Revisiting resource deadlocks

- There are non-shared computer resources
  - Maybe more than one instance
  - Printers, Semaphores, Tape drives, CPU
- Processes need access to these resources
  - Acquire resource
    - If resource is available, access is granted
    - If not available, the process is blocked
  - Use resource
  - Release resource
- Undesirable scenario:
  - Process A acquires resource 1, and is waiting for resource 2
  - Process B acquires resource 2, and is waiting for resource 1
  ⇒ Deadlock!

Deadlocks with resources

Definition:
Deadlock exists among a set of processes if
- Every process is waiting for an event
- This event can be caused only by another process in the set
- Event is the acquire of release of another resource

For example: Locks

Object X, Y:

Process A code:
```c
/* initial compute */
X.acquire();
Y.acquire();
... use X and Y ...
Y.release();
X.release();
```

Process B code:
```c
/* initial compute */
Y.acquire();
X.acquire();
... use X and Y ...
X.release();
Y.release();
```

Can we do this for resource wait?

- Observation: the conditions won't be identical
- In particular, a resource-wait cycle might not imply that a deadlock has occurred

Reminder: Conditions for Process-Wait Deadlocks to arise

- Mutual Exclusion
  - At least one resource must be held in non-sharable mode
- Hold and wait
  - There exists a process holding a resource, and waiting for another
- No preemption
  - Resources cannot be preempted
- Circular wait
  - There exists a set of processes \([P_1, P_2, \ldots, P_n]\), such that
    - \(P_i\) is waiting for \(P_{i+1}\), \(P_{i+2}\), \ldots, and \(P_{i+k}\) for \(P_i\)
Deadlock can be described using a resource allocation graph, RAG.
The RAG consists of:
- set of vertices \( V = P \cup R \)
- where \( P = \{P_1, P_2, \ldots, P_n\} \) of processes and \( R = \{R_1, R_2, \ldots, R_m\} \) of resources.
- Request edge: directed edge from a process to a resource.
  - \( P_i \rightarrow R_j \) implies that \( P_i \) has requested \( R_j \).
- Assignment edge: directed edge from a resource to a process.
  - \( R_j \rightarrow P_i \) implies that \( R_j \) has been allocated to \( P_i \).

If the graph has no cycles, deadlock cannot exist.
If the graph has a cycle, deadlock may exist.

Dealing with Deadlocks

- "Reactive" Approaches: break deadlocks if they arise
  - Periodically check for evidence of deadlock
    - For example, using a graph reduction algorithm
    - Or just using timeout on the lock acquire operations
  - Then need a way to recover
    - Could blue screen and reboot the computer
    - Perhaps a thread can give up on what it was trying to do
  - Database systems always have a way to "back out" by "aborting" (rolling back) uncompleted activities
    - This lets them abort and then retry if a deadlock arises

Deadlock Prevention

- Can the OS prevent deadlocks?
- Prevention: Negate one of necessary conditions.
  - Let's try one by one... Mutual exclusion
    - Make resources sharable
    - Not always possible: concurrency conflicts may arise
  - Example of a way to "share" a resource
    - "Initiate work to be done asynchronously"
    - Later the O/S will do a notification when task finishes

Deadlock Prevention

- Hold and wait
  - One option: if you need to wait, must release resources, then re-acquire them after wait is finished (very awkward)
  - Or simply request everything all at once in one shot

These both have issues
- First approach is inefficient (endlessly acquires/releases the same things. Also attempt to reacquire a resource may fail)
- In second, what if you don’t know what resources will be needed until you actually run the code?
- Starvation (if you request lots of very popular resources)
- Low utilization (Might ask for things you don’t end up needing)
The last option is best

- Many systems use this last approach
  - Impose some kind of ordering on resources, like alphabetical by name, or by distance from the root of a tree, or by position on a queue
  - Ask for them in a fixed order (like smaller to larger)
- This does assume a code structure that respects the rules... if you can't do so, the approach may not be feasible in your application

Banker’s Algorithm

- Avoids deadlock using an idea similar to the way banks manage credit cards
  - For each process there is a “line of credit” corresponding to its maximum use of each kind of resource
    - E.g. “Sally can borrow up to $10,000 plus up to 22,900 and $5,000”
    - “Process P can use up to 10Gb of memory, and up to 10Gb of disk storage”
  - Each separate resource would have its own limit.
  - Banker needs to be sure that if customers pay their bills, it can pay the merchants. Banker’s algorithm uses the identical idea for resources.

