ARE YOU COMING TO BED?

I CAN'T. THIS IS IMPORTANT.

WHAT?

SOMEONE IS WRONG ON THE INTERNET.
Reminders: February 24, 2014

• Assignment 2 is due next Friday, March 7

• Lectures on March 5 and March 7 will instead be help sessions in the lab to complete the assignment
  • another lab beside CSC 2-19 will be booked as well

• Any questions about anything?
Some response to your feedback

- I really liked a lot of the suggestions
- Make deadlines more consistent: smaller exercises every week
- I’ll try to talk louder and slower
- Answers to Thought Questions (probably in class)
- Unfortunately, I have no control on whether its a morning or afternoon class
- Posting lectures earlier and improving pacing of lab lectures
- Vagueness in assignments/exercises, labs and demos
  - I’ll try to make specifications more clear, demos slower with more explanation
- Theory in lectures too far from assignments and labs
  - This is intentional, mostly as there is not enough time and too many OSs to properly bridge the gap. Goal is to give a foundation for you to learn more later
Some clarifications due to your feedback

• All demoed code is posted online under Reference Materials

• Some nervousness about final exam; we’ll go through a practice final during the last week of classes

• Some nervousness about C programming skills; you’re probably smarter than you give yourself credit for

• We will do an (anonymous) code review in pairs so you can see how others code

• If you cannot make office hours, please email me and we can set up a time; come ask me questions and get help early
Example where deadlocks really matter

- Database systems do a lot of read and writes (more so than your laptop or other operating environments)

- With many many concurrent reads and writes and corresponding locks, there is a higher chance of getting into a deadlocked state

- e.g. Process P0 and P1 both want to write to two tables

\[
\begin{align*}
P_0 & \quad \text{wait}(S); \\
& \quad \text{wait}(Q); \\
& \quad \ldots \\
& \quad \text{signal}(S); \\
& \quad \text{signal}(Q); \\
\end{align*}
\]

\[
\begin{align*}
P_1 & \quad \text{wait}(Q); \\
& \quad \text{wait}(S); \\
& \quad \ldots \\
& \quad \text{signal}(Q); \\
& \quad \text{signal}(S); \\
\end{align*}
\]
Deadlock prevention typical in databases

- System is by design guaranteed not to deadlock
- This design is usually guaranteed by avoiding circular waits by imposing an ordering on resources
- e.g. let lock S be first in order, lock Q second. Is the below now possible?

```
P_0
wait(S);
wait(Q);
...
signal(S),
signal(Q);
P_1
wait(Q);
wait(S);
...
signal(Q);
signal(S);
```
How can we more formally check if the ordering is enough to prevent deadlock?

- For strong correctness results, we usually need mathematically rigorous analysis
- Back to our favourite tool for analyzing deadlocks: resource allocation graphs!
- We can see what happens to a resource allocation graph when we impose an ordering on the resource vertices
Recall: resource allocation graphs

- Formal description using resource allocation graphs

Set of processes,

\[ P = \{ P_1, P_2, \ldots, P_n \} \]

Set of resources,

\[ R = \{ R_1, R_2, \ldots, R_n \} \]

Vertex set: \( V = P \cup R \)

Request edge: \( P_i \rightarrow R_j \)

Assignment edge: \( R_j \rightarrow P_i \)
Another example: is there a deadlock?

Multiple resource unit case: No Deadlock—yet!

Because, either $P_2$ or $P_4$ could relinquish a resource allowing $P_1$ or $P_3$ (which are currently blocked) to continue. $P_2$ is still executing, even if $P_4$ requests $R_1$. 
Another example: is there a deadlock?

First graph

Slight modification on First graph
Preventing deadlock: what happens if we impose a strict ordering on resources?

• Then we can ensure that we will not have any cycles, which means we cannot have a deadlock!

