Deadlock

Concurrency Issues

- Past lectures:
  - Problem: Safely coordinate access to shared resource
  - Solutions:
    - Use semaphores, monitors, locks, condition variables
    - Coordinate access *within* shared objects

- What about coordinated access *across* multiple objects?
  - If you are not careful, it can lead to *deadlock*

- Today's lecture:
  - What is deadlock?
  - How can we address deadlock?

Deadlocks
Motivating Examples

- Two *producer* processes share a buffer but use a different protocol for accessing the buffers

```
Producer1() {
  P(emptyBuffer)
  P(producerMutexLock)
   :
}
```

```
Producer2(){
  P(producerMutexLock)
  P(emptyBuffer)
   :
}
```

- A postscript interpreter and a visualization program compete for memory frames

```
PS_Interpreter() {
  request(memory_frames, 10)
  <process file>
  request(frame_buffer, 1)
  <draw file on screen>
}
```

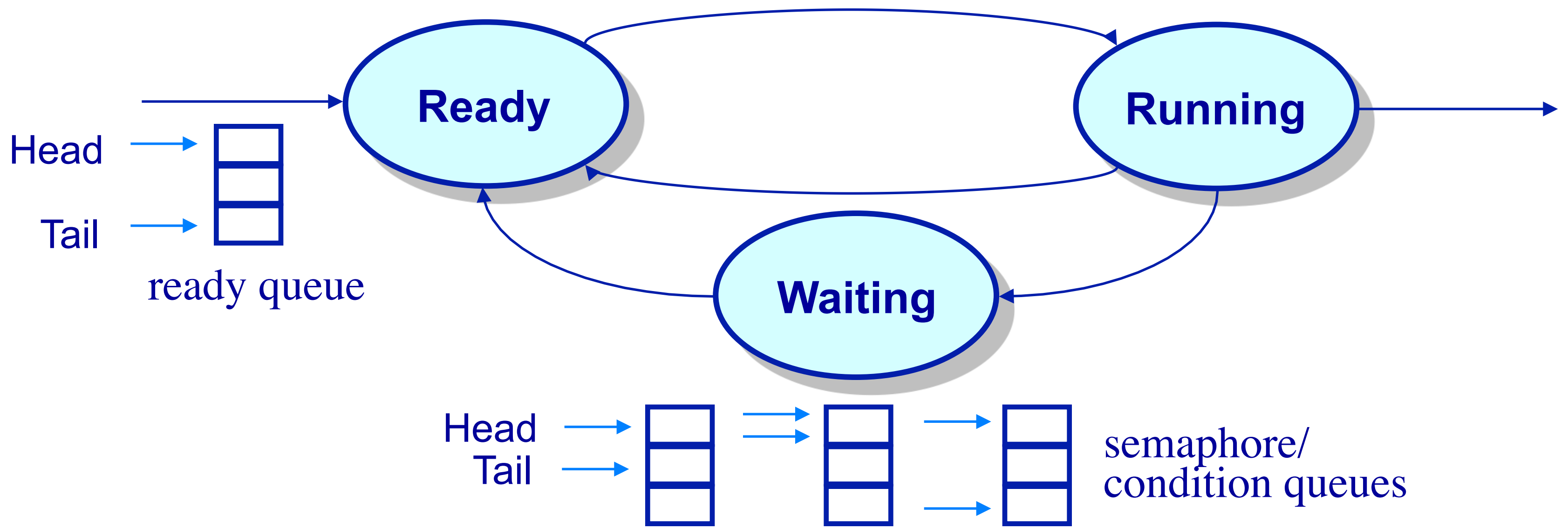
```
Visualize() {
  request(frame_buffer, 1)
  <display data>
  request(memory_frames, 20)
  <update display>
}
```

The TENEX Case

- If a process requests all systems buffers, operator console tries to print an error message
- To do so
  - lock the console
  - request a buffer

DUH!

