Monitors & Condition Synchronization & Semaphores

INF2140 Parallel Programming. Lecture 6: **Chapter 5 Part II**

Feb. 29, 2012

Plan today: Monitors and Semaphores

- repetition of **semaphores**
- more on implementation of **SemaDemo** example
- **Bounded Buffer** example
  - condition synchronization
  - with semaphores
- so-called *nested semaphores*
  - deadlock
- monitor invariants

Repetition: Monitors

- encapsulates data and control
- provides mutual exclusion
  - FSP: a monitor is a process
  - Java: a monitor is a (passive) object
- Conditional synchronization
  - FSP: simply using conditions
  - Java: synchronized methods with
    - while-testing, wait, notify and notifyAll,
    - using the build-in queue of the monitor object.
- Semaphores (see below)

5.2 Semaphores

- Semaphores are widely used for dealing with inter-process synchronization in operating systems. Semaphore **s** is like an integer variable that can take only non-negative values.
- The only operations permitted on **s** are **up(s)** and **down(s)**. 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

modelling semaphores

To ensure analyzability, 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.
```

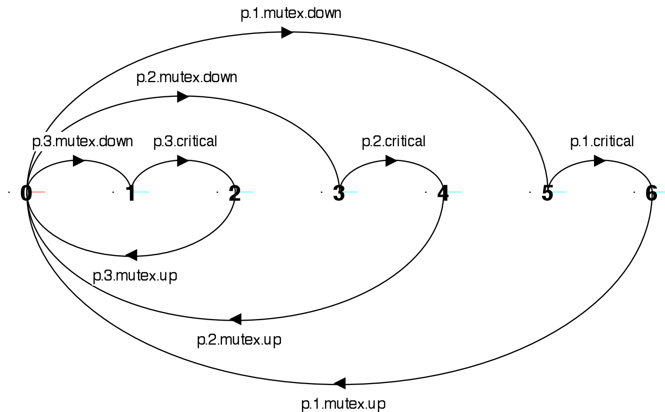
semaphore demo - model

Three processes **p[1..3]** use a shared semaphore **mutex** to ensure mutually exclusive access (action **critical**) to some 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.
- Note: **binary semaphore** (N=1)
- Tested last time with a test process.

semaphore demo - model



semaphores in Java

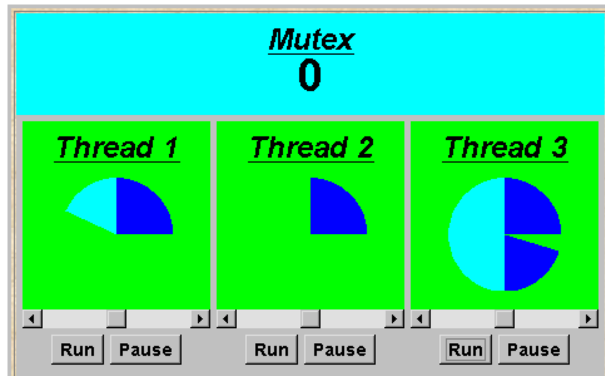
```
public class Semaphore {
  private int value;

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

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

- Semaphores are passive objects, therefore implemented as monitors.
- Is it safe to use notify() here rather than notifyAll()?
- Note: each semaphore implemented by a separate object (and its *waiting queue*).

SEMADEMO display



SEMADEMO

What if we adjust the time that each thread spends in its **critical section**?

- large resource requirement - more conflict?
  (eg. more than 67% of a rotation)?
- small resource requirement - no conflict?
  (eg. less than 33% of a rotation)?

Hence the time a thread spends in its critical section should be kept as short as possible.

SEMADEMO program - revised ThreadPanel class

```
public class ThreadPanel extends Panel {
//construct display with title, rotating arc color c
  public ThreadPanel(String title, Color c) {...}
//hasSlider == true creates panel with slider
  public ThreadPanel
  (String title, Color c, boolean hasSlider) {...}
//rotate display of currently running thread 6 deg.
//return false when in initial color, otherw. true
  public static boolean rotate()
          throws InterruptedException {...}
//rotate display of currently running thread by deg.
  public static void rotate(int degrees)
          throws InterruptedException {...}
//create a new thread with target r and start it
  public void start(Runnable r) {...}
//stop the thread using Thread.interrupt()
  public void stop() {...}
                    }
```

