Monitors

Johannes Åman Pohjola
CSIRO’s Data61 (and UNSW)
Term 2 2021
Where we are at

In the last lecture, we saw a generalisation of *locks* called a *semaphore*, with a particular analysis of the *producer consumer problem*.

In this lecture, we will look at another concurrency abstraction called a *monitor*, designed to ameliorate some of the problems with semaphores.
Main Disadvantages of Semaphores

1. **Lack of structure**: when building a large system, responsibility is diffused among implementers. Someone forgets to call `signal` \(\rightarrow\) possible deadlock.
Main Disadvantages of Semaphores

1. **Lack of structure**: when building a large system, responsibility is diffused among implementers. Someone forgets to call `signal` $\rightarrow$ possible deadlock.

2. **Global visibility**: when something goes wrong, the whole program must be inspected $\rightarrow$ deadlocks are hard to isolate.
Main Disadvantages of Semaphores

1. **Lack of structure**: when building a large system, responsibility is diffused among implementers. Someone forgets to call `signal` ⇒ possible deadlock.

2. **Global visibility**: when something goes wrong, the whole program must be inspected ⇒ deadlocks are hard to isolate.

**Solution**

*Monitors* concentrate one responsibility into a single module and encapsulate critical resources. They offer more structure than semaphores; more control than *await*. 
Monitors

History:
- In the literature: Brinch Hansen (1973) and Hoare (1974)
- languages — Concurrent Pascal (1975)... Java, Pthreads library

Definition
Monitors are a generalisation of objects (as in OOP).
- May encapsulate some private data —all fields are private
- Exposes one or more operations — akin to methods.
- Implicit mutual exclusion—each operation invocation is implicitly atomic.
- Explicit signaling and waiting through condition variables.
Our Counting Example

Algorithm 2.1: Atomicity of monitor operations

```
monitor CS
    integer n ← 0

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

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: loop ten times</td>
<td>q1: loop ten times</td>
</tr>
<tr>
<td>p2: CS.increment</td>
<td>q2: CS.increment</td>
</tr>
</tbody>
</table>
Program structure

\[
\text{monitor}_1 \ldots \text{monitor}_M \\
\text{process}_1 \ldots \text{process}_N
\]

- processes interact indirectly by using the same monitor
- processes call monitor procedures
- at most one call active in a monitor at a time — by definition
- explicit signaling using condition variables
- \textit{monitor invariant}: predicate about local state that is true when no call is active
Condition variables

**Definition**

*Condition variables* are named FIFO queues of blocked processes.
**Condition variables**

*Definition*

*Condition variables* are *named FIFO queues* of blocked processes.

Processes executing a procedure of a monitor with condition variable cv can:

- voluntarily suspend themselves using `waitC(cv)`,
- unblock the first suspended process by calling `signalC(cv)`, or
- test for emptiness of the queue: `empty(cv)`.

*Warning*

The exact semantics of these differ between implementations!
Condition variables

Definition

*Condition variables* are *named FIFO queues* of blocked processes.

Processes executing a procedure of a monitor with condition variable cv can:

- voluntarily suspend themselves using **waitC**(cv),
- unblock the first suspended process by calling **signalC**(cv), or
- test for emptiness of the queue: **empty**(cv).

Warning

The exact semantics of these differ between implementations!
Algorithm 2.2: Semaphore simulated with a monitor

**monitor** Sem

integer s ← k

**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>
State Diagram for the Semaphore Simulation

- **p1: Sem.wait**, q1: Sem.wait, 1, ⟨⟩
- **p2: Sem.signal**, q1: Sem.wait, 0, ⟨⟩
- **p2: Sem.signal**, blocked, 0, ⟨q⟩
- **p1: Sem.wait**, q2: Sem.signal, 0, ⟨⟩
- **blocked, q2: Sem.signal**
  0, ⟨p⟩
Algorithm 2.3: 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(V, buffer)
            signalC(notEmpty)
    operation take()
        datatype W
        if buffer is empty
            waitC(notEmpty)
            W ← head(buffer)
            W ← head(buffer)
        signalC(notFull)
        return W
```
Algorithm 2.3: Producer-consumer . . . (continued)

