COMP 150-CCP Concurrent Programming

Lecture 10: Introduction to Semaphores

Dr. Richard S. Hall
rickhall@cs.tufts.edu

Semaphores

Semaphores (Dijkstra 1968) are widely used for dealing with inter-process synchronization in operating systems. A semaphore ***s*** is an integer variable that can hold only non-negative values.

The only operations permitted on ***s*** are ***up(s)*** (**V** = vrijgeven = release) and ***down(s)*** (**P** = passeren = pass). Blocked processes are held in a FIFO queue.

down(s)*:** if (s*** > 0) then decrement ***s***
else block execution of the calling process

up(s)*:** if (processes blocked on ***s) then awaken one of them
else increment ***s***

Modeling Semaphores

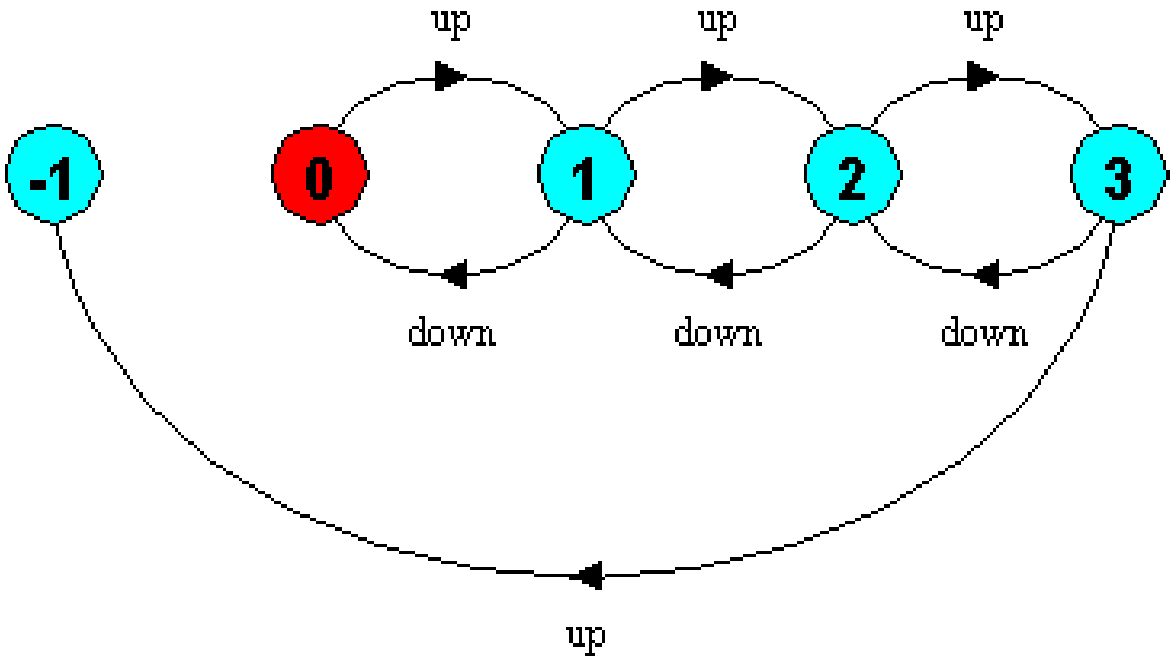
To ensure analyzable models, we only model semaphores that take a finite range of values. If this range is exceeded then we regard this as an **ERROR**. **N** is the initial value.

```
const Max = 3
range Int = 0..Max

SEMAPHORE(N=0)  = SEMA[N],
SEMA[v:Int]     = (up->SEMA[v+1]
                  |when(v>0) down->SEMA[v-1]),
SEMA[Max+1]     = ERROR.
```

LTS?

Modeling Semaphores



Action `down` is only accepted when value `v` of the semaphore is greater than 0.

Action `up` is not guarded.

Trace to a violation

`up` → `up` → `up` → `up`

Semaphore Example

Three processes `p[1..3]` use a shared ***mutex*** semaphore to ensure mutually exclusive access to ***critical region*** (i.e., access to some shared resource).

```
LOOP = (mutex.down->critical->mutex.up->LOOP).
||SEMADEMO = (p[1..3]:LOOP
              ||{p[1..3]}::mutex:SEMAPHORE(1)).
```