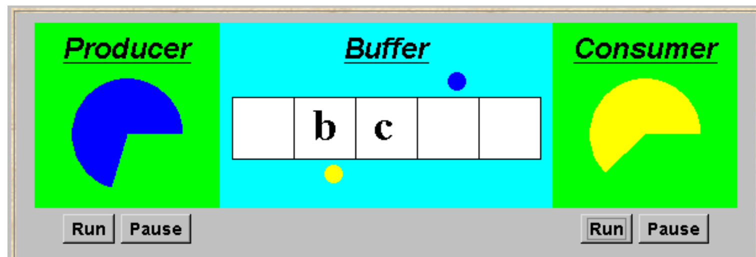
SEMADEMO program - class MutexLoop

```
class MutexLoop implements Runnable {
  Semaphore mutex;
  MutexLoop (Semaphore sema) {mutex=sema;}
  public void run() {
    try {
      while(true)  {
        while(!ThreadPanel.rotate());
        mutex.down(); // get mutual exclusion
        while(ThreadPanel.rotate());
                      // critical actions
        mutex.up();   // release mutual excl.
      }
    } catch(InterruptedException e){}
}                                      }
```

Comments: SEMADEMO program - class `MutexLoop`

- Threads and semaphore are created by the applet `start()` method.
- `ThreadPanel.rotate()` returns false while executing non-critical actions (dark color) and true otherwise.

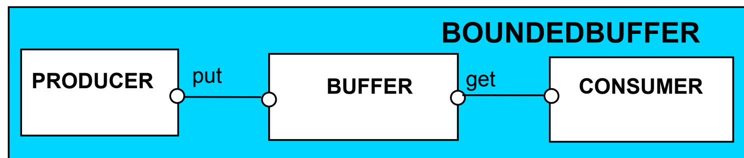
5.3 Bounded Buffer



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.

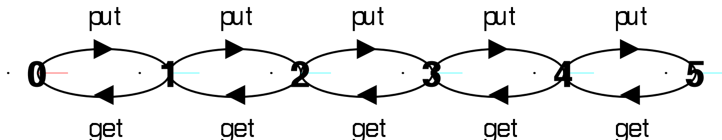
It can be used to smooth out transfer rates between the producer and consumer. (see car park example)

Bounded Buffer - a data -independent model



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

LTS:



Bounded Buffer - a data-independent model

```
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).
```

- Note: simple condition synchronization.
- All values abstracted away!
- Here only one producer and consumer.

Bounded Buffer - a data-independent model - multiple users

```
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 = ({a,b,c}:PRODUCER||{a,b,c}:CONSUMER
                  ||{a,b,c}::BUFFER(5)).
```

- Note: 7 processes
- still works as expected?

Bounded Buffer program - buffer monitor

```
public interface Buffer <E> {...}
class BufferImpl<E> implements Buffer<E> {...
  public synchronized void put(E o)
            throws InterruptedException {
    while (count==size) wait();
    buf[in] = o; ++count; in = (in+1)%size;
    notifyAll(); }
  public synchronized E get()
            throws InterruptedException {
    while (count==0) wait();
    E o = buf[out];
    buf[out]=null; --count; out=(out+1)%size;
    notifyAll();
    return (o);
  }                                    }
```

Comments: Bounded Buffer program - buffer monitor

Notes:

- We separate the interface to permit an alternative implementation later.
- Is it safe to use `notify()` here rather than `notifyAll()`?

Bounded Buffer program - producer process

```
class Producer implements Runnable {
  Buffer buf;
  String alphabet="abcdefghijklmnopqrstuvwxyz";
  Producer(Buffer b) {buf = b;}
  public void run() {
    try {
      int ai = 0;
      while(true) {
        ThreadPanel.rotate(12);
        buf.put(alphabet.charAt(ai));
        ai=(ai+1) % alphabet.length();
        ThreadPanel.rotate(348); }
    } catch (InterruptedException e){}
} }
```

Bounded Buffer program - Consumer process

Similarly Consumer which calls **buf.get()**.

