UTA

Today's Agenda

- HW 2 Out, Due on Oct. 15
- Presentation Signup
- Quick Review
- Continue on Monitor

UTA

Monitor

- Introduction
- Monitors in Java
- Signaling Disciplines
- Implementing Monitors
- Monitor-based Solutions

UTA

Motivation

Semaphores were defined before the concepts of data encapsulation and information hiding were introduced.

In addition, a semaphore can be used for both mutual exclusion and conditional synchronization, which makes it difficult to distinguish them.

UTA

What is it?

Monitor is a high-level synchronization construct that supports data encapsulation and information hiding.

It encapsulates shared data, operations on the data, and the required synchronization for accessing the data.

Important: Threads are active, while monitors are passive.

UTA

OO Definition

A monitor class N with m data members, n condition variables, and k methods.

```
monitor N {
          private Data d1, d2, ..., dm;                       // data members
          private ConditionVariable v1, v2, ..., vn ;         // condition vars

          public N () { ... };                                // constructor
          public/private void M1 (...) { ... }                // access methods
          public/private void M2 (...) { ... }
   ...
          public/private void Mk (...) { ...}
}
```


UTA

Mutual Exclusion

Mutual exclusion is provided automatically by monitor implementation.

If a thread calls a monitor method, but another thread is already executing inside the monitor, the calling thread waits in an entry queue.

UTA

Conditional Synchronization

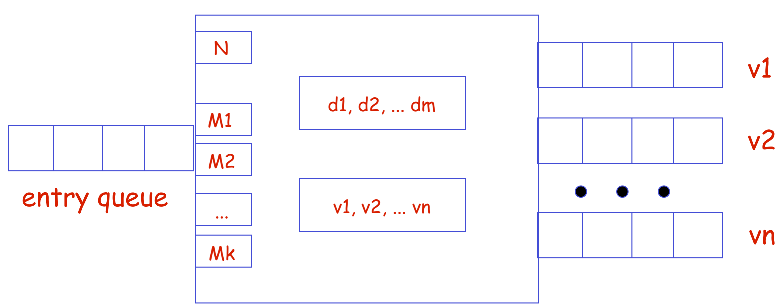
Conditional synchronization is programmed using condition variables.

A condition variable denotes a queue of threads that are waiting for a specific condition to become true.

A monitor has one entry queue and one queue associated with each condition variable.

UTA

Graphic View



UTA

Wait Operation

A thread that is executing inside a monitor method blocks itself on condition variable v by executing v.wait ().

Execution of operation wait releases mutual exclusion and blocks the thread on the rear of the queue for v.

Important: Threads blocked on a condition variable are outside the monitor.

UTA

Signal Operation

A thread blocked on condition variable v is awakened by the execution of v.signal ().

If there are no threads blocked on v, this signal operation has no effect; otherwise, it awakens the thread at the front of the queue for v.

UTA

Signaling Discipline

There are different types of signaling disciplines. For now, we will assume the "signal and continue", or SC, discipline.

The thread executes a signal operation to awaken a waiting thread, the thread continues executing in the monitor and the awakened thread is moved to the entry queue.

UTA

Empty & Length Operations

The execution of v.empty () returns true if the queue for v is empty, and false otherwise.

The execution of v.length () returns the length of the queue for v.

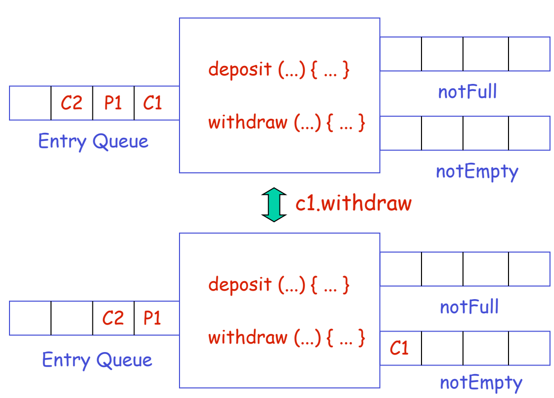
UTA

An Example

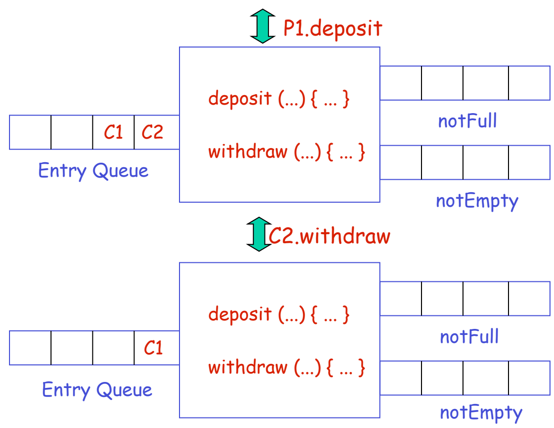
```
monitor BoundedBuffer {
    private int fullSlots = 0;
    private int capacity = 0;
    private int[] buffer = null;
    private int in = 0, out = 0;
    private ConditionVariable notFull = new ConditionVariable ();
    private ConditionVariable notEmpty = new ConditionVariable ();

    public BoundedBuffer (int capacity) {
        this.capacity = capacity;
        buffer = new int [ capacity ];
    }
    public void deposit (int value) {
        while (fullSlots == capacity) {
            notFull.wait ();
        }
        buffer[in] = value;
        in = (in + 1) % capacity;
        ++ fullSlots;
        notEmpty.signal ();
    }
    public int withdraw () {
        int value;
        while (fullSlotes == 0) {
            notEmpty.wait ();
        }
        value = buffer[out];
        out = (out + 1) % capacity;
        -- fullSlots
        notFull.signal();
        return value;
    }
}
```


UTA

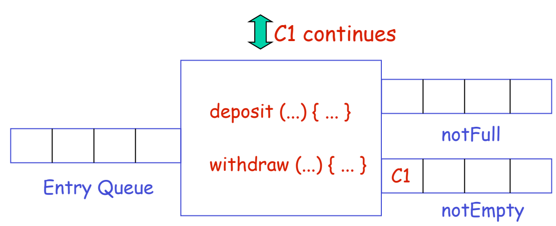
An Example (cont'd)



An Example (cont'd)



An Example (cont'd)



UTA

An Example (cont'd)

What if we change while to if in methods deposit and withdraw?

