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 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)

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 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
    - 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

<table>
<thead>
<tr>
<th>p: CS.increment</th>
<th>q: CS.increment</th>
</tr>
</thead>
</table>
| 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

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

(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 - signalling process gets mutex
- If no one was waiting, **signal is lost** (no memory)
  - Allows process that lost mutex in SignalC to proceed again?

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 awaked 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
Lecture 8: Monitors

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

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

loop forever
if not p1
    q1
else
    q2

p1: Som.wait
q1: Som.signal
p2: Som.signal
q2: Som.signal
```

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

```
monitor PC
bufferType buffer = empty
condition notEmpty
condition notFull
void append_tail(bufferType head)
operation append(dataType V)
    buffer = full
    waitC(notFull)
    V = head(buffer)
    sign = 1
    operation take
    dataType W
    buffer is empty
    waitC(notEmpty)
    head(buffer) = W
    W = head(buffer)
    signalC(notFull)
```

buffer hidden, synchronization hidden (easy to write code)
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

BACI Monitors

- **wait**
  - IRR
  - Queue not FIFO
  - Baton passing

- **Also**
  - wait() with priority: 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 IR, use while-loops
  - notifyAll() will wake-up all waiting processes
    - Must check the conditions again
    - No order guaranteed – starvation is possible

Java Monitors

```java
class PCMonitor {
    final int N = 5;
    int oldest = 0, newest = 0;
    volatile int count = 0;
    int buffer[] = new int[N];
    synchronized void append(int v) {
        try {
            temp = buffer[oldest];
        } catch (InterruptedException e) {
        }
        buffer[oldest] = v;
        newest = (newest + 1) % N;
        count = count + 1;
        notifyAll();
    }
}
```

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

```java
protected
void StartWrite() {
    if (writing || readers > 0) {
        writeOkToWrite;
        writing = 1;
    }
}
```

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

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

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

Monitor Summary

- Automatic Mutex
Mutex 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