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 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
  - 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 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
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?
    - Allows process that lost mutex in SignalC 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**  \( \text{SignalC} (CV) \)
  - 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 awaked process resumes (becomes active again)
    - No priority or priority over arrivals for sem. mutex?

- **Signal and Wait**  \( \text{SignalC} (CV) \)
  - 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

Algorithm 7.2: Semaphore simulated with a monitor

```
monitor Sem
integer s ← 1 (mutex sem)
condition notZero
operation wait
  if s = 0
    waitC(notZero)
    s ← s - 1
operation signal
  s ← s + 1
  signalC(notZero)
```

Semaphore counter kept separately, initialized before any process active.

What if signalC comes first?

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>non-critical section</td>
<td>non-critical section</td>
</tr>
<tr>
<td>p1: Sem.wait</td>
<td>q1: Sem.wait</td>
</tr>
<tr>
<td>critical section</td>
<td>critical section</td>
</tr>
<tr>
<td>p2: Sem.signal</td>
<td>q2: Sem.signal</td>
</tr>
</tbody>
</table>
**Problem with/without IRR**

- No IRR, e.g., \( E = S = W \) or \( E < W < S \)
  - Process \( P \) waits in \( \text{WaitC}() \)
  - Process \( P \) released from \( \text{WaitC} \), 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**

- Initial state: \( s = 0 \)
- \( P \) and \( Q_1 \) compete, \( Q_1 \) wins, \( Q_1 \) enters \( CS \), \( s = 0 \), \( P \) waits
- \( Q_2 \) gets in, finds \( s = 1 \), sets \( s = 0 \), enters \( CS \)
- \( P \) waits for mutex here
- FIX: must test for condition again

---

**Table:**

<table>
<thead>
<tr>
<th>P</th>
<th>p</th>
<th>Q1, Q2</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>non-critical section</td>
<td>loop forever</td>
</tr>
<tr>
<td>q1: Sem.wait</td>
<td>critical section</td>
<td>q2: Sem.signal</td>
</tr>
</tbody>
</table>
Algorithm 7.2: Semaphore simulated with a monitor

<table>
<thead>
<tr>
<th>Monitor Sem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer s ← 1</td>
</tr>
<tr>
<td>Condition notZero</td>
</tr>
<tr>
<td>Operation wait</td>
</tr>
<tr>
<td>If s = 0</td>
</tr>
<tr>
<td>waitC(notZero)</td>
</tr>
<tr>
<td>s ← s − 1</td>
</tr>
<tr>
<td>Operation signal</td>
</tr>
<tr>
<td>s ← s + 1</td>
</tr>
<tr>
<td>signalC(notZero)</td>
</tr>
</tbody>
</table>

No immediate resumption requirement, E = S = W

a) P & Q1 compete, Q1 wins, Q1 enters CS, s=0, P waits

b) Q1 signals P, s=1
c) P waits for mutex here
d) Q2 gets in, finds s=1, sets s=0, enters CS
e) P advances, sets s = -1,

P          Q1, Q2          q
| loop forever |
| non-critical section |
| p1: Sem.wait |
| critical section |
| p2: Sem.signal |

P          Q1, Q2          q
| loop forever |
| non-critical section |
| q1: Sem.wait |
| critical section |
| q2: Sem.signal |

FIX: must test for condition again
Algorithm 7.2: Semaphore simulated with a monitor

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

- a) P & Q1 compete, Q1 wins, Q1 enters CS, \( s=0 \), P waits
- b) Q1 signals P, \( s=1 \)
- c) P waits for mutex here
- d) Q2 gets in, finds \( s=1 \), sets \( s=0 \), enters CS
- e) P advances, sets \( s=-1 \), enters CS

\[\begin{array}{c|cc}
\text{P} & \text{Q1, Q2} & \text{q} \\
\hline
\text{loop forever} & \text{loop forever} & \\
\text{non-critical section} & \text{non-critical section} & \\
p1: & \text{Sem.wait} & q1: \text{Sem.wait} \\
& \text{critical section} & \text{critical section} \\
p2: & \text{Sem.signal} & q2: \text{Sem.signal} \\
\end{array}\]

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

**Producer**
- void append_tail()
- datatype D
- loop forever
- p1: D ← produce
- p2: PC.append(D)

buffer hidden, synchronization hidden
(easy-to-write code)

**Consumer**
- datatype D
- loop forever
- q1: D ← PC.take
- q2: consume(D)

Discuss
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
    - 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