UTA

Monitor

- Introduction
- Monitors in Java
- Signaling Disciplines
- Implementing Monitors
- Monitor based Solutions

UTA

Java Monitor (1)

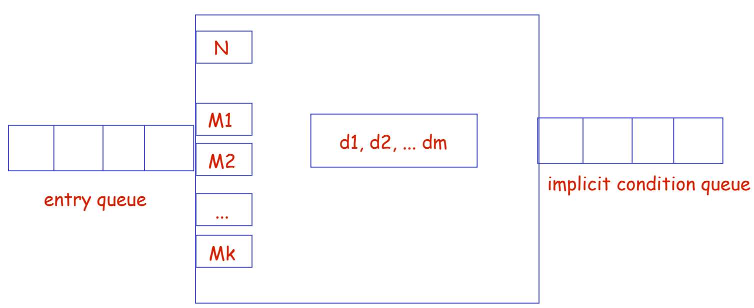
Java built-in monitor is different from the monitors we have defined.

Mutual exclusion is only enforced for synchronized methods.

There are no explicit condition variables in Java. Instead, each object is associated with only one implicit condition variable.

UTA

Java Monitor (2)



UTA

Java Monitor (3)

- Each Java object is implicitly associated with a lock.
- A thread must hold an object's lock before it can execute a wait, notify, or notifyAll operations.
- When a thread executes wait, it releases the object's lock and waits in the implicit condition queue.
- notify operations do not guarantee FCFS.
- A notified thread must reacquire the lock before it resumes execution.

UTA

notify vs. notifyAll

The fact that Java monitors have only one condition variable has important implications.

As threads waiting on the same implicit condition variable may be waiting for different conditions to become true, notify operations must be handled carefully.

notify vs. notifyAll (cont'd)

```
public final class BinarySemaphore extends Semaphore {
    public BinarySemaphore (int initialPermits) {
        super(initialPermits);
        if (initialPermits != 0 || initialPermits != 1) {
            throw new IllegalArgumentException("initial value must be 0 or 1.");
        }
    }
    synchronized public void P () {
        while (permits == 0) {
            try { wait (); } catch (InterruptedException ex) {};
        }
        permits = 0;
        notifyAll ();
    }
    synchronized public void V () {
        while (permits == 1) {
            try { wait (); } catch (InterruptedException ex) {};
        }
        permits = 1;
        notifyAll ();
    }
}
```


notify vs. notifyAll (cont'd)

What happens if we change notifyAll to notify?

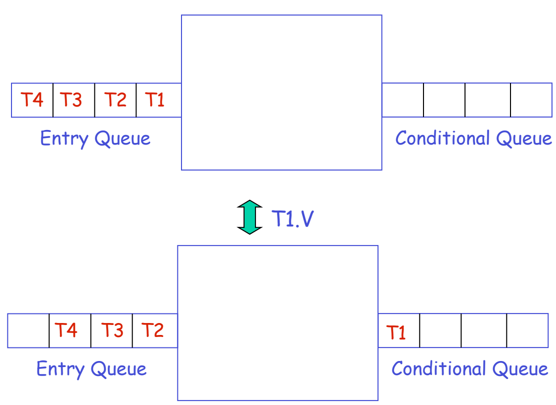
```
public final class BinarySemaphore extends Semaphore {
    public BinarySemaphore (int initialPermits) {
        super(initialPermits);
        if (initialPermits != 0 && initialPermits != 1) {
            throw new IllegalArgumentException("initial value must be 0 or 1.");
        }
    }
    synchronized public void P () {
        while (permits == 0) {
            try { wait (); } catch (InterruptedException ex) {};
        }
        permits = 0;
        notify ();
    }
    synchronized public void V () {
        while (permits == 1) {
            try { wait (); } catch (InterruptedException ex) {};
        }
        permits = 1;
        notify ();
    }
}
```

notify vs notifyAll (cont'd)

Let s be a binary semaphore initialized as 1.

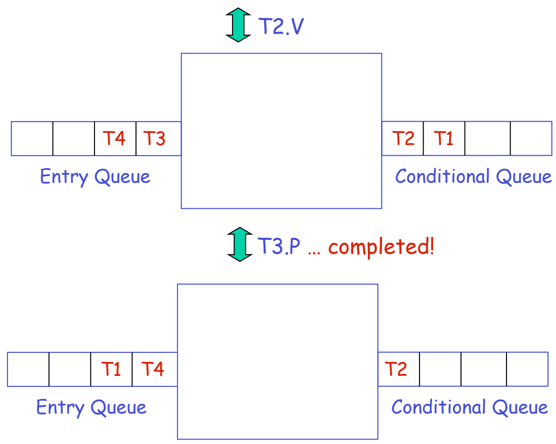
T1	T2	T3	T4
s.V	s.V	s.P	s.P

notify vs. notifyAll (cont'd)



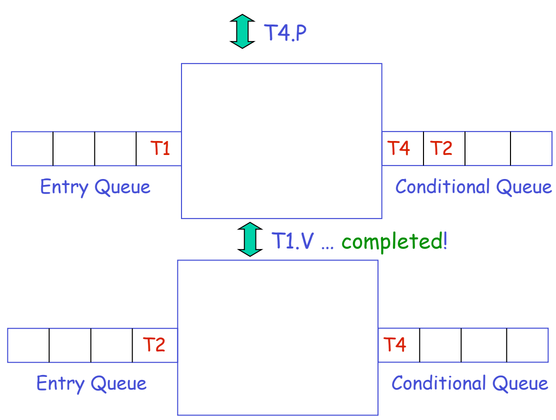
UTA

notify vs. notifyAll (cont'd)

