Lesson 8

Monitors

Ch 7 [BenA 06]

Monitors
Condition Variables
BACI and Java Monitors
Protected Objects

Monitor Concept

- High level concept
  - Semaphore is low level concept
- Want to encapsulate
  - Shared data and access to it
  - Operations on data
  - Mutex and synchronization
- Problems solved
  - Which data is shared?
  - Which semaphore is used to synchronize processes?
  - Which mutex is used to control critical section?
  - How to use shared resources?
  - How to maximize parallelizable work?
- Other approaches to the same (similar) problems
  - Conditional critical regions, protected objects, path expressions, communicating sequential processes, synchronizing resources, guarded commands, active objects, rendezvous, Java object, Ada package, ...

Other approaches to the same problems

- Conditional critical regions, protected objects, path expressions, communicating sequential processes, synchronizing resources, guarded commands, active objects, rendezvous, Java object, Ada package, ...

Monitor (Hoare 1974)

- Encapsulated data and operations for it
  - Abstract data type, object
  - Public methods are the only way to manipulate data
  - Monitor methods can manipulate only monitor or parameter data
  - Global data outside monitor is not accessible
  - Monitor data structures are initialized at creation time and are permanent
  - Concept "data" denotes here often to synchronization data only
  - Actual computational data processing often outside monitor
  - Concurrent access possible to computational data
  - More possible parallelism in computation

Monitor

- Automatic mutex for monitor methods
  - Only one method active at a time (invoked by some process)
  - May be a problem: limits possible concurrency
  - Monitor should not be used for work, but preferably just for synchronization
  - Other processes are waiting
  - To enter the monitor (in mutex), or
  - Inside the monitor in some method
    - waiting for a monitor condition variable become true
    - waiting for mutex after release from condition variable
  - No queue, just set of competing processes
  - Implementation may vary
- Monitor is passive
  - Does not do anything by itself
  - No own executing threads
  - Exception: code to initialize monitor data structures
  - Methods can be active only when processes invoke them

Algorithm 7.1: Atomicity of monitor operations

```c
monitor CS
    int n = 0
    operation increment
        int temp = n + 1
        n = temp
    end operation increment
end monitor CS
```

```
waitC
    p
    q
    pl: CS.increment
    q1: CS.increment
```

Monitor Condition Variables

- For synchronization inside the monitor
  - Must be hand-coded
  - Not visible outside
  - Looks simpler than really is
- Condition CV
- WaitC (CV)
- SignalC (CV)
Declaration and WaitC

- **Condition CV**
  - Declare new condition variable
  - No value, just fifo queue of waiting processes
- **WaitC (CV)**
  - Always suspends, process placed in queue
  - Unlocks monitor mutex
    - Allows someone else into monitor?
    - Allows another process awakened from (another?) WaitC to proceed?
  - When awakened, waits for mutex lock to proceed
    - Not really ready-to-run yet

SignalC

- **Wakes up first waiting process, if any**
  - Which one continues execution in monitor (in mutex)?
    - The process doing the signalling?
    - The process just woken up?
    - Some other processes trying to get into monitor? No.
  - Two signalling disciplines (two semantics)
    - Signal and continue - signalling process keeps mutex
    - Signal and wait - signalled process gets mutex
  - If no one was waiting, signal is lost (no memory)
    - Advanced signalling (with memory) must be handled in some other manner

Signaling Semantics

- **Signal and Continue**
  - Signaller process continues
  - Mutex can not terminate at signal operation
  - Awakened (signalled) process will wait in mutex lock
  - With other processes trying to enter the semaphore
    - May not be the next one active
    - Many control variables signalled by one process?
  - Condition waited for may not be true any more once awakened process resumes (becomes active again)
  - No priority or priority over arrivals for sem. mutex?

- **Signal and Wait**
  - Awakened (signalled) process executes immediately
    - Mutex baton passing
    - No one else can get the mutex lock at this time
  - Condition waited for is certainly true when process resumes execution
  - Signaller waits in mutex lock
    - With other processes trying to enter the semaphore
    - No priority, or priority over arrivals for mutex?
    - Process may lose mutex at any signal operation
    - But does not lose, if no one was waiting!
    - Problem, if critical section would continue over SignalC

ESW-Priorities in Monitors

- Another way to describe signal/wait semantics
  - Instead of fifo, signal-and-continue, signal-and-wait
- Processes in 3 dynamic groups
  - Priority depends on what they are doing in monitor
    - E = priority of processes entering the monitor
    - S = priority of a process signalling in SignalC
    - W = priority of a process waiting in WaitC
  - E < S < W (highest pri), i.e., IRR
    - Processes waiting in WaitC have highest priority
    - Entering new process have lowest priority
    - IRR = immediate resumption requirement
    - Signal and urgent wait
    - Classical, usual semantics
    - New arrivals can not starve those inside
Problem with/without IRR

- No IRR, e.g., E=S=W or E<W<S
  - Process P waits in WaitC()
  - Process P released from WaitC, but is not executed right away
  - Waits in monitor mutex (semaphore?)
  - Signaller or some other process changes the state that P was waiting for
  - P is executed in wrong state
- IRR
  - Signalling process may lose mutex!

