# Monitors

CS511

# Review

- We've seen that semaphores are an efficient tool to solve synchronization problems
- However, they have some drawbacks
  1. They are low-level constructs
     - It is easy to forget an acquire or release
  2. They are not related to the data
     - They can appear in any part of the code

# Monitors

- Combines ADTs and mutual exclusion
  - Proposed by Tony Hoare [1974]
- Adopted in many modern PLs
  - Java
  - C#

# Main Ingredients

- A set of operations encapsulated in modules
- A unique lock that ensures mutual exclusion to all operations in the monitor
- Special variables called condition variables, that are used to program conditional synchronization

# Counter Example

- Construct a counter with two operations: inc() and dec().
- No two threads should be able to simultaneously modify the value of the counter
  - Think of a solution using semaphores
  - A solution using monitors

# Counter using Semaphores

```
class Counter {
  private int c = 0;
  private Semaphore mutex = new Semaphore(1);

  public void inc() {
     mutex.acquire();
     c++;
     mutex.release();
  }
  public void dec() {
     mutex.acquire();
     c--;
     mutex.release();
  }
}
```

# Counter using Monitors

```
monitor Counter {

  private int counter = 0;

  public void inc() {
    counter++;
  }

  public void dec() {
    counter--;
  }

}
```

- The monitor comes equipped with a lock or mutex that allows at most one thread to execute its operations
- Note:
  - This is pseudocode (not Hydra nor Java)
  - We will see how to write monitors in Java later

# Condition Variables

- Apart from the lock, there are condition variables associated to the monitor
- They have two operations:
  - `Cond.wait()`
  - `Cond.signal()`
- Just like in semaphores, they have an associated queue of blocked processes.

# Condition Variables

`Cond.wait()`

- always blocks the process and places it in the waiting queue of the variable `Cond`.
- When it blocks, it releases the mutex on the monitor.

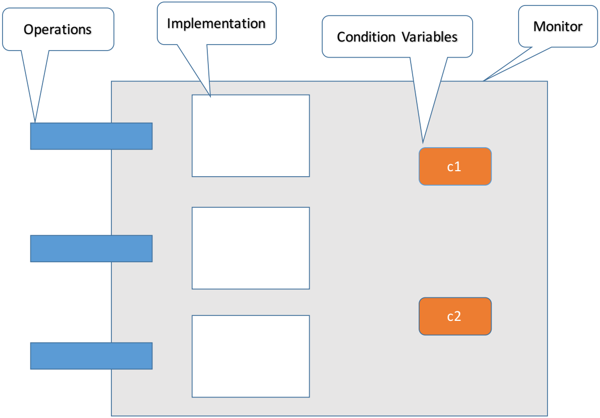
`Cond.notify()`

- Unblocks the first process in the waiting queue of the variable `Cond` and continues execution
- If there are no processes in the waiting queue, it has no effect.

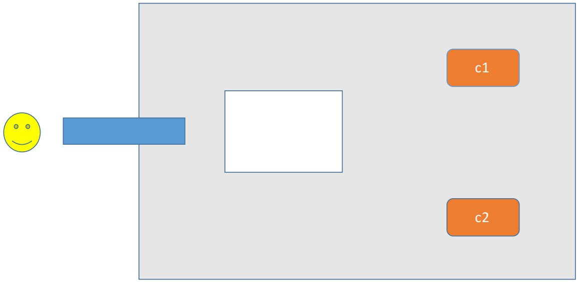
# Example: Buffer (Pseudocode)

```
monitor Buffer {
  private condition spaceAvailable; // wait until space available
  private condition dataAvailable;  // wait until data available

  private Object data = null;    // shared data

  public Object read() {
    if (data == null)
      dataAvailable.wait();
    aux = data;
    data = null;
    spaceAvailable.signal();
    return aux;
  }

  public void write(Object o) {
    if (data != null)
      spaceAvailable.wait();
    data = o;
    dataAvailable.signal();
  }

}
```