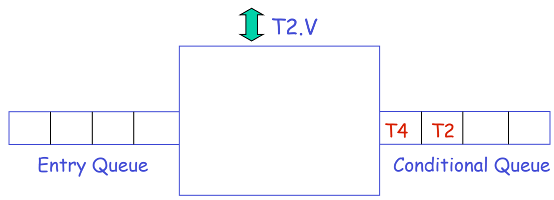


UTA

notify vs. notifyAll (cont'd)



notify vs. notifyAll (cont'd)



notify vs. notifyAll (cont'd)

Using notifyAll instead of notify unless the following requirements are met:

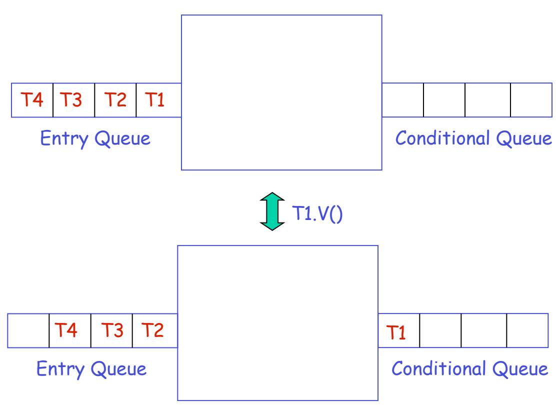
- All waiting threads are waiting on the exact same condition.
- Each notification should enable exactly one thread to continue.

while vs if

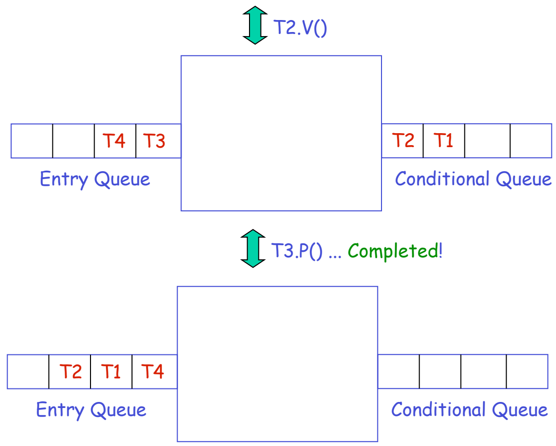
What if we change while to if?

```
public final class BinarySemaphore extends Semaphore {
    public BinarySemaphore (int initialPermits) {
        super(initialPermits);
        if (initialPermits != 0 || initialPermits != 1) {
            throw new IllegalArgumentException("initial value must be 0 or 1.");
        }
    }
    synchronized public void P () {
        if (permits == 0) {
            try { wait (); } catch (InterruptedException ex) {};
        }
        permits = 0;
        notifyAll ();
    }
    synchronized public void V () {
        if (permits == 1) {
            try { wait (); } catch (InterruptedException ex) {};
        }
        permits = 1;
        notifyAll ();
    }
}
```

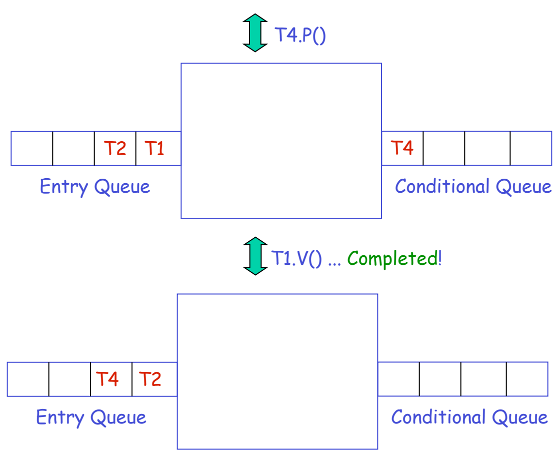

while vs. if (cont'd)



while vs. if (cont'd)

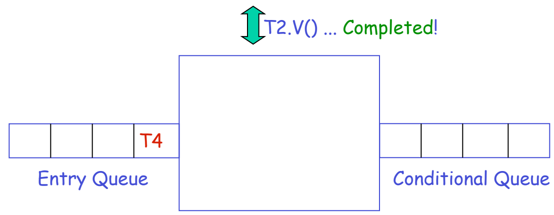


while vs. if (cont'd)



UTA

while vs. if (cont'd)



UTA

while vs. if (cont'd)

In general, with the SC discipline, use while instead of if, unless the following requirement is met:

- After a thread T is awakened, no other threads can change the condition T was waiting on before T re-enter the monitor.

UTA

Revisit the BB example

What if we change while to if and/or notify to notifyAll?

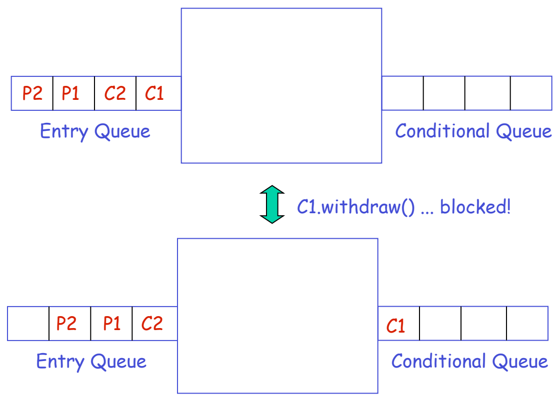
UTA

Revisit BB (cont'd)

