Last Class: Monitors

- Monitor wraps operations with a mutex
- Condition variables release mutex temporarily
- C++ does not provide a monitor construct, but monitors can be implemented by following the monitor rules for acquiring and releasing locks
- It is possible to implement monitors with semaphores

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

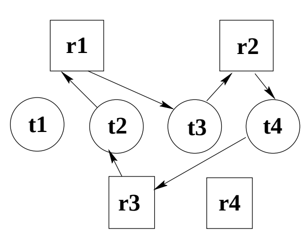
Necessary Conditions for Deadlock

Deadlock can happen if all the following conditions hold.

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.

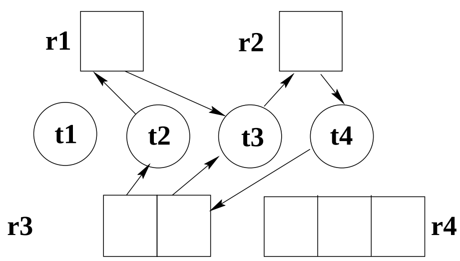
Deadlock Detection Using a Resource Allocation Graph

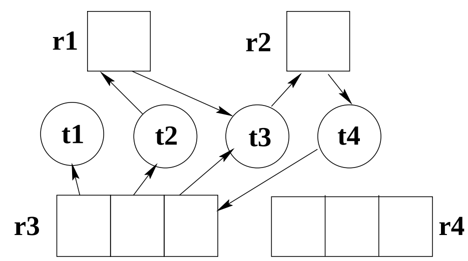
- We define a graph with vertices that represent both resources $\{r_1, \ldots r_m\}$ and threads $\{t_1, \ldots, t_n\}$.
  - A directed edge from a thread to a resource, $t_i \rightarrow r_j$ indicates that t_i has requested that resource, but has not yet acquired it (*Request Edge*)
  - A directed 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)

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 $\{t_1, \ldots, t_n\}$ 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

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!!)