• Definition 1 in hand-out: a resource allocation graph

• Definition 2 in hand-out: total resource ordering
  1. resources have a strict ordering, i.e. value of $R_i < R_j$ for $i < j$
  2. each process is assigned the same value as highest valued resource that is allocated to it
  3. process can only request resources with higher value than itself
**Proof:** resource ordering ensures no cycles

- **Proof by contradiction:** assume $G$ has a directed cycle $C$
- Then there must be a vertex $v_i$ with lowest value in $C$
  - if there are two vertices with lowest value, then pick earlier one in cycle
  - cannot be more than two vertices with lowest value (exercise: why not?)
- Therefore there is some vertex $v_j$ with directed edge into $v_i$ such that $\text{Value}(v_i) < \text{Value}(v_j)$
Proof: resource ordering ensures no cycles

- $v_j$ has a directed edge to $v_i$ such that $\text{Value}(v_i) < \text{Value}(v_j)$

- Case 1: $v_i$ is a process
  - then $v_j$ must be a resource; this is a contradiction, since a directed edge from $v_j$ to $v_i$ means $v_j$ is allocated to $v_i$ so $\text{Value}(v_i)$ must be at least as large as $\text{Value}(v_j)$

- Case 2: $v_i$ is a resource
  - then $v_j$ must be a process; this is a contradiction, since a directed edge from $v_j$ to $v_i$ means $v_j$ is requesting $v_i$, but process $v_j$ can only request resources with higher value

- Since both cases are not possible, there cannot exist a cycle $C$ in resource allocation graph $G$
Why is this useful?

- Let’s us rigorously show that by imposing a total ordering on resources, we guarantee our system is deadlock-free.
- We can do cool things like use previous saved resource allocation graphs from our system to automatically find a good ordering on resources (using an order sorting algorithm for direct acyclic graphs).
- This resource order is based on previous behaviour of processes requesting resources, rather than on our guess of what might be a good ordering.
What happens if we do not impose a strict ordering on resources?

- Could have some resources that have the same weight
- More practical but also more complicated case, now cannot have the same strong statement
  - e.g. if all vertices have same weight, then back to no ordering
- With advances in graph theory, might still be able to say something rigorous for weak orderings in graphs
Video break: brought to you by another fantastic classmate!
Strategy 2: Prevent Deadlocks

- Many different ways to prevent deadlocks, from
- very restrictive but easy to guarantee no deadlock to
- less restrictive but harder to guarantee no deadlock
- Enforcing a strict ordering on resources is effective for guarantees but might be restrictive
Deadlock Avoidance

- Deadlock avoidance allows the four necessary conditions (i.e. does not design the system to prevent them from occurring) but at runtime makes judicious choices to ensure the system remains deadlock-free.

- Dynamically decide whether current resource request will be granted; will not grant it if could lead to deadlock.

- Therefore, deadlock avoidance requires:
  - a notion of a “safe” state
  - some knowledge of future requests for process resources.
Safe state for deadlock avoidance

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

- System is in **safe state** if there exists a sequence \(<P_1, P_2, ..., P_n>\) of ALL the processes in the systems 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 \(j < i\).

- That is:
  - 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, and so on.
Slight confusion about “safe sequence”

- The safe sequence (e.g. \(<P1,P3,P4,P2,P0>\)) does NOT mean that the processes have to be run in that order.

- Rather, it means that the current assignment and request edges for those processes will not result in deadlock because there exists some order that shows that those processes can complete and progressively release resources.

- A safe sequence ensures there is no circular wait, because it ensures that each process can individually still request up to its maximum number of resources in order to complete.

