Monitors

February 4, 2019

- Previous methods
- Monitors
- Monitors and Java

Previous methods

1: Previous methods

1.1: Shared variables

- complex algorithms
- time wasting — busy waits
- primitive (atomic) actions are too weak

Previous methods

1.2: Semaphores

- easy to forget to use a semaphore (in complex programmes)
- forgetting a `poll()` results in lack of mutual exclusion
- forgetting a `vote()` often leads to deadlock
- nesting of `poll()`s and `vote()`s is risky
- cannot test value of semaphore without (possibly) blocking

Need separate primitives for mutual exclusion and synchronistion.

Monitors

2: Monitors

A more structured and more user-friendly method of communication.

- generalisation of module concept
- only one process may execute procedures at any one time
- new data type **condition**

Monitors

2.1: The condition data type

Two operations:

- `cond.wait()`
  Go to sleep (join the queue for `cond`) and release the monitor
- `cond.signal()`
  If the queue for `cond` is not empty, wake the *first* process in the queue. Otherwise no effect

Producer/consumer

2.2: Producer/Consumer
2.2.1: The Monitor

```
monitor ProducerConsumer <T> {// this is not Java

    // Conditions for testing
    condition notFull, notEmpty;

    // The buffer
    Buffer<T> buffer;
    ...// see below for producer/consumer code
}
```

Producer/consumer

2.2.2: The Producer

```
public void producer(T item) {
    if (buffer.full()) {// is the buffer full?
        notFull.wait(); // if so, wait (and release the monitor)
    }
    buffer.put(item); // add the item to the buffer
    notEmpty.signal(); // signal that the buffer is not empty
}
```

Producer/consumer

2.2.3: The Consumer

```
public T consumer() {
    if (buffer.empty()) {// is the buffer empty?
        notEmpty.wait(); // if so, wait (and release the monitor)
    }
    T item = buffer.take(); // get the item from the buffer
    notFull.signal(); // signal that the buffer is not full
    return item;
}
```

Problems: Embedded signals

2.3: Problems with monitors
2.3.1: Embedded signals

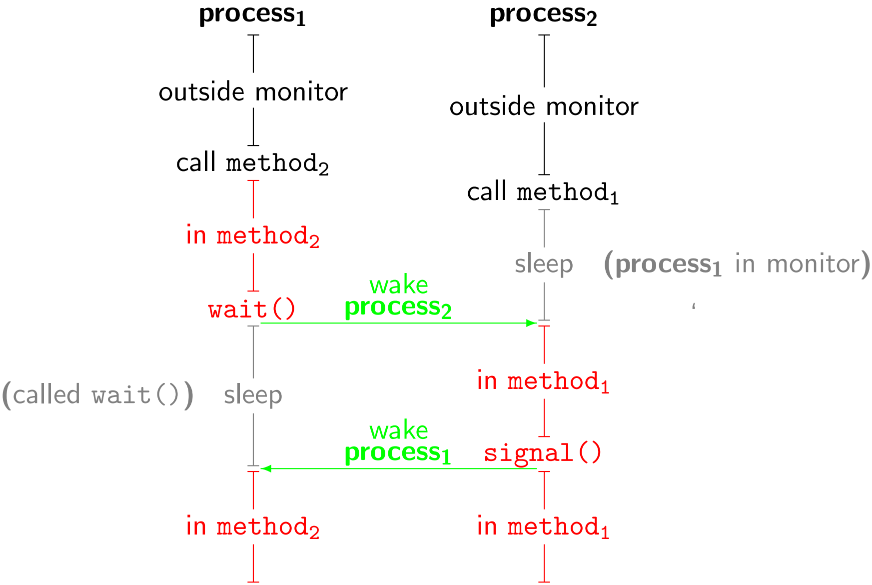
```
monitor demo {
    condition cond;

    void method₁() {...; cond.signal(); ... }

    void method₂() {...; cond.wait(); ... }
}
```

Can lead to both processes in the monitor simultaneously...