Deadlock
Definition



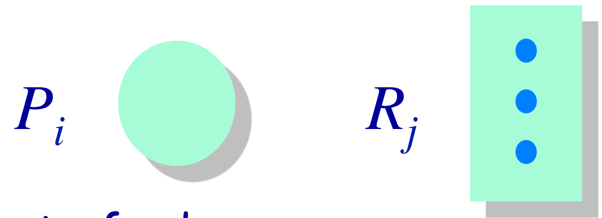
- A set of processes is deadlocked when every process in the set is waiting for an event that can only be generated by some process in the set

- Starvation vs. deadlock
  - Starvation: threads wait indefinitely (e.g., because some other thread is using a resource)
  - Deadlock: circular waiting for resources
  - Deadlock → starvation, but not the other way

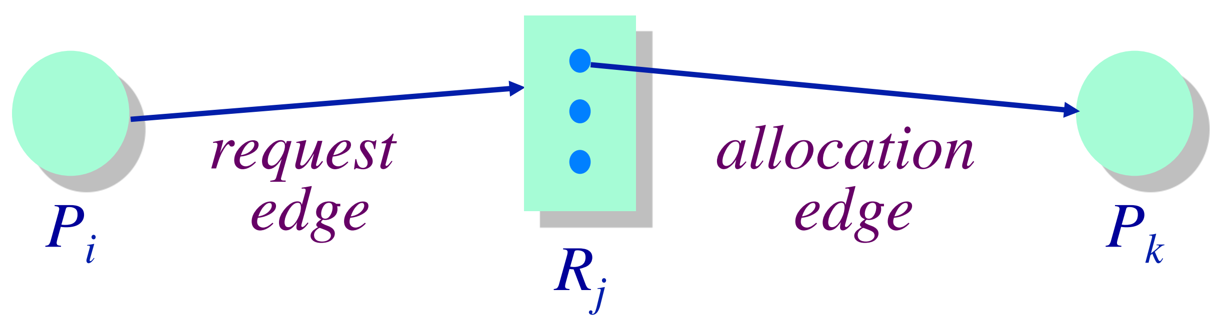
A Graph Theoretic Model of Deadlock

The resource allocation graph (*RAG*)

- Basic components of any resource allocation problem
  - Processes and resources
- Model the state of a computer system as a directed graph
  - $G = (V, E)$
  - V = the set of vertices = $\{P_1, \ldots, P_n\} \cup \{R_1, \ldots, R_m\}$