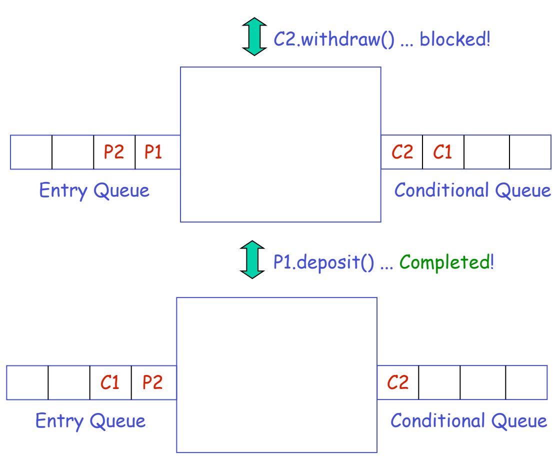
```
final class BoundedBuffer {
    private int fullSlots = 0;
    private int capacity = 0;
    private int[] buffer = null;
    private int in = 0, out = 0;

    public BoundedBuffer (int capacity) {
        this.capacity = capacity;
        buffer = new int [ capacity ];
    }
    public synchronized void deposit (int value) {
        while (fullSlots == capacity) {
            try { wait(); } catch (InterruptedException ex) {}
        }
        buffer[in] = value;
        in = (in + 1) % capacity;
        if (fullSlots ++ == 0) {
            notify ();
    }
    public synchronized void withdraw () {
        int value;
        while (fullSlotes == 0) {
            try { wait (); } catch (InterruptedException ex) {}
        }
        value = buffer[out];
        out = (out + 1) % capacity;
        if (fullSlots -- == capacity) {
            notify ()
        }
        return value;
    }
}
```

Revisit BB

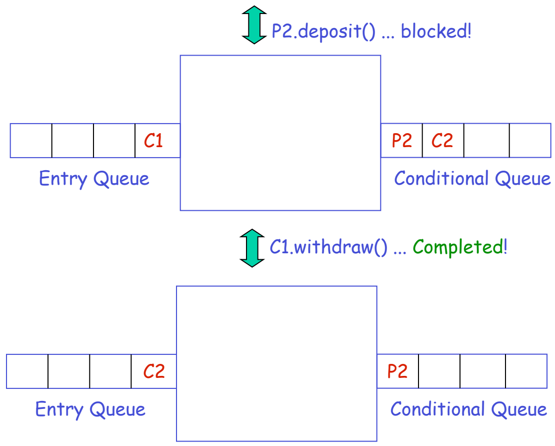
Assume the buffer has one slot and is empty.



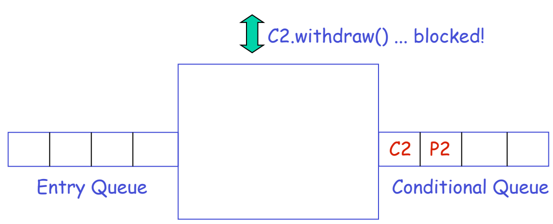
Revisit BB (cont'd)



Revisit BB (cont'd)



Revisit BB (cont'd)



UTA

Revisit MutexLock implementation

Why do we use notify instead of notifyAll, and if instead of while?

UTA

MutexLock

```
public final class MutexLock {
        private Thread owner = null;
        private int waiting = 0;
        public int count = 0;
        public boolean free = true;

        public synchronized void lock () {
              if (free) {
                    count = 1; free = false; owner = Thread.currentThread ();
              }
              else if (owner == Thread.currentThread()) { ++ count;  }
              else {
                    ++ waiting;
                    try { wait(); } catch (InterruptedException ex) {}
                    count = 1; owner = Thread.currentThread ()
              }
        }

        public synchronized void unlock () {
              if (owner != null) {
                    if (owner == Thread.currentThread ()) {
                          -- count;
                          if (count == 0) {
                                owner = null;
                                if (waiting > 0) {
                                      -- waiting;
                                      notify();
                                }
                                else { free = true; return; }
                          }
                          else return;
                    }
              }
              throw new OwnerException ();
        }
}
```

UTA

MutexLock (cont'd)

```
public final class MutexLock {
        private Thread owner = null;
        private int waiting = 0;
        public int count = 0;
        public boolean free = true;

        public synchronized void lock () {
            if (free) {
                count = 1; free = false; owner = Thread.currentThread ();
            }
            else if (owner == Thread.currentThread()) { ++ count;  }
            else {
                ++ waiting;
                try { wait(); } catch (InterruptedException ex) {}
                free = false;
                count = 1; owner = Thread.currentThread ()
            }
        }
        public synchronized void unlock () {
            if (owner != null) {
                if (owner == Thread.currentThread ()) {
                    -- count;
                    if (count == 0) {
                        owner = null;
                        if (waiting > 0) {
                            -- waiting;
                            free = true;
                            notify();
                        }
                        else { free = true; return; }
                    }
                    else return;
                }
            }
            throw new OwnerException ();
        }
}
```


UTA

synchronized block

Consider the following code segment:

```
synchronized (o) {
   /* block of code */
}
```

synchronized block (cont'd)

```
public synchronized void foo (o) {
   /* block of code */
}
```

⇕

```
public void foo (o) {
   synchronized (this) {
      /* block of code */
   }
}
```


synchronized block (cont'd)

```
public final class BinarySemaphore {
    int vPermits = 0, pPermits = 0;
    Object allowP = null, allowV = null;

    public BinarySemaphore (int initialPermits) {
        if (initialPermits != 0 || initialPermits != 1) {
            throw new IllegalArgumentException("initial value must be 0 or 1.");
        }
        pPermits = initialPermits;
        vPermits = 1 – vPermits;
        allowP = new Object ();
        allowV = new Object ();
    }
    public void P () {
        synchronized (allowP) {
            -- pPermits;
            if (pPermits < 0) {
                try { allowP.wait (); } catch (InterruptedException ex) {};
            }
        }
        synchronized (allowV) {
            ++ vPermits;
            if(vPermits <= 0) {
                allowV.notify ();
            }
        }
    }
    public void V () {
        synchronized (allowV) {
            -- vPermits;
            if (vPermits < 0) {
                try { allowV.wait (); } catch (InterruptedException ex) {};
            }
        }
        synchronized (allowP) {
            ++ pPermits;
            if(pPermits <= 0) {
                allowP.notify ();
            }
        }
    }
}
```

UTA

Monitor

- Introduction
- Monitors in Java
- Signaling Disciplines
- Implementing Monitors
- Monitor based Solutions

UTA

Signaling Discipline

A thread blocked on a condition variable c is awakened by the execution of c.signal ().

