Deadlocks and Starvation

Readings: Chapter on Deadlocks in Tanenbaum. You can skip Deadlock Detection and Banker's Algorithm.

- We looked at solving synchronization problems using monitors and semaphores.
- Unfortunately, problems can arise

Example 1:

Recall our solution to the reader-writer problem. It was possible for readers to wait indefinitely if new writers kept coming in.

On the other hand, writers would not wait indefinitely as long as ready threads are served in order.

Why?

Example 2:

Thread A	Thread B
lock(x) lock(y) Use resource X and Y Unlock(y) Unlock(x)	Lock(y) Lock(x) Use resource X and Y Unlock(x) Unlock(y)

In this case, it is possible that both Thread A and Thread B wait indefinitely for each other, with no progress being made.

What is common and different between the above examples?

Common aspect: Both problems involve threads *waiting* for *resources to become available. They can also involve*

- **Resources:** things needed by a thread to do its job a thread *waits* for resources
  - e.g., locks, AW+WW = 0 (i.e., database is free from active or waiting writers), disk blocks, memory pages

- **Indefinite wait:** In both examples, a thread may end up waiting indefinitely.

Differences between the two examples: The type of waiting is different.

- **Starvation**

  A thread may wait indefinitely because other threads keep coming in and getting the requested resources before this thread does. Note that resource is being actively used and the thread will stop waiting if other threads stop coming in.

- **Deadlocks**

  A group of threads are waiting for resources held by others in the group. None of them will ever make progress.

Example 1 has starvation, but Example 2 does not.

A solution to a synchronization problem suffers from the starvation problem if starvation is a *possibility*. Usually, differences in priorities can lead to starvation. Lower priority threads starve if higher priority threads keep requesting the resources.

A solution suffers from the deadlock problem if a deadlock is a *possibility*.

In Example 2, will a deadlock always occur?

In Example 2, can a deadlock occur?

Realistically, on any one run of the two threads, is a deadlock a low-probability event or a high-probability event?
What is the sequence of events that must occur for a deadlock to happen, assuming Thread 1 starts out first?

If a deadlock does happen, is it a serious event?

Avoiding starvation:

- Switch priorities so that every thread has a chance to have high priority.
    - E.g., Readers give priority to waiting writers, but active writers give priority to waiting readers. When both are waiting, they will end up alternating.
    - Raise priority if a thread has been waiting too long.
    - Use FIFO order among competing requests

Preventing deadlocks:

To understand how to prevent deadlocks, one must figure out necessary conditions for a deadlock to occur. There are four necessary

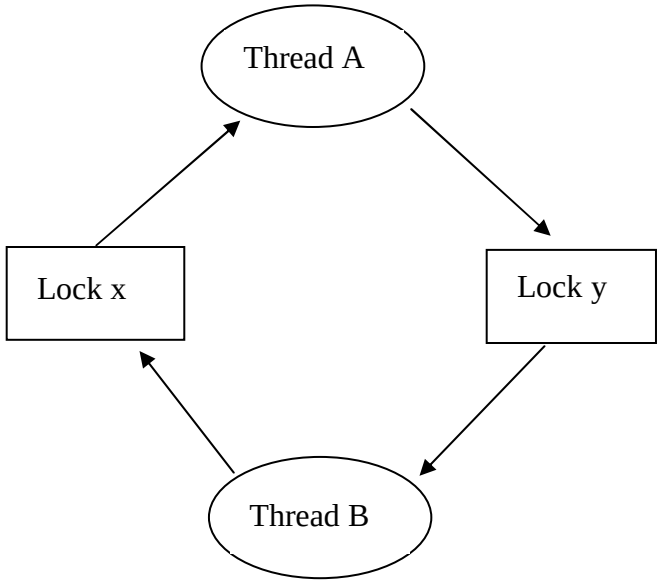
conditions, all of which must hold, for a deadlock to occur. If any one condition fails, then you cannot have a deadlock.

1) **Limited (shared) resource**
   i.e., not enough to serve all threads simultaneously. Otherwise there's no waiting.

2) **No preemption:** Preemption means you forcefully take away the resource from someone. (Hard to do that with locks.)

