Last Class: Deadlocks

- Necessary conditions for deadlock:
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait

- Ways of handling deadlock
  - Deadlock detection and recovery
  - Deadlock prevention
  - Deadlock avoidance

Today

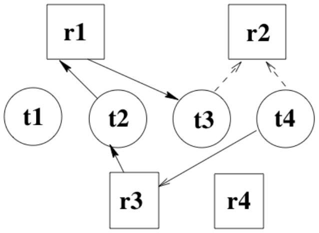
- Deadlock Avoidance: Banker's algorithm

- Synchronization wrap-up

- Exam review

Deadlock Avoidance

- Claim edges: an edge from a thread to a resource that may be requested in the future
- 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

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

Banker's Algorithm:Resource Allocation

```
public void synchronized allocate (int request[m], int i) {
  // request  contains the resources being requested
  // 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]>
 } }
```

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: requires O(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

Example (contd)

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

- What is the contents of the Need matrix?

	A B C
P_0	
P_1	
P_2	
P_3	

- Is the system in a safe state? Why?

Example: solutions

- How many resources of type (A,B,C)? (3,14,11)
  resources = total + avail
- What is the contents of the need matrix?
  Need = Max - Allocation.

	A B C
P_0	0 0 0
P_1	0 7 5
P_2	1 0 0
P_3	0 0 2

- Is the system in a safe state? Why?
- Yes, because the processes can be executed in the sequence P_0, P_2, P_1, P_3, even if each process asks for its maximum number of resources when it executes.

Example (contd)

- 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?

Example: solutions

- If a request from process P_1 arrives for additional resources of (0,5,2), can the Banker's algorithm grant the request immediately? Show the system state, and other criteria.

  Yes. Since

1. $(0,5,2) \le (1,5,2)$, the Available resources, and
2. $(0,5,2) + (1,0,0) = (1,5,2) \le (1,7,5)$, the maximum number P_1 can request.
3. The new system state after the allocation is:

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

and the sequence P_0, P_2, P_1, P_3 satisfies the safety constraint.

Synchronization Wrap up

- Low-Level Synchronization Primitives: hardware support

	Advantages	**Disadvantages**
Load/Store		
Interrupt Disable		
Test&Set		

- What can the OS do with these low-level primitives? the user?

High-Level Synchronization Primitives

- **Locks:**
  - Value: Initially lock is always free.
  - Acquire: Guarantees only one thread has lock; if another thread holds the lock, the acquiring thread waits, else the thread continues
  - Release: Enables another thread to get lock. If threads are waiting, one gets the lock, else, the lock becomes free.
- **Semaphores:**
  - Value: Initialization depends on problem.
  - Wait: Decrements value, Thread continues if value ≥ 0 (semaphore is available), otherwise, it waits on semaphore
  - Signal: unblocks a process on the wait queue, otherwise, increments value
  - A *counting semaphore* enables simultaneous access to a fixed number of resources
- **What is the relationship between semaphores and locks?**

High-Level Synchronization Primitives: Monitors

- *Monitor Locks* provide mutual exclusion to shared data.
  - Lock.Acquire -- wait until lock is free, then grab it.
  - Lock.Release -- unlock, and wake up any thread waiting in Acquire.
  - Always acquire lock before accessing shared data structure.
  - Always release lock when finished with shared data.
  - Lock is initially free.
- *A Condition Variable* is a queue of threads waiting for something inside a critical section. Operations:
  - Wait() - atomically release lock, go to sleep
  - Signal() - wake up waiting thread (if one exists) and give it the lock
  - Broadcast() - wake up all waiting threads
- **Rule:** thread must hold the lock when doing condition variable operations.

Deadlocks

- Necessary conditions for deadlock:
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait

- Ways of handling deadlock
  - Deadlock detection and recovery
  - Deadlock prevention
  - Deadlock avoidance

Exam Review

Processes and Threads

Topics you should understand:

1. What is a process?
2. What is a process control block? What is it used for? What information does it contain?
3. What execution states can a process be in? What do they mean? What causes a process to change execution states?
4. How does the OS keep track of processes?
5. What is a context switch? What happens during a context switch? What causes a context switch to occur?
6. What is the difference between a process and a thread?
7. What is the difference between a kernel thread and a user-level thread?
8. How are processes created? Fork() and Exec()
    1. Write pseudo-code for process creation using fork

CPU Scheduling

Topics you should understand:

1. What are FCFS, Round Robin, SJF, Multilevel Feedback Queue, and Lottery Scheduling algorithms?
2. What are the advantages and disadvantages of each?
3. What is preemptive scheduling? What is non-preemptive scheduling? Which scheduling algorithms can be preemptive?
4. What is a time slice? What effect does a very small time slice have? What effect does a very large time slice have?
5. What is an I/O bound process? What is a CPU bound process? Is there any reason to treat them differently for scheduling purposes?

CPU Scheduling

Things you should be able to do:

1. Given a list of processes, their arrival time, the lengths of their CPU and I/O bursts, and their total CPU time, you should be able to compute their completion time and waiting time for each scheduling algorithm we have discussed.
2. Given a variation to a scheduling algorithm we studied, discuss what impact you would expect that variation to have.

Synchronization

Topics you should understand:

1. Why do we need to synchronize processes/threads?
2. What is mutual exclusion?
3. What is a critical section?
4. What is a lock? What do you need to do to use a lock correctly?
5. What is a semaphore? What are the three things a semaphore can be used for?
6. What is a monitor? What is a condition variable? What are the two possible resumption semantics after a condition variable has been signaled? What are the advantages and disadvantages of each?
7. What is busy waiting?
8. How can interrupts be manipulated to support the implementation of critical sections? What are the advantages and disadvantages?
9. What is test&set? How can a test&set instruction be used to support the implementation of critical sections? What are the advantages and disadvantages?

Synchronization

Things you should be able to do:

1. Given some code that uses locks, semaphores, or monitors, you should be able to explain whether you believe it works. In particular, does it guarantee mutual exclusion where appropriate, does it avoid starvation, and does it avoid deadlock?

Deadlocks

Topics you should understand:

1. What are the four necessary conditions for deadlock to occur?
2. What is the difference between deadlock detection and deadlock prevention?
3. After detecting deadlock, what options are conceivable for recovering from deadlock?
4. What is a safe state? What is the difference between an unsafe state and a deadlocked state?

Deadlocks

Things you should be able to do:

1. Given some code, reason about whether or not it is possible for deadlock to occur.
2. Given a state consisting of resources allocated to processes, processes waiting on resources, and available resources, determine if the processes are deadlocked.
3. Given a state consisting of resources allocated to processes, maximum resource requirements of processes, and available resources, determine if the state could lead to deadlock.
4. Given a state consisting of resources allocated to processes, maximum resource requirements of processes, and available resources, and a request for additional resources from a process, determine if the request can be safely satisfied.
5. Given some code that might deadlock, describe how you might change the algorithm to prevent deadlock.

General Skills

- You should be able to read C++/Java code.
- You will be asked to write pseudo code with synchronization.
- You will **not** be asked detailed questions about any specific operating system, such as Unix, Windows, Mac OS X ...