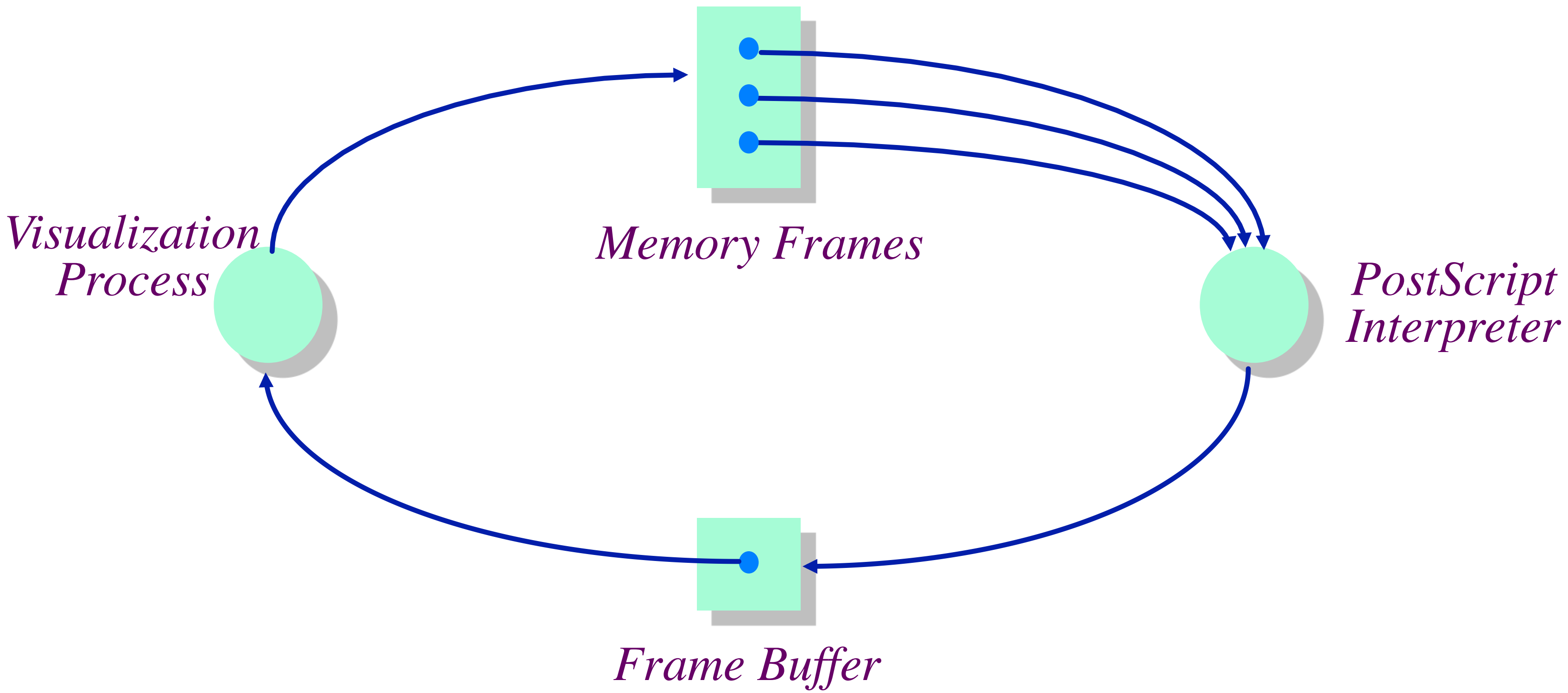
  - E = the set of edges =
    {*edges from a resource to a process*} ∪
    {*edges from a process to a resource*}



Resource Allocation Graphs
Examples

- A PostScript interpreter that is waiting for the frame buffer lock and a visualization process that is waiting for memory

