Monitors

- A monitor is a construct that encapsulates variables and access methods
- Only one thread at a time is allowed to be active inside the monitor. This mutual exclusion is implicit in monitors
- Each monitor object has a lock (binary semaphore). The compiler generates code to acquire the lock at the beginning of each method and to release the lock at the end

Disadvantages of Semaphores

- Semaphores are prone to mistakes
- Critical sections may spread all over the program
- It is difficult to distinguish mutual exclusion and condition synchronization

Condition Variables

- Condition variables are used for condition synchronization in monitors. They provide two operations: wait and signal
- A wait on a condition variable releases the lock and blocks on the queue for that condition variable
- Each signal on a condition variable awakens one thread on the queue for that condition variable
Condition Variables

- Condition variables have no values and they differ with semaphores in two ways.
- A signal on an empty condition variable has no effect whereas a V always increments the value of a semaphore.
- A wait on a condition variable always blocks whereas a P always decrements the value of a semaphore and blocks only when that value is non-positive.

Signaling Disciplines

- **Signal and exit**: the thread that signals on a condition variable need to leave the monitor immediately. The thread awakened continues to execute inside the monitor.
- **Signal and continue**: the thread that signals on a condition variable continues to execute. The thread awakened resumes its execution after the signaler leaves the monitor.

Signal and Exit

- After signaling on a condition variable, the signaler must exit the monitor immediately by executing a return statement.
- The signalled thread is given priority to proceed inside the monitor over those threads waiting to enter the monitor.
- It is guaranteed that the signalled thread finds the condition still true when it resumes execution inside the monitor.

Bounded Buffers

```java
class BoundedBuffer extends MyObject {
    private int size = 0;
    private double[] buf = null;
    private int front = 0, rear = 0, count = 0;
    private ConditionVariable notFull = null;
    private ConditionVariable notEmpty = null;
    public BoundedBuffer(int size) {
        this.size = size;
        buf = new double[size];
        notFull = new ConditionVariable();
        notEmpty = new ConditionVariable();
    }
}
```
Parallel Programming

**Bounded Buffers**

```java
public synchronized void deposit(double data) {
    if (count == size) wait(notFull);
    buf[rear] = data;
    rear = (rear + 1) % size; count++;
    if (count == 1) notify(notEmpty);
}
public synchronized double fetch() {
    double result;
    if (count == 0) wait(notEmpty);
    result = buf[front];
    front = (front + 1) % size; count--;
    if (count == size - 1) notify(notFull);
    return result;
}
```

**Dining Philosophers**

```java
class DiningServer extends MyObject {
    private boolean checkStarving = false;
    private int numPhils = 0;
    private int[] state = null;
    private ConditionVariable[] self = null;
    private static final int THINKING = 0, HUNGRY = 1, STARVING = 2, EATING = 3;
    public DiningServer(int numPhils, boolean checkStarving) {
        this.numPhils = numPhils;
        this.checkStarving = checkStarving;
        state = new int[numPhils];
        for (int i = 0; i < numPhils; i++) state[i] = THINKING;
        self = new ConditionVariable[numPhils];
        for (int i = 0; i < numPhils; i++) self[i] = new ConditionVariable();
    }
}
```

```java
public void takeForks(int i) {
    hungryAndGetForks(i);
}
public void putForks(int i) {
    finishedEating(i);
    checkForkDown(left(i)); checkForkDown(right(i));
}
private final int left(int i) {
    return (numPhils + i - 1) % numPhils;
}
private final int right(int i) {
    return (i + 1) % numPhils;
}
```

```java
private void seeIfStarving(int k) {
    if (state[k] == HUNGRY && state[left(k)] != STARVING &&
        state[right(k)] != STARVING) {
        state[k] = STARVING;
    } else if (checkStarving)
        seeIfStarving(k);
}
```

```java
private void test(int k, boolean checkStarving) {
    if (state[left(k)] == EATING && state[left(k)] != STARVING &&
        state[k] == HUNGRY) {
        state[k] = STARVING && state[right(k)] != EATING;
        state[k] = EATING;
    } else if (checkStarving)
        seeIfStarving(k);
}
```
### Dining Philosophers

```java
private synchronized void hungryAndGetForks(int i) {
    state[i] = HUNGRY;
    test(i, false);
    if (state[i] != EATING) wait(self[i]);
}

private synchronized void finishedEating(int i) {
    state[i] = THINKING;
}

private synchronized void checkForkDown(int i) {
    test(i, checkStarving);
    if (state[i] == EATING) notify(self[i]);
}
```