If there are no threads blocked on c, the signal operation has no effect. Otherwise, the signal operation awakens the thread in the front of the queue for c.

What happens next depends on the signaling disciplines.

UTA

Signal-and-Continue (SC)

When a thread executes c.signal ().

- If there are no threads waiting on c, this operation has no effects.
- Otherwise, the signaling thread awakens one thread waiting on c, and continues execution inside monitor.
- The awakened thread does not resume execution immediately. Instead, it is moved from the conditional queue to the entry queue.

UTA

Signal-and-Continue (SC) (cont'd)

When a thread executes c.wait:

- The thread releases mutual exclusion to allow a thread on the entry queue to enter the monitor.
- The thread then blocks itself on the condition queue for c.

UTA

Signal-and-Continue (SC) (cont'd)

When a thread exits a monitor:

❑ The thread releases mutual exclusion to allow a thread on the entry queue to enter the monitor.

UTA

Signal-and-Urgent-Wait (SU)

When a thread executes c.signal ().

❑ If there are no threads waiting on c, this operation has no effects.

❑ Otherwise, the signaling thread awakens one thread waiting on c, and blocks itself in a queue called the reentry queue.

❑ The awakened thread reenters the monitor immediately.

UTA

Signal-and-Urgent-Wait (SU) (cont'd)

When a thread executes c.wait ().

- If the reentry queue is not empty, the thread awakens a thread from the reentry queue before it blocks itself on the queue for c.
- Otherwise, the thread releases mutual exclusion (to allow a thread on the entry queue to enter the monitor) and then blocks itself on the queue for c.

UTA

Signal-and-Urgent-Wait (SU) (cont'd)

When a thread exits a monitor thread

- If the reentry queue is not empty, it awakens a thread from the reentry queue.
- Otherwise, it releases mutual exclusion (to allow a thread on the entry queue to enter the monitor).

UTA

Signal-and-Exit (SE)

SE is a special case of SU. When a thread executes an SE signal operation, it exits the monitor immediately.

Therefore, a signal statement is either the last statement of a method or it is followed immediately by a return statement.

The awakened thread is the next thread to enter the monitor.

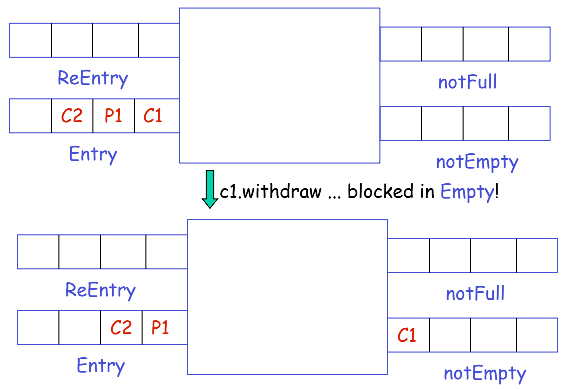
UTA

Signal-and-Exit (SE)

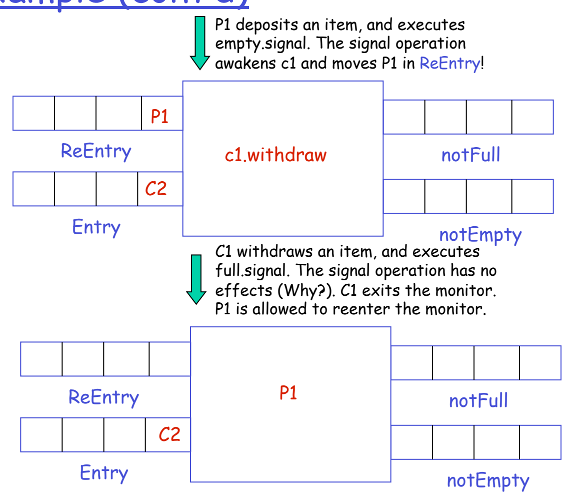
It has turned out that many signal statements do appear as the last statement in monitor methods.

Using SE semantics for these signal statements is more efficient. (Why?)

SU Example

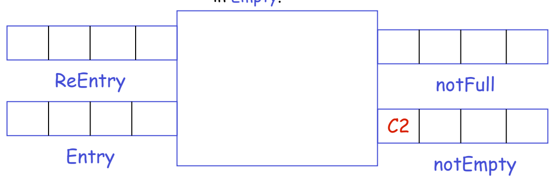


SU Example (cont'd)

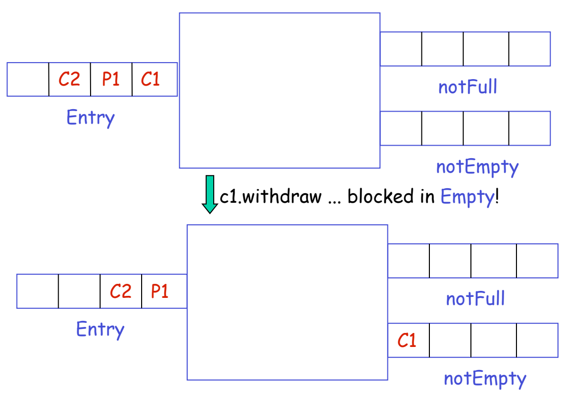


SU Example (cont'd)

