Deadlocks

- A deadlock is a situation where a set of threads/processes ceases to make progress because each is waiting for another in the set.
- Avoiding deadlocks is usually essential to the correct functioning of a concurrent system.
- But, avoiding deadlocks is often not easy.
- In this topic we will see some common methods for avoiding and resolving deadlocks.
- We will focus on Java programs, but the methods generally apply to other situations involving deadlocks as well.
- In practice, the best solution will depend on the exact features of the system and often a combination of methods is required, or even a completely new approach.

When can deadlocks occur?

- Deadlocks can occur in any concurrent system where processes wait for each other and a cyclic chain can arise with each process waiting for the next one in the chain.
- More specifically, deadlocks can occur in any system that satisfies the four *Coffman conditions* (due to E. G. Coffman in 1971):
  1. *Mutual Exclusion Condition*: a resource is either assigned to one process or it is available
  2. *Hold and Wait Condition*: processes already holding resources may request new resources
  3. *No Preemption Condition*: only a process holding a resource may release it
  4. *Circular Wait Condition*: two or more processes form a circular chain where each process requests a resource that the next process in the chain holds
- Concurrent Java programs may satisfy these conditions, with the object locks/monitors as the resources, so they may deadlock.
- We can avoid deadlocks by arranging our program so that one of the Coffman conditions doesn't hold.

Deadlock Prevention Technique 1

Technique 1: Avoiding holding multiple locks

- If no thread attempts to hold more than one lock, then no deadlocks can occur since the Hold and Wait condition is invalidated.
- For some systems, this is easy to arrange, but for others there is a genuine need to lock multiple objects at the same time.
- In such cases, minimizing the number of situations where threads hold multiple locks will often still reduce the number of situations that need to be resolved by other techniques, so is good practice.

Deadlock Prevention Technique 2

Technique 2: Using coarser-grained locks

- One way to modify a program to avoid holding multiple locks is to replace uses of multiple locks by a single lock.
- For example, the lock on a class object may be used instead of the individual locks on instances of the class.
- This can also sometimes avoid the Circular Wait Condition even when threads hold multiple locks.

```
class NoDeadlock {
      private int n;
      public NoDeadlock(int n) {this.n = n;}
      public boolean eq(NoDeadlock other) {
           synchronized(NoDeadlock.class)
               { return n==other.getN(); }
      }
      public int getN() {
           synchronized(NoDeadlock.class) { return n; }
      }
}
```

Deadlock Prevention Technique 2: Issues

- Combining into coarser-grained locks may result in threads waiting when they do not really need to, hence concurrency is reduced, which may be undesirable.
- New deadlocks may arise, defeating the purpose of the technique.
  - E.g., if have locks A1, A2 and B.
  - Combining A1 and A2 will allow this sequence to deadlock, when previously it could not:
    - Thread 1 wants A1 then B
    - Thread 2 wants B then A2
- We can always resolve the new deadlocks by combining more locks, until we ultimately use only a single lock for the whole program. (E.g., we could combine A1, A2 and B into a single lock.)
- But, this reduces the concurrency so much that it is generally undesirable, and anyway library code will often use its own locks.
- On the other hand, if threads only ever hold locks for very short periods of time, then perhaps a single lock is reasonable.

Deadlock Prevention Technique 3

Technique 3: Using finer-grained locks

- This is the opposite of technique 2.
- A common use of this technique is to replace a lock on a whole object by a number of locks for its parts.
- E.g., instead of locking a collection object like an array, the individual objects in the collection may be locked as appropriate.
- Just like technique 2, this may introduce deadlocks.
- Finer-grained locking tends to be more complex, so this method is only recommended where it is quite clear that it will work.

Deadlock Prevention Technique 4

Technique 4: Minimizing the holding of locks

- The default style in Java holds locks on objects whenever a method for the object is executing.
- Often some methods can be safely unsynchronized, and this may prevent deadlocks.
- It is not safe to do this if the method accesses multiple instance variables of the object, or the same variable many times, and it assumes that the object has not changed in between.
- Similarly, this is not safe if the object is modified temporarily to an inconsistent state without holding the lock.
- Often only a part of a method needs to be synchronized, since the other parts do not access instance variables, or do so only safely.

```
public int findData() {
     Boolean here=false;
     int n;
     synchronized (this) {
         here = this.dataIsHere;
         if(here) { n=this.getData(); }
     }
     if(!here) { n=otherObject.findData(); }
     return n;
}
```


Deadlock Prevention Technique 5

Technique 5: Reordering lock acquisition

- If we require threads to always acquire locks in a particular order, then no deadlock can occur.
- The Circular Wait Condition is avoided, since we cannot have a circular chain if threads can only wait for locks which come *after* the locks they've already acquired.
- This may be easy, or it may require major restructuring the code.
- Or, often it is practically impossible, since there is no practical way to predict which locks may be acquired in future.

```
class NoDeadlock2 {
      private int n;
      public NoDeadlock2(int n) {this.n = n;}
      public boolean eq(NoDeadlock2 b) {
           if(System.identityHashCode(this) >
              System.identityHashCode(b) )
                return other.eq(this);       // Reverses order
            else
                synchronized(this) { return n==other.getN(); }
      }
      public synchronized int getN() { return n; }
}
```

