Monitors

Abhik Roychoudhury
CS 3211
National University of Singapore

Modified from Kramer and Magee’s lecture notes.
Reading material: Chapter 5 of Textbook.

monitors & condition synchronization

Concepts: monitors:
- encapsulated data + access procedures
- mutual exclusion + condition synchronization
- single access procedure active in the monitor
- nested monitors

Models: guarded actions

Practice: private data and synchronized methods (exclusion).
- wait(), notify() and notifyAll() for condition synch.
- single thread active in the monitor at a time

The concept of monitors

Brings in the concept of protected or private data.
The protected data is accessed by several threads via operations.
Protected data cannot be accessed without invoking the operations.
Each operation is executed atomically.
A monitor thus represents a passive object, whose operations are invoked by various active objects — the threads.

A schematic monitor

Monitor X{
  int n = 0;
  operation increment{
    int tmp;
    tmp = n; n = tmp+1;
  }
}

Diagrammatic view of monitor X

The critical section code is encapsulated inside monitor operations, not replicated inside processes.

Extending monitors with conditions

Monitor operations may involve waiting on conditions (these are simple boolean expressions).
When such conditions become true, the waiting threads are notified (using wait, notify feature of Java).
Thus, each such condition has a waiting queue of blocked processes.
The schematic for conditional wait / notify are:

wait_on_cond(Cond)
  append p, the current proc. to queue for Cond
  p.state = blocked
  monitor.lock = released

signal_to_cond(Cond){
  if queue for Cond != empty{
    remove head of queue, let it be process x;
    x.state = ready
  }
}

Producer consumer problem

Producer: blocks if buffer is full.
Consumer: Blocks if buffer is empty.
Schematic Producer-Consumer

```
monitor PC{
    buffer = empty;
    condition notFull, notEmpty;
    operation produce(v){
        if buffer is full{
            wait_on_cond(notFull)
        } else {
            add v to tail of buffer;
            signal_to_cond(notEmpty)
        }
    }
    operation consume(){
        if buffer is empty{
            wait_on_cond(notEmpty)
        } else {
            remove w from head of buffer;
            signal_to_cond(notFull);
            return w;
        }
    }
}
```

Producer                         Consumer
while (1){
    d = get_new_item;
    PC.produce(d);
}
while (1){
    d = PC.consume();
    put_item(d);
}

Monitors in Java

Not a default construct.
Need to be programmed as a new class with private data (the data being protected) and synchronized methods.

Blocking of processes is supported by wait()
Unblocking of processes is supported by notify(), notifyAll()

wait() can throw exceptions, so we will add code to catch them.

Producer-consumer in Java

```
class PCMonitor{
    final int N = 5;
    int Oldest = 0, Newest = 0;
    volatile int Count = 0;
    int Buffer[] = new int[N];
    synchronized void produce(int v){
        while (Count == N) try{ wait();} catch(InterruptedException e) {}
        Buffer[Newest] = v;  
        Newest = (Newest + 1) % N;
        Count++;  notifyAll();
    }
    synchronized int consume(){
        int tmp;
        while (Count == 0) try{ wait();} catch(InterruptedException e) {}
        tmp = Buffer[Oldest]; Oldest = (Oldest + 1) % N;
        Count--; notifyAll();
        return tmp;
    }
}
```

Readers-Writers Problem

Several processes accessing a common resource.
Accessing processes grouped into two categories.
Readers: do not exclude other readers, exclude writers.
Writers: exclude all other processes while accessing.

How to give a solution using monitors?

Schematic Readers-Writers

```
monitor RW{
    int reader, writer;
    condition OKtoRead, OKtoWrite;
    operation StartRead{
        if writers !=0 || not empty(OKtoWrite){
            wait_on_cond(OKtoRead);
        } else{
            reader++;  signal_to_cond(OKtoRead);
        }
    }
    operation EndRead{
        reader--;  if reader == 0 { signal_to_cond(OKtoWrite);}
    }
    operation StartWrite{
        if writers!=0 || readers != 0 {
            wait_on_cond(OKtoWrite);
        } else{
            writer++;  signal_to_cond(OKtoWrite);
        }
    }
    operation EndWrite{
        writer--;  if empty(OKtoRead){ signal_to_cond(OKtoWrite);}
    }
}
```

Correctness of Readers-Writers

R ≡ Number of readers
W ≡ Number of writers

Invariant property

\( (R > 0 \Rightarrow W = 0) \land (W < 1) \land (W = 1 \Rightarrow R = 0) \)

Prove that it is preserved by each of the operations of the RW monitor.
Doing it in Java

Java has no mechanism for waiting on a specific condition. We can call the \texttt{wait()} method of any Java object, which suspends the current thread. The thread is said to be "waiting on" the given object.