<table>
<thead>
<tr>
<th>producer</th>
<th>consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>datatype D</td>
<td>datatype D</td>
</tr>
<tr>
<td><strong>loop</strong> forever</td>
<td><strong>loop</strong> forever</td>
</tr>
<tr>
<td>p1: D ← produce</td>
<td>q1: D ← PC.take</td>
</tr>
<tr>
<td>p2: PC.append(D)</td>
<td>q2: consume(D)</td>
</tr>
</tbody>
</table>
The Immediate Resumption Requirement

**Question**: When a condition variable is signalled, who executes next? *It depends!*

Diagram showing the state of condition variables and signaling.
Signaling disciplines

Precedences:

- $S$: the signaling process
- $W$: waiting on a condition variable
- $E$: waiting on entry

Signal and Urgent Wait

In Hoare’s paper, $E < S < W$. This is also called the *immediate resumption requirement* (IRR). That is, a signalling process must wait for the signalled process to exit the monitor (or wait on a condition variable) before resuming. Signalling gives up control!

Signal and Continue

In Java, pthreads, and many other implementations, $E = W < S$. This means that signalling processes continue executing, and signalled processes await entry to the monitor at the same priority as everyone else.
Diagram for monitors

- Entry
- Queue
- Executing in monitor
- Condition variable
- Queue
- Monitor free
- SC
- SW
- Wait
- Call
- Return
Simulating Monitors in Promela 1

1 bool lock = false;
2
3 typedef Condition {
4   bool gate;
5   byte waiting;
6 }
7 inline enterMon() {
8   atomic {
9     !lock;
10    lock = true;
11   }
12 }
13
14 inline leaveMon() {
15    lock = false;
16 }
inline waitC(C) {
    atomic {
        C.waiting++;  
        lock = false; /* Exit monitor */
        C.gate; /* Wait for gate */
        lock = true; /* IRR */
        C.gate = false; /* Reset gate */
        C.waiting--;  
    }
}
Simulating Monitors in Promela 3

```c
inline signalC(C) {
    atomic {
        if
            /* Signal only if waiting */
            :: (C.waiting > 0) ->
                C.gate = true;
            !lock; /* IRR - wait for released lock */
                lock = true; /* Take lock again */
            :: else
                fi;
        
    }
}

#define emptyC(C) (C.waiting == 0)
```
Monitors in Java

An object in Java can be made to approximate a monitor with one waitset (i.e. unfair) condition variable and no immediate resumption:

- A method is made mutually exclusive using the synchronized keyword.
Monitors in Java

An object in Java can be made to approximate a monitor with one waitset (i.e. unfair) condition variable and no immediate resumption:

- A method is made mutually exclusive using the `synchronized` keyword.
- Synchronized methods of an object may call their `wait()` to suspend until `notify()` is called, analogous to condition variables.
Monitors in Java

An object in Java can be made to approximate a monitor with one waitset (i.e. unfair) condition variable and no immediate resumption:

- A method is made mutually exclusive using the `synchronized` keyword.
- Synchronized methods of an object may call their `wait()` to suspend until `notify()` is called, analogous to condition variables.
- **No immediate resumption requirement** means that waiting processes need to re-check their conditions!
Monitors in Java

An object in Java can be made to approximate a monitor with one waitset (i.e. unfair) condition variable and no immediate resumption:

- A method is made mutually exclusive using the `synchronized` keyword.
- Synchronized methods of an object may call their `wait()` to suspend until `notify()` is called, analogous to condition variables.
- **No immediate resumption requirement** means that waiting processes need to re-check their conditions!
- **No strong fairness guarantee** about wait lists, meaning any arbitrary waiting process is awoken by `notify()`.
Monitors in Java

An object in Java can be made to approximate a monitor with one waitset (i.e. unfair) condition variable and no immediate resumption:

- A method is made mutually exclusive using the `synchronized` keyword.
- Synchronized methods of an object may call their `wait()` to suspend until `notify()` is called, analogous to condition variables.
- **No immediate resumption requirement** means that waiting processes need to re-check their conditions!
- **No strong fairness guarantee** about wait lists, meaning any arbitrary waiting process is awoken by `notify()`.