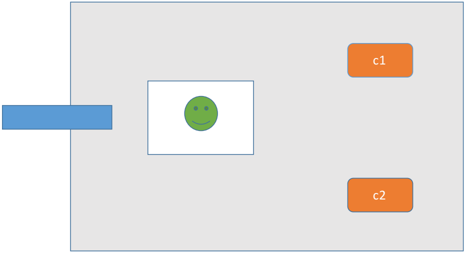
# Graphically



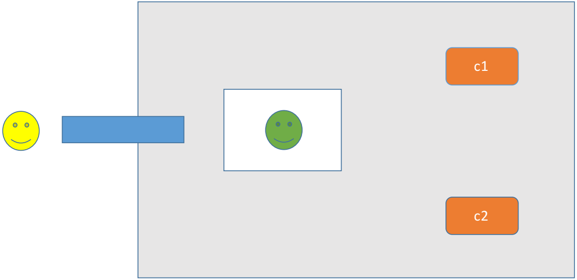
# Typical Behavior



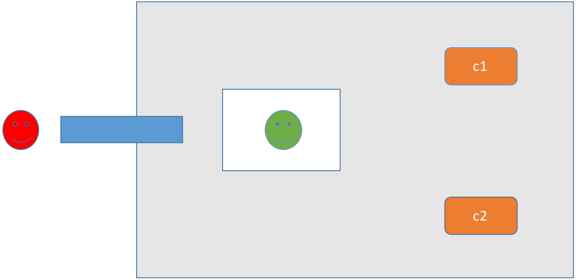
# Typical Behavior



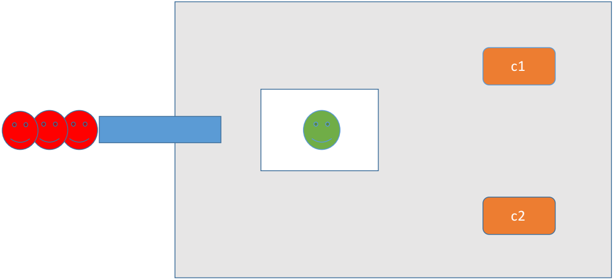
# Typical Behavior



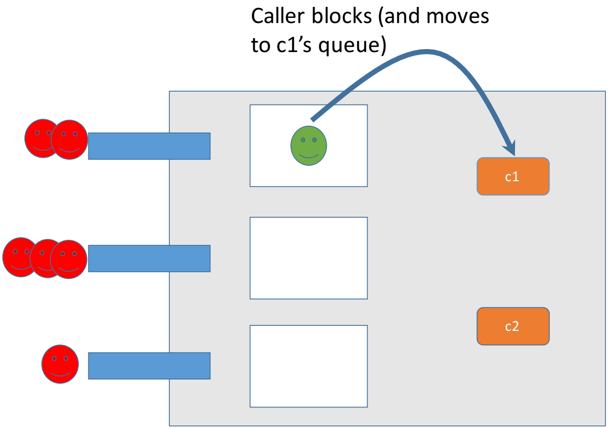
# Typical Behavior



# Typical Behavior

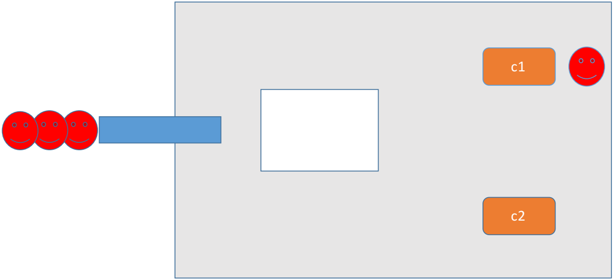


# Wait



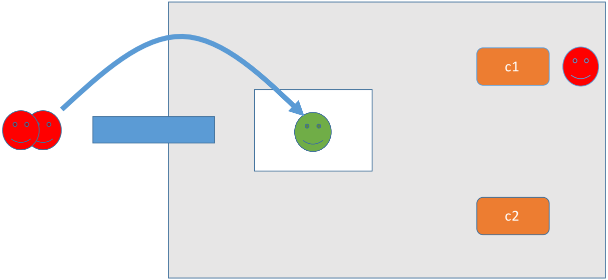

- Blocks process currently executing and associates it to variable's queue
- Upon blocking frees the lock allowing the entry of other processes