Another thread can call the \texttt{notify()} method of the same Java object. This "wakes up" one of the threads waiting on that object. If wrong process is notified it will return itself to the set of waiting processes.

```
synchronized method1(){
   while (x==0) wait();
}
synchronized method2(){
   while (y==0) wait();
}
synchronized method3(…){
   if (…) x = 1 else y = 1;
   notifyAll();
}
```

So Far ...

A basic idea of what monitor is

-- protected data
-- atomic access via methods

Basic Examples to show usage of monitors

-- Producer-consumer
-- Readers-writers

Encoding of monitors on top of Java

**Now**
More advanced programming with monitors.

5.1 Condition synchronization

A controller is required for a carpark, which only permits cars to enter when the carpark is not full and does not permit cars to leave when there are no cars in the carpark. Car arrival and departure are simulated by separate threads.

```
carpark model

♦ Events or actions of interest? arrive and depart
♦ Identify processes. arrivals, departures and carpark control
♦ Define each process and interactions (structure).
```

```
carpark model

CARPARKCONTROL(N=4) = SPACES[N],
   SPACES[i:0..N] = (when(i>0) arrive->SPACES[i-1] | when(i<N) depart->SPACES[i+1]).

ARRIVALS = (arrive->ARRIVALS).
DEPARTURES = (depart->DEPARTURES).
CARPARK = (ARRIVALS | CARPARKCONTROL(4) | DEPARTURES).

Guarded actions are used to control arrive and depart.

LTS?
```
Carpark LTS

\[
\text{CARPARKCONTROL(N=4)} = \text{SPACES}[N],
\text{SPACES}[i:0..N] = \begin{cases} 
\text{arrive->SPACES}[i-1] \\
\text{depart->SPACES}[i+1]
\end{cases}
\]

ARRIVALS = (arrive->ARRIVALS).
DEPARTURES = (depart->DEPARTURES).

\[
\text{||CARPARK} = (\text{ARRIVALS}||\text{CARPARKCONTROL(4)}||\text{DEPARTURES}).
\]

Car park program

- Model - all entities are processes interacting by actions
- Program - need to identify threads and monitors
  - thread - active entity which initiates (output) actions
  - monitor - passive entity which responds to (input) actions.

For the carpark:

![Car park diagram]

Car park program

Arrivals and Departures implement Runnable, CarParkControl provides the control (condition synchronization).

Instances of these are created by the start() method of the CarPark applet.

```java
public void start() {
    CarParkControl c = new DisplayCarPark(carDisplay, Places);
    arrivals.start(new Arrivals(c));
    departures.start(new Departures(c));
}
```

Car park program - Arrivals and Departures threads

```java
class Arrivals implements Runnable {
    CarParkControl carpark;
    Arrivals(CarParkControl c) { carpark = c; }
    public void run() {
        try {
            while (true) {
                ThreadPanel.rotate(330);
                carpark.arrive();
                ThreadPanel.rotate(30);
            }
        } catch (InterruptedException e) {
        }
    }
}
```

How do we implement the control of CarParkControl?

```java
class CarParkControl {
    protected int spaces;
    protected int capacity;
    CarParkControl(int capacity) {
        capacity = spaces = n;
    }
    synchronized void arrive() {
        --spaces;
    }
    synchronized void depart() {
        ++spaces;
    }
}
```

How do we implement the control of CarParkControl?
condition synchronization in Java

Java provides a thread **wait queue** per monitor (actually per object) with the following methods:

- **public final void notify()** - Wakes up a single thread that is waiting on this object's queue.
- **public final void notifyAll()** - Wakes up all threads that are waiting on this object's queue.
- **public final void wait()** - `throws InterruptedException` - Waits to be notified by another thread. The waiting thread releases the synchronization lock associated with the monitor. When notified, the thread must wait to reacquire the monitor before resuming execution.

