Parallel Programming
Programowanie równoległe

Lecture 3: Monitors.

Paweł Rzążewski

Disadvantages of semaphores

Although semaphores allow us to solve the critical section problem, they have some disadvantages:

- as a structural concept, they do not work well with object-oriented programming,
- solving even a simple task requires many semaphores,
- they are used for mutual exclusion and condition synchronization – the programs are hard to read and analyze.

Monitors – in a perfect world

A monitor is a class (or an object of this class) with the following properties:

- all fields, variables, methods are private, except for the so-called entry methods,
- entry methods are public,
- at most one entry method may be executed at the same time.

Monitors – in a perfect world

A monitor is a class (or an object of this class) with the following properties:

- all fields, variables, methods are private, except for the so-called entry methods,
- entry methods are public,
- at most one entry method may be executed at the same time.

The monitor can be in one of two states:

- busy – some entry method is executed,
- free – no entry method is executed.

If the monitor is busy and a process P wants to enter it (by calling an entry method), P is suspended in the *entry set*.

Waiting set

Additionally, we have some pre-defined type *QUEUE*, which is a first-in-first-out collection of processes. Each of these queues has two methods:

- `delay()` (wait, sleep) – the process calling this method is suspended in the waiting set. The monitor becomes free.
- `continue()` (signal, notify) – one process from the waiting set is awaken. It continues from the place, where it called `delay()`.

Each of these methods can only be called inside an entry method.

Notifying the queue

Suppose we have the monitor with one *QUEUE* q.

```
entry A():
    q.delay()
    work1()
```

```
entry B():
    q.notify()
    work2()
```

The process $P1$ calls $A()$ and then process $P2$ calls $B()$.

Notifying the queue

Suppose we have the monitor with one *QUEUE* q.

```
entry A():
    q.delay()
    work1()
```

```
entry B():
    q.notify()
    work2()
```

The process $P1$ calls $A()$ and then process $P2$ calls $B()$.

After calling `q.notify()`, the control is **immediately** passed to $P1$. Thus `work2()` is never executed!

Notifying the queue

Suppose we have the monitor with one *QUEUE* q.

```
entry A():
    q.delay()
    work1()
```

```
entry B():
    q.notify()
    work2()
```

The process $P1$ calls $A()$ and then process $P2$ calls $B()$.

After calling `q.notify()`, the control is **immediately** passed to $P1$. Thus `work2()` is never executed!

A good practice is to call `notify()` as the last operation in an entry method. If this is not possible (e.g. we want to notify many processes), some additional work is needed.

Producer and consumer using monitors

Recall that we have a number of producers and consumers using a single-element buffer b. The producers write data into the buffer. The consumers read the produced data. If the buffer is full, a producer should wait. If the buffer is empty, a consumer should wait.
We define a monitor class *Buffer*, b is an object of this class.

Producer:
repeat:
    $x \leftarrow$ produce()
    b.put(x)

Consumer:
repeat:
    b.get(x)
    consume(x)

Producer and consumer using monitors – ctd.

Monitor Buffer

`int` *full*

product *buffer*

QUEUE *qc*

QUEUE *qp*

```
entry put(x):
    if (full = 1)
        qp.delay()
    buffer ← x
    full ← 1
    qc.notify()
```

```
entry get(x):
    if (full = 0)
        qc.delay()
    x ← buffer
    full ← 0
    qp.notify()
```

What about a bigger buffer?

Design a monitor solution for

- producer-consumer problem with an infinite buffer,
- producer-consumer problem with a k-element buffer,
- producer-consumer problem with a k-element buffer, where copying takes much time.

Readers-writers with priority to writers

What about the readers-writers problem?

Readers-writers with priority to writers

What about the readers-writers problem?

We shall use the following variables:
integer *nr* – the number of **reading** readers,
integer *nw* – the number of **waiting** writers,
boolean flag *w* – is true when a writer is writing,
two *QUEUE*s *qr* and *qw* of readers and writers.