- It is not enough to that the processes are not deadlocked with their current allocation, since they might need to request more to complete and then create a deadlock.
### Example of unsafe allocation

One resource type, multiple instances: 12 tape drives

<table>
<thead>
<tr>
<th></th>
<th>Maximum Needs</th>
<th>Current Needs</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

- If P2 is given one more tape drive, then no longer safe
- What if P0 now needs remaining 5 tape drives to complete; since there are only 2 available, P0 waits
- What if P2 also now needs remaining 6 tape drives to complete; since there are only 2 available, P2 waits
- Even if P1 completes and releases resources, P0 & P2 deadlocked
Deadlock avoidance algorithms

• Single instance of each resource type → use a resource allocation graph and see if a new request creates a cycle

• It takes $O(|V| + |E|)$ time to determine if a graph has a cycle (e.g. using Tarjan’s strongly connected components algorithm)

• Multiple instance of a resource type

• find a knot (with a cycle) in a graph; a “knot” is sometimes defined differently to either mean it includes a cycle or not (at most $O(n \times m^2)$)

• Banker’s algorithm
Banker’s algorithm

- Used when there are multiple instances of a resource
- 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
Data structures for Banker’s Algorithm

Let \( n \) = number of processes, and \( m \) = number of resources types.

- **Available**: Vector of length \( m \). If available \([j] = k\), there are \( k \) instances of resource type \( R_j \) available

- **Max**: \( n \times m \) matrix. If \( Max[i,j] = k \), then process \( P_i \) may request at most \( k \) instances of resource type \( R_j \)

- **Allocation**: \( n \times m \) matrix. If \( Allocation[i,j] = k \) then \( P_i \) is currently allocated \( k \) instances of \( R_j \)

- **Need**: \( n \times m \) matrix. If \( Need[i,j] = k \), then \( P_i \) may need \( k \) more instances of \( R_j \) to complete its task

\[
Need[i,j] = Max[i,j] - Allocation[i,j]
\]
Banker’s algorithm: check if system in safe state

1. Let \textit{Work} and \textit{Finish} be vectors of length \( m \) and \( n \), respectively. Initialize:

   \[
   \text{Work} = \text{Available} \\
   \text{Finish}[i] = \text{false} \text{ for } i = 0, 1, \ldots, n-1
   \]

2. Find an \( i \) such that both:
   (a) \( \text{Finish}[i] = \text{false} \)
   (b) \( \text{Need}_i \leq \text{Work} \)

   If no such \( i \) exists, go to step 4

3. \( \text{Work} = \text{Work} + \text{Allocation}_i \)
   \( \text{Finish}[i] = \text{true} \)
   go to step 2

4. If \( \text{Finish}[i] == \text{true} \) for all \( i \), then the system is in a safe state
Banker’s algorithm: check if system in safe state

```java
for (int i = 0; i < n; i++) {
    // first find a thread that can finish
    for (int j = 0; j < n; j++) {
        if (!finish[j]) {
            boolean temp = true;
            for (int k = 0; k < m; k++) {
                if (need[j][k] > work[k])
                    temp = false;
            }

            if (temp) {
                // if this thread can finish
                finish[j] = true;
                for (int x = 0; x < m; x++)
                    work[x] += work[j][x];
            }
        }
    }
}
```

Runtime: \( O(m \times n^2) \)
Banker’s algorithm: resource-request algorithm

\( \text{Request}_i \) = request vector for process \( P_i \). If \( \text{Request}_i[j] = k \) then process \( P_i \) wants \( k \) instances of resource type \( R_j \)

1. If \( \text{Request}_i \leq \text{Need}_i \) go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If \( \text{Request}_i \leq \text{Available} \), go to step 3. Otherwise \( P_i \) must wait, since resources are not available

3. Pretend to allocate requested resources to \( P_i \) by modifying the state as follows:

\[
\begin{align*}
\text{Available} & = \text{Available} – \text{Request}_i; \\
\text{Allocation}_i & = \text{Allocation}_i + \text{Request}_i; \\
\text{Need}_i & = \text{Need}_i – \text{Request}_i;
\end{align*}
\]

- If safe \( \Rightarrow \) the resources are allocated to \( P_i \)
- If unsafe \( \Rightarrow \) \( P_i \) must wait, and the old resource-allocation state is restored
Example: Request Algorithm

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true

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