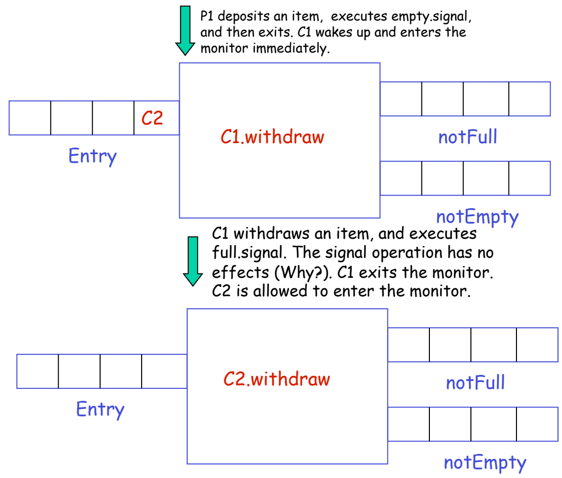
P1 has no more statements to execute. P1 exits the monitor. C2 is allowed to enter. C2 is then blocked in Empty.



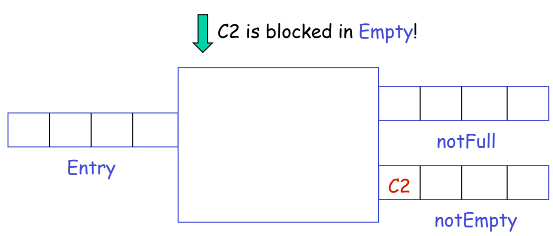
SE Example



SE Example (cont'd)



SE Example (cont'd)



SC vs. SU

What is the essential difference between SC and SU?

Implications

If a thread executes an SU signal to notify another thread that a certain condition is true, this condition remains true when the signaled thread reenters the monitor. (However, this does not hold for an SC signal. Why?)

Implications: A wait operation does not have to be placed in a while loop.

UTA

Example

```
monitor CountingSemaphore {

    private int permits;
    private ConditionVariable permiteAvail = new ConditionVariable ();

    public CoutingSemaphore (int initialPermits) {
        permits = initialPermits;
    }

    public void P () {
        if (permits == 0) {
            permitAvail.wait ();
        }
        -- permits;
    }

    public void V () {
        ++ permits
        permitAvail.signal ();
    }
}
```

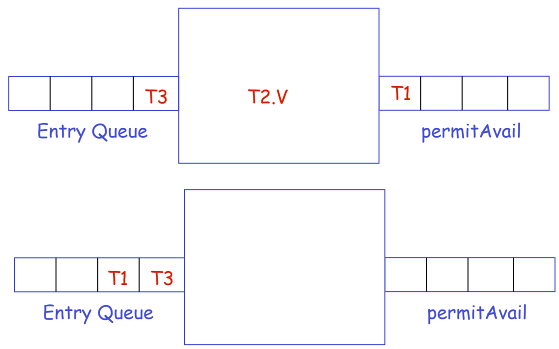

UTA

Example (cont'd)

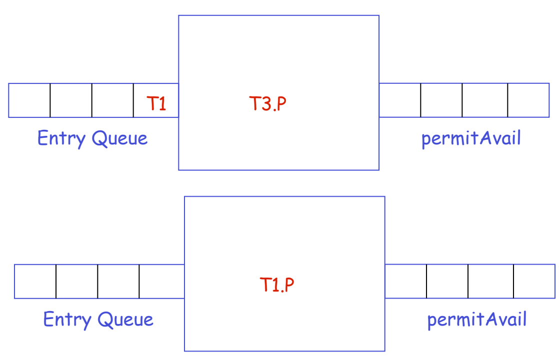
The simulation is incorrect if SC signals are used. Why?

Example (cont'd)

Assume that T1 is blocked on permitAvail in P, T2 has just executed ++permits in V, and T3 is waiting in the entry queue to execute a P operation.



Example (cont'd)



UTA

Example (cont'd)

What is the fundamental reason for this error?

The condition permits > 0 is true when T1 is awakened. However, T1 does not resume execution immediately.

T1 has to compete for mutual exclusion with other threads. When T1 acquires mutual exclusion and reenters the monitor, the condition permits > 0 is no longer true. (The condition was falsified by T2.)

UTA

Example (cont'd)

Can we fix this problem such that it still works with SC signal?

UTA

SC vs SU vs SE

With SC, a wait operation usually has to be placed in a while loop. This means that the relevant condition must be reevaluated after a wait operation.

With SU, an if statement can be used. However, the signaling thread will be blocked in the reentry queue, and needs to reenter the monitor before it exits. This represents an extra context switch.

With SE, both the cost of an extra evaluation and that of an extra context switch can be avoided. (What is the catch?)

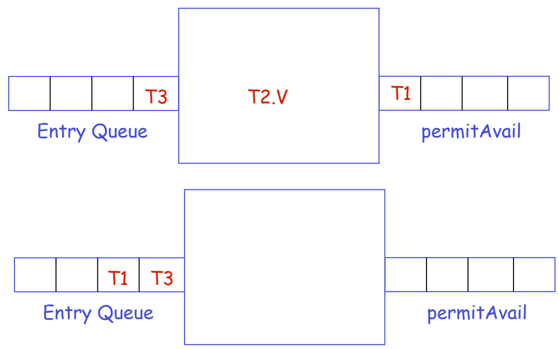
UTA

Revisit the example

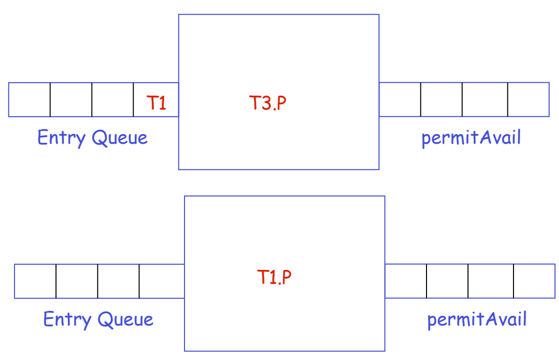
Is there any other problem if we change if to while, assuming SC discipline?