Safe State

- We’ll say that the system (the bank) is in a safe state if we know that there is some schedule that lets us run every process to completion
  - When a process completes it releases its resources
  - In effect, Sally pays her credit card bill, letting the bank collect the money needed to pay Brooks Brothers, where Harry just bought some shirts
  - Not every state is safe. Bank is conservative: it makes you wait (when making a purchase) if granting that request right now would leave it in an unsafe state

Safe State with Resources

- Consider a system with processes \(P_1, P_2, \ldots, P_n\).
  - Let’s say that an “execution order” is just an ordering on these processes, perhaps \(P_2, P_1, P_3\)
  - If we know the maximum resource needs for each process, we can ask if a given execution order makes sense
    - E.g. to run \(P_1\), perhaps we need a maximum of 10Gb disk space
    - We can ask: do we actually have that much available?
  - Of course once \(P_1\) finishes, it will release that space

Safe State with Resources

- Consider a system with processes \(P_1, P_2, \ldots, P_n\).
  - Let’s say that an “execution order” is just an ordering on these processes, perhaps \(P_2, P_1, P_3\)
  - So: \(P_1\) must be executable “now” (we can satisfy its maximum need), but then will release resources it holds
  - Then \(P_2\) must be executable (if we reclaim \(P_1\)’s resources, we’ll be able to satisfy \(P_2\)’s worst-case needs)
  - … etc until every process is able to complete
Safe State with Resources

- A state is said to be safe, if it has an execution sequence 
  \( \{P_0, P_1, ..., P_j\} \), such that for each \( P_j \)
  the resources that \( P_j \) can still request can be satisfied by the
  currently available resources plus the resources held by all
  \( P_i \) where \( i < j \)
- How do we turn this definition into an algorithm?
  - The idea is simple: keep track of resource allocations
  - If a process makes a request
    - Grant it if (and only if) the resulting state is safe
    - Delay it if the resulting state would be unsafe

Confusing because...

- Keep in mind that the actual execution may not be the
  one that the bank used to convince itself that the state
  is safe
- For example, the banker’s algorithm might be looking
  at a request for disk space by process \( P_j \)
  - So it thinks “What if I grant this request?”
  - Computes the resulting resource allocation state
  - Then finds that \( \{P_j, P_{j-1}, ..., P_0\} \) is a possible execution
  - ... so it grants \( P_j \)’s request. Yet the real execution doesn’t
    have to be \( \{P_j, P_{j-1}, ..., P_0\} \) – this was just a worst case option

Safe State Example

- Suppose there are 12 tape drives

<table>
<thead>
<tr>
<th></th>
<th>max need</th>
<th>current usage</th>
<th>could ask for</th>
</tr>
</thead>
<tbody>
<tr>
<td>p0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>p1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>p2</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

- 3 drives remain

- current state is safe because a safe sequence exists: \( <p_1, p_0, p_2> \)
  - \( p_0 \) can complete with current resources
  - \( p_1 \) can complete with current + \( p_0 \)
  - \( p_2 \) can complete with current + \( p_0 \)
- if \( p_2 \) requests 1 drive, then it must wait to avoid unsafe state.

Safe State Example

- Suppose there are 12 tape drives

<table>
<thead>
<tr>
<th></th>
<th>max need</th>
<th>current usage</th>
<th>could ask for</th>
</tr>
</thead>
<tbody>
<tr>
<td>p0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>p1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>p2</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

- 3 drives remain

- current state is safe because a safe sequence exists: \( <p_1, p_0, p_2> \)
  - \( p_0 \) can complete with current resources
  - \( p_1 \) can complete with current + \( p_0 \)
  - \( p_2 \) can complete with current + \( p_0 \)
- if \( p_2 \) requests 1 drive, then it must wait to avoid unsafe state.

