Agenda

- Assignment 3 out, due November 10
- Today's lecture
  - Java Synchronization: 6.8
  - Introduction to Deadlocks: 7.1, 7.2, 7.3, 7.4
  - No office hours for today
- Next lecture
  - More on Deadlocks
  - Detection, Avoidance, etc….

Common Concurrency Mechanisms

- Semaphore – integer value used for signaling among processes.
  - Initialize, increment, decrement operations may be performed on a semaphore
  - Operations are atomic
  - Decrement operation may result in the blocking of a process
  - Increment operation may result in the unblock of a process
  - Binary Semaphore – takes only the values 0 and 1
- Mutex – similar to a binary semaphore
  - With the restriction that the process that locks the mutex must be the one to unlock it
  - Condition variable: a data type used to block a process/thread until a particular condition holds

Common Concurrency Mechanisms

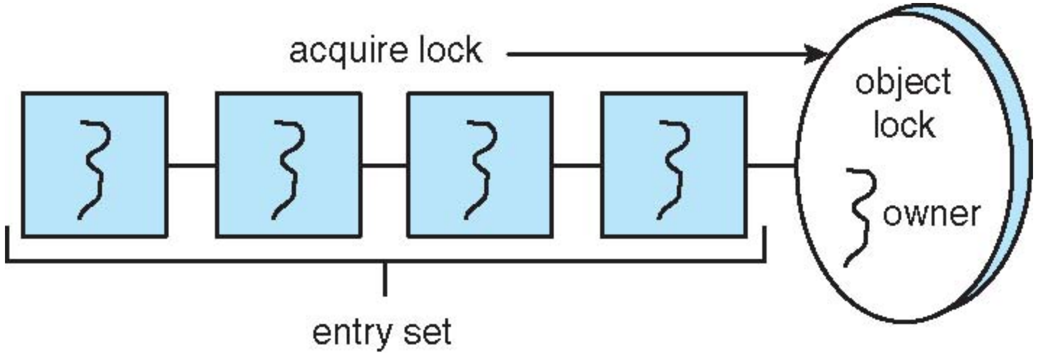
- Monitors – Language specific synchronization
  - Provide a fundamental guarantee that only one process may be in a monitor at any time
  - In Java, the lock associated with an object is actually a monitor
- Implemented at the compiler/language level (hidden from the user)
  - The compiler must ensure that the property is preserved
  - Implementation of mutual exclusion is system dependent
  - Can be implemented with semaphores, locks, or other mechanisms
- The critical section is inside the monitor, to execute CS a thread
  - Enters monitor
    - Wait if there is already a thread in the monitor
  - Executes critical section
  - Leaves critical section
    - Once a thread leaves, next waiting thread can enter

Java Synchronization

- Java provides synchronization at the language-level.
- Each Java object has an associated lock.
- This lock is acquired by invoking a **synchronized** method.
- This lock is released when exiting the **synchronized** method.
- Threads waiting to acquire the object lock are placed in the **entry set** for the object lock.

Java Synchronization

- Each object has an associated **entry set**.



Java Synchronization wait/notify()

- When a thread invokes **wait():**

  1. The thread releases the object lock;
  2. The state of the thread is set to Blocked;
  3. The thread is placed in the **wait set** for the object.

- When a thread invokes **notify()**:

  1. An arbitrary thread T from the wait set is selected;
  2. T is moved from the wait to the entry set;
  3. The state of T is set to Runnable.

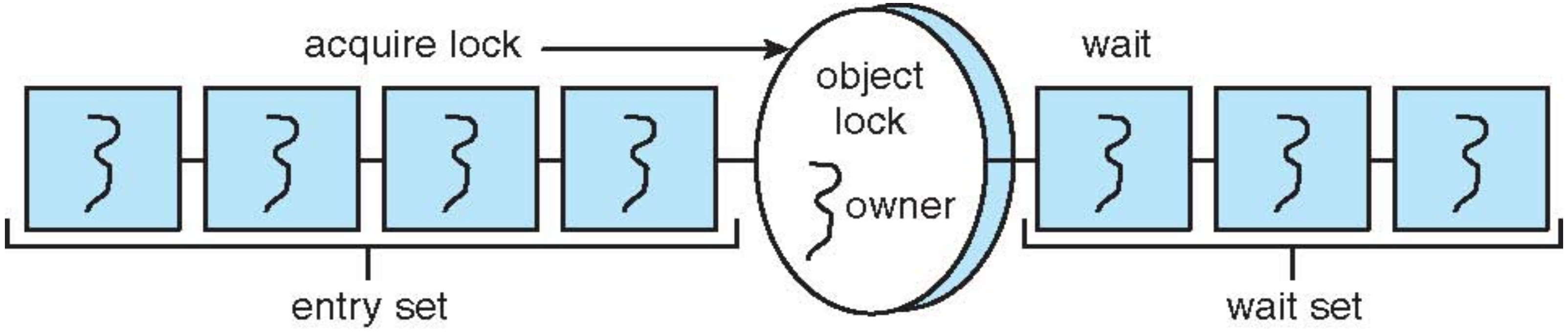
Java Synchronization wait/notify()

- When a thread invokes **wait():**

  1. The thread releases the object lock;
  2. The state of the thread is set to Blocked;
  3. The thread is placed in the **wait set** for the object.

