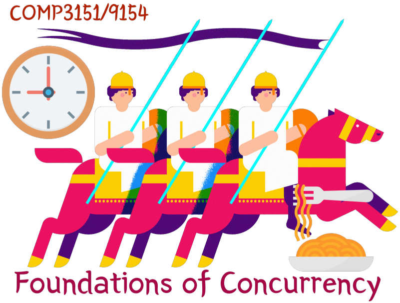


Monitors

Johannes Åman Pohjola
CSE, UNSW
Term 2 2022

Where we are at

Last lecture, we saw a generalisation of *locks* called *semaphores*.

In this lecture, we'll look at another concurrency abstraction, designed to ameliorate some problems with semaphores: *monitors*.

Main Disadvantages of Semaphores

1. **Lack of structure**: when building a large system, responsibility is diffused among implementers. Someone forgets to call `signal` $\Longrightarrow$ 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` $\Longrightarrow$ possible deadlock.
2. **Global visibility**: when something goes wrong, the whole program must be inspected $\Longrightarrow$ 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` $\Longrightarrow$ possible deadlock.
2. **Global visibility**: when something goes wrong, the whole program must be inspected $\Longrightarrow$ 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
```

p	q
p1: **loop** ten times	q1: **loop** ten times
p2: CS.increment	q2: CS.increment

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
- *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).

Condition variables

Definition

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

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

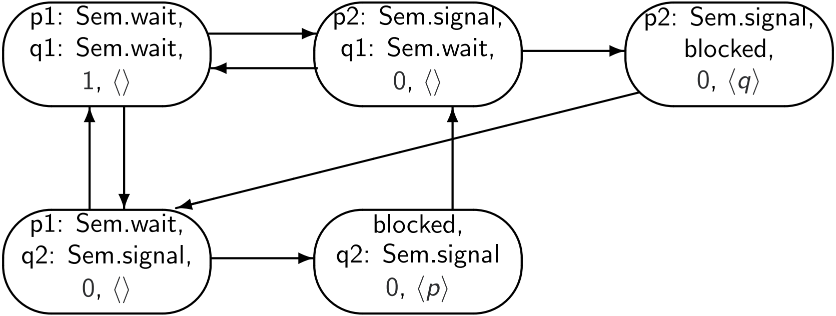
- voluntarily suspend themselves using **waitC**(cv),
- unblock the first suspended process by calling **signalC**(cv), or
- test for emptiness of the queue: **empty**(cv).

Warning

The exact semantics of these differ between implementations!

Algorithm 2.2: Semaphore simulated with a monitor	
monitor Sem	
integer s $\leftarrow$ k	
condition notZero	
operation wait	
if s = 0	
waitC(notZero)	
s $\leftarrow$ s $-$ 1	
operation signal	
s $\leftarrow$ s $+$ 1	
signalC(notZero)	
p	**q**
loop forever	**loop** forever
non-critical section	*non-critical section*
p1: Sem.wait	q1: Sem.wait
critical section	*critical section*
p2: Sem.signal	q2: Sem.signal

State Diagram for the Semaphore Simulation



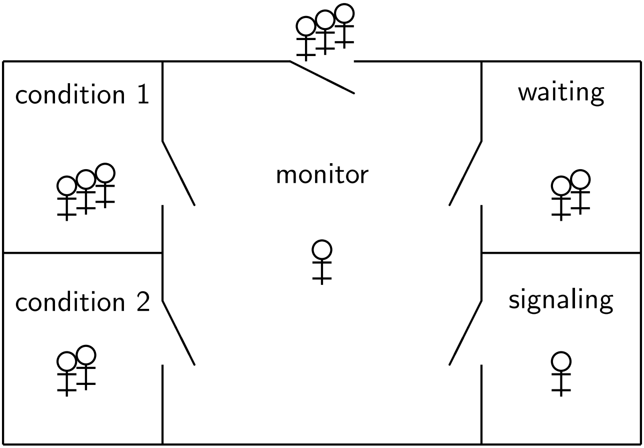
Algorithm 2.3: Producer-consumer (finite buffer, monitor)