Algorithm 7.2: Semaphore simulated with a monitor

No immediate resumption requirement, \( E = S = W \)

<table>
<thead>
<tr>
<th>Monitor Sem</th>
<th>P</th>
<th>Q1, Q2</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer ( s )</td>
<td>-1</td>
<td>operation wait</td>
<td></td>
</tr>
<tr>
<td>if ( s = 0 )</td>
<td>waitC(notZero)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( s )</td>
<td>( s = s + 1 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>signalC(notZero)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop forever</td>
<td>non-critical section</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p1: Som.wait</td>
<td>q1: Sem.wait</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p2: Som.signal</td>
<td>q2: Sem.signal</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Algorithm 7.3: Producer-consumer (finite buffer, monitor)

No IRR semantics (important assumption)

<table>
<thead>
<tr>
<th>Producer</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>datatype D</td>
<td>datatype D</td>
</tr>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>p1: P ← produce</td>
<td>q1: D ← PC.take</td>
</tr>
<tr>
<td>p2: PC.append(D)</td>
<td></td>
</tr>
<tr>
<td>buffer_hider, synchronization hidden (easy-to-write code)</td>
<td></td>
</tr>
</tbody>
</table>

Concurrent Programming (RIO)
Discussion

- Look at previous slide, Alg. 7.3
- Assume now: no IRR
  - What does it mean?
  - Do you need to change the code? How?
  - Changes in monitor ("server")?
  - Changes in producer/consumer ("clients")?
  - Will it work with multiple producers/consumers?
  - Exactly where can any producer/consumer process be suspended?

Other Monitor Internal Operations

- Empty(CV)
  - Returns TRUE, iff CV-queue is empty
  - Might do something else than wait for your turn ....
- Wait(CV, rank)
  - Priority queue, release in priority order
  - Small rank number, high priority
- MinRank(CV)
  - Return rank for first waiting process (or 0 or whatever?)
- Signal_all(CV)
  - Wake up everyone waiting

Changes in monitor ("server")?
Changes in producer/consumer ("clients")?

Will it work with multiple producers/consumers?
Exactly where can any producer/consumer process be suspended?

Algorithm 7.3: Readers and writers with a monitor

- monitor RW
  - integer readers = 0
  - integer writers = 0
  - condition OKtoRead, OKtoWrite
  - operation StartRead
    - wait(OKtoRead)
    - writers ≠ 0 or not empty(OKtoWrite)
    - wait(OKtoWrite)
    - readers ← readers + 1
    - signal(OKtoRead)
  - operation EndRead
    - readers ← readers - 1
    - if readers = 0
    - signal(OKtoWrite)
    - else signal(OKtoRead)

- operation StartWrite
  - writers ≠ 0 or readers ≠ 0
  - wait(OKtoWrite)
  - wait(OKtoRead)
  - writers ← writers + 1
  - operation EndWrite
    - writers ← writers - 1
    - if empty(OKtoRead)
    - then signal(OKtoWrite)
    - else signal(OKtoRead)

- If writer finishing, and 1 writer and 2 readers waiting, who is next?
- 3 processes waiting in OKtoRead. Who is next?
- 3 processes waiting in OKtoWrite. Who is next?
- If writer finishing, and 1 writer and 2 readers waiting, who is next?

Algorithm 7.4: Readers and writers with a monitor

- monitor RW
  - integer readers = 0
  - integer writers = 0
  - condition OKtoRead, OKtoWrite
  - operation StartRead
    - wait(OKtoRead)
    - writers ≠ 0 or not empty(OKtoWrite)
    - wait(OKtoWrite)
    - readers ← readers + 1
    - signal(OKtoRead)
  - operation EndRead
    - readers ← readers - 1
    - if readers = 0
    - signal(OKtoWrite)
    - else signal(OKtoRead)

- operation StartWrite
  - writers ≠ 0 or readers ≠ 0
  - wait(OKtoWrite)
  - wait(OKtoRead)
  - writers ← writers + 1
  - operation EndWrite
    - writers ← writers - 1
    - if empty(OKtoRead)
    - then signal(OKtoWrite)
    - else signal(OKtoRead)

- If writer finishing, and 1 writer and 2 readers waiting, who is next?
- 3 processes waiting in OKtoRead. Who is next?
- 3 processes waiting in OKtoWrite. Who is next?
- If writer finishing, and 1 writer and 2 readers waiting, who is next?