- When a thread invokes **notify()**:

  1. An arbitrary thread T from the wait set is selected;
  2. T is moved from the wait to the entry set;
  3. The state of T is set to Runnable.



Java Synchronization

- Synchronized insert() and remove () methods – Incorrect!

- Thread stays in runnable state
- Allows JVM to select another thread to run

```
// Producers call this method
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE)
      Thread.yield();

    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    ++count;
}

// Consumers call this method
public synchronized E remove() {
    E item;

    while (count == 0)
      Thread.yield();

    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    --count;

    return item;
}
```

- NO busy waiting
- Prone to Livelock!

- Synchronized insert() method – Correct!

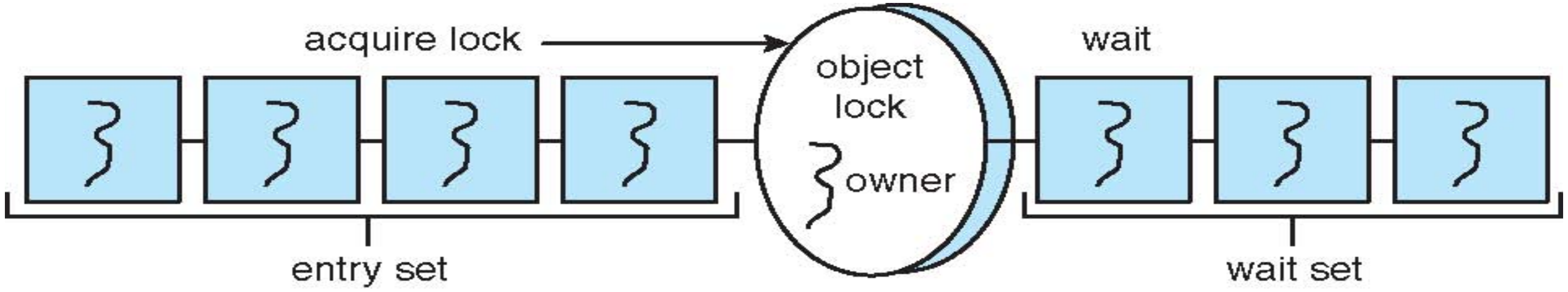
- putting the thread to sleep
- Block thread and put it in a queue

```
// Producers call this method
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE) {
      try {
        wait();
      }
      catch (InterruptedException e) { }
    }

    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    ++count;

    notify();
}
```

- Wake up a waiting process
- May not notify the correct thread

Java Synchronization

- The call to **notify()** selects an arbitrary thread from the wait set. It is possible the selected thread is in fact not waiting upon the condition for which it was notified.
- The call **notifyAll()** selects all threads in the wait set and moves them to the entry set.
- In general, **notifyAll()** is a more conservative strategy than **notify()**.



Concurrency Features in Java

- Prior to Java 5, the only concurrency features in Java were Using synchronized/wait/notify.

- Beginning with Java 5, new features were added to the API:
  - Reentrant Locks
    - `Lock key = ReentrantLock();`
    - `Lock(), unlock()`
  - Semaphores
    - `Semaphore s = new Semaphore(1);`
  - Condition Variables
    - `Condition cVar = key.newCondition();`

Concurrency Features in Java

- Prior to Java 5, the only concurrency features in Java were Using synchronized/wait/notify.

- Beginning with Java 5, new features were added to the API:
  - Reentrant Locks

```
Lock key = new ReentrantLock();

key.lock();
try {
   // critical section
}
finally {
   key.unlock();
}
```

Concurrency Features in Java

- Prior to Java 5, the only concurrency features in Java were Using synchronized/wait/notify.