Deadlock Prevention Technique 6

Technique 6: Using alternative locking instead of `synchronized`

- `java.util.concurrent` contains a number of alternatives.
  - Simple re-entrant locks
  - Readers and writers locks
  - Semaphores
  - And more
- Generally these have the advantage that you can check whether a lock is held before requesting it, and avoid simply blocking until the lock is released.
- Also, often specialized locks like Readers and Writers will avoid deadlocks and increase concurrency.
- Additionally, you can request and release these locks anywhere, instead of only following the block structure of your code.
  [This is useful for many of the other techniques, such as 4 and 5.]
- However, this also means you need to be very careful that locks are always released.
- This includes when exceptions are thrown, so usually a finally clause is required.

Example for Technique 6

- As an example, the following code periodically polls to obtain locks on two objects, but releases one lock if the other is busy.

```
import java.util.concurrent.lock.ReentrantLock;
class NoDeadlock3 {
     private int n;
     protected ReentrantLock lock=new ReentrantLock();
     public NoDeadlock2(int n) {this.n = n;}

     private boolean getOurLocks(NoDeadlock3 other) {
          Boolean myLock, yourLock; // initially false
          try {
              myLock = lock.tryLock();
              yourLock = other.lock.tryLock();
          } finally {
              if (! (myLock && yourLock)) {
                  if (myLock) { lock.unlock(); }
                  if (yourLock) { other.lock.unlock(); }
              }
          }
          return myLock && yourLock;
     }
     public boolean eq(NoDeadlock3 other) {
          while(!getOurLocks(other)) { Thread.sleep(1); }
          Boolean ret = (n==other.getN());
          lock.unlock();  other.lock.unlock();
          return ret;
     }
}
```


Technique 6: Other Aspects of Alternative Locks

- The `ReentrantLock` class also supports a form of wait and notify via methods and using objects that represent condition variables.
- There is also support for finding the thread that holds a lock, and even finding all the threads waiting on a lock or on a condition variable.
- If none of the lock classes in `java.util.concurrent` are suitable for a particular purpose, support for writing your own is in `java.util.concurrent.locks.LockSupport`.

Deadlock Prevention Technique 7

Technique 7: Releasing and reacquiring locks in order

- This is basically a refinement of technique 5.
- Technique 6 is required in order to release locks without exiting a synchronized block or method.
- The basic idea is to always acquire locks in a particular order.
- When an out of order lock is need, locks after the new lock are released, the new lock is requested, then the released locks are re-requested.
- This always avoids deadlocks, can be implemented in a central *lock manager* class, and at first appears quite general.
- Alas, releasing locks and reacquiring them may be unsafe since a thread is usually holding locks for a reason.
  - It may be in the middle of modifying an object.
  - Or, it may depend on an object not being modified.

Deadlock Prevention Technique 8

Technique 8: Using interrupts and/or timed waits

- In this technique, deadlocks are avoided by aborting waits.
- The simplest case is a timed wait that takes some alternative action if the thread waits too long for a lock.
- Using the `Thread.interrupt` method can achieve a similar effect in some cases: if one thread knows that another should complete some task reasonably quickly, it can interrupt it if it doesn't.
- An interrupted thread will raise an exception.
- In either case, we need some way of failing gracefully to a point where we hold no locks.
- This failure process is usually called a *Rollback*.
- Generally the thread will try again once it has released all locks.

Deadlock Prevention Technique 9

Technique 9: Using an event loop instead of threads

- This technique avoids using threads all together, or at least minimizes their use, and is very common for GUIs, etc.
- Instead of using multiple threads, a single thread runs in a loop that handles one "event" each time.
- Each event handler should take a relatively short amount of time.
- Since events can occur while the thread is busy, generally a queue of events is used, with the thread handling each in turn.
- This technique can even be used to roughly simulate a new thread: in an event handler we can add a runnable object on the end of the queue, which we later execute in the event loop.
- Events that require I/O or a longer period can similarly put a *continuation* runnable object on the queue that completes the handling of the event.
- This isn't too far from implementing our own thread scheduler.
- It has the advantage that no interruptions occur, except when an event handler explicitly gives up control.
- Since there are no threads, locks are never needed, but the programming style required does not suit all systems.

Deadlock Prevention Technique 10

Technique 10: Restricted threads and asynchronous messages

- This technique restricts so that each thread always runs within a particular object, or group of objects, and is the only thread in that object/group.
- Communication between threads is via simple messages, rather than threads moving in to other objects.
- To prevent deadlocks, threads should not wait for a return value from a message, instead a separate message will be used.
- In the meantime, a thread may respond to other messages.
- Like in the previous technique, each thread generally has a queue of messages that it responds to in turn.
- With this scheme, deadlocks cannot occur, but the style required can be awkward.
- This scheme is quite common for distributed programming, and we will come back to it later.