V = {*PS interpret*, *visualization*} ∪ {*memory frames*, *frame buffer lock*}



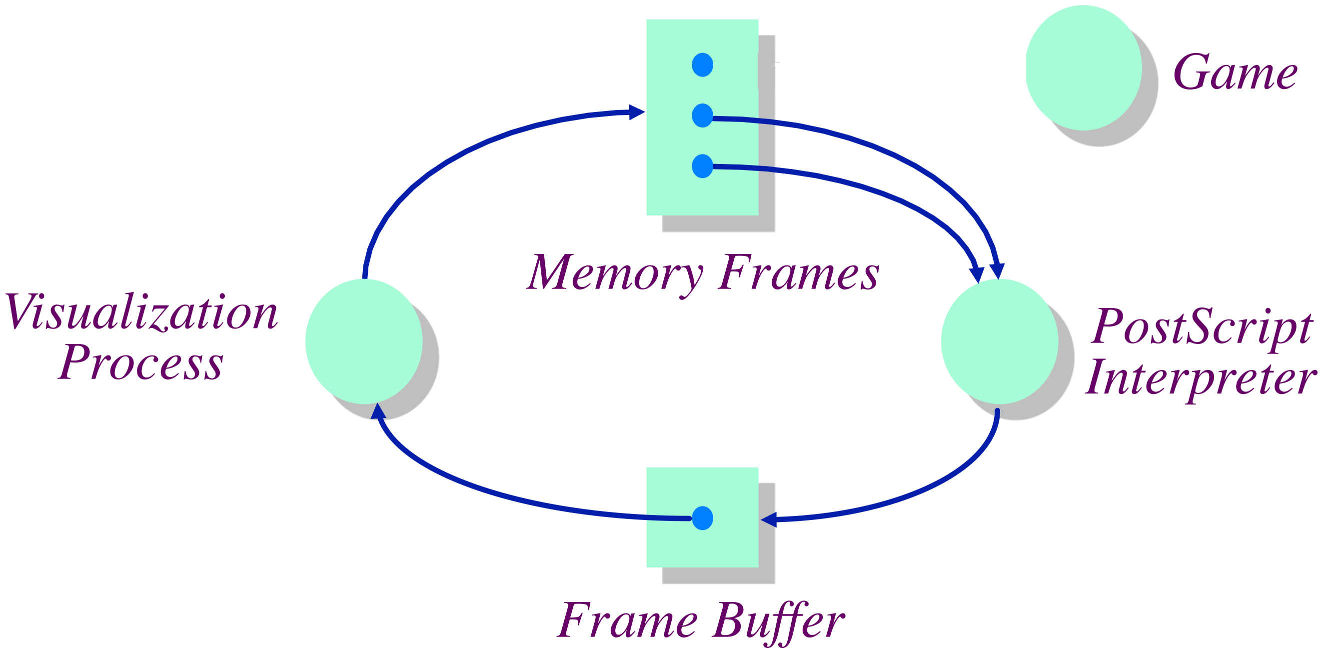
A Graph Theoretic Model of Deadlock

Resource allocation graphs & deadlock

- Theorem: *If a resource allocation graph does not contain a cycle then no processes are deadlocked*

A cycle in a *RAG is* a necessary condition for deadlock

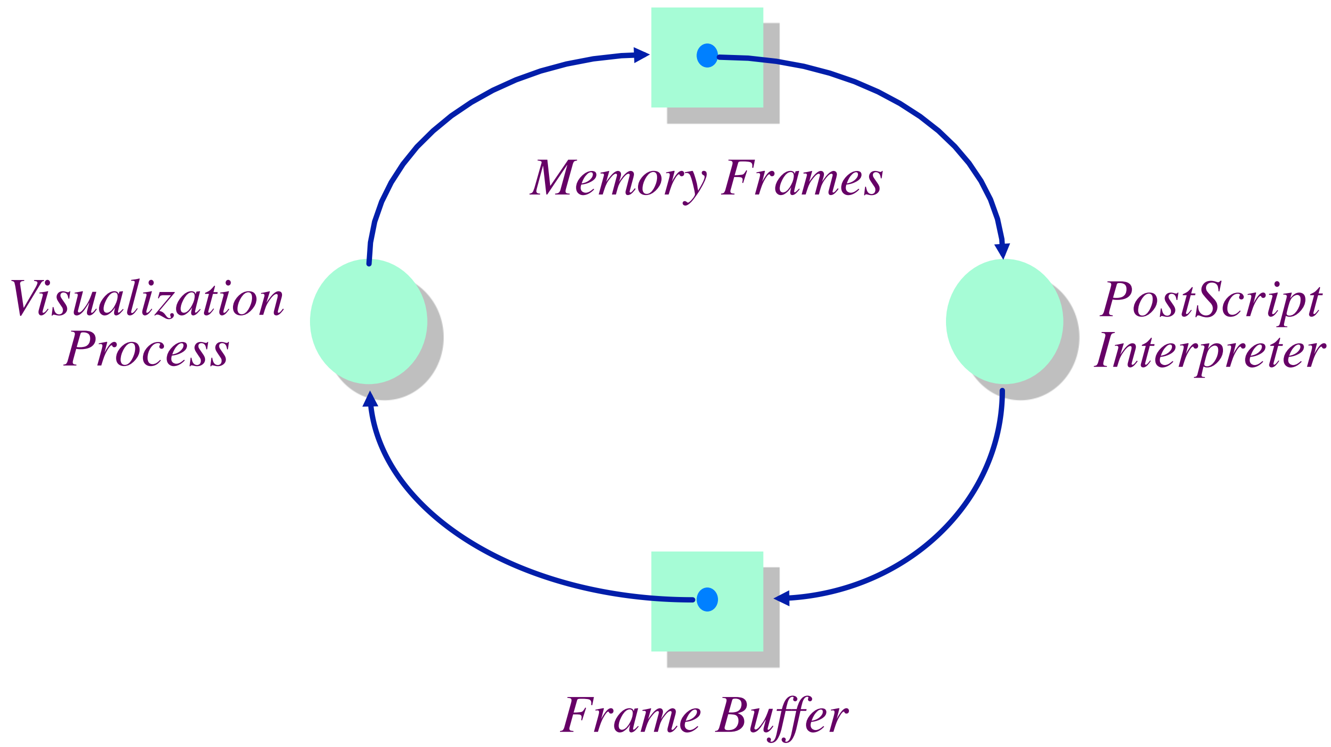
Is the existence of a cycle a sufficient condition?