We refer to a thread **entering** a monitor when it acquires the mutual exclusion lock associated with the monitor and **exiting** the monitor when it releases the lock.

**Wait()** - causes the thread to exit the monitor, permitting other threads to enter the monitor.

**FSP: when cond act -> NEWSTAT**

Java: `public synchronized void act() { while (!cond) wait(); while (!cond) wait(); // modify monitor data notifyAll(); }`

The `while` loop is necessary to retest the condition `cond` to ensure that `cond` is indeed satisfied when it re-enters the monitor. `notifyall()` is necessary to awaken other thread(s) that may be waiting to enter the monitor now that the monitor data has been changed.

### CarParkControl - condition synchronization

```java
public synchronized void act() throws InterruptedException {
    while (spaces==0) wait();
    --spaces;
    notify();
}
```

Why is it safe to use `notify()` here rather than `notifyAll()`?

### Monitors are passive

**Active** entities (that initiate actions) are implemented as **threads**. **Passive** entities (that respond to actions) are implemented as **monitors**.

Each guarded action in the model of a monitor is implemented as a synchronized method which uses a `while` loop and `wait()` to implement the guard. The `while` loop condition is the negation of the model guard condition.

Changes in the state of the monitor are signaled to waiting threads using `notify()` or `notifyAll()`.
5.2 Semaphores

Semaphores are widely used for dealing with inter-process synchronization in operating systems. Semaphore is an integer variable that can take only non-negative values.

The only operations permitted on a semaphore are `up()` and `down()`. 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 \( s \)

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.
```

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)?

```
PUBLIC class Semaphore {
        private int value;
        public Semaphore (int initial) {
                value = initial;
        }
        synchronized public void up() {
                ++value;
                notify();
        }
        synchronized public void down() throws InterruptedException {
                while (value== 0) wait();
                --value;
        }
}
```

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.)
7.3 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.
bounded buffer - a data-independent model

\[
\text{BUFFER}(N=5) = \text{COUNT}[0], \\
\text{COUNT}[1:0..N] = (\text{when } (i<N) \text{ put} \rightarrow \text{COUNT}[i+1] \\
\text{when } (i>0) \text{ get} \rightarrow \text{COUNT}[i-1] \\
\}
\]

PRODUCER = (put \rightarrow \text{PRODUCER}) \\
CONSUMER = (get \rightarrow \text{CONSUMER}) \\
|| \text{BOUNDEDBUFFER} = (\text{PRODUCER} \mid \text{BUFFER}(5) \mid \text{CONSUMER}) \]

public interface Buffer {} \\
class BufferImpl implements Buffer { \\
\text{sync} \\
public synchronized void put(Object o) \\
throws InterruptedException { \\
\text{buf}[\text{in}] = o; \text{++count}; \text{in}=(\text{in}+1)\%\text{size}; \text{notify}(); \\
} \\
\text{sync} \\
public synchronized Object get() \\
throws InterruptedException { \\
\text{empty}.down(); \text{full}.down(); \\
Object o = \text{buf}[\text{out}]; \text{buf}[\text{out}]=\text{null}; \text{--count}; \text{out}=(\text{out}+1)\%\text{size}; \text{notify}(); \text{return} (o); \\
} \\
}

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 implements Buffer { \\
\text{full} = \text{new Semaphore}(0); \\
\text{empty} = \text{new Semaphore}(\text{size}); \\
}

\text{full}.down(); \text{empty}.up(); \\
\text{empty} is decremented during a put operation, which is blocked if empty is zero; \text{full} is decremented by a get operation, which is blocked if full is zero.
nested monitors - bounded buffer 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?

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 - revised bounded buffer model

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

Does this behave as desired?

Minimized LTS?

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.

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

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 ≤ spaces ≤ N
Semaphore Invariant: 0 ≤ value
Buffer Invariant:
    0 ≤ count ≤ size
    and 0 ≤ in < size
    and 0 ≤ out < size
    and 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 + condition 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