### Readers and Writers

```java
class Database extends MyObject {
    private int numReaders = 0;
    private boolean isWriting = false;
    private ConditionVariable OKtoRead = new ConditionVariable();
    private ConditionVariable OKtoWrite = new ConditionVariable();

    public synchronized void startRead(int i) {
        if (isWriting) wait(OKtoRead);
        else if (!empty(OKtoWrite)) {
            wait(OKtoRead);
            numReaders++;
            notify(OKtoRead);
        }
    }

    public synchronized void endRead(int i) {
        numReaders--;
        if (numReaders == 0) notify(OKtoWrite);
    }

    public synchronized void startWrite(int i) {
        if (numReaders != 0 || isWriting) wait(OKtoWrite);
        isWriting = true;
    }

    public synchronized void endWrite(int i) {
        isWriting = false;
        if (!empty(OKtoRead)) notify(OKtoRead);
        else notify(OKtoWrite);
    }
```

### Signal and Continue

- After signaling on a condition variable, the signaler does not require to exit the monitor immediately.
- The signaled thread is not given priority to proceed inside the monitor over those threads waiting to enter the monitor.
- It is not guaranteed that the signaled thread finds the condition still true when it resumes execution inside the monitor.
**Condition Variables in Java**

- Each *object* (and hence each *monitor*) has one (anonymous) *condition variable*
- Inside a monitor, the functions `wait()`, `notify()`, and `notifyAll()` automatically refer to the condition variable of the monitor
- `Notify()` moves to the ready queue one of the waiting threads
- `NotifyAll()` moves to the ready queue all of the waiting threads

**Bounded Buffers**

```java
public synchronized void deposit(double value) {
    if (count == numSlots)
        try { wait(); } catch (InterruptedException e) {}
    buffer[putIn] = value; putIn = (putIn+1) % numSlots; count++;
    if (count == 1) notify();
}
public synchronized double fetch() {
    double value;
    if (count == 0)
        try { wait(); } catch (InterruptedException e) {}
    value = buffer[takeOut];
    takeOut = (takeOut+1) % numSlots; count--;
    if (count == numSlots-1) notify();
    return value;
}
```

**Dining Philosophers**

```java
public synchronized void takeForks(int i) {
    state[i] = HUNGRY;
    test(i, false);
    while (state[i] != EATING)
        try { wait(); } catch (InterruptedException e) {}
}

public synchronized void putForks(int i) {
    state[i] = THINKING;
    test(left(i), checkStarving);
    test(right(i), checkStarving);
    notifyAll();
}
```

```java
private void seeIfStarving(int k) {
    if (state[k] == HUNGRY && state[left(k)] != STARVING &&
        state[right(k)] != STARVING) {
        state[k] = STARVING;
        System.out.println("philosopher " + k + " is STARVING");
    }
}
```
Parallel Programming

Readers and Writers

class Database extends MyObject {
    private int numReaders = 0;
    private int numWriters = 0;
    private int numWaitingReaders = 0;
    private int numWaitingWriters = 0;
    private boolean okToWrite = true;
    private long startWaitingReadersTime = 0;
    public Database() { super("rwDB"); }
    // methods
}

Readers and Writers

public synchronized void startRead(int i) {
    long readerArrivalTime = 0;
    if (numWaitingWriters > 0 || numWriters > 0) {
        numWaitingReaders++; okToWrite = false;
        while (!okToWrite)
            try { wait(); } catch (InterruptedException e) { }
        numWaitingReaders--;
    }
    numReaders++;
}

public synchronized void endRead(int i) {
    numReaders--;
    okToWrite = numReaders == 0;
    if (okToWrite) notifyAll();
}