- Beginning with Java 5, new features were added to the API:
  - Semaphores

```
Semaphore sem = new Semaphore(1);

try {
   sem.acquire();
   // critical section
}
catch (InterruptedException ie) { }
finally {
   sem.release();
}
```

Condition Variables

- A Condition Variable is created by first creating a ReentrantLock and invocking its newCondition () method:

```
Lock lock = new ReentrantLock();
Condition [] condVars = new Condition [5];
condVar[0] = lock.newCondition ();
```

- Once this is done, it is possible to invoke the await() and signal() methods

```
/**
 * myNumber is the number of the thread
 * that wishes to do some work
 */
public void doWork(int myNumber) {
  lock.lock();

  try {
    /**
     * If it's not my turn, then wait
     * until I'm signaled
     */
    if (myNumber != turn)
      condVars[myNumber].await();

    // Do some work for awhile . . .

    /**
     * Finished working. Now indicate to the
     * next waiting thread that it is their
     * turn to do some work.
     */

    turn = (turn + 1) % 5;
    condVars[turn].signal();
  }
  catch (InterruptedException ie) { }
  finally {
    lock.unlock();
  }
}
```

Synchronization Issues

- Synchronization is the other main use for locks, semaphores, and monitors
- Basic Idea: Threads wait for each other until certain conditions are met
  - Example: the Producer/Consumer problem.
- One of the things that happens when proper synchronization does not occur is deadlock

Resources

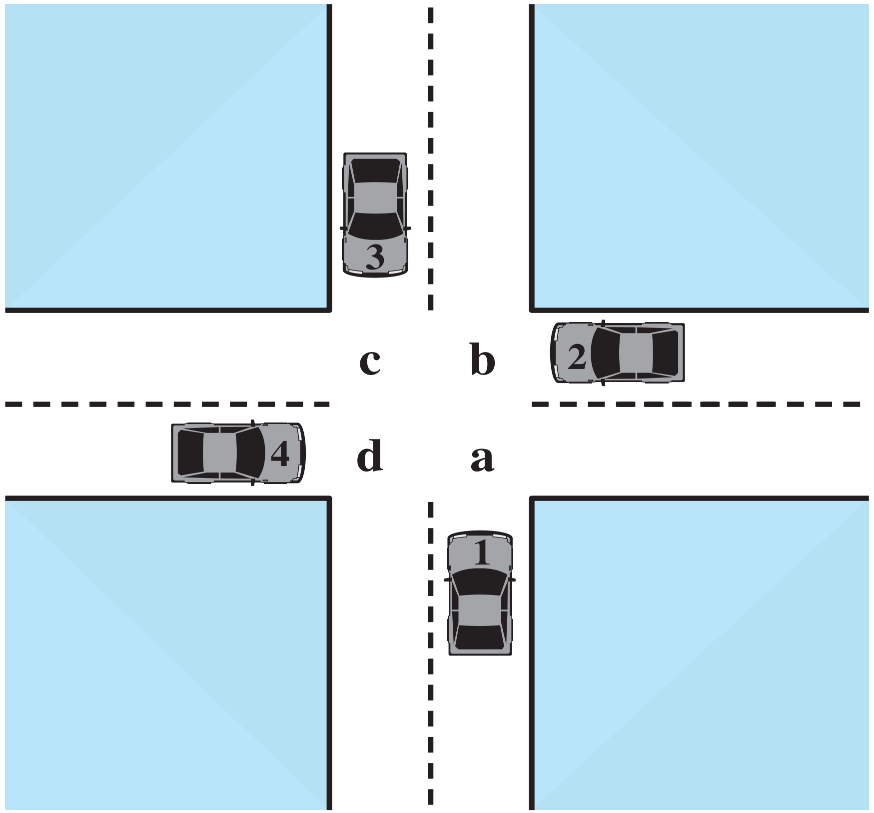
- A process may use one of more resources at a time.
- Sequence of events required to use a resource
  1. **Request the resource**
     - If the resource is not available, the requesting process is blocked
  2. **Use the resource**
     - The process using the resource has exclusive lock on the resource
  3. **Release the resource**
     - When done using the resource, process releases its lock on the resource
- Resource example: Memory, CPU, DVD reader, ….

What is a Deadlock?