# Wait



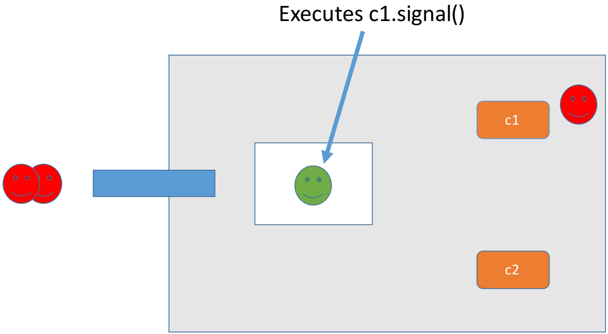

- Blocks process currently executing and associates it to variable's queue
- Upon blocking frees the lock allowing the entry of other processes

# Wait




- Blocks process currently executing and associates it to variable's queue
- Upon blocking frees the lock allowing the entry of other processes

# Signal

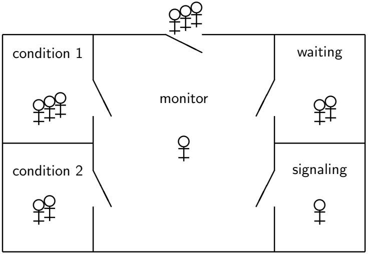



- Signalling process continues to execute after signalling on `c1`?
- Processes waiting in `c1`'s queue start immediately running inside the monitor?
- What about the processes blocked on entry to the monitor?

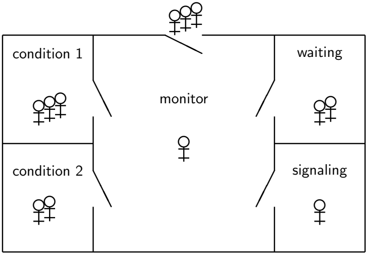
# Signal (cont.)

States that a process can be in:

- Waiting to enter the monitor
- Executing within the monitor (only one)
- Blocked on condition variables
- Queue of processes just released from waiting on a condition variable
- Queue of processes that have just completed a `signal` operation



# Signal




- Precedence of
  - signalling processes $S$
  - waiting processes $W$
  - processes blocked on entry $E$
- Two strategies:
  - Signal and Urgent Wait: $E < S < W$ (classical monitors, aka immediate resumption requirement – IRR)
  - Signal and Continue: $E = W < S$ (Java)

# Signal and Urgent Wait

$$E < S < W$$

- When a process blocked on a condition variable is signaled, it immediately begins executing ahead of the signaling process
- Rationale: Signaling process changed the state of the monitor so that the condition now holds
- Waiting process resumes immediately (at the instruction immediately following the call to wait that blocked it) and hence need not check the condition
- Cons: signaling process unnecessarily delayed (unless signal is the last operation)

# Examples

- Before analyzing the other strategy for signal, namely Signal and Continue, we present another example of a monitor
- We already considered the readers and writers problem with buffer of size 0
- We now consider the same problem with a buffer of size $n$
- Finally, we show how monitors can be used to define semaphores

Monitors

Examples

Monitors in Java

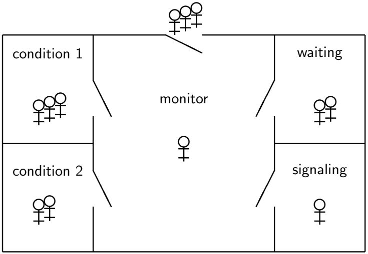
# Exercise: N Dimensional Buffer

```
monitor Buffer {
  private Object[] data = new Object[N+1];
  private int begin = 0, end = 0;
  private condition untilNotFull, untilAvailable;

  public void push(Object o) {
    if (isFull())
        untilNotFull.wait();
    data[begin] = o;
    begin = next(begin);
    untilAvailable.notify()
  }

  public Object pop() {
    if (isEmpty())
       untilAvailable.wait();
    Object result = data[end];
    end = next(end);
    untilNotFull.notify();
    return result;
  }

  private boolean isEmpty() { return begin == end; }
  private boolean isFull()  { return next(begin) == end; }
  private int next(int i)   { return (i+1)%(N+1); }
```

