Monitors

Ch 7 [BenA 06]

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 by Monitor:
  - 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, …

Monitor (Hoare 1974)

- Encapsulated data and operations for it
  - Abstract data type, object
  - Public methods are the only way to manipulate data
  - Public 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 usually 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 just for synchroniz.
  - 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 or losing execution turn when signaling to 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

```
monitor CS
    integer n = 0
    operation increment
    integer temp
    temp = n
    n = temp + 1
    p: CS.increment
    q: CS.increment
```

- Automatic mutex solution
  - Solution with busy-wait, disable interrupts, or suspension!
  - Internal to monitor, user has no handle on it, might be useful to know
    - Only one procedure active at a time – which one?
  - No ordered queue to enter monitor
  - Starvation is possible, if many processes continuously trying to get in

Monitor Condition Variables

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

(Fig. 5.15 [Stal05])
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 SignalC(CV)**
  - 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 SignalC(CV)**
  - 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 signaling semantics
  - Define priority order for monitor mutex
  - 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 (1/3)

<table>
<thead>
<tr>
<th>( p )</th>
<th>( q )</th>
<th>( p )</th>
<th>( q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>non-critical section</td>
<td>loop forever</td>
<td>non-critical section</td>
</tr>
<tr>
<td>( p_1 ): Sem.wait</td>
<td>( q_1 ): Sem.wait</td>
<td>( p_2 ): Sem.signal</td>
<td>( q_2 ): Sem.signal</td>
</tr>
</tbody>
</table>

### Algorithm 7.2: Semaphore simulated with a monitor (2/3)

<table>
<thead>
<tr>
<th>( p )</th>
<th>( q )</th>
<th>( p )</th>
<th>( q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>non-critical section</td>
<td>loop forever</td>
<td>non-critical section</td>
</tr>
<tr>
<td>( p_1 ): Sem.wait</td>
<td>( q_1 ): Sem.wait</td>
<td>( p_2 ): Sem.signal</td>
<td>( q_2 ): Sem.signal</td>
</tr>
</tbody>
</table>

### Algorithm 7.2: Semaphore simulated with a monitor (3/3)

<table>
<thead>
<tr>
<th>( p )</th>
<th>( q )</th>
<th>( p )</th>
<th>( q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>non-critical section</td>
<td>loop forever</td>
<td>non-critical section</td>
</tr>
<tr>
<td>( p_1 ): Sem.wait</td>
<td>( q_1 ): Sem.wait</td>
<td>( p_2 ): Sem.signal</td>
<td>( q_2 ): Sem.signal</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>producer</th>
<th>consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>producer</td>
<td>consumer</td>
</tr>
</tbody>
</table>

buffer hidden, synchronization hidden (easy-to-write code)
### Better Producer/Consumer Monitor

- No work in monitor

**Producer (size N buffer)**

- Loop forever
  - \( D \leftarrow \text{Produce} \)
  - \( \text{PC.append} \_\text{ok}(N) \)
  - \( \text{append} \_\text{tail(buffer,D)} \)
  - \( \text{PC.append} \_\text{done}() \)

**Consumer**

- Loop forever
  - \( \text{PC.take} \_\text{ok}() \)
  - \( D \leftarrow \text{head} \_\text{buffer} \)
  - \( \text{PC.take} \_\text{done}() \)
  - \( \text{consume}(D) \)

**Monitor PC**

- \( \text{buf} \_\text{cnt} = 0 \)
- \( \text{condition notEmpty, notFull} \)

- **Operation** \( \text{append} \_\text{ok}(N) \)
  - if \( \text{buf} \_\text{cnt} = N \)
    - \( \text{waitC(notFull)} \)
  - \( \text{buf} \_\text{cnt}++ \)

- **Operation** \( \text{append} \_\text{done}() \)
  - \( \text{signalC(notEmpty)} \)

- **Operation** \( \text{take} \_\text{ok}() \)
  - if \( \text{buf} \_\text{cnt} = 0 \)
    - \( \text{waitC(notEmpty)} \)
    - \( \text{buf} \_\text{cnt}-- \)

- **Operation** \( \text{take} \_\text{done}() \)
  - \( \text{signalC(notFull)} \)

---

### Other Monitor Internal Operations

- **Empty(CV)**
  - Returns TRUE, if 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 (e.g.)

- **Minrank(CV)**
  - Return rank for first waiting process (or 0 or whatever?)