Resources for Java Programming

See also Vladimir’s videos introducing concurrent programming in Java, available on the course website.
Shared Data

Consider the *Readers and Writers* problem, common in any database:

**Problem**

We have a large data structure (i.e. a structure that cannot be updated in one atomic step) that is shared between some number of writers who are updating the data structure and some number of readers who are attempting to retrieve a coherent copy of the data structure.
Shared Data

Consider the *Readers and Writers* problem, common in any database:

**Problem**

We have a *large data structure* (i.e. a structure that cannot be updated in one atomic step) that is shared between some number of writers who are updating the data structure and some number of readers who are attempting to retrieve a coherent copy of the data structure.

**Desiderata:**

- We want *atomicity*, in that each update happens in one go, and updates-in-progress or partial updates are not observable.
Shared Data

Consider the *Readers and Writers* problem, common in any database:

**Problem**

We have a large data structure (i.e. a structure that cannot be updated in one atomic step) that is shared between some number of writers who are updating the data structure and some number of readers who are attempting to retrieve a coherent copy of the data structure.

**Desiderata:**

- We want *atomicity*, in that each update happens in one go, and updates-in-progress or partial updates are not observable.
- We want *consistency*, in that any reader that starts after an update finishes will see that update.
Shared Data

Consider the *Readers and Writers* problem, common in any database:

**Problem**

We have a large data structure (i.e. a structure that cannot be updated in one atomic step) that is shared between some number of writers who are updating the data structure and some number of readers who are attempting to retrieve a coherent copy of the data structure.

**Desiderata:**

- We want *atomicity*, in that each update happens in one go, and updates-in-progress or partial updates are not observable.
- We want *consistency*, in that any reader that starts after an update finishes will see that update.
- We want to minimise *waiting*. 
A Crappy Solution

Treat both reads and updates as critical sections — use any old critical section solution (semaphores, etc.) to sequentialise all reads and writes to the data structure.
A Crappy Solution

Treat both reads and updates as critical sections — use any old critical section solution (semaphores, etc.) to sequentialise all reads and writes to the data structure.

Observation

Updates are *atomic* and reads are *consistent* — but reads can’t happen concurrently, which leads to unnecessary *contention*.
A Better Solution

Use a *monitor* with two condition variables (a la Ben-Ari’s chapter 7).