Safe State Example

<table>
<thead>
<tr>
<th>process</th>
<th>holding max claims</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4 6</td>
</tr>
<tr>
<td>B</td>
<td>4 11</td>
</tr>
<tr>
<td>C</td>
<td>2 9</td>
</tr>
</tbody>
</table>

- unallocated: 2
- deadlock-free sequence: \( A, C, B \)
- if \( C \) makes only 6 requests

However, this sequence is not safe:
If \( C \) should have 7 instead of 6 requests, deadlock exists.

Res. Alloc. Graph Algorithm

- Recall our resource allocation graphs... in fact the Banker’s Algorithm works by finding a graph reduction sequence:
  - For a requested resource it computes the resulting resource allocation graph in which every process requests its maximum need
  - Then checks to see if that graph can be reduced. If so the state is safe and the request is granted. If not the request must wait.
  - Graph reduction order is the “safe schedule”
Banker’s Algorithm

- So...
  - A process pre-declares its worst-case needs
  - Then it asks for what it “really” needs, a little at a time
  - The algorithm decides when to grant requests
- It delays a request unless:
  - It can find a sequence of processes...
  - ... such that it could grant their outstanding need...
  - ... so they would terminate...
  - ... letting it collect their resources...
  - ... and in this way it can execute everything to completion!

Banker’s Algorithm

- Decides whether to grant a resource request.
- Data structures:
  - n: integer # of processes
  - m: integer # of resources
  - available[1..m] available[i] is # of avail resources of type i
  - max[1..n,1..m] max demand of each Pi for each Ri
  - allocation[1..n,1..m] current allocation of resource Rj to Pi
  - need[1..n,1..m] max # resource Rj that Pi may still request
  - let request[] be vector of # of resource Rj Process Pi wants

Basic Algorithm

1. If request[] > need[] then
   error (asked for too much)
2. If request[] > available[] then
   wait (can’t supply it now)
3. Resources are available to satisfy the request
   - Let’s assume that we satisfy the request. Then we would have:
     available = available - request[]
     allocation[] = allocation[] + request[]
     need[] = need[] - request[]
   - Now, check if this would leave us in a safe state:
     if yes, grant the request,
     if no, then leave the state as is and cause process to wait.

Safety Check

free[1..m] = available /* how many resources are available */
finish[1..n] = false (for all i) /* none finished yet */

Step 1: Find an i such that finish[i]=false and need[]= 0
  /* find a proc that can complete its request now */
  if no such i exists, go to step 3 /* we’re done */

Step 2: Find an i:
  finish[i] = true /* done with this process */
  free[] = free[] + allocation[]
  /* assume this process were to finish, and its allocation
  go back to the available list */

Step 3: If finish[i] = true for all i, the system is safe. Else Not

Banker’s Algorithm: Example

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

this is a safe state: safe sequence <P1, P3, P4, P2, P0>

Suppose that P1 requests (1,0,2)
- add it to P1’s allocation and subtract it from Available
### Banker’s Algorithm: Example

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Max</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

This is still safe: safe seq <P1, P3, P4, P0, P2>

In this new state, P4 requests (3, 0, 2)
not enough available resources
Po requests (0, 2, 0)
let's check resulting state

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Max</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>P1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

This is unsafe state (why?)
So Po's request will be denied

Problems with Banker’s Algorithm?

### Problems with Banker’s Alg.

- May be hard to figure out the maximum needs
  - If too conservative, Bank doesn't allow any parallelism
  - But if too optimistic, a process could exceed its limit
    - It can request a bigger limit (a bigger "line of credit")
    - We can grant that request if the state would still be safe
    - But we might not be able to do so, and in that case the process would have to wait, or be terminated
- Some real systems use Banker’s Algorithm but it isn’t very common. Many just impose limits
  - If resource exhaustion occurs, they blue screen

### Deadlock summary

- We've looked at two kinds of systems
  - Process-wait situations, where "process P is waiting for process Q" – common when using locks
  - Resource-wait situations, where "Process P needs resource R" – more general
- We identified necessary conditions for deadlock in the process-wait case
- We found ways to test for deadlock
- We developed ways to build deadlock-free systems, such as ordered requests and Banker’s Algorithm

### Real systems?

- Some real systems use these techniques
- Others just recommend that you impose time-limits whenever you wait, for anything
  - But you need to decide what you’ll do when a timeout expires!
- Database transactions are a very effective option, but only if you are working with databases or files.