Algorithm 7.5: Dining philosophers with a monitor

- monitor ForkMonitor
  - integer array[0,4] fork = [2, ..., 2]
  - condition array[0,4] OKtoEat
  - operation takeFork(integer i)
    - if fork[i] ≠ 2
    - wait(OKtoEat[i])
    - fork[i] = fork[i] + 1
    - fork[i-1] = fork[i-1] + 1
    - operation releaseFork(integer i)
    - fork[i] = fork[i] + 1
    - fork[i-1] = fork[i-1] + 1
  - Both at once!

- Is order important?
- Signalizing semantics? IRQ -> mutex will break here!
- When executed?
- Much later? Semantics?
- What changes were needed, if E=S=W semantics were used?

Algorithm 7.6: Readers and writers with a monitor

- Algorithm 7.4: Readers and writers with a monitor
  - Compare to Lesson 7, slide 26

- Algorithm 7.5: Dining philosophers with a monitor
  - Compare to Lesson 7, slide 27

BACI Monitors

- When executed?
- Much later? Semantics?
- What changes were needed, if E=S=W semantics were used?

- Operation: StartRead
  - No need for counts dr, dw

- Operation: StartWrite
  - No need for counts dr, dw

- Operation: EndRead
  - No need for counts dr, dw

- Operation: EndWrite
  - No need for counts dr, dw

- Algorithm 7.6: Readers and writers with a monitor
  - Compare to Lesson 7, slide 27

- Algorithm 7.7: Dining philosophers with a monitor
  - Compare to Lesson 7, slide 28
Concurrent Programming (RIO)

Lecture 8: Monitors

Readers and Writers in C:

```c
#include <stdio.h>

#define MAXREADERS 5
#define MAXWRITERS 5

int readers = 0, writing = 0;
condition OKtoRead, OKtoWrite;

void StartRead() {
    if (writing || (empty(OKtoRead)))
        wait(OKtoRead);
    readers = readers + 1;
    signal(OKtoWrite);
}

void EndRead() {
    readers = readers - 1;
    if (readers == 0)
        signal(OKtoRead);
    RW.EndRead();
}

void StartWrite() {
    if (writing || (readers != 0))
        wait(OKtoWrite);
    writing = 1;
    RW.StartWrite();
    ... write data base ... RW.EndWrite();
    signal(OKtoRead);
}
```

Java Monitors

- No real support
- Emulate monitor with normal object with all methods synchronized
- Emulate monitor condition variables operations with Java wait(), notifyAll(), and try/catch.
  - Generic wait-operation
  - “E = W < S” signal semantics
    - No IRR, use while-loops
  - notifyAll() will wake-up all waiting processes
    - Must check the conditions again
    - No order guaranteed – starvation is possible

PlusMinus with Java Monitor

- Simple Java solution with monitor-like code
  - PlusMinus_mon.java
  - Better: make data structures visible only to "monitor" methods?

Monitor Summary

+ Automatic Mutex
+ Hides complexities from monitor user
- Internal synchronization with semantically complex condition variables
  - With IRR semantics, try to place signalC at the end of the method
  - Without IRR, mutex ends with signalC
- Does not allow for any concurrency inside monitor

- Monitor should be used only to control concurrency
- Actual work should be done outside the monitor
Protected Objects

- Like monitor, but condition variable definitions implicit and coupled with `when-expression` on which to wait
  - Automatic mutex control for operations (as in monitor)
- **Barrier**, fifo queue
  - Evaluated only (always!) when some operation terminates within mutex
  - Signaller exiting
  - Implicit signalling
  - Do not confuse with barrier synchronization!

```
operation StartWrite when not writing and readers = 0
writing := true
```

Algorithm 7.6: Readers and writers with a protected object

```
protected object RW

integer readers := 0
boolean writing := false

procedure StartWrite when not writing

begin
  readers := readers + 1;
  writing := true;
  if writing then
    readers := readers - 1;
  end if

procedure EndWrite

begin
  writing := false;

procedure StartRead

begin
  if writing then
    writers := writers - 1;
  end if

procedure EndRead

begin

```

Summary

- **Monitors**
  - Automatic mutex, no concurrent work inside monitor
  - Need concurrency – do actual work outside monitor
  - Internal synchronization with condition variables
    - Similar but different to semaphores
    - Signalling semantics varies
    - No need for shared memory areas
    - Enough to invoke monitor methods in (prog. lang.)library
- **Protected Objects**
  - Avoids some problems with monitors
    - Automatic mutex and signalling
    - Can signal only at the end of method
    - Wait only in barrier at the beginning of method
    - No mutex breaks in the middle of method
    - Barrier evaluation may be costly
    - No concurrent work inside protected object
    - Need concurrency – do actual work outside protected object