A Graph Theoretic Model of Deadlock

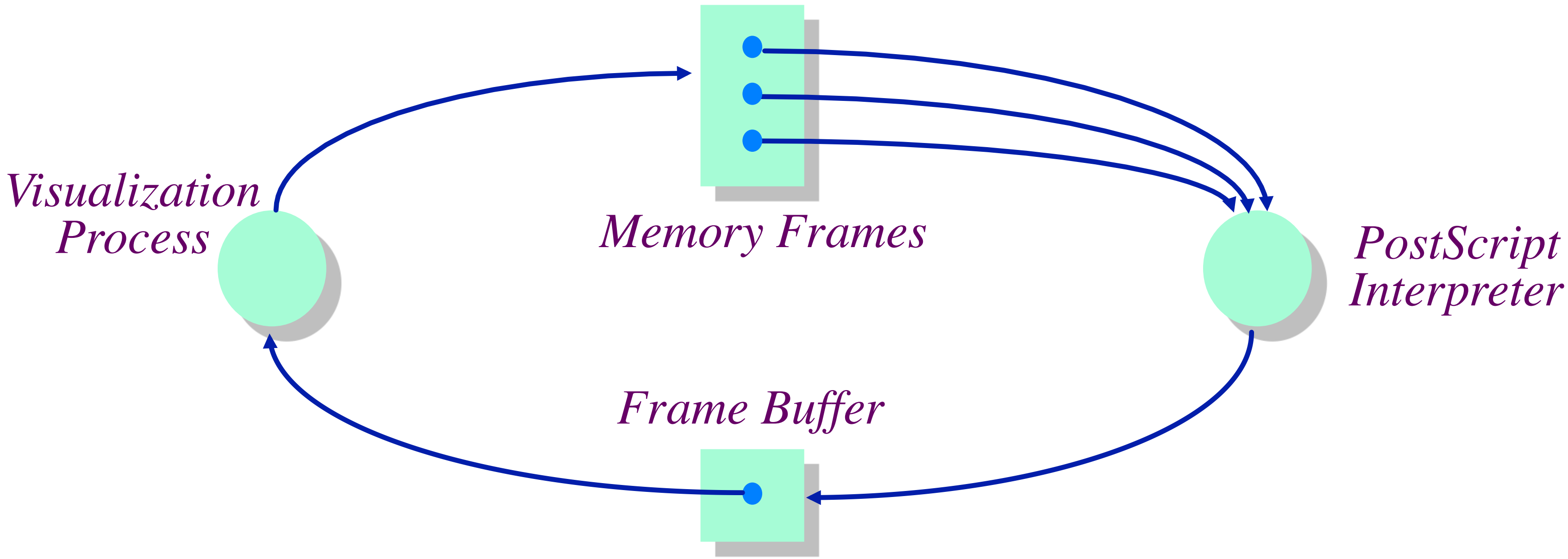
Resource allocation graphs & deadlock

- Theorem: *If there is only a single unit of all resources then a set of processes are deadlocked iff there is a cycle in the resource allocation graph*



Using the Theory

An operational definition of deadlock



- A set of processes are deadlocked *iff* the following conditions hold simultaneously
  1. Mutual exclusion is required for resource usage (serially useable)
  2. A process is in a "hold-and-wait" state
  3. Preemption of resource usage is not allowed
  4. Circular waiting exists (a cycle exists in the *RAG*)

Dealing With Deadlock
Deadlock prevention & avoidance

- Adopt some resource allocation protocol that ensures deadlock can never occur
  - Deadlock prevention/avoidance
    - Guarantee that deadlock will never occur
    - Generally breaks one of the following conditions:
      - Mutex
      - Hold-and-wait
      - No preemption
      - Circular wait *This is usually the weak link*
  - Deadlock detection and recovery
    - Admit the possibility of deadlock occurring and periodically check for it
    - On detecting deadlock, abort
      - Breaks the no-preemption condition

What does the RAG for a lock look like?