Revisit the example (cont'd)

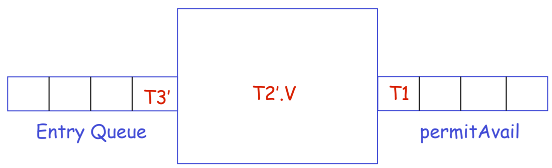
Assume that T1 is blocked on permitAvail in P, T2 has just executed ++permits in V, and T3 is waiting in the entry queue to execute a P operation.



Revisit the example (cont'd)

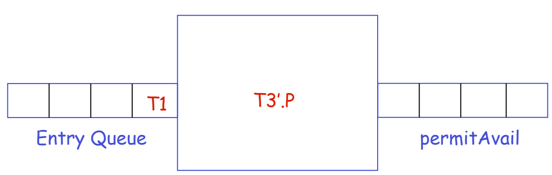


Revisit the example (cont'd)



Revisit the example (cont'd)

At this point, we assume that thread T2' arrives and starts to execute a V operation inside the monitor. Before T2' exits the monitor, thread T3' arrives and waits in the entry queue to execute a P operation. Similarly, T3' will complete its P operation, and T1 will be blocked again.



Alternative implementation

```
monitor class CountingSemaphore {

    private int permits;
    private ConditionVariable permiteAvail = new ConditionVariable ();

    public CoutingSemaphore (int initialPermits) {
        permits = initialPermits;
    }

    public void P () {
        -- permits;
        if (permits < 0) {
            permitAvail.wait ();
        }
    }

    public void V () {
        ++ permits
        permitAvail.signal ();
    }
}
```


Alternative Implementation (cont'd)

Is this implementation correct with SU discipline?
How about SC discipline?

UTA

Monitor

- Introduction
- Monitors in Java
- Signaling Disciplines
- Implementing Monitors
- Monitor based Solutions

UTA

SC Signaling

Each public monitor method is implemented as follows:

```
public ReturnType foo (...) {
   mutex.P ();
   /* body of foo */
   mutex.V ();
}
```

SC Signaling (cont'd)

```
final class ConditionVariable {
   private CountingSemaphore threadQue = new CoutingSemaphore (0);
   private int numWaitingThreads = 0;
   public void waitC () {
      numWaitingThreads ++;
      threadQue.VP (mutex); // { mutex.V (); threadQue.P () }
      mutex.P ();
   }
   public void signalC () {
      if (numWaitingThreads > 0) {
         numWaitingThreads --;
         threadQue.V ();
      }
   }
   public void signalCall () {
      while (numWaitingThreads > 0) {
         -- numWaitingThreads;
         threadQueue.V ();
      }
   }
   public boolean empty () { return numWaitingThreads == 0; }
   public int length () { return numWaitingThreads; }
}
```


SU Signaling

Each public monitor method is implemented as follows:

```
public ReturnType foo (...) {
   mutex.P ();
   /* body of foo */
   if (reentryCount > 0)
      reentry.V ();
   else mutex.V ();
}
```

UTA

SU Signaling (cont'd)

```
final class ConditionVariable {
   private CountingSemaphore threadQue = new CoutingSemaphore (0);
   private int numWaitingThreads = 0;
   public void waitC () {
      numWaitingThreads ++;
      if (reentryCount > 0) threadQue.VP (reentry); // { reentry.V(); threadQue.P() }
      else threadQue.VP (mutex); // { mutex.V(); threadQue.P() }
      -- numWaitingThreads;
   }
   public void signalC () {
      if (numWaitingThreads > 0) {
         ++ reentryCount;
         reentry.VP (threadQue); // { threadQue.V(); reentry.P() }
         -- reentryCount;
      }
   }
   public boolean empty () { return numWaitingThreads == 0; }
   public int length () { return numWaitingThreads; }
}
```


UTA

A monitor toolbox for Java

A monitor toolbox is a program unit that can be used to simulate the monitor construct.

The Java monitor toolbox consists of two classes, MonitorSC and MonitorSU.

UTA

Why a monitor toolbox?

A simulated monitor can be implemented in languages that do not support monitors directly.

Different versions of monitors can be created for different signaling disciplines.

A simulated monitor can be extended to support testing and debugging.

UTA

How to use a Java monitor toolbox

A regular Java class can be made into a monitor class by doing the following:

- extend class MonitorSC or MonitorSU
- use operations enterMonitor and exitMonitor at the start and end of each public method.
- declar as many ConditionVariables as needed
- use operations waitC () and signalC/signalCall () on ConditionVariables

MonitorSC

```
public class MonitorSC {
    private BinarySemaphore mutex = new BinarySemaphore (1);
    protected final class ConditionVariable {
        private CountingSemaphore threadQue = new CountingSemaphore (0);
        private int numWaitingThreads = 0;
        public void signalC () {
            if (numWaitingThreads > 0) {
                numWaitingThreads --;
                threadQue.V();
            }
        }
        public void signalCall () {
            while (numWaitingThreads > 0) {
                -- numWaitingThreads;
                threadQue.V();
            }
        }
        public void waitC () {
            numWaitingThreads ++;
            threadQue.VP(mutex); // { mutex.V(); threadQue.P() }
            mutex.P();
        }
        public boolean empty () { return numWaitingThreads == 0; }
        public int length () { return numWaitingThreads; }
    }

    protected void enterMonitor () { mutex.P(); }
    protected void exitMonitor () { mutex.V(); }
}
```


