Objective for today:
Formal treatment of deadlock.

Dealing with Deadlock
It can be prevented by breaking one of the prerequisite conditions (review):
- Mutually exclusive use of resources
  - Example: Allowing shared access to read-only files (readers/writers problem from readers point of view)
- Circular waiting
  - Example: Define an ordering on resources and acquire them in order (lower numbered fork first)
- Hold and wait
- No pre-emption

Dealing with Deadlock (cont.)
Let it happen, then detect it and recover
- Via externally-imposed preemption of resources
Avoid dynamically by monitoring resource requests and denying some.
- Banker’s Algorithm ...
The Zax Deadlock Example

Deadlock Theory

State of resource allocation captured in 
Resource Graph
- Bipartite graph model with 
  a set \( P \) of vertices 
  representing processes 
  and 
  a set \( R \) for resources.
- Directed edges
  - \( R_i \rightarrow P_j \) means \( R_i \) alloc to \( P_j \)
  - \( P_j \rightarrow R_i \) means \( P_j \) requests \( R_i \)
- Resource vertices contain units of the resource

Deadlock defined on graph:
- \( P_i \) is blocked in state \( S \) if there is no operation \( P_i \) can perform
- \( P_i \) is deadlocked if it is blocked in all reachable states from \( S \)
- \( S \) is safe if no reachable state is a deadlock state (i.e., having some deadlocked process)

Reusable Resources

State transitions by operations:
- Granting a request
- Making a new request if all outstanding requests satisfied
Deadlock Theory

- Cycle in graph is a necessary condition
  - no cycle $\rightarrow$ no deadlock.
- No deadlock iff graph is completely reducible
  - Intuition: Analyze graph, asking if deadlock is inevitable from this state by simulating most favorable state transitions.

The Zax Deadlock Example

Deadlock Detection Algorithm

Let $U$ be the set of processes that have yet to be reduced. Initially $U = P$. Consider only reusable resources.

while (there exist unblocked processes in $U$)  
  { Remove unblocked $P_i$ from $U$; 
    Cancel $P_i$’s outstanding requests;  
    Release $P_i$’s allocated resources;  
    if (* possibly unblocking other $P_j$ in $U$ *)  
    if ($U \neq \lambda$) signal deadlock;  
  }


Deadlock Detection Example

Another Example

Another Example

Is there an unblocked process to start with?
Another Example

With and without $P_2$

Another Example

With and without $P_2$

Another Example

With and without $P_2$
Another Example

Is there an unblocked process to start with?
With and without P₂

Consumable Resources

- Not a fixed number of units, operations of producing and consuming (e.g. messages)
- Ordering matters on applying reductions
  - Reducing by producer makes “enough” units, to

Consumable Resources

- Not a fixed number of units, operations of producing and consuming (e.g. messages)
- Ordering matters on applying reductions
  - Reducing by producer makes “enough” units, to
  - Start with P₂
Consumable Resources

• Not a fixed number of units, operations of producing and consuming (e.g. messages)
• Ordering matters on applying reductions
  – Reducing by producer makes “enough” units, to
  – Start with $P_2$
• Not reducible

![Diagram of Consumable Resources]

Consumable Resources

• Not a fixed number of units, operations of producing and consuming (e.g. messages)
• Ordering matters on applying reductions
  – Reducing by producer makes “enough” units, to
  – Start with $P_2$

Consumable Resources

• Not a fixed number of units, operations of producing and consuming (e.g. messages)
• Ordering matters on applying reductions
  – Reducing by producer makes “enough” units, to
  – Start with $P_1$
Consumable Resources

- Not a fixed number of units, operations of producing and consuming (e.g. messages)
- Ordering matters on applying reductions
  - Reducing by producer makes “enough” units, to
  - Start with P₁
Deadlock Detection & Recovery

- Continuous monitoring and running this algorithm are expensive.
- What to do when a deadlock is detected?
  - Abort deadlocked processes (will result in restarts).
  - Preempt resources from selected processes, rolling back the victims to a previous state (undoing effects of work that has been done)
  - Watch out for starvation.

Avoidance - Banker’s Algorithm

- Each process must declare its maximum claim on each of the resources and may never request beyond that level.
- When a process places a request, the Banker decides whether to grant that request according to the following criteria:
  - “If I grant this request, then there is a run on the bank (everyone requests the remainder of their maximum claim), will we have deadlock?”

Representing the State

- $n$ processes, $m$ resources
- $\text{avail}[m] - \text{avail}[i]$ is the number of available units of $R_i$
- $\text{max}[n,m] - \text{max}[i,j]$ is claim of $P_i$ for $R_j$
- $\text{alloc}[n,m] - \text{alloc}[i,j]$ is current allocation of $R_j$ to $P_i$
- $\text{need}[n,m] = \text{max}[n,m] - \text{alloc}[n,m]$ - the rest that can be requested.
Basic Outline of Algorithm

if (request[i,j] > avail[j]) defer;
//Sufficient resources for request
//pretend to grant request
avail[j] = avail[j] - request[i,j];
alloc[i,j] = alloc[i,j] + request[i,j];
need[i,j] = need[i,j] - request[i,j];
if (safe state) grant; else defer;
Basic Outline of Algorithm

if (request[i,j] > avail[j]) defer;
//Sufficient resources for request
//pretend to grant request
    avail[j] = avail[j] - request[i,j];
    alloc[i,j] = alloc[i,j] +
        request[i,j];
    need[i,j] = need[i,j] -
        request[i,j];
if (safe state) grant; else defer;