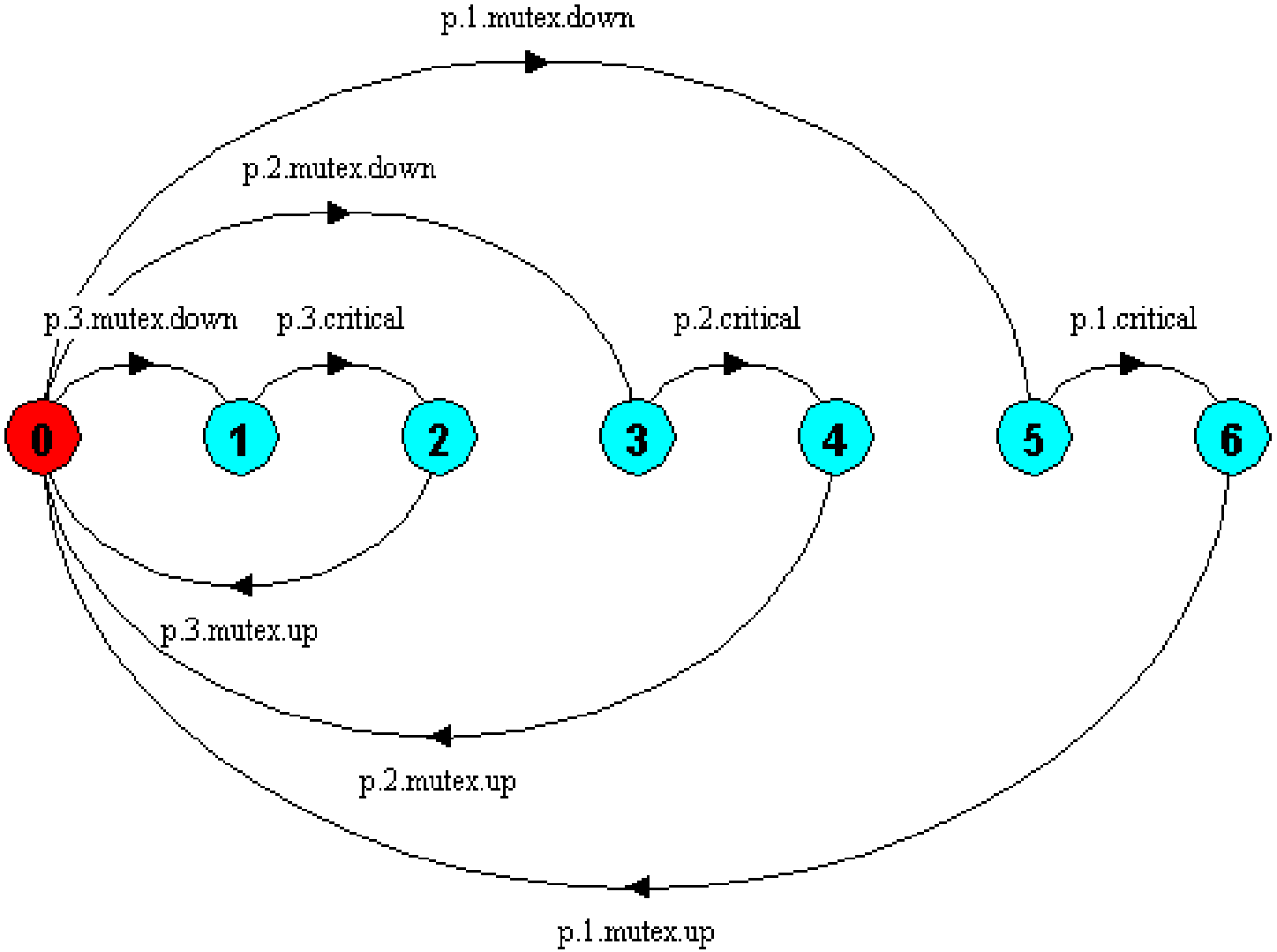
For mutual exclusion, the semaphore initial value is `1`.
Why?

Is the `ERROR` state reachable for `SEMADEMO`***?***

Is a *binary* semaphore sufficient (i.e., `Max=1`)***?***

LTS?

Semaphore Example



Semaphores in Java

Semaphores are passive objects, therefore implemented as ***monitors***.

(In practice, semaphores are a low-level mechanism often used in implementing the higher-level monitor construct.)

```
public class Semaphore {
  private int value;

  public Semaphore (int initial)
    {value = initial;}

  public synchronized void up() {
     ++value;
     notify();
  }

  public synchronized void down()
       throws InterruptedException {
    while (value == 0) wait();
    --value;
  }
}
```

Why `notify()` not `notifyAll()` ?
Why no `notify()` in `down()` ?

Bounded Buffer Example

A bounded buffer consists of a fixed number of slots. Items are put into the buffer by a ***producer*** process and removed by a ***consumer*** process. It can be used to smooth out transfer rates between the ***producer*** and ***consumer***.

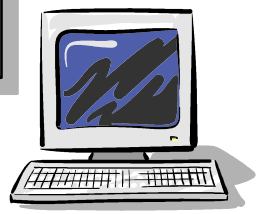
Semaphore Bounded Buffer Model

The behavior of BOUNDEDBUFFER is independent of the actual data values, and so can be modeled in a data-independent manner.

```
const Max = 5
range Int = 0..Max

SEMAPHORE ...as before...

BUFFER =   (put -> empty.down -> full.up ->BUFFER
           |get -> full.down -> empty.up ->BUFFER
           ).
PRODUCER = (put -> PRODUCER).
CONSUMER = (get -> CONSUMER).
||BOUNDEDBUFFER = (PRODUCER|| BUFFER || CONSUMER
                  ||empty:SEMAPHORE(5)
                  ||items:SEMAPHORE(0))
                  @{put,get}.
```



Java Semaphore Bounded Buffer

We use two semaphores **full** and **empty** to reflect the state of the buffer, instead of condition variables.