**Requirements**

We will need atomicity and consistency, and multiple reads to execute concurrently. Still, we don’t want to allow updates to execute concurrently with reads, to prevent partial updates from being observed by a reader.
Algorithm 2.4: 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)
```
Algorithm 2.4: Readers and writers with a monitor (continued)

**operation** StartWrite

- if writers $\neq 0$ or readers $\neq 0$
  - `waitC(OKtoWrite)`
  - `writers ← writers + 1`

**operation** EndWrite

- `writers ← writers - 1`
- if empty(OKtoRead)
  - then `signalC(OKtoWrite)`
  - else `signalC(OKtoRead)`

<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: <em>read the database</em></td>
<td>q2: <em>write to the database</em></td>
</tr>
<tr>
<td>p3: RW.EndRead</td>
<td>q3: RW.EndWrite</td>
</tr>
</tbody>
</table>
Proving Atomicity

Essentially we desire **mutual exclusion** of writers with any other process (writer or reader).
Proving Atomicity

Essentially we desire mutual exclusion of writers with any other process (writer or reader).
Like any safety property, we can prove it by gathering invariants. Let $R$ be the number of active readers and $W$ be the number of active writers.
Proving Atomicity

Essentially we desire mutual exclusion of writers with any other process (writer or reader).
Like any safety property, we can prove it by gathering invariants.
Let $R$ be the number of active readers and $W$ be the number of active writers.
  - readers $= R \geq 0$ and writers $= W \geq 0$, trivially.
Proving Atomicity

Essentially we desire mutual exclusion of writers with any other process (writer or reader).
Like any safety property, we can prove it by gathering invariants. Let \( R \) be the number of active readers and \( W \) be the number of active writers.

- readers = \( R \geq 0 \) and writers = \( W \geq 0 \), trivially.
- \( (R > 0 \Rightarrow W = 0) \land (W \leq 1) \land (W = 1 \Rightarrow R = 0) \).

This is preserved across the eight possible transitions in this system: the four monitor operations running unhindered, and the four partial operations resulting from a signal. See Ben-Ari p159 for details.
Proving Atomicity

Essentially we desire mutual exclusion of writers with any other process (writer or reader). Like any safety property, we can prove it by gathering invariants. Let $R$ be the number of active readers and $W$ be the number of active writers.

- readers $= R \geq 0$ and writers $= W \geq 0$, trivially.
- $(R > 0 \Rightarrow W = 0) \land (W \leq 1) \land (W = 1 \Rightarrow R = 0)$.

This is preserved across the eight possible transitions in this system: the four monitor operations running unhindered, and the four partial operations resulting from a signal. See Ben-Ari p159 for details.

Liveness Properties

We may also wish to prove some analogue of starvation freedom as Ben-Ari does on p160, however this relies on a fair bit of handwaving, as without a concrete monitor implementation it’s hard to know whether starvation is possible!
Complication

Now suppose we don’t want readers to wait (much) while an update is performed. Instead, we’d rather they get an *older version* of the data structure.
Complication

Now suppose we don’t want readers to wait (much) while an update is performed. Instead, we’d rather they get an older version of the data structure.

**Trick:** Rather than update the data structure in place, a writer creates *their own local copy* of the data structure, and then merely updates the (shared) *pointer* to the data structure to point to their copy.

Johannes: Draw on the board
Reading and Writing

Complication

Now suppose we don’t want readers to wait (much) while an update is performed. Instead, we’d rather they get an older version of the data structure.

**Trick:** Rather than update the data structure in place, a writer creates their own local copy of the data structure, and then merely updates the (shared) pointer to the data structure to point to their copy.

Johannes: Draw on the board

**Atomicity**  The only shared write is now just to one pointer.

**Consistency**  Reads that start before the pointer update get the older version, but reads that start after get the latest.
Persistent Data Structures

Copying is $\mathcal{O}(n)$ in the worst case, but we can do better for many tree-like types of data structure.

Example (Binary Search Tree)
Persistent Data Structures

Copying is $O(n)$ in the worst case, but we can do better for many tree-like types of data structure.

Example (Binary Search Tree)
Persistent Data Structures

Copying is $O(n)$ in the worst case, but we can do better for many tree-like types of data structure.

Example (Binary Search Tree)
Persistent Data Structures

Copying is $O(n)$ in the worst case, but we can do better for many tree-like types of data structure.

Example (Binary Search Tree)
Purely Functional Data Structures

Persistent data structures that exclusively make use of copying over mutation are called *purely functional* data structures. They are so called because operations on them are best expressed in the form of mathematical functions that, given an input structure, return a new output structure:

\[
\begin{align*}
\text{insert } v \text{ Leaf} &= \text{Branch } v \text{ Leaf Leaf} \\
\text{insert } v \text{ (Branch } x \text{ l r)} &= \text{if } v \leq x \text{ then} \\
&\quad \text{Branch } x \text{ (insert } v \text{ l) r} \\
&\quad \text{else} \\
&\quad \text{Branch } x \text{ l (insert } v \text{ r)}
\end{align*}
\]

Purely functional programming languages like Haskell are designed to facilitate programming in this way.
Next lecture, we’ll be looking at message-passing, the foundation of distributed concurrency.

This homework involves Java programming. There are some resources to assist you on the course website.

Assignment 1 is also coming out this week, hopefully tonight.