Deadlock — Last Time

Deadlock can exist if and only if the following conditions hold.

1. **Mutual Exclusion** - at least one thread must hold a resource in non-sharable mode, i.e., the resource may only be used by one thread at a time.
2. **Hold and Wait** - at least one thread holds a resource and is waiting for other resource(s) to become available. A different thread holds the resource(s).
3. **No Preemption** - A thread can only release a resource voluntarily; another thread or the OS cannot force the thread to release the resource.
4. **Circular wait** - A set of waiting threads $\{t_1, \ldots, t_n\}$ where t_i is waiting on t_{i+1} $(i = 1\ to\ n)$ and t_n is waiting on t_1.

Deadlock can occur when threads acquire multiple resources (printers, disks, etc.), perform work, and then release their resources.

Deadlock prevention algorithms check resource requests and possibly availability to prevent deadlock.

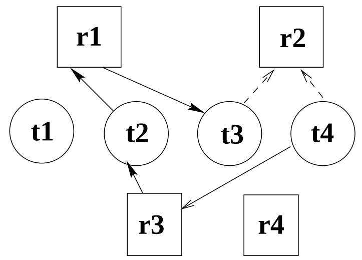
Deadlock detection and recovery finds instances of deadlock and takes some action to eliminate the deadlock.

Today

Deadlock prevention algorithms

Preventing Deadlock

- We can use a variant of the resource allocation graph that includes “claim” edges from threads to resources they may request.
- Satisfying a request results in converting a claim edge to an allocation edge and changing its direction.
- A cycle in this extended resource allocation graph indicates an unsafe state.
- If the allocation would result in an unsafe state, the allocation is denied even if the resource is available. The claim edge is converted to a request edge and the thread waits.
- This solution does not work for multiple instances of the *same* resource.



Banker's Algorithm:

- This algorithm handles multiple instances of the same resource.
- Force threads to provide advance information about what resources they may need for the duration of the execution.
- The resources requested may not exceed the total available in the system.
- The algorithm allocates resources to a requesting thread if the allocation leaves the system in a safe state.
- Otherwise, the thread must wait.

Preventing Deadlock with Banker's Algorithm (Java)

```
class ResourceManager {
  int n;         // # threads
  int m;         // # resources
  int avail[m],  // # available resources of each type
  max[n,m],      // # resources thread may want
  alloc[n,m],    // # resources thread is using
  need[n,m],     // additional resources thread may request

  public void synchronized
    allocate (int request[m], int i) {
    // request is a vector of the requested resources
    // i is the thread making the request

    if (request > need[i])  //vector comparison
      error();  // Can't request more than you declared
    else while (request[i] > avail)
      wait();   // Insufficient resources available

    // enough resources exist to satisfy the requests
    // See if the request would lead to an unsafe state
    avail = avail - request; // vector additions
    alloc[i] = alloc[i] + request;
    need[i] = need[i] - request;

    while ( !safeState () ) {
      // if this is an unsafe state,
      // undo the allocation and wait
      <undo the changes to avail, alloc[i], and need[i]>
      wait ();
      <redo the changes to avail, alloc[i], and need[i]>
  } }
```

Preventing Deadlock with Banker's Algorithm

Safety Check

```
private boolean safeState () {
  boolean work[m] = avail[m];  // accommodate all resources
  boolean finish[n] = false;   // none finished yet

  // find a process that can complete its work now

  while (find i such that finish[i] == false
       and need[i] <= work) { // vector operations
    work = work + alloc[i]
    finish[i] = true;
  }

  if (finish[i] == true for all i) {
     return true;
  }
  else {
     return false;
  }
}
```

Worst case, this algorithm requires an order of mn^2 operations to determine if the system is safe.

Example using Banker's Algorithm

System snapshot:

	Max			Allocation			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	0	1	0	0	1			
P_1	1	7	5	1	0	0			
P_2	2	3	5	1	3	5			
P_3	0	6	5	0	6	3			
total				2	9	9	1	5	2

1. How many resources are there of type (A,B,C)?

2. What is the contents of the Need matrix?

	A	B	C
P_0			
P_1			
P_2			
P_3			

3. Is the system in a safe state? Why?

4. If a request from process P_1 arrives for additional resources of (0,5,2), can the Banker's algorithm grant the request immediately?

What would be the new system state after the allocation?

	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	0	1									
P_1	1	7	5									
P_2	2	3	5									
P_3	0	6	5									
total												

What is a sequence of process execution that satisfies the safety constraint?

Summary of Deadlock Algorithms

- Deadlock can occur in any multi-threaded program (not just an OS) with the following conditions:
  - Mutual exclusion
  - Hold & wait
  - No resource preemption
  - Circular wait
- Detecting/preventing deadlock requires a global view of all resources, whereas semaphores give a localized view of an individual resource.
- Predicting the future is hard and programmers tend to overestimate their needs.
- Deadlock detection and avoidance as in the Banker's Algorithm are rarely implemented due to their expense.
- Many systems partition resources and then prevent deadlock by guaranteeing that one of the necessary conditions cannot hold for each class.

Next Time:

Big picture on synchronization mechanisms and their use.