5.4 Nested Monitors

Suppose that, in place of using the **count** variable and condition synchronization directly, we instead use two semaphores **full** and **empty** to reflect the state of the buffer.

```
class SemaBuffer <E> implements Buffer <E> {
  ...
  Semaphore full;  //counts number of items
  Semaphore empty; //counts number of spaces

  SemaBuffer(int size) {
    this.size = size; buf =(E[])new Object[size];
    full = new Semaphore(0);
    empty= new Semaphore(size);
} }
```

Nested Monitors - bounded buffer program

```
synchronized public void put(E o)
                throws InterruptedException {
    empty.down();
    buf[in] = o;
    ++count; in=(in+1)%size;
    full.up();
  }
  synchronized public E get()
                throws InterruptedException{
    full.down();
    E o =buf[out]; buf[out]=null;
    --count; out=(out+1)%size;
    empty.up();
    return (o);
  }
```

Nested Monitors - bounded buffer program

Comments:

- **empty** is decremented during a `put` operation, which is blocked if empty is zero;
- **full** is decremented by a `get` operation, which is blocked if full is zero.
- Does this behave as desired?

Nested Monitors - bounded buffer FSP model

```
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) ||full:SEMAPHORE(0)
                         )@{put,get}.
```

- Does this behave as desired?

Nested Monitors - bounded buffer model

LTSA analysis predicts a possible DEADLOCK:

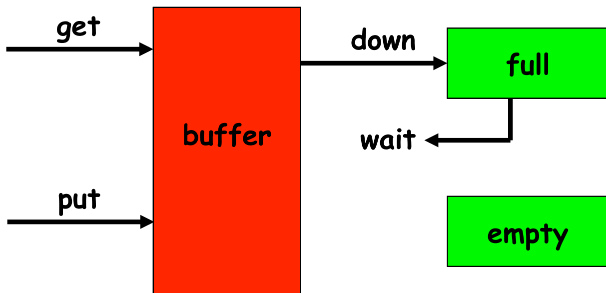
```
Composing
 potential DEADLOCK
States Composed: 28 Transitions: 32 in 60ms
Trace to DEADLOCK:
      get
```

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

- This situation is known as the nested monitor problem.

Nested Monitors - bounded buffer program

```
synchronized public Object get()
                 throws InterruptedException{
    full.down(); // if no items, block!
    ...
  }
```



Nested Monitors - revised bounded buffer program

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.

```java
public void put(E o)
                throws InterruptedException {
    empty.down();
    synchronized(this){
      buf[in] = o; ++count; in=(in+1)%size;
    }
    full.up();
  }
```

Nested Monitors - revised bounded buffer model

```
BUFFER    =  (put -> BUFFER
             |get -> BUFFER).
PRODUCER =(empty.down->put->full.up->PRODUCER).
CONSUMER =(full.down->get->empty.up->CONSUMER).

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

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?
- Minimized LTS?

Nested Monitors model - multiple consumers/producers

```
BUFFER =  (put -> BUFFER
          |get -> BUFFER).

PRODUCER =(empty.down->put->full.up->PRODUCER).
CONSUMER =(full.down->get->empty.up->CONSUMER).

||BOUNDEDBUFFER = ({a,b,c}:PRODUCER || {a,b,c}:CONSUMER
                  || {a,b,c}::BUFFER
                  || {a,b,c}::empty:SEMAPHORE(5)
                  || {a,b,c}::full :SEMAPHORE(0))
                 @{{a,b,c}.put,get}.
```

- Does this behave as desired?

5.5 Monitor Invariants

An **invariant** for a monitor is an assertion concerning the variables it encapsulates. This assertion must hold whenever there is no thread executing inside the monitor, i.e., on thread **entry** to and **exit** from a monitor.

- CarParkControl Invariant: $0 \leq spaces \leq N$
- Semaphore Invariant: $0 \leq value$
- Buffer Invariants:
  - $0 \leq count \leq size$
  - $0 \leq in < size$
  - $0 \leq out < size$
  - $in = (out + count)$ **modulo** $size$

Invariants can be helpful in reasoning about correctness of monitors using a logical *proof-based* approach. Generally we prefer to use a *model-based* approach amenable to mechanical checking .

Summary

- Concepts
  - **monitors**:
    - encapsulated data + access procedures
    - mutual exclusion + synchronization
  - nested monitors
- Model
  - guarded actions
- Practice
  - private data and synchronized methods in Java
  - `wait()`, `notify()` and `notifyAll()` for condition synchronization
  - single thread active in the monitor at a time