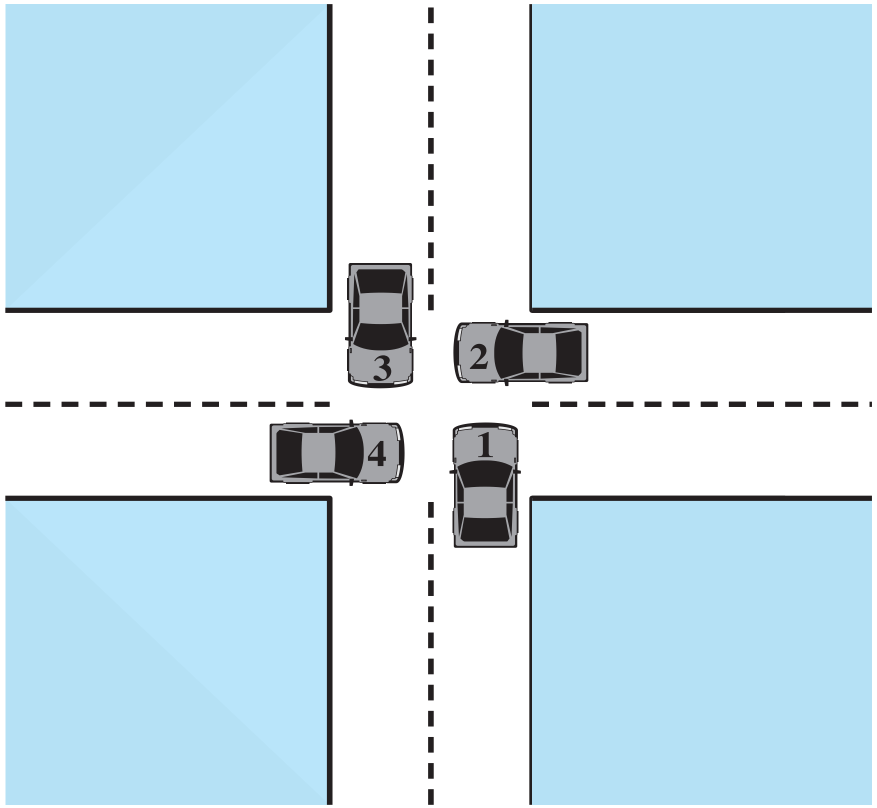
A set of processes is deadlocked if …

- Each process in the set waiting for an event
- That event can be caused only by another process

Deadlock Example



(a) Deadlock possible



(b) Deadlock

Deadlock Examples

Deadlock Approaches

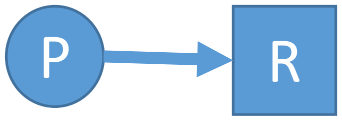
There is no single effective strategy for all types of deadlocks

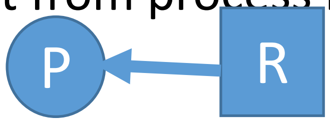
- Deadlock Prevention
  - Disallow one of the three conditions for deadlock occurrence, or
  - Prevent circular wait condition
- Deadlock Avoidance
  - Do not grant a resource request if this allocation might lead to deadlock
- Deadlock Detection
  - Grant resources when possible
  - Periodically check for the presence of deadlock
  - Take action to recover

Resource Allocation Graph (RAG)

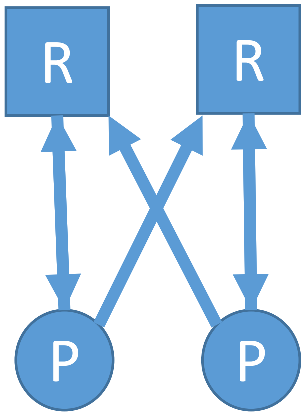
- Directed graph used to depict a state of the system resources and processes
- Nodes: Processes 

, Resources 

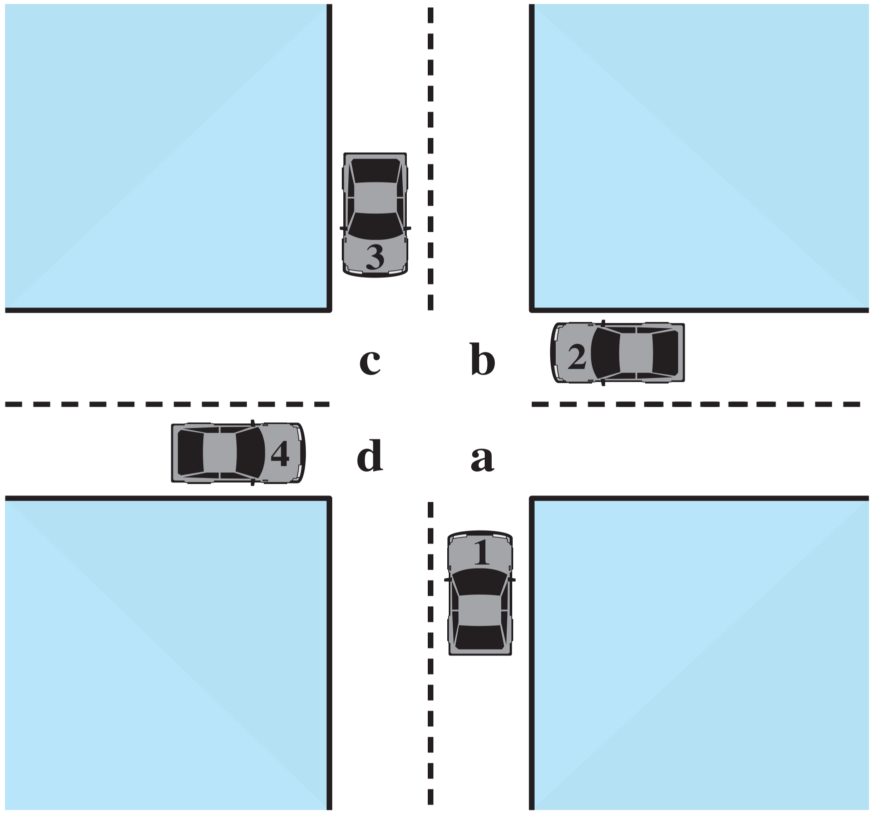
- Edges:
  - Request edge: 