Readers and Writers

public synchronized void startWrite(int i) {
    if (numReaders > 0 || numWriters > 0) {
        numWaitingWriters++; okToWrite = false;
        while (!okToWrite)
            try { wait(); } catch (InterruptedException e) { }
        numWaitingWriters--;
    }
    okToWrite = false; numWriters++;
}

public synchronized void endWrite(int i) {
    numWriters--; okToWrite = numWaitingReaders == 0;
    startWaitingReadersTime = age();
    notifyAll();
}

Semaphores

public abstract class Semaphore {
    protected int value = 0;
    protected Semaphore() { value = 0; }
    protected Semaphore(int initial) { value = initial; }
    public synchronized void P() {
        value--;
        if (value < 0)
            try { wait(); } catch (InterruptedException e) { }
    }
    public synchronized void V() {
        value++;
        if (value <= 0) notify();
    }
}
Parallel Programming

Binary Semaphores

```java
public final class BinarySemaphore extends Semaphore {
    public BinarySemaphore() { super(); }
    public BinarySemaphore(int initial) {
        super (initial != 0 ? 1:0);
    }
    public BinarySemaphore(boolean initial) {super(initial ? 1:0);}
    public final synchronized void V() { super.V();
        if (value > 1) value = 1;
    }
}
```

Counting Semaphores

```java
public final class CountingSemaphore extends Semaphore {
    public CountingSemaphore() { super(); }
    public CountingSemaphore(int initial) { super(initial); }
}
```

Locks

```java
class Lock {
    private Thread owner = null;
    public Lock() { super(); }
    public synchronized void lock() {
        Thread who = Thread.currentThread();
        if (owner == who) return;
        else if (owner == null) {
            owner = who; return;
        } else /* owner != who && owner != null */ {
            while (owner != null) {
                try { wait(); } catch (InterruptedException e) { }
            }
            owner = who; return;
        }
    }
}
```

Locks

```java
public synchronized void unlock() throws IllegalMonitorStateException {
    Thread who = Thread.currentThread();
    if (owner == null) return;
    else if (owner != who) {
        throw new IllegalMonitorStateException();
    } else /* owner == who */ {
        owner = null;
        notify();
    }
```
Notification Objects

Shared object:
    Object obj = new object();
In one thread:
    synchronized (obj) {
        if (!condition)
            try { obj.wait(); } catch (InterruptedException e) {} 
        ... 
    }
In another thread:
    synchronized (obj) {
        if (condition) obj.notify(); 
    }

Bounded Buffers

private Object conveyD = null, conveyF = null;
public void deposit(double value) {
    synchronized (conveyD) {
        spaces--; 
        if (spaces < 0)
            try { conveyD.wait(); } catch (InterruptedException e) {} 
        buffer[putIn] = value; 
        putIn = (putIn + 1) % numSlots; 
    }
    synchronized (conveyF) {
        elements++; 
        if (elements <= 0) conveyF.notify(); 
    }
}

public double fetch() {
    double value; 
    synchronized (conveyF) {
        elements--; 
        if (elements < 0)
            try { conveyF.wait(); } catch (InterruptedException e) {} 
        value = buffer[takeOut]; 
        takeOut = (takeOut + 1) % numSlots; 
    }
    synchronized (conveyD) {
        spaces++; 
        if (spaces <= 0) conveyD.notify(); 
    }
    return value; 
}
Dining Philosophers

```java
class DiningServer extends MyObject {
    private boolean checkStarving = false;
    private int numPhils = 0;
    private int[] state = null;
    private Object[] convey = null;
    private static final int
        THINKING = 0, HUNGRY = 1, STARVING = 2, EATING = 3;
    public DiningServer(int numPhils, boolean checkStarving) {
        super("DiningServer");
        this.numPhils = numPhils;
        this.checkStarving = checkStarving;
        state = new int[numPhils];
        for (int i = 0; i < numPhils; i++) state[i] = THINKING;
        convey = new Object[numPhils];
        for (int i = 0; i < numPhils; i++) convey[i] = new Object();
    }
}
```