We create a separate buffer interface to permit alternative implementations.

```
public interface Buffer {…}

public class SemaBuffer implements Buffer {
  …

  Semaphore items;  //counts number of items
  Semaphore spaces; //counts number of spaces

  public SemaBuffer(int size) {
    this.size = size; buf = new Object[size];
    items = new Semaphore(0);
    spaces = new Semaphore(size);
  }
…
}
```

Java Semaphore Bounded Buffer

spaces is decremented during the **put()** operation, which is blocked if **spaces** is zero; **items** is decremented by the **get()** operation, which is blocked if **items** is zero.

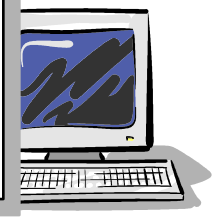
```
public synchronized void put(Object o)
                throws InterruptedException {
  spaces.down();
  buf[in] = o;
  ++count; in = (in+1) % size;
  items.up();
}

public synchronized Object get()
                 throws InterruptedException {
  items.down();
  Object o = buf[out]; buf[out] = null;
  --count; out = (out+1) % size;
  spaces.up();
  return o;
}
```

Java Bounded Buffer Producer

```
public class Producer implements Runnable {
  private Buffer buf;
  private String alphabet = "abcdefghijklmnopqrstuvwxyz";
  private boolean paused = true;
  public Producer(Buffer b) { buf = b; }
  public synchronized void pause() {
    paused = !paused;
    notify();
  }
  public void run() {
    try {
      int ai = 0;
      while (true) {
        synchronized (this) {
          while (paused) { wait(); }
        }
        buf.put(new Character(alphabet.charAt(ai)));
        ai = (ai + 1) % alphabet.length();
        Thread.sleep(500);
      }
    } catch (InterruptedException ex) { }
  }
}
```

Consumer **is similar but calls** ***buf.get().***



Nested Monitor Problem

LTSA analysis predicts a possible ***deadlock***:

```
Composing
 potential DEADLOCK
States Composed: 28 Transitions: 32 in 60ms
Trace to DEADLOCK:
    get
```

The `Consumer` tries to `get` a character, but the buffer is empty. It blocks and releases the lock on the semaphore `items`. The `Producer` tries to `put` a character into the buffer, but also blocks. ***Why?***

This situation is known as the ***nested monitor problem***.

Nested Monitor Program Fix

The only way to avoid it in Java is by careful design. In this example, the deadlock can be removed by ensuring that the monitor lock for the buffer is not acquired until ***after*** semaphores are decremented.

```
public void put(Object o)
   throws InterruptedException {
   spaces.down();
   synchronized (this){
      buf[in] = o; ++count;
      in = (in+1) % size;
   }
   items.up();
}
```

Nested Monitor Model Fix

```
BUFFER =   (put -> BUFFER
           |get -> BUFFER).
PRODUCER =
     (spaces.down->put->items.up->PRODUCER).
CONSUMER =
     (items.down->get->spaces.up->CONSUMER).
```

The semaphore actions have been moved to the producer and consumer. This is exactly as in the implementation where the semaphore actions are outside the monitor .

Does this behave as desired?

Nested Monitor Program Fix

Or perhaps use a mutex semaphore, rather than mixing monitors and semaphores.

```
...
Semaphore items;  //counts number of items
Semaphore spaces; //counts number of spaces
Semaphore mutex;  //control access to critical region
public SemaBuffer(int size) {
  this.size = size; buf = new Object[size];
  items = new Semaphore(0);
  spaces = new Semaphore(size);
  mutex = new Semaphore(1);
}
public void put(Object o) throws InterruptedException {
  spaces.down();
  mutex.down();
  buf[in] = o; ++count;
  in = (in+1) % size;
  mutex.up();
  items.up();
}
...
```

Monitor Bounded Buffer Model

Since monitors are higher level than semaphores, the BOUNDEDBUFFER model and Java implementation are more straightforward using monitors.

```
BUFFER(N=5) = COUNT[0],
COUNT[i:0..N]
    = (when (i<N) put->COUNT[i+1]
      |when (i>0) get->COUNT[i-1]
      ).

PRODUCER = (put->PRODUCER).
CONSUMER = (get->CONSUMER).

||BOUNDEDBUFFER = (PRODUCER
                   ||BUFFER(5)||CONSUMER).
```

(similar to the Parking Lot example)

Java Monitor Bounded Buffer

```
class MonitorBuffer implements Buffer {
      …
  public synchronized void put(Object o)
              throws InterruptedException {
    while (count == size) wait();
    buf[in] = o; ++count; in = (in+1) % size;
    notifyAll();
  }

  public synchronized Object get()
              throws InterruptedException {
    while (count == 0) wait();
    Object o = buf[out];
    buf[out] = null; --count; out = (out+1) % size;
    notifyAll();
    return o;
  }
}
```