# Monitor that Defines a Semaphore

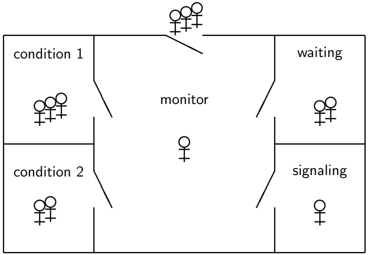
```
monitor Semaphore {
  private condition noCero;
  private int permissions;

  public Semaphore(int n) {
    this.permissions = n;
  }

  public void acquire() {
    if (permissions == 0)
      noCero.wait();
    permissions--;
  }

  public void release() {
    permissions++;
    noCero.signal();
  }

}
```

# Strategies for Signal




- Recall from above:
  - Precedence of signalling processes $S$
  - Precedence of waiting processes $W$
  - Precedence of processes blocked on entry $E$
- Two strategies for signal:
  - Signal and Urgent Wait: $E < S < W$ (classical monitors, aka immediate resumption requirement – IRR)
  - Signal and Continue: $E = W < S$ (Java)
- We already considered Signal and Urgent Wait
- We now consider Signal and Continue

# Signal and Continue: $E = W < S$ (Java)




- Process from $S$ which executes the signal continues execution
- Process from $W$ which is unblocked joins competition for the lock
- Problem: signaling process can modify the condition after it executed the signal

# Signal and Continue

- Must re-check the condition

```
public void acquire() {
  while (permissions == 0)
    noCero.wait();
  permissions--;
}
```

- Risk: introduce starvation.
- Seems less intuitive, but is preferred discipline today
  - Simpler formal semantics
  - Compatible with priority policies

# Signal and Continue

```java
public void acquire() {
  while (permissions == 0)
    noCero.wait();
  permissions--;
}

public void release() {
  permissions++;
  noCero.signal();
}
```

# Signal and Continue

```
public void acquire() {
  while (permissions == 0)
    noCero.wait();
  permissions--;
}

public void release() {
  permissions++;
  noCero.signal();
}
```

- Is it fair?
- What happens with a process that is waiting to acquire the lock?

# Signal and Continue

- Not fair because a process outside the monitor could steal the permission
- Possible measure: pass the permission on to a blocked process on the condition variable
- This requires that the process that executes the signal detects whether there are blocked processes on the associated condition variable

# Monitor that defines a Semaphore

```
public void acquire() {
  if (permissions == 0) {
    waiting++;
    noCero.wait();
    waiting--;
  } else {
    permissions--;
  }
}

public void release() {
  if (waiting == 0)
    permissions++;
  else // else case does not increment permissions
    noCero.signal();
}
```

Monitors

Examples

Monitors in Java

# Monitors in Java

- Every class has a lock and a unique condition variable
  - Pros: convenient encapsulation (lock and condition variable cannot be tampered with)
  - Cons: Using multiple condition variables may benefit efficiency
- Methods `wait`, `notify` and `notifyAll` belong to the class `Object`
- Must use `synchronize` keyword in each method of the monitor
  - Guarantees mutual exclusion
  - Allows operations on condition variables to be invoked

# N Dimensional Buffer in Java

```
class Buffer {
  private Object[] data = new Object[N+1];
  private int begin = 0, end = 0;

  public synchronized void push(Object o) {
    while (isFull()) wait();
    data[begin] = o;
    begin = next(begin);
    notifyAll();
  }

  public synchronized Object pop() {
    while (isEmpty()) wait();
    Object result = data[end];
    end = next(end);
    notifyAll();
    return result;
  }

  private boolean isEmpty() { return begin == end; }
  private boolean isFull()  { return next(begin) == end; }
  private int next(int i)   { return (i+1)%(N+1); }

}
```