MonitorSU

```
public class MonitorSU {
    private BinarySemaphore mutex = new BinarySemaphore (1);
    private BinarySemaphore reentry = new BinarySemaphore (0);
    private int reentryCount = 0;
    protected final class ConditionVariable {
        private CountingSemaphore threadQue = new CountingSemaphore (0);
        private int numWaitingThreads = 0;
        public void signalC () {
            if (numWaitingThreads > 0) {
                ++ reentryCount;
                reentry.VP (threadQue); // { threadQue.V(); reentry.P() }
                -- reentryCount;
            }
        }
        public void signalC_and_exitMonitor () {
            if (numWaitingThreads > 0) threadQue.V();
            else if (reentryCount > 0) reentry.V();
            else mutex.V()
        }
        public void waitC () {
            numWaitingThreads ++;
            if (reentryCount > 0) threadQue.VP (reentry); // { reentry.V(); threadQue.P() }
            else threadQue.VP(mutex); // { mutex.V(); threadQue.P() }
            -- numWaitingThreads;
        }
        public boolean empty () { return numWaitingThreads == 0; }
        public int length () { return numWaitingThreads; }
    }

    protected void enterMonitor () { mutex.P(); }
    protected void exitMonitor () {
        if (reentryCount > 0) reentry.V ();
        else mutex.V();
    }
}
```

Example

```
final class BoundedBuffer extends MonitorSC {
   ...
   private ConditionVariable notFull = new ConditionVariable ();
   private ConditionVariable notEmpty = new ConditionVariable ();
   ...

   public void deposit (int value) {
      enterMonitor ();
      while (fullSlots == capacity) {
         notFull.waitC ();
      }
      buffer[in] = value;
      in = (in + 1) % capacity;
      ++ fullSlots;
      notEmpty.signalC ();
      exitMonitor ();
   }
   ...
}
```


Example (cont'd)

```
final class BoundedBuffer extends MonitorSU {
   ...
   private ConditionVariable notFull = new ConditionVariable ();
   private ConditionVariable notEmpty = new ConditionVariable ();
   ...

   public void deposit (int value) {
      enterMonitor ();
      while (fullSlots == capacity) {
         notFull.waitC ();
      }
      buffer[in] = value;
      in = (in + 1) % capacity;
      ++ fullSlots;
      notEmpty.signalC_and_exitMonitor ();
   }
   ...
}
```

UTA

Monitor

- Introduction
- Monitors in Java
- Signaling Disciplines
- Implementing Monitors
- Monitor based Solutions

UTA

Dining Philosopher

```
while (true) {
  /* thinking */
  dp.pickUp (i);
  /* eating */
  dp.putDown (i)
}
```

Dining Philosophers (Solution 1)

```
monitor class DiningPhilosopher {
    final int n = ... ; // number of philosophers
    final int thinking = 0, hungry = 1, eating = 2;
    int state[] = new int [n];
    ConditionVariable[] self = new ConditionVariable [n];
    DiningPhilosopher () {
        for (int i = 0; i < n; i ++) state[i] = thinking;
        for (int j = 0; j < n; j ++) self[j] = new ConditionVariable ();
    }
    public void pickUp (int i) {
        state[i] = hungry;
        test(i);
        if (state[i] != eating) self[i].wait ();
    }
    public void putDown (int i) {
        state[i] = thinking;
        test( (i-1) % n);
        test( (i+1) % n);
    }
    private void test (int k) {
        if ( (state[k] == hungry) && (state[(k-1)%n] != eating) && (state[(k+1)%n] != eating)) {
            state[k] = eating;
            self[k].signal ();
        }
    }
}
```


Dining Philosophers (Solution 2)

```
monitor class DiningPhilosopher {
    final int n = ... ; // number of philosophers
    final int thinking = 0, hungry = 1, starving = 2, eating = 3;
    int state[] = new int [n];
    ConditionVariable[] self = new ConditionVariable [n];
    DiningPhilosopher () {
        for (int i = 0; i < n; i ++) state[i] = thinking;
        for (int j = 0; j < n; j ++) self[j] = new ConditionVariable ();
    }
    public void pickUp (int i) {
        state[i] = hungry;
        test(i);
        if (state[i] != eating) self[i].wait ();
    }
    public void putDown (int i) {
        state[i] = thinking;
        test( (i-1) % n);
        test( (i+1) % n);
    }
    private void test (int k) {
        if ( (state[k] == hungry || state[k] == starving) && (state[(k-1)%n] != eating &&
state[(k-1)%n] != starving) && (state[(k+1)%n] != eating && state[(k+1)%n] != starving)) {
            state[k] = eating;
            self[k].signal ();
        }
        else if ((state[k] == hungry) && (state[(k-1)%n] != starving) && (state[(k+1)%n] !=
starving)) {
            state[k] = starving;
        }
    }
}
```

UTA

R>W.1

```
monitor class RW {
    int readerCount = 0; boolean writing = false;
    ConditionVariable readerQue = new ConditionVariable ();
    ConditionVariable writerQue = new ConditionVariable ();
    int singaledReaders = 0;
    public void startRead () {
        if (writing) {
            readerQue.wait ();
            -- signaledReaders;
        }
        ++ readerCount;
    }
    public void endRead () {
        -- readerCount;
        if (readerCount == 0 && signaledReaders == 0)
            writerQue.signal ();
    }
    public void startWrite () {
        while (readerCount > 0 || writing || !readerQue.empty() || signaledReaders > 0)
            writeQue.wait ();
        writing = true;
    }
    public void endWrite () {
        writing = false;
        if (!readerQue.empty ()) {
            signaledReaders = readerQue.length();  readerQue.signalAll ();
        }
        else
            writerQue.signal ();
    } }
```


UTA

R > W.1 (cont'd)

read

```
rw.start_read ();
    /* read shared data */
rw.end_read ();
```

write

```
rw.start_write ();
    /* read shared data */
rw.end_write ();
```