3) **Hold and wait:** Threads in a deadlock hold a resource, and wait for another resource

4) **Circular wait:**


Thread A
Lock x
Lock y
Thread B

- thread A is waiting for resource y
- resource y is held by thread B
- thread B is waiting for resource x
- resource x is held by thread A

The above is called a **wait-for** graph

Are the conditions independent? Or does one condition imply another?

If they are not independent, why list all of them?

Example 3: circular waiting for resources

e.g. both EECS 484 and EECS 482 are full

- you want to switch from EECS 484 to EECS 482, but want to be sure that you end up in either 484 or 482
- someone else wants to switch from EECS 482 to EECS 484 :-)

To switch, you want to add the new course, then drop the old course
(don't want to drop the old course and not be able to add the new course)

Both of you are waiting for each other to drop the course, will wait forever

The shared resources in this case is spot in the two classes.

Deadlock always leads to starvation (but not necessarily vice versa)

Example #4: dining philosophers
(This is a classic problem, also due to Dijkstra)

5 philosphers sitting around a round table, 1 chopstick in between each (5 chopsticks total)

each philosopher needs two chopsticks to eat

Algorithm for philosopher:

wait for right chopstick to be free, then pick it up
wait for left chopstick to be free, then pick it up
eat
put both chopsticks down

Can this deadlock? If so, how?

What to do about deadlocks?

1) Ignore them (this is actually a pretty common solution, because the others are hard and costly)

   If you have two processes waiting for each other, those processes just hang

   other processes can run fine, though

2) detect and fix

3) prevent

Detect and fix:

- detect (the easy part)
  - Scan the wait-for graph and detect cycles

- fix (the hard part)
  1) shoot the thread, force it to give up the resources that it's holding

     This isn't always possible. e.g., if you shoot a thread that's holding a lock, the shared data is left in an inconsistent state.

     2) roll-back actions of 1 or more threads, try again common technique in databases (transactions)

     This solves the "inconsistent state" problem in #1, by rolling back the thread to before it held the lock

Not always possible to roll back a thread. E.g., how do you undo the firing of a missile?

Deadlock prevention (and avoidance):

Basic idea is to get rid of one of the 4 necessary conditions

1) Make resources unlimited (or at least enough that there's no waiting)

- Frequently impossible, but this is the best solution
- Even if you can't make it so deadlock is impossible, you can make it very unlikely. E.g., large classrooms. Still possible to have full classes, but unlikely
- Sometimes, if you cannot make resources unlimited, you can reduce the number of threads competing for them.

Dining Philosophers:

2) Allow preemption
   - can preempt CPU (save its state in its thread control block and switch, get back to it later)
   - can preempt memory, e.g., by saving it to disk and loading it back later
   - can't preempt the holding of a lock

Usually not applicable to synchronization problems because it is dangerous to preempt locks. A thread may be in the middle of a critical section.

3) Eliminate hold and wait

- Eliminate hold: if can't satisfy resource right away, thread dies. E.g., phone company. If it can't get all the lines needed to call California, you get a busy signal.

- Time-out requests for each lock and retry? Will this work in Example 2?

  This is just another form of waiting if some resources are still held. This is sometime called a *livelock*.

- On a time-out, threads must be prepared to back out all the way, giving up all the held resources, and retry.

- Eliminate wait while holding: Request all resources at beginning. E.g., grab both chopsticks at same time (or grab one chopstick. If the other is busy, drop the one you grabbed. This could be viewed as undoing its recent actions)

  - hard to predict the future. Tend to overestimate how many resources need. Inefficient, because it reserves all resources at beginning

4) Circular chain of requests

Impose an ordering between resources and ensure that you always request resources in order.

E.g., three resources X, Y, and Z.

Order them: E.g., X, Y, Z.

Thread 1: Requires both Y and X
Requests X followed by Y.

Thread 2: Requires both X and Z
Requests X followed by Z

Thread 3: Requires X, Y, and Z.
Requests X, followed by Y, followed by Z.

Question: Why does it work?

Question: Could the resources have been ordered as Y, Z, X?

Exercise: Develop deadlock-free solutions to the Dining Philospher Problem by attacking each of the four conditions.