```
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)
      signalC(notFull)
      return W
```

Algorithm 2.3: Producer-consumer . . . (continued)	
producer	**consumer**
datatype D	datatype D
loop forever	**loop** forever
p1: D ← produce	q1: D ← PC.take
p2: PC.append(D)	q2: consume(D)

The Immediate Resumption Requirement

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



Signaling disciplines

Precedences:

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

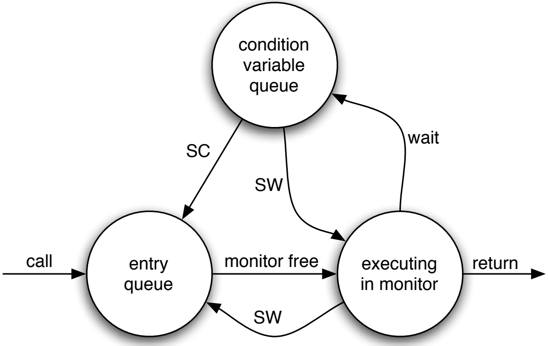
Signal and Urgent Wait

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



Simulating Monitors in Promela 1

```
bool lock = false;

typedef Condition {
   bool gate;
   byte waiting;
}
inline enterMon() {
   atomic {
      !lock;
      lock = true;
   }
}
inline leaveMon() {
   lock = false;
}
```

Simulating Monitors in Promela 2