Dining Philosophers

```java
public void takeForks(int i) {
    synchronized (convey[i]) {
        if (hungryAndGetForks(i)) return;
        else try {convey[i].wait();} catch (InterruptedException e) {} } }
}
```

Dining Philosophers

```java
private synchronized boolean hungryAndGetForks(int i) {
    state[i] = HUNGRY;
    test(i, false);
    return state[i] == EATING;
}
```

Dining Philosophers

```java
private void forkAvailable(int i) {
    synchronized (convey[i]) {
        convey[i].notify();
    }
}
```
Readers and Writers

class Database extends MyObject {
    private int numReaders = 0;
    private boolean isWriting = false;
    private Vector waitingReaders = new Vector();
    private Vector waiting Writers = new Vector();
    public Database() { super("rwDB"); }
    // methods
}

Readers and Writers

public void startRead(int i) {
    Object convey = new Object();
    synchronized (convey) {
        if (cannotReadNow(convey))
            try { convey.wait(); } catch (InterruptedException e) {} 
    }
    private synchronized boolean cannotReadNow(Object convey) {
        boolean status;
        if (isWriting || numReaders > 0) {
            waitingReaders.addElement(convey); status = true;
        } else { isWriting = true; status = false; }
        return status;
    }
}

Readers and Writers

public void startWrite(int i) {
    Object convey = new Object();
    synchronized (convey) {
        if (cannotWriteNow(convey))
            try { convey.wait(); } catch (InterruptedException e) {} 
    }
    private synchronized boolean cannotWriteNow(Object convey) {
        boolean status;
        if (isWriting || numReaders > 0) {
            waiting Writers.addElement(convey); status = true;
        } else { isWriting = true; status = false; }
        return status;
    }
}

Readers and Writers

public synchronized void endRead(int i) {
    numReaders--;
    if (numReaders == 0 && waitingWriters.size() > 0) {
        synchronized (waitingWriters.elementAt(0)) {
            waitingWriters.elementAt(0).notify();
        }
        waitingWriters.removeElementAt(0);
        isWriting = true;
    }
}

Readers and Writers
Readers and Writers

```java
public synchronized void endWrite(int i) {
    isWriting = false;
    if (waitingReaders.size() > 0) {
        while (waitingReaders.size() > 0) {
            synchronized (waitingReaders.elementAt(0)) {
                waitingReaders.elementAt(0).notify();
            }
            waitingReaders.removeElementAt(0); numReaders++;
        }
    } else if (waitingWriters.size() > 0) {
        synchronized (waitingWriters.elementAt(0)) {
            waitingWriters.elementAt(0).notify();
        }
        waitingWriters.removeElementAt(0); isWriting = true;
    }
}
```

Implementing Monitors with Semaphores

- Each monitor includes a binary semaphore that permits only one thread at a time to be active in the monitor
  
  semaphore mutex = 1;
  ...
  service-method, { 
    P(mutex);
    ...
    V(mutex);
  }

Named Condition Variables

```java
public class ConditionVariable {
    public void waitCV(Object monitor) {} 
    public void notifyCV(Object monitor) {} 
    public boolean emptyCV(Object monitor) { return true; } 
}

// MyObject
protected static void wait(ConditionVariable cv) { cv.waitCV(); }
protected static void notify(ConditionVariable cv) { cv.notifyCV(); }
protected static boolean empty(ConditionVariable cv) {
    return cv.emptyCV();
}
```

Implementing Monitors with Semaphores

- Each condition variable cond, is implemented with a semaphore SEM, and an integer counter COUNT, both initially zero
- Signal-and-continue
  
  wait():  
  COUNT++;  
  V(mutex);  
  if (COUNT > 0) {
    P(SEM);  
    V(SEM);  
    P(mutex);
  }
  signal():
Implementing Monitors with Semaphores

- Signal-and-exit
  - wait():
  - COUNT_i++; 
  - V(mutex);
  - P(SEM_i);

- signal():
  - if (COUNT_i > 0) {
  - COUNT_i--; 
  - V(SEM_i);
  - } else V(mutex);