- **Signal_all(CV)**
  - Wake up everyone waiting
  - If IRR, who gets mutex turn? Highest rank?
  - 1st in queue? Last in queue?

---

### Readers and Writers with Monitor

**Readers**

- Many can read concurrently
- No writers allowed with readers

**Writers**

- Only one can write at a time
- No readers allowed at that time

**Data base**

- \( \text{read()} \)
- \( \text{write()} \)

**Monitor to control access to database**

**reader**

- \( \text{p1: RWStartRead} \)
- \( \text{p2: read the database} \)
- \( \text{p3: RWEndRead} \)

**writer**

- \( \text{q1: RWStartWrite} \)
- \( \text{q2: write to the database} \)
- \( \text{q3: RWEndWrite} \)

---

### Algorithm 7.4: Readers and writers with a monitor

**monitor RW**

- integer \( \text{readers} = 0 \)
- integer \( \text{writers} = 0 \)
- condition \( \text{OKtoRead, OKtoWrite} \)

**operation StartRead**

- \( \text{writers} = 0 \) or not empty(OKtoWrite)
- \( \text{waitC(OKtoRead)} \)
- \( \text{readers} = \text{readers} + 1 \)
- \( \text{signalC(OKtoRead)} \)

**operation EndRead**

- \( \text{readers} = \text{readers} - 1 \)
- if \( \text{readers} = 0 \)
    - \( \text{signalC(OKtoWrite)} \)
    - else \( \text{signalC(OKtoRead)} \)

**monitor RW**

- \( \text{writers} = 0 \) or not readers \( \neq 0 \)
- \( \text{waitC(OKtoWrite)} \)
- \( \text{writers} = \text{writers} + 1 \)

**operation StartWrite**

- \( \text{writers} = 0 \) or readers \( \neq 0 \)
- \( \text{waitC(OKtoRead)} \)

**operation EndWrite**

- \( \text{writers} = \text{writers} - 1 \)
- if empty(OKtoRead)
  - then signalC(OKtoWrite)
  - else signalC(OKtoRead)

- 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 \([0..4]\) \( \text{fork} = [2, 2, 2, 2, 2] \)
- condition array \([0..4]\) \( \text{OKtoEat} \)

**operation takeFork(integer i)**

- if \( \text{fork}[i] \neq 2 \)
  - \( \text{waitC(OKtoEat[i])} \)
  - \( \text{OKtoEat[i]} = 0 \)
- \( \text{fork}[i] = \text{fork}[i] - 1 \)

**operation releaseFork(integer i)**

- if \( \text{fork}[i] = 0 \)
  - \( \text{signalC(OKtoEat[i])} \)

**Signaling semantics? IRR — mutexes will break here!**

**Deadlock free? Why? Starvation possible.**

**Is order Important?**

- When executed? Much later? Semantics?

**What changes were needed, if E=S=W semantics were used?**

---

### BACI Monitors

**wait**

- IRR
- Queue not FIFO
- Baton passing

**Also**

- \( \text{wait()} \) with priority
  - \( \text{waitC(OKtoWrite, 1)} \)
- Default priority = 10 (big number, high priority ??)
Readers and Writers in C:

```c
void StartWrite() {
    if (writing || (readers != 0)) {
        wait(OKtoRead);
        writing = 1;
        signal(OKtoWrite);
    } else {
        signal(OKtoRead);
    }
}

RW.StartRead();
... read data base ...
RW.EndRead();

RW.StartWrite();
... write data base ...
RW.EndWrite();

readers have priority, writer may starve
```

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

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
    - With 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

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
    - With 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 is exiting
  - Implicit signalling
  - Do not confuse with barrier synchronization!

Algorithm 7.6: Readers and writers with a protected object

```
protected object RW
  integer readers := 0;
  boolean writing := false;
  procedure StartRead when not writing
    readers := readers + 1;
    when (writing) then
      begin
        writing := true;
        end
  operation EndRead
    readers := readers - 1;
  operation StartWrite when not writing and readers = 0
    writing := true;
```

```
reader
  loop forever
    RW StartRead
  read the database
  RW EndRead
```

```
writer
  loop forever
    RW StartWrite
  write to the database
  RW EndWrite
```

```
E-W semantics
```

- What if many barriers become true? Which one resumes?

**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 – all tested with every signal?
  - No concurrent work inside protected object
  - Need concurrency – do actual work outside protected object