Last Class: Synchronization Problems

- Reader Writer
  - Multiple readers, single writer
  - In practice, use read-write locks
- Dining Philosophers
  - Need to hold multiple resources to perform task

Real-world Examples

- Producer-consumer
  - Audio-Video player: network and display threads; shared buffer
  - Web servers: master thread and slave thread
- Reader-writer
  - Banking system: read account balances versus update
- Dining Philosophers
  - Cooperating processes that need to share limited resources
    - Set of processes that need to lock multiple resources
      - Disk and tape (backup),
    - Travel reservation: hotel, airline, car rental databases

Today: Deadlocks

- What are deadlocks?
- Conditions for deadlocks
- Deadlock prevention
- Deadlock detection

Deadlocks

- **Deadlock:** A condition where two or more threads are waiting for an event that can only be generated by these same threads.
- Example:

```
        Process A:                         Process B:
    printer.Wait();                  disk.Wait();
    disk.Wait();                       printer.Wait();

    // copy from disk                  // copy from disk
    // to printer                      // to printer

    printer.Signal();          printer.Signal();
    disk.Signal();             disk.Signal();
```

Deadlocks: Terminology

- **Deadlock** can occur when several threads compete for a finite number of resources simultaneously
- **Deadlock prevention** algorithms check resource requests and possibly availability to prevent deadlock.
- **Deadlock detection** finds instances of deadlock when threads stop making progress and tries to recover.
- **Starvation** occurs when a thread waits indefinitely for some resource, but other threads are actually using it (making progress).

  => Starvation is a different condition from deadlock

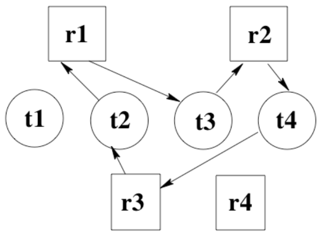
Necessary Conditions for Deadlock

Deadlock can happen if all the following conditions hold.

- **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.
- **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).
- **No Preemption:** A thread can only release a resource voluntarily; another thread or the OS cannot force the thread to release the resource.
- **Circular wait:** A set of waiting threads $\{t_1, ..., t_n\}$ where t_i is waiting on t_{i+1} ($i = 1$ to n) and t_n is waiting on t_1.

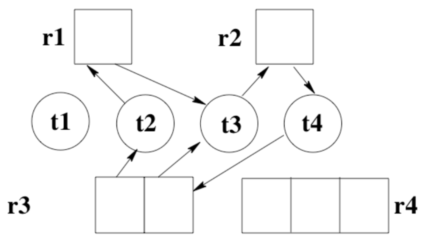
Deadlock Detection Using a Resource Allocation Graph

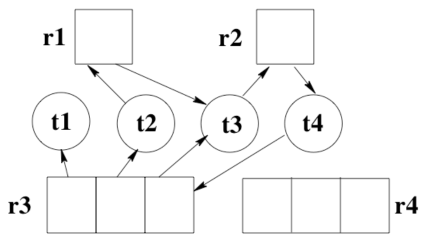
- We define a graph with vertices that represent both resources $\{r_1, ..., r_m\}$ and threads $\{t_1, ..., t_n\}$.
  - A directed edge from a thread to a resource, $t_i \rightarrow r_j$ indicates that t_i has requested that resource, but has not yet acquired it (*Request Edge*)
  - A directed edge from a resource to a thread $r_j \rightarrow t_i$ indicates that the OS has allocated r_j to t_i (*Assignment Edge*)
- If the graph has no cycles, no deadlock exists.
- If the graph has a cycle, deadlock might exist.



Deadlock Detection Using a Resource Allocation Graph

- What if there are multiple interchangeable instances of a resource?
  - Then a cycle indicates only that deadlock *might* exist.
  - If any instance of a resource involved in the cycle is held by a thread not in the cycle, then we can make progress when that resource is released.