Problems: Embedded signals



Problems: Embedded signals

Three possible solutions:

- simple: signal and leave
- complex: signal and continue
- even more complex: signal and urgent wait

Problems: Embedded signals, Signal and leave

2.3.1 A: Simple: Signal and leave

- `signal` *must* be last statement in a monitor
- signalling process will leave monitor immediately after `signal()`
- onus on programmer (or compiler?)

Problems: Embedded signals, Signal and continue

2.3.1 B: Complex: Signal and continue

- `signal()` wakes a waiting process, if any, and adds it to those contending for the monitor
- signaling process continues until it releases the monitor
- a contending process, if any, obtains the monitor

Problems: Embedded signals, Signal and urgent wait

2.3.1 C: Signal and urgent wait

Process signaling sleeps if necessary.

Two possibilities on `cond.signal()`:

- No processes waiting on `cond`
  - signaler proceeds
- Processes waiting on `cond`
  - signaler sleeps
  - wake waiting process
  - wake signaler when this process exits monitor
  - no other process may enter until woken process *and* signaler leave monitor

Problems: Nested monitors, Mutual calls

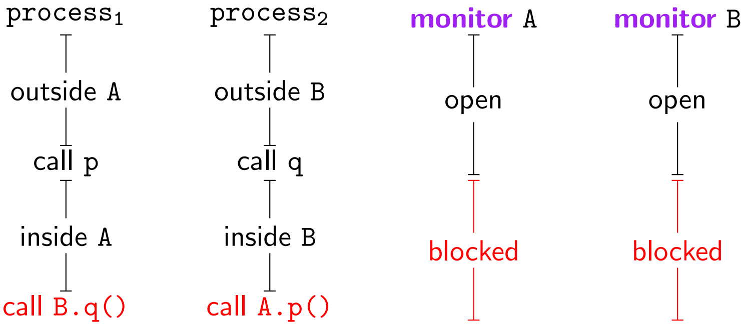
2.3.2: Nested monitors
2.3.2 A: Mutual calls

```
monitor A {
    void p() {
        ...; B.q(); ...
    }
}
```

```
monitor B {
    void q() {
        ...; A.p(); ...
    }
}
```

Can lead to deadlock...

Problems: Nested monitors, mutual calls



Solution

Assign *priorities* to monitors — no calls to monitors of lower priority allowed.

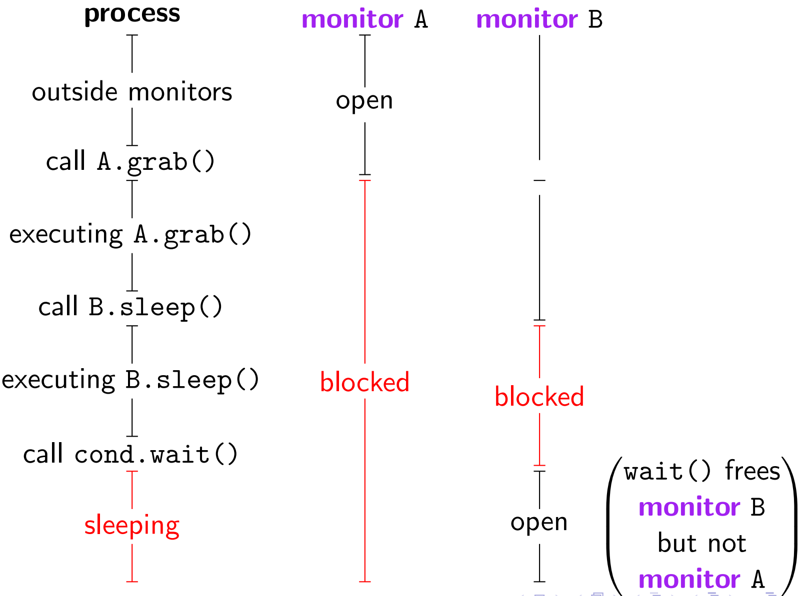
Problems: Nested monitors, Blocking waits

2.3.2 B: Blocking waits

```
monitor A {
    void grab() {
        ...; B.sleep(); ...;
    }
    void release() {
        ...; B.wake(); ...
    }
}
```

```
monitor B {
    condition cond;
    void sleep() {
        ...; cond.wait(); ...;
    }
    void wake() {
        ...; cond.signal(); ...
    }
}
```

Problems: Nested monitors, Blocking waits



Problems: Nested monitors, Blocking waits

Solution:

Process leaves *all* monitors when it blocks — however must then enter multiple monitors when freed (as an atomic action).