<table>
<thead>
<tr>
<th>reader</th>
<th>writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: RW.StartRead</td>
<td>q1: RW.StartWrite</td>
</tr>
<tr>
<td>p2: read the database</td>
<td>q2: write to the database</td>
</tr>
<tr>
<td>p3: RW.EndRead</td>
<td>q3: RW.EndWrite</td>
</tr>
</tbody>
</table>

outside monitor!
Algorithm 7.4: Readers and writers with a monitor

\[
\text{monitor RW}
\]
\[
\begin{align*}
\text{integer readers} & \leftarrow 0 \\
\text{integer writers} & \leftarrow 0 \\
\text{condition OKtoRead, OKtoWrite}
\end{align*}
\]

\[
\begin{array}{ll}
\text{operation StartRead} & \\
\text{if writers} \neq 0 \text{ or not empty(OKtoWrite)} & \\
\text{waitC(OKtoWrite)} & \\
\text{readers} & \leftarrow \text{readers} + 1 \\
\text{signalC(OKtoRead)} & \\
\end{array}
\]

\[
\begin{array}{ll}
\text{operation EndRead} & \\
\text{readers} & \leftarrow \text{readers} - 1 \\
\text{if readers} = 0 & \\
\text{signalC(OKtoWrite)} & \\
\end{array}
\]

- 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

\[
\text{monitor ForkMonitor}
\]
\[
\begin{align*}
\text{integer array[0..4] fork} & \leftarrow [2, \ldots, 2] \\
\text{condition array[0..4] OKtoEat} & \\
\text{operation takeForks(integer i)} & \\
\text{if fork[i] \neq 2} & \\
\text{waitC(OKtoEat[i])} & \\
\text{fork[i+1]} & \leftarrow \text{fork[i+1]} - 1 \\
\text{fork[i-1]} & \leftarrow \text{fork[i-1]} - 1 \\
\end{align*}
\]

\[
\begin{align*}
\text{operation releaseForks(integer i)} & \\
\text{fork[i+1]} & \leftarrow \text{fork[i+1]} + 1 \\
\text{fork[i-1]} & \leftarrow \text{fork[i-1]} + 1 \\
\text{if fork[i+1] = 2} & \\
\text{signalC(OKtoEat[i+1])} & \\
\text{if fork[i-1] = 2} & \\
\text{signalC(OKtoEat[i-1])} & \\
\end{align*}
\]

- Number of forks available to philosopher i
  - Both at once!
- Signaling semantics? IRR \rightarrow mutex will break here!
- When executed? Much later? Semantics?

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

- waitc
  - IRR
  - Queue not FIFO
  - Baton passing

- Also
  - `waitc()` with priority:
    ```
    waitc (OKtoWrite, 1);
    ```
  - Default priority = 10 (big number, high priority ?)

---

```c
monitor RW {
  int readers = 0, writing = 0; (typo fix)
  condition OKtoRead, OKtoWrite;

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

---

```c
void StartWrite() {
  if (writing || (readers != 0))
    waitc(OKtoWrite);
  writing = 1;
}

void EndWrite() {
  writing = 0;
  if (empty(OKtoRead))
    signalc(OKtoWrite);
  else
    signalc(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

```java
class PCMMonitor {
    final int N = 5;
    int Oldest = 0, Newest = 0;
    volatile int Count = 0;
    int Buffer[] = new int[N];

    synchronized void Append(int V) {
        while (Count == N)
            try {
                wait();
            } catch (InterruptedException e) {} // <---
        Buffer[N] = V;
        Newest = (Newest + 1) % N;
        Count = Count + 1;
        notifyAll();
    }

    synchronized int Take() {
        int temp;
        while (Count == 0)
            try {
                wait();
            } catch (InterruptedException e) {} // <---
        temp = Buffer[Oldest];
        Oldest = (Oldest + 1) % N;
        Count = Count - 1;
        notifyAll();
        return temp;
    }
}
```
PlusMinus with Java Monitor

- Simple Java solution with monitor-like code
  - Plusminus_mon.java

```
vera: javac Plusminus_mon.java
vera: java Plusminus_mon
```

- 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!

```
condition OKtoWrite;
void StartWrite() {
    if (writing || (readers != 0))
        wait(OKtoWrite);
    writing = 1;
}  
```

(operation StartWrite when not writing and readers = 0
(protected object)}
Algorithm 7.6: Readers and writers with a protected object

---

protected object RW

integer readers ← 0

boolean writing ← false

operation StartRead when not writing

readers ← readers + 1

operation EndRead

readers ← readers − 1

operation StartWrite when not writing and readers = 0

writing ← true

operation EndWrite

writing ← false

---

 Mutex semantics?

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

---

Readers and Writers as ADA Protected Object

Continuous flow of readers will starve writers.

How would you change it to give writers priority?
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