Readers-writers with priority to writers

```
entry reader-start()
  if nw > 0 or w = true
     qr.delay()
  nr ← nr + 1
  qr.notify()


entry reader-end()
  nr ← nr − 1
  if nr = 0
     qw.notify()
```

Readers-writers with priority to writers

```
entry reader-start()
  if nw > 0 or w = true
     qr.delay()
  nr ← nr + 1
  qr.notify()

entry reader-end()
  nr ← nr − 1
  if nr = 0
     qw.notify()
```

```
entry writer-start()
  if nr > 0 or w = true
     nw ← nw + 1
     qw.delay()
     nw ← nw − 1
  w ← true

entry writer-end()
  w ← false
  if nw > 0
     qw.notify()
  qr.notify()
```

Readers-writers with priority to writers

```
entry reader-start()
  if nw > 0 or w = true
     qr.delay()
  nr ← nr + 1
  qr.notify()

entry reader-end()
  nr ← nr − 1
  if nr = 0
     qw.notify()
```

```
entry writer-start()
  if nr > 0 or w = true
     nw ← nw + 1
     qw.delay()
     nw ← nw − 1
  w ← true

entry writer-end()
  w ← false
  if nw > 0
     qw.notify()
  qr.notify()
```

Why do we call *qr*.notify() in the last line of `writer-end()`?

Dining philosophers and monitors

Again, we have a monitor with the following variables (all indices are computed cyclically):
$Fork[1...n]$ – states of forks (Free or Busy)
$Wait[1...n]$ – indicates is the given philosopher is waiting
$QWait[1...n]$ – an array of $QUEUE$s where the philosophers wait

```
entry try-to-take-forks(i)
if Fork[i] = B or Fork[i + 1] = B
    Wait[i] ← true
    QWait[i].delay()
    Wait[i] ← false
Fork[i] ← B
Fork[i + 1] ← B
```

```
entry put-down-forks(i)
Fork[i] ← F
Fork[i + 1] ← F
if Wait[i − 1] = true and Fork[i − 1] = F
    QWait[i − 1].notify()
if Wait[i + 1] = true and Fork[i + 2] = F
    QWait[i + 1].notify()
```

Dining philosophers and monitors

Again, we have a monitor with the following variables (all indices are computed cyclically):
$Fork[1...n]$ – states of forks (Free or Busy)
$Wait[1...n]$ – indicates is the given philosopher is waiting
$QWait[1...n]$ – an array of $QUEUE$s where the philosophers wait

```
entry try-to-take-forks(i)
if Fork[i] = B or Fork[i + 1] = B
    Wait[i] ← true
    QWait[i].delay()
    Wait[i] ← false
Fork[i] ← B
Fork[i + 1] ← B
```

```
entry put-down-forks(i)
Fork[i] ← F
Fork[i + 1] ← F
if Wait[i − 1] = true and Fork[i − 1] = F
    QWait[i − 1].notify()
if Wait[i + 1] = true and Fork[i + 2] = F
    QWait[i + 1].notify()
```

Is it ok?

Dining philosophers and monitors

Again, we have a monitor with the following variables (all indices are computed cyclically):
$Fork[1...n]$ – states of forks (Free or Busy)
$Wait[1...n]$ – indicates is the given philosopher is waiting
$QWait[1...n]$ – an array of $QUEUE$s where the philosophers wait

```
entry try-to-take-forks(i)
if Fork[i] = B or Fork[i + 1] = B
    Wait[i] ← true
    QWait[i].delay()
    Wait[i] ← false
Fork[i] ← B
Fork[i + 1] ← B
```

```
entry put-down-forks(i)
Fork[i] ← F
Fork[i + 1] ← F
if Wait[i − 1] = true and Fork[i − 1] = F
    QWait[i − 1].notify()
if Wait[i + 1] = true and Fork[i + 2] = F
    QWait[i + 1].notify()
```

Is it ok?
What happens if **both** neighbors can start eating?

Dining philosophers and monitors – corrected

$Fork[1...n]$ – states of forks (Free or Busy)
$Wait[1...n]$ – indicates is the given philosopher is waiting
$QWait[1...n]$ – an array of $QUEUE$s where the philosophers wait
$wait$ – a flag indicating that the previous neighbor can be awaken

```
entry put-down-forks(i)
Fork[i] ← F
Fork[i + 1] ← F
if Wait[i − 1] = true and Fork[i − 1] = F
    wait ← true
if Wait[i + 1] = true and Fork[i + 2] = F
    QWait[i + 1].notify()
if wait = true
    wait ← false
    QWait[i − 1].notify()
```