```
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

```
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 immmediate 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 immmediate 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 immmediate 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 immmediate 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 immmediate 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 which cannot be updated in one atomic step. It is shared between many writers and many readers.

Shared Data

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

Problem

We have a large data structure which cannot be updated in one atomic step. It is shared between many writers and many readers.

Desiderata:

- *Atomicity*. An update should happen 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 which cannot be updated in one atomic step. It is shared between many writers and many readers.

Desiderata:

- *Atomicity*. An update should happen in one go, and updates-in-progress or partial updates are not observable.
- *Consistency*. 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 which cannot be updated in one atomic step. It is shared between many writers and many readers.

Desiderata:

- *Atomicity*. An update should happen in one go, and updates-in-progress or partial updates are not observable.
- *Consistency*. Any reader that starts after an update finishes will see that update.
- Minimal *waiting*.

A Crappy Solution

Treat both reads and updates as critical sections — use any old critical section solution 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 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

A *monitor* with two condition variables (à la Ben-Ari chapter 7).

Requirements

1. *Atomicity* and *consistency* (still)
2. Multiple reads can execute concurrently.
3. If someone writes: no concurrent reads or writes.

Algorithm 2.4: Readers and writers with a monitor

monitor RW
  integer readers $\leftarrow$ 0
  integer writers $\leftarrow$ 0
  condition OKtoRead, OKtoWrite
  operation StartRead
    if writers $\neq$ 0 or not **empty**(OKtoWrite)
      waitC(OKtoRead)
    readers $\leftarrow$ readers $+$ 1
    signalC(OKtoRead)
  operation EndRead
    readers $\leftarrow$ 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 $\leftarrow$ writers + 1

operation EndWrite
  writers $\leftarrow$ writers − 1
  if empty(OKtoRead)
    then signalC(OKtoWrite)
    else signalC(OKtoRead)

reader	writer
p1: RW.StartRead	q1: RW.StartWrite
p2: *read the database*	q2: *write to the database*
p3: RW.EndRead	q3: RW.EndWrite

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.

- $\text{readers} = R \geq 0$ and $\text{writers} = W \geq 0$, 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.
- $(R > 0 \Rightarrow W = 0) \wedge (W \leq 1) \wedge (W = 1 \Rightarrow R = 0)$.
  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. This gets a bit handwavy. Without a concrete monitor implementation, it's hard to know whether starvation is possible!

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.

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: A writer creates *their own local copy* of the data structure, and then 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: A writer creates *their own local copy* of the data structure, and then updates the (shared) *pointer* to the data structure to point to their copy.

Johannes: Draw on the board

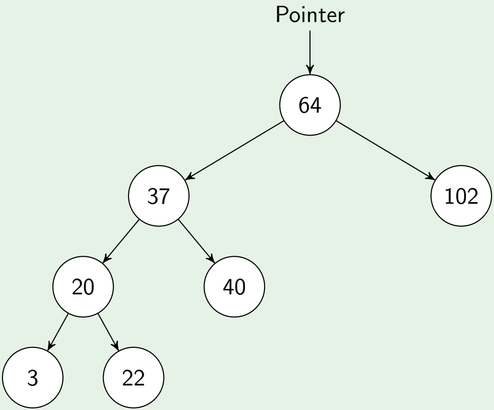
Atomicity The only shared write is now just to one pointer.

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.

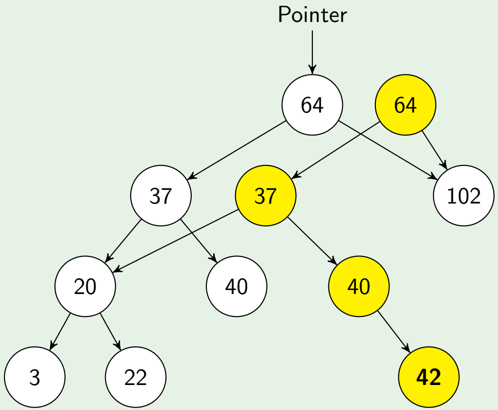
Example (Binary Search Tree)



Persistent Data Structures

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

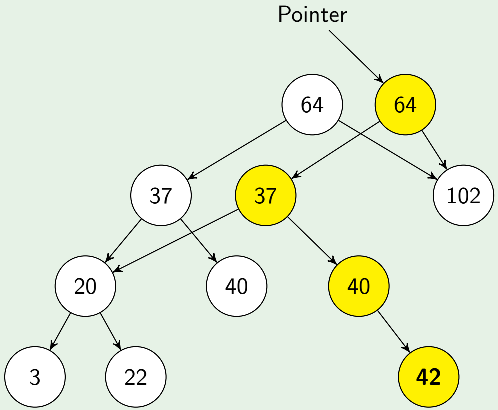
Example (Binary Search Tree)



Persistent Data Structures

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

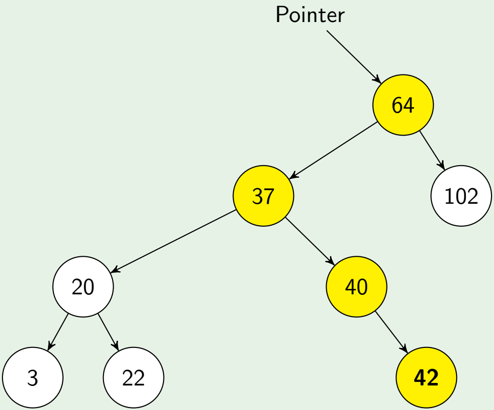
Example (Binary Search Tree)



Persistent Data Structures

Copying is $\mathcal{O}(n)$ 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:

$$\begin{array}{lcl} \mathit{insert}\ v\ \mathrm{Leaf} & = & \mathrm{Branch}\ v\ \mathrm{Leaf}\ \mathrm{Leaf} \\ \mathit{insert}\ v\ (\mathrm{Branch}\ x\ l\ r) & = & \textsf{if}\ v \leq x\ \textsf{then} \\ & & \quad \mathrm{Branch}\ x\ (\mathit{insert}\ v\ l)\ r \\ & & \textsf{else} \\ & & \quad \mathrm{Branch}\ x\ l\ (\mathit{insert}\ v\ r) \end{array}$$

Purely functional programming languages like Haskell are designed to facilitate programming in this way.

What Now?

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 out this week, hopefully tonight.