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, …

Semaphore problems
• forget P or V
• extra P or V
• wrong semaphore
• forget to use mutex
• used for mutex and for synchronization
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
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?
    - 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

Discuss
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
- Classical, usual semantics
  - New arrivals cannot 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)

p | q
--|--
loop forever | loop forever
| non-critical section | non-critical section
| Sem.wait | Sem.wait
| critical section | critical section
| Sem.signal | Sem.signal
```

No need for “if anybody waiting…”

What if signalC comes first?
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 (2)

monitor Sem
  integer s ← 1
  condition notZero
  operation wait
    while (s = 0)
      waitC(notZero)
      s ← s - 1
  operation signal
    s ← s + 1
    signalC(notZero)

P, P & Q1 compete, Q1 wins, Q1 enters CS, s=0,
P waits
Q1 signals P, s=1
P waits for mutex here
Q2 gets in, finds s=1, sets s=0, enters CS
P advances, sets s = -1, enters CS

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>p1: Sem.wait</td>
<td>critical section</td>
<td>q1: Sem.wait</td>
</tr>
<tr>
<td>p2: Sem.signal</td>
<td></td>
<td>q2: Sem.signal</td>
</tr>
</tbody>
</table>

Discuss
Algorithm 7.2: Semaphore simulated with a monitor

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

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

<table>
<thead>
<tr>
<th>P</th>
<th>Q1, Q2</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>

b) Q1 signals P, \( s = 1 \)
e) P advances, sets \( s = -1 \), enters CS

d) Q2 gets in, finds \( s = 1 \), sets \( s = 0 \), enters CS

c) P waits for mutex here

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

<table>
<thead>
<tr>
<th>P</th>
<th>p</th>
<th>Q1, Q2</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>p1: Sem.wait</td>
<td>critical section</td>
<td>q1: Sem.wait</td>
<td>critical section</td>
</tr>
<tr>
<td>p2: Sem.signal</td>
<td></td>
<td>q2: Sem.signal</td>
<td></td>
</tr>
</tbody>
</table>

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

- IRR semantics (important assumption)
- producer
  - void append_tail()
  - bufferType head()
  - datatype D
  - loop forever
  - p1: D ← produce
  - p2: PC.append(D)
- consumer
  - datatype D
  - loop forever
  - q1: D ← PC.take
  - q2: consume(D)

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

internal procedures in monitor, no waitC in them (important design feature)

Discuss
Better Producer/Consumer Monitor

- No work in monitor

**Producer** (size N buffer)

- Loop forever
  - D <- Produce
  - PC.append_ok(N)
  - append_tail(buffer, D)
  - PC_append_done()

**Consumer**

- Loop forever
  - PC.take_ok()
  - D <- head(buffer)
  - PC_take_done()
  - consume(D)

**Monitor PC**

```
Int buf_cnt = 0;
condition notEmpty, notFull;

Operation append_ok(N)
  if (buf_cnt==N)
    waitC(notFull)
    buf_cnt++;

Operation append_done()
  signalC(notEmpty)

Operation take_ok()
  if (buf_cnt==0)
    waitC(notEmpty)
    buf_cnt--;

Operation take_done()
  signalC(notFull)
```

**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 (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

---

Algorithm 7.4: Readers and writers with a monitor

```plaintext
monitor RW
  integer readers ← 0
  integer writers ← 0
  condition OKtoRead, OKtoWrite

  operation StartRead
    if writers ≠ 0 or not empty(OKtoWrite)
      waitC(OKtoRead)
    readers ← readers + 1
    signalC(OKtoRead)

  operation EndRead
    readers ← readers - 1
    if readers = 0
      signalC(OKtoWrite)

  operation StartWrite
    if writers ≠ 0 or readers ≠ 0
      waitC(OKtoWrite)
    writers ← writers + 1

  operation EndWrite
    writers ← 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

```plaintext
monitor ForkMonitor

integer array[0..4] fork [2, ..., 2]
condition array[0..4] OKtoEat
operation takeForks(integer i)
if fork[i] ≠ 2
    waitC(OKtoEat[i]) ← IRR?
fork[i+1] ← fork[i+1] + 1
fork[i-1] ← fork[i-1] + 1

operation releaseForks(integer i)
fork[i+1] ← fork[i+1] - 1
fork[i-1] ← fork[i-1] - 1

philosopher i
loop forever
p1: think
p2: takeForks(i)
p3: eat
p4: releaseForks(i)
```

Number of forks available to philosopher i

Both at once!


Is order Important?

When executed?

Much later? Semantics?

Signaling semantics?

IRR → mutex will break here!

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

---

BACI Monitors

- `waitc`
  - IRR
  - Queue not FIFO
  - Baton passing

- Also
  - `waitc()` with priority: `waitc(OKtoWrite, 1)`
  - Default priority = 10 (big number, high priority ??)
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
    ```java
    vera: javac Plusminus_mon.java
    vera: java Plusminus_mon
    ```
    - Better: make data structures visible only to "monitor" methods?

```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)
      wait();

    try {
      Buffer[Oldest] = V;
      Newest = (Newest + 1) % N;
      Count = Count + 1;
    } catch (InterruptedException e) {}
    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;
}
```
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
    - **signal** 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

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

```ada
protected RW is
  entry StartRead;
  procedure EndRead;
  entry StartWrite;
  procedure EndWrite;
private
  Readers: Natural := 0;
  Writing: Boolean := false;
end RW;

protected body RW is
  entry StartRead
    when not Writing is
    begin
      Readers := Readers + 1;
      EndStartRead;
  end StartRead;

  procedure EndRead is
  begin
    Readers := Readers - 1;
    EndEndRead;
  end EndRead;

  entry StartWrite
    when not Writing and Readers = 0 is
    begin
      Writing := true;
      EndStartWrite;
  end StartWrite;

  procedure EndWrite is
  begin
    Writing := false;
    EndEndWrite;
  end EndWrite;
end RW;
```

**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