Detect Deadlock and Then Correct It

- Scan the resource allocation graph for cycles, and then break the cycles.
- Different ways of breaking a cycle:
  - Kill all threads in the cycle.
  - Kill the threads one at a time, forcing them to give up resources.
  - Preempt resources one at a time rolling back the state of the thread holding the resource to the state it was in prior to getting the resource. This technique is common in database transactions.
- Detecting cycles takes $O(n^2)$ time, where n is $|T| + |R|$. When should we execute this algorithm?
  - Just before granting a resource, check if granting it would lead to a cycle? (Each request is then $O(n^2)$.)
  - Whenever a resource request can't be filled? (Each failed request is $O(n^2)$.)
  - On a regular schedule (hourly or ...)? (May take a long time to detect deadlock)
  - When CPU utilization drops below some threshold? (May take a long time to detect deadlock)
- What do current OS do?
  - Leave it to the programmer/application.

Deadlock Prevention

Prevent deadlock: ensure that at least one of the necessary conditions doesn't hold.

1. **Mutual Exclusion:** make resources sharable (but not all resources can be shared)
2. **Hold and Wait:**
   - Guarantee that a thread cannot hold one resource when it requests another
   - Make threads request all the resources they need at once and make the thread release all resources before requesting a new set.
3. **No Preemption:**
   - If a thread requests a resource that cannot be immediately allocated to it, then the OS preempts (releases) all the resources that the thread is currently holding.
   - Only when all of the resources are available, will the OS restart the thread.
   - *Problem:* not all resources can be easily preempted, like printers.
4. **Circular wait:** impose an ordering (numbering) on the resources and request them in order.

Deadlock Prevention with Resource Reservation

- Threads provide advance information about the maximum resources they may need during execution
- Define a sequence of threads $\{t_1, ..., t_n\}$ as *safe* if for each t_i, the resources that t_i can still request can be satisfied by the currently available resources plus the resources held by all t_j, $j < i$.
- A *safe state* is a state in which there is a safe sequence for the threads.
- An unsafe state is not equivalent to deadlock, it just may lead to deadlock, since some threads might not actually use the maximum resources they have declared.
- Grant a resource to a thread is the new state is safe
- If the new state is unsafe, the thread must wait even if the resource is currently available.
- This algorithm ensures no circular-wait condition exists.

Example

- Threads t_1, t_2, and t_3 are competing for 12 tape drives.
- Currently, 11 drives are allocated to the threads, leaving 1 available.
- The current state is *safe* (there exists a safe sequence, $\{t_1, t_2, t_3\}$ where all threads may obtain their maximum number of resources without waiting)
  - t_1 can complete with the current resource allocation
  - t_2 can complete with its current resources, plus all of t_1's resources, and the unallocated tape drive.
- t_3 can complete with all its current resources, all of t_1 and t_2's resources, and the unallocated tape drive.

	max need	in use	could want
t_1	4	3	1
t_2	8	4	4
t_3	12	4	8

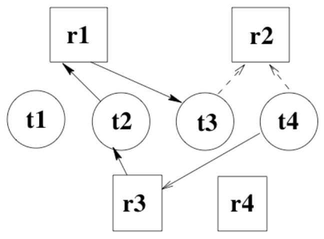
Example (contd)

- If t_3 requests one more drive, then it must wait because allocating the drive would lead to an unsafe state.
- There are now 0 available drives, but each thread might need at least one more drive.

	max need	in use	could want
t_1	4	3	1
t_2	8	4	4
t_3	12	5	7

Deadlock Avoidance using Resource Allocation Graph

- 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.

Summary

- Deadlock: situation in which a set of threads/processes cannot proceed because each requires resources held by another member of the set.
- Detection and recovery: recognize deadlock after it has occurred and break it.
- Avoidance: don't allocate a resource if it would introduce a cycle.
- Prevention: design resource allocation strategies that guarantee that one of the necessary conditions never holds
- Code concurrent programs very carefully. This only helps prevent deadlock over resources managed by the program, not OS resources.
- Ignore the possibility! (Most OSes use this option!!)

Exam 1 Review

Computer and OS Architecture

- Things you should know:
- Moore's Law, OS evolution
- Key hardware features and what they are used for
- OS architecture: monolithic, layered, microkernel, modular
- System calls

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?

- How are processes created? Fork() and Exec()
  - 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?

General Skills

- You should be able to read 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 ...