Dining philosophers and monitors – corrected

$Fork[1...n]$ – states of forks (Free or Busy)
$Wait[1...n]$ – indicates is the given philosopher is waiting
$QWait[1...n]$ – an array of $QUEUE$s where the philosophers wait
$wait$ – a flag indicating that the previous neighbor can be awaken

```
entry put-down-forks(i)
Fork[i] ← F
Fork[i + 1] ← F
if Wait[i − 1] = true and Fork[i − 1] = F
    wait ← true
if Wait[i + 1] = true and Fork[i + 2] = F
    QWait[i + 1].notify()
if wait = true
    wait ← false
    QWait[i − 1].notify()
```

```
entry try-to-take-forks(i)
if Fork[i] = B or Fork[i + 1] = B
    Wait[i] ← true
    QWait[i].delay()
    Wait[i] ← false
Fork[i] ← B
Fork[i + 1] ← B
if wait = true
    wait ← false
    QWait[i − 2].notify()
```

Monitors in Java

There are several ways of achieving similar behavior in Java. The most similar to our perfect-world-monitors is as follows.

Entry functions

```
 final Lock lock = new ReentrantLock();

public void Method(...)  {
   lock.lock();
   try {
   ...
   } finally {
   lock.unlock();
   }
}
```

Producer and consumer using monitors – again

Monitor Buffer

`int` *full*

product *buffer*

QUEUE *qc*

QUEUE *qp*

```
entry put(x):
    if (full = 1)
        qp.delay()
    buffer ← x
    full ← 1
    qc.notify()
```

```
entry get(x):
    if(full = 0)
        qc.delay()
    x ← buffer
    full ← 0
    qp.notify()
```

Producer and consumer in Java

```
 final Lock lock = new ReentrantLock();
final Condition empty = lock.newCondition();
final Condition full = lock.newCondition();
Object buffer;
```

```
 public void put(Object x) throws
InterruptedException {
   lock.lock();
   try {
   if (buffer == null) empty.await();
   buffer = x;
   full.signal();
   } finally {
   lock.unlock(); }
}
```

Producer and consumer in Java

```
 final Lock lock = new ReentrantLock();
final Condition empty = lock.newCondition();
final Condition full = lock.newCondition();
Object buffer;
```

```
 public void get(Object x) throws
InterruptedException {
   lock.lock();
   try {
   if (buffer != null) full.await();
   Object x = buffer;
   buffer = null;
   empty.signal();
   return x;
   } finally {
   lock.unlock(); }
}
```

Conditional variables and locks

The conditional variables are related to locks:
`final Condition empty = lock.newCondition();`.
Calling `await()` on the variable unlocks the lock. On the other hand, when awake a process using `signal()`, the lock becomes locked.

Monitors in C#

The simplest way to implement entry methods is by using `lock`.

`object o = (whatever)`

```
public void entryMethod() {
  lock(o)
  {
     work();
  }
}
```

which is a syntax sugar for:

```
public void entryMethod() {
  Monitor.Enter(o);
  work();
  Monitor.Exit(o);
}
```

Monitors in C#, ctd.

Waiting and waking is done by:

```
Monitor.Wait(o);
Monitor.Pulse(o);
```

Monitor Buffer

```
object o = new object();

int get():
    lock(o) {
        if (empty)
            Monitor.Wait(o);
        take_element();
    }

entry put(x):
    lock(o) {
        add_element();
        Monitor.Pulse(o);
    }
```

Monitors in C#, ctd.

Waiting and waking is done by:

`Monitor.Wait(o);`

`Monitor.Pulse(o);`

Monitor Buffer

```
object o = new object();

int get():
    lock(o) {
        if (empty)
            Monitor.Wait(o);
        take_element();
    }
```

```
entry put(x):
    lock(o) {
        add_element();
        Monitor.Pulse(o);
    }
```

Introducing multiple queues is more complicated, see:

`https://github.com/CodeExMachina/ConditionVariable`.

Spurious wakeup

It may happen, that the process awaiting on some conditional variable is awaken, even if no process signalled it. This is called a *spurious wakeup* and happens in many implementations of conditional variables (Java, POSIX, Win API).

Thus we should always use:
`while (`$buffer != null$`) full.await();`
instead of
`if (`$buffer != null$`) full.await();`