- $Request_i = \text{request vector for process } P_i$
- If $Request_i[j] = k$ then process $P_i$ wants $k$ instances of resource type $R_j$
  
  1. If $Request_i \leq Need_i$ go to step 2
     Otherwise, raise error condition, since process has exceeded its maximum claim
  
  2. If $Request_i \leq Available$, go to step 3
     Otherwise $P_i$ must wait, since resources are not available
  
  3. Pretend to allocate requested resources to $P_i$:
     
     $Available = Available - Request_i;$
     $Allocation_i = Allocation_i + Request_i;$
     $Need_i = Need_i - Request_i;$

     If safe $\Rightarrow$ the resources are allocated to $P_i$
     If unsafe $\Rightarrow$ $P_i$ must wait, restore old resource allocation
Example of Banker’s Algorithm

- 5 processes P₀ through P₄
- 3 resource types:
  - A (10 instances), B (5 instances), C (7 instances)
- Snapshot at time Tᵢ:

<table>
<thead>
<tr>
<th>Process</th>
<th>Allocation</th>
<th>Max</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>
</tr>
<tr>
<td>P₀</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td>3 3 2</td>
</tr>
<tr>
<td>P₁</td>
<td>2 0 0</td>
<td>3 2 2</td>
<td></td>
</tr>
<tr>
<td>P₂</td>
<td>3 0 2</td>
<td>9 0 2</td>
<td></td>
</tr>
<tr>
<td>P₃</td>
<td>2 1 1</td>
<td>2 2 2</td>
<td></td>
</tr>
<tr>
<td>P₄</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td></td>
</tr>
</tbody>
</table>
Check for Safety

- Matrix \textit{Need} is defined to be \textit{Max} – \textit{Allocation}

<table>
<thead>
<tr>
<th>Process</th>
<th>Need</th>
<th>\textit{Allocation}</th>
<th>\textit{Max}</th>
<th>\textit{Available}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>(P_0)</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>0 1 0</td>
</tr>
<tr>
<td>(P_1)</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2 0 0</td>
</tr>
<tr>
<td>(P_2)</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>9 0 2</td>
</tr>
<tr>
<td>(P_3)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2 1 1</td>
</tr>
<tr>
<td>(P_4)</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>4 3 3</td>
</tr>
</tbody>
</table>

- The system is in a safe state since the sequence \(< P_1, P_3, P_4, P_2, P_0 >\) satisfies safety criteria.
$P_1$ Requests (1,0,2)

- Let’s advance the system with $P_1$ making request (1,0,2)
- Check that $Request \leq Available$ (that is, (1,0,2) $\leq$ (3,3,2) $\Rightarrow$ true)

<table>
<thead>
<tr>
<th>Process</th>
<th>Allocation</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
<td></td>
</tr>
<tr>
<td>$P_0$</td>
<td>0 1 0</td>
<td>7 4 3</td>
<td>2 3 0</td>
</tr>
<tr>
<td>$P_1$</td>
<td>3 0 2</td>
<td>0 2 0</td>
<td></td>
</tr>
<tr>
<td>$P_2$</td>
<td>3 0 2</td>
<td>6 0 0</td>
<td></td>
</tr>
<tr>
<td>$P_3$</td>
<td>2 1 1</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>$P_4$</td>
<td>0 0 2</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

- Executing safety algorithm shows that sequence $<P_1, P_3, P_4, P_0, P_2>$ satisfies safety requirement
- Can request for (3,3,0) by $P_4$ be granted?
- Can request for (0,2,0) by $P_0$ be granted?
Deadlock Detection

- Allow system to enter deadlock state

- Detection algorithm determines if the resource allocation graph is in a deadlock state

- If it is, a recovery scheme is used to get out of it
  - Assume that it is possible to, in fact, recover
  - It is possible that this might require a rollback to a prior consistent state
Single Instance of Each Resource Type

- Need to maintain a *wait-for* graph
  - Nodes are processes
  - \( P_i \rightarrow P_j \) if \( P_i \) is waiting for \( P_j \)

- Periodically invoke an algorithm that searches for a cycle in the *wait-for* graph
  - If there is a cycle, there exists a deadlock

- An algorithm to detect a cycle in a graph requires an order of \( n^2 \) operations, where \( n \) is the number of vertices in the graph
Resource-Allocation and Wait-for Graph

Resource-Allocation Graph

Corresponding wait-for graph
Several Instances of a Resource Type

- *Available*: A vector of length $m$ indicates the number of available resources of each type.

- *Allocation*: An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.

- *Request*: An $n \times m$ matrix indicates the current request of each process.
  - If $\text{Request}_{i}[j] = k$, then process $P_i$ is requesting $k$ more instances of resource type $R_j$. 
Detection Algorithm

1. Let $Work$ and $Finish$ be vectors of length $m$ and $n$, respectively. Initialize:
   (a) $Work = Available$
   (b) For $i = 1, 2, \ldots, n$, if $Allocation_i \neq 0$, then $Finish[i] = false$; otherwise, $Finish[i] = true$

2. Find an index $i$ such that both:
   (a) $Finish[i] == false$
   (b) $Request_i \leq Work$
   If no such $i$ exists, go to step 4

3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   go to step 2

4. If $Finish[i] == false$, for some $i$, $1 \leq i \leq n$, then the system is in deadlock state
   If $Finish[i] == false$, then $P_i$ is deadlocked

Algorithm requires an order of $O(m \times n^2)$ operations to detect whether the system is in deadlocked state.
Example of Detection Algorithm

- Five processes $P_0$ through $P_4$
- Three resource types
  - $A$ (7 instances), $B$ (2 instances), and $C$ (6 instances)
- Snapshot at time $T_0$:

<table>
<thead>
<tr>
<th>Process</th>
<th>Allocation</th>
<th>Request</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>
</tr>
<tr>
<td>$P_0$</td>
<td>0 1 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2 0 0</td>
<td>2 0 2</td>
<td></td>
</tr>
<tr>
<td>$P_2$</td>
<td>3 0 3</td>
<td>0 0 0</td>
<td></td>
</tr>
<tr>
<td>$P_3$</td>
<td>2 1 1</td>
<td>1 0 0</td>
<td></td>
</tr>
<tr>
<td>$P_4$</td>
<td>0 0 2</td>
<td>0 0 2</td>
<td></td>
</tr>
</tbody>
</table>

- Sequence $<P_0, P_2, P_3, P_1, P_4>$ gives $Finish[i] = true$ for all $i$
Deadlock?

- $P_2$ requests an additional instance of type C

<table>
<thead>
<tr>
<th>Process</th>
<th>Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>$P_0$</td>
<td>0</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2</td>
</tr>
<tr>
<td>$P_2$</td>
<td>0</td>
</tr>
<tr>
<td>$P_3$</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0</td>
</tr>
</tbody>
</table>

- State of system?
  - Can reclaim resources held by process $P_0$, but insufficient resources to fulfill other processes requests
  - Deadlock exists, consisting of processes $P_1$, $P_2$, $P_3$, and $P_4$
Detection Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle

- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock
Deadlock Recovery – Process Termination

- Aborting 1 or more processes will release their resources
- Different schemes
  - Abort all deadlocked processes
  - Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?
Deadlock Recovery – Resource Preemption

- Selecting a victim
  - Attempt to minimize cost

- Rollback
  - Return to some safe state
  - Restart process for that state

- Starvation
  - Same process may always be picked as victim, include number of rollback in cost factor
Next Class

- Midterm review