Synchronization and Deadlock

*Chapters 5 and 6 of* [OS4e],
*Chapters 7 and 8 of* [OSC]:
- Java Synchronization
- Deadlock

Java Synchronization

Semaphores
- supported, in some form or other, by most operating systems
- sufficiently powerful to solve critical-section problem and other concurrency-control problems
- rather low-level (like *gotos*), so easy to misuse
  - e.g., forget to *signal* upon leaving critical section
  - e.g., get confused between *wait* and *signal*

Java instead associates *lock* with each object:
- only one method or block of code *synchronized* to an object may execute (i.e., own its lock) at a time
Application programmer
- declares object instead of semaphore
  
  Object x = new Object();
- encapsulates critical section instead of waiting at
  start and signalling at end
  
  synchronized(x) { ... };

This improves program structure, but even
better is for
- shared data, and code that manipulates it, to be
  encapsulated in an object (monitor)
  - in keeping with object-oriented programming
  - thread placed in object's entry queue when it invokes
    synchronized method and another thread owns lock

Sometimes, however, programmer would like
thread to wait until a certain condition is met
- e.g., consider a bounded-buffer, with put method
  invoked repeatedly by Producer thread, and get
  method invoked by Consumer
  - Producer should wait until buffer is not full
  - Consumer should wait until buffer is not empty

Thread can't simply yield, because it owns lock!

Java solves this by providing wait() method
- lock is released
- thread's state becomes Blocked
- thread placed on object's wait queue
Java also provides `notify()` method

- one thread (if any) transferred from wait queue to entry queue
- that thread’s state becomes Ready
- current thread continues executing

E.g. Bounded-Buffer Producer-Consumer problem can be solved in Java using the List interface (see http://java.sun.com/docs/books/tutorial/collections/), as follows

N.B. UNIX `pipe` is bounded buffer containing sequence of bytes

```java
import java.util.*;
public class BoundedBuffer {

    int capacity;
    List buffer;

    public BoundedBuffer(int c) {

        // Constructs an empty list with capacity c.
        capacity = c;
        buffer = new ArrayList(c);
    }
}
```
public synchronized void put(Object o) {
    if (buffer.size() == capacity) {
        try { wait(); }
        catch (InterruptedException e) {  }
    }
    buffer.add(o);
    notify();
}

public synchronized Object get() {
    Object o = null;
    if (buffer.isEmpty()) {
        try { wait(); }  
        catch (InterruptedException e) {  }
    }
    try { o = buffer.remove(0); }  
    catch (IndexOutOfBoundsException e) { }  
    notify();  
    return o;
}
In general, there may be several threads in wait queue and entry queue

Notified thread should re-check the condition on which it was waiting because

- it may be waiting for different condition from that established by notifying thread
- another thread may have got in first and negated the condition again

N.B. in Tony Hoare’s version of the monitor from the mid 1970s

- multiple condition variables can be declared, each with its own wait queue
- thread starts executing immediately it is notified

So to be on the safe side, use

```java
while (!condition) {
    try {
        wait();
    } catch (InterruptedException e) { }
}
```

Another problem is that if “wrong” thread notified, “right” thread might wait forever!

Instead use Java’s `notifyAll()` method
notifyAll() is safer, but more expensive, to use than notify() because
- all threads are transferred from wait queue to entry queue
- those threads’ state becomes Ready
- current thread continues executing

E.g. Readers-Writers problem can be solved in Java, as follows, where
- readerCount keeps track of number of readers
- writerCount is 0 or 1
- at most one of these counts is positive
- Reader thread performs `startRead before endRead`
- Writer thread performs `startWrite before endWrite`

```java
public class AccessControl {
    private int readerCount, writerCount;

    public AccessControl() {
        readerCount = 0;
        writerCount = 0;
    }
}
```
public synchronized void startRead() {
    while (writerCount == 1) {
        try {
            wait();
        } catch (InterruptedException e) {  }
        readerCount++;
    }
    public synchronized void endRead() {
        readerCount--;
        if (readerCount == 0) notifyAll();
    }
    public synchronized void startWrite() {
        while (readerCount > 0 || writerCount == 1) {
            try {
                wait();
            } catch (InterruptedException e) {  }
            writerCount = 1;
        }
        public synchronized void endWrite() {
            writerCount = 0;
            notifyAll();
        }
    }
Deadlock

Recall that files, I/O devices, regions of memory, semaphores, etc., are OS-managed resources that are held by processes

- request resource by making system call
  e.g., open file, allocate memory, wait on semaphore

- use resource

- release resource by making system call
  e.g., close file, free memory, signal on semaphore

Another kind of resource is the Java object
  – managed by JVM when using synchronized methods

*deadlock*

- situation in which set of waiting processes (or threads) cannot make progress because they are waiting for each other to release resources

To be “waiting for each other to release resources” means *n* processes *hold-and-wait*, where *n* > 1, forming *circular wait*

- *P*<sub>0</sub> is requesting *R*<sub>0</sub>, which is held by *P*<sub>1</sub> and cannot be shared by both (*mutual exclusion*)

- ...

- *P*<sub>n-1</sub> is requesting *R*<sub>n-1</sub>, which is held by *P*<sub>0</sub> and cannot be shared by both
Deadlock recovery involves forcing processes to release resources
- resource is said to be preempted
- operating systems commonly abort all deadlocked processes!

Example of possible deadlock in Java:
- one thread executes
  ```java
  synchronized(x) {
    synchronized(y) {
      ... }
  }
  ```
- other thread executes
  ```java
  synchronized(y) {
    synchronized(x) {
      ... }
  }
  ```

![Figure 6.5 Circular Wait](image-url)
Example suggests strategy for *deadlock prevention*:

- linear ordering is placed on resources
- when process requests resource $R$, it must have all resources lower in the ordering that are required if it is to use and release $R$

Besides deadlock recovery and deadlock prevention, *deadlock avoidance* can sometimes be arranged:

- need to know maximum resource requirements of processes
- must ensure there are always sufficient resources for one process to complete execution and release its resources when granting resource

Dining Philosophers problem (due to Dijkstra) is often used to illustrate deadlock:

- 5 philosophers
- 5 forks
- philosopher must acquire own and right-hand neighbour’s fork in order to eat spaghetti
- philosopher puts down both forks after eating

Danger is that all philosophers pick up their own fork and reach out for their neighbour’s...

Deadlock can be prevented by employing footman who ensures at most 4 philosophers can be seated at any time!