Deadlock Avoidance

Resource Ordering

- Recall this situation. How can we avoid it?

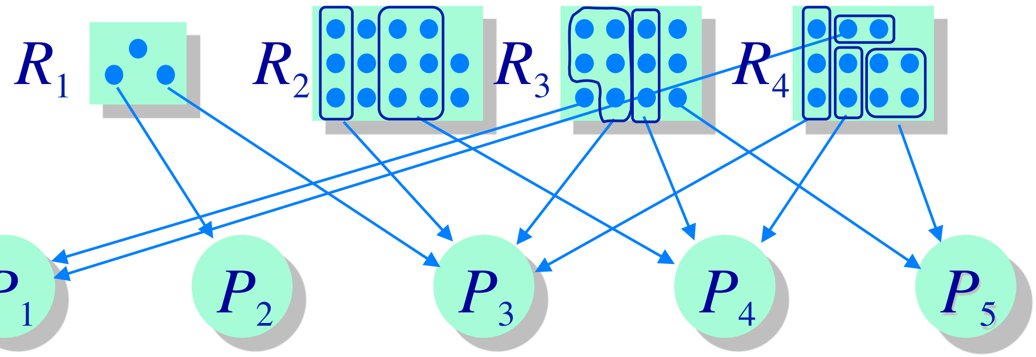
```
Producer1() {
  P(emptyBuffer)
  P(producerMutexLock)
   :
}
```

```
Producer2(){
  P(producerMutexLock)
  P(emptyBuffer)
   :
}
```

- Eliminate circular waiting by ordering all locks (or semaphores, or resoruces). All code grabs locks in a predefined order. Problems?
  - *Maintaining global order is difficult, especially in a large project.*
  - *Global order can force a client to grab a lock earlier than it would like, tying up a resource for too long.*
  - *Deadlock is a global property, but lock manipulation is local.*

Deadlock Detection & Recovery

Recovering from deadlock



- Abort all deadlocked processes & reclaim their resources
- Abort one process at a time until all cycles in the *RAG* are eliminated
- Where to start?
  - Select low priority process
  - Processes with most allocation of resources
- Caveat: ensure that system is in consistent state (e.g., transactions)
- Optimization:
  - Checkpoint processes periodically; rollback processes to checkpointed state

Dealing With Deadlock

Deadlock avoidance – Banker's Algorithm

- Examine each resource request and determine whether or not granting the request can lead to deadlock

Define a set of vectors and matrices that characterize the current state of all resources and processes

- *resource allocation state matrix*

  $Alloc_{ij}$ = the number of units of resource *j* held by process *i*

- *maximum claim matrix*

  Max_{ij} = the maximum number of units of resource *j* that the process *i* will ever require simultaneously

- *available vector*

  $Avail_j$ = the number of units of resource *j* that are unallocated

$$\begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \\ \vdots \\ P_p \end{array}
\begin{array}{ccccc} R_1 & R_2 & R_3 & \ldots & R_r \\ n_{1,1} & n_{1,2} & n_{1,3} & \ldots & n_{1,r} \\ n_{2,1} & n_{2,2} & & & \\ n_{3,1} & & \ddots & & \vdots \\ \vdots & & & & \\ n_{p,1} & & \ldots & & n_{p,r} \end{array}$$

$$\langle n_1, n_2, n_3, \ldots, n_r \rangle$$

Dealing With Deadlock

Deadlock detection & recovery

- What are some problems with the banker's algorithm?
  - Very slow $O(n^2m)$
  - Too slow to run on every allocation. What else can we do?

- Deadlock prevention and avoidance:
  - Develop and use resource allocation mechanisms and protocols that prohibit deadlock

- Deadlock detection and recovery:
  - Let the system deadlock and *then* deal with it
    - Detect that a set of processes are deadlocked
    - Recover from the deadlock