Java

3: Monitors and Java

3.1: Introduction

Java — synchronised methods, rather than synchronised classes.

Java keywords:

synchronized method is synchronised (executing thread owns object's monitor)

wait wait until another thread invokes **notify** or **notifyall** (invoking thread must own object's monitor)

notify allow a waiting thread, if one exists, to compete for object's monitor (current thread continues until it relinquishes monitor — signal and continue)

notifyall wake all waiting threads

3.2: Java Code
3.2.1: Carpark control

```
class CarParkControl {

    protected int spaces;
    protected int capacity;

    public CarParkControl(int n) {
        capacity = spaces = n;
    }
    ...// see below for enter() and leave() methods
}
```

3.2.1 A: enter()

```
synchronized void enter() throws InterruptedException {
    if (spaces == 0) {
        wait();
    }
    --spaces;
    notify();
}
```

Java

3.2.1 B: leave()

```
synchronized void leave() throws InterruptedException {
    if (spaces == capacity) {
        wait();
    }
    ++spaces;
    notify();
}
```

Java

3.2.2: Cars arriving

```
class Arrivals extends Thread {

    CarParkControl control;

    public Arrivals(CarParkControl control) {
        this.control = control;
    }

    public void run() {
        try {
            while (true) {
                carpark.enter();
            }
        }  catch (InterruptedException e) {}
    }
}
```

3.2.3: Cars leaving

```
class Departures extends Thread {

    CarParkControl control;

    public Departures(CarParkControl control) {
        this.control = control;
    }

    public void run() {
        try {
            while (true) {
                carpark.leave();
            }
        }  catch (InterruptedException e) {}
    }
}
```

3.2.4: The car park

```
public class CarPark {

    public CarPark(int n) throws InterruptedException {
        CarParkControl control = new CarParkControl(n);
        Arrivals arrivals = new Arrivals(control);
        Departures departures = new Departures(control);
        arrivals.start(); departures.start();
        arrivals.join(); departures.join();
    }
}
```

Java

3.3: Synchronised statements

Java also has **synchronized** statements. Need to specify the object whose monitor lock is to be used.

Java

```java
private SomeType thing1, thing2;
private Object lock1 = new Object(), lock2 = new Object();

public void access1() {
    synchronized(lock1) {
        // do something to thing1
    }
}
public void access2() {
    synchronized(lock2) {
        // do something to thing2
    }
}
```

3.4: Conditions

Java synchronisation provides you with *one* wait-set per object.
Use `Conditions` and `Locks` for multiple wait-sets.

```
final Lock lock = new ReentrantLock();
final Condition notFull = lock.newCondition();
final Condition notEmpty = lock.newCondition();

final T[] buffer = new T[...];
final int putIndex = 0; takeIndex = 0; numberOfElements = 0;
```

Java

```
public void put(T item) throws InterruptedException {
    lock.lock();
    try {
        while (count == buffer.length) notFull.await();
        buffer[putIndex] = item;
        putIndex = (putIndex+1)%buffer.length;
        numberOfElements++;
        notEmpty.signal();
    } finally {
        lock.unlock();
    }
}
```

Java

```java
public T take() throws InterruptedException {
    lock.lock();
    try {
        while (count == 0) notEmpty.await();
        T item = buffer[takeIndex];
        takeIndex = (takeIndex+1)%buffer.length;
        numberOfElements--;
        notFull.signal();
        return item;
    } finally {
        lock.unlock();
    }
}
```

End of monitors lecture