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)

- **Elliot**
- **Algol-60**
- **Sir Charles**

- 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
**Algorithm 7.1: Atomicity of monitor operations**

```plaintext
monitor CS
  integer n ← 0

  operation increment
  integer temp
  temp ← n
  n ← temp + 1

  p1: 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)
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** $SignalC( CV )$
  - 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?
Signaling Semantics

- **Signal and Wait**  \textit{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
    - \textit{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)

<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>
```

Semaphore counter kept separately, initialized before any process active.

No need for "if anybody waiting...". What if `signalC` comes 1st?
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 (2)

No immediate resumption requirement, E = S = W

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

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

FIX: must test for condition again

<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>

Discuss
Algorithm 7.2: Semaphore simulated with a monitor (1/3)

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

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

- **a)** \( P \) & \( Q_1 \) compete, \( Q_1 \) wins,
  \( Q_1 \) enters CS, \( s=0 \),
  \( P \) waits

- **b)** \( Q_1 \) signals \( P \), \( s=1 \)

- **c)** \( P \) waits for mutex here

- **d)** \( Q_2 \) gets in,
  finds \( s=1 \),
  sets \( s=0 \),
  enters CS

- **e)** \( P \) advances,
  sets \( s = -1 \),
  enters CS

<table>
<thead>
<tr>
<th></th>
<th>( P )</th>
<th>( P )</th>
<th>Q1, Q2</th>
<th>( q )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>loop forever</td>
<td>non-critical section</td>
<td>loop forever</td>
<td>non-critical section</td>
</tr>
<tr>
<td>p1:</td>
<td>Sem.wait</td>
<td>critical section</td>
<td>q1:</td>
<td>Sem.wait</td>
</tr>
<tr>
<td>p2:</td>
<td>Sem.signal</td>
<td></td>
<td>q2:</td>
<td>Sem.signal</td>
</tr>
</tbody>
</table>
Algorithm 7.2: Semaphore simulated with a monitor (2/3)

No immediate resumption requirement, $E = S = W$

```
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>$p$</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>

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

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

178.2.2012 Copyright Teemu Kerola 2012

---

**monitor Sem**
- integer s ← 1
- condition notZero
- operation wait
  - while (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>

---

- 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

---

FIX: must test for condition again

---

No immediate resumption requirement, E = S = W
Algorithm 7.3: Producer-consumer (finite buffer, monitor)

monitor PC
bufferType buffer ← empty
condition notEmpty
condition notFull

operation append(datatype V)
  if buffer is full
    waitC(notFull)
  append_tail(V, buffer) ; typo in book
  signalC(notEmpty)

operation take()
  datatype W
  if buffer is empty
    waitC(notEmpty)
  W ← head(buffer)
  signalC(notFull)
  return W

producer
  datatype D
  loop forever
  p1: D ← produce
  p2: PC.append(D)

consumer
  datatype D
  loop forever
  q1: D ← PC.take
  q2: consume(D)

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

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

IRR semantics (important assumption)
Better Producer/Consumer Monitor

- No work in monitor

### Producer (size \( N \) buffer)

- Loop forever
  - \( D \leftarrow \text{Produce} \)
  - \( \text{PC.append\_ok}(N) \)
  - \( \text{append\_tail}(\text{buffer, } D) \)
  - \( \text{PC-append\_done()} \)

### Consumer

- Loop forever
  - \( \text{PC.take\_ok()} \)
  - \( D \leftarrow \text{head}(\text{buffer}) \)
  - \( \text{PC-take\_done()} \)
  - \( \text{consume}(D) \)

### Monitor PC

```plaintext
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)
```

8.2.2012 Copyright Teemu Kerola 2012
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

---

**Reader**

<table>
<thead>
<tr>
<th>p1:</th>
<th><code>RW.StartRead</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>p2:</td>
<td><code>read the database</code></td>
</tr>
<tr>
<td>p3:</td>
<td><code>RW.EndRead</code></td>
</tr>
</tbody>
</table>

**Writer**

<table>
<thead>
<tr>
<th>q1:</th>
<th><code>RW.StartWrite</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>q2:</td>
<td><code>write to the database</code></td>
</tr>
<tr>
<td>q3:</td>
<td><code>RW.EndWrite</code></td>
</tr>
</tbody>
</table>

---

Monitor to control access to database

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

```
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])
    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
    if fork[i+1] = 2
        signalC(OKtoEat[i+1])
    if fork[i−1] = 2
        signalC(OKtoEat[i−1])
```

---

**Number of forks available to philosopher i**

**Philosopher i**

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

---

**Is order important?**

**Deadlock free? Why?**

Starvation possible.

**Signaling semantics?**

IRR → mutex will break here!

**When executed? Much later? Semantics?**

Both at once!
BACI 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))
            waitc(OKtoRead);
        readers = readers + 1;
        signalc(OKtoRead);
    }
}
```
```
monitor RW {
    int readers = 0, writing = 0; (typo fix)
    condition OKtoRead, OKtoWrite;

    void StartRead() {
        if (writing || !empty(OKtoWrite))
            waitc(OKtoRead);
        readers = readers + 1;
        signalc(OKtoRead);
    }

    void EndRead() {
        readers = readers - 1;
        if (readers == 0)
            signalc(OKtoWrite);
    }

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

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

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
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) {
        while (Count == N)
            try {
                wait();
            } catch (InterruptedException e) {}  
        Buffer[Newest] = V;
        Newest = (Newest + 1) % N;
        Count = Count + 1;
    }
}

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

  http://www.cs.helsinki.fi/u/kerola/rio/Java/examples/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!

```
condition OKtoWrite;

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

*operation* StartWrite *when not* writing *and readers = 0*

```
writing ← true
```

(monitor)
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

reader

loop forever
  RW.StartRead
  read the database
  RW.EndRead

writer

loop forever
  RW.StartWrite
  write to the database
  RW.EndWrite

• 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?

```ada
package 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;
    end StartRead;

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

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

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