    - denotes a request from process P for resource R
  - Assignment edge: 

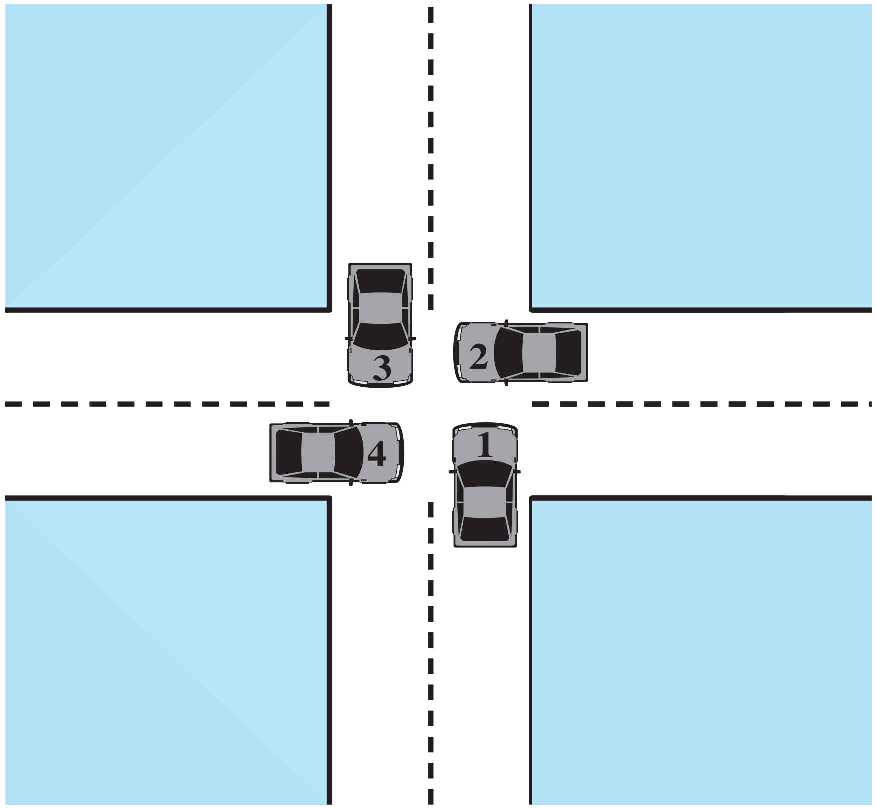
    - denotes process P holding (using) resource R
- Cycles in an RAG represent potential deadlocks.



Construct the RAG for case (b)



(a) Deadlock possible



(b) Deadlock

Conditions for Deadlock

- Mutual Exclusion
  - Only one process may use a resource at a time
  - Processes that request a resource being used are forced to block
- Hold-and-Wait
  - A process may attempt to acquire more than one resource.
  - A process may hold allocated resources while awaiting assignment of other resources.
- No Pre-emption (no interference from the system)
  - No resource can be forcibly removed from a process holding it
- Circular Wait
  - There is a set of processes and resources such as there is a cycle in the resource allocation graph that cannot be broken

How to Deal with Deadlocks?

1. **Ignore** the problem, maybe it will go away
   - (UNIX and Windows model)
2. **Prevention**, by structurally negating one of the four required conditions
3. **Detection and recovery**. Let deadlocks occur, detect them, and take action
4. **Dynamic avoidance** by careful resource allocation
   - System is given advance information about what resources the process needs. The process is not executed until it can be done safely

Deadlock Prevention Strategy

Idea: Prevent one of the four conditions from occurring

- Mutual Exclusion
- Hold and Wait
- No Preemption
- Circular Wait

Indirect method

Direct method