SWEN90004
Modelling Complex Software Systems

Monitors; Java summary

Nic Geard

Lecture Con.04

Overview

Synchronized methods and monitors

Semaphores are easier to use than shared protocol variables, but they are still a low-level and unstructured primitive and don't scale well.

For that reason, concurrent programming languages offer higher-level synchronization primitives. In the case of Java:

- Synchronized methods/objects: a method or an object can be declared `synchronized`, which means only one process can execute or modify it at any one time.
- Monitors: a set of synchronized methods and data (an object or module) that queue processes trying to access the data.

Synchronization in Java

```
class Counter {
  int value = 0;

  void increment() {
    int temp = value;
    try { Thread.sleep(1); }
    catch (Exception e) {}
    value = temp + 1;
  }
}
```

Here a `Counter` instance will increment an integer using the `increment()` method. Note that the variable `value` is read, then the process sleeps, and then the value is incremented and stored. What happens if two threads want to call `increment()` simultaneously?

Synchronization in Java

```
class UseCounter extends Thread {
  Counter c;

  public UseCounter(Counter c) {
    this.c = c;
  }

  public void run () {
    for (int i = 0; i < 5; i++) {
      c.increment();
    }
  }

  public static void main() {
    ...
  }
}
```

Synchronization in Java

```
public static void main() {
  Counter c = new Counter();
  UseCounter c1 = new UseCounter(c);
  UseCounter c2 = new UseCounter(c);
  c1.start();
  c2.start();
  try {
    c1.join();
    c2.join();
  }
  catch (InterruptedException e) {}
  System.out.println(c.value);
}
```

Synchronized methods

The `synchronized` keyword declares a method or object as being executable or modifiable by only one process at a time. If a method is declared as `synchronized`, in effect, it marks this method as a critical section.

We inherit `Counter` and modify the code by simply inserting the keyword, and run it again (changing the types of `c` in `UseCounter`):

```
class SynchedCounter extends Counter {
  synchronized void increment() {
    int temp = value;
    try { Thread.sleep(1); }
    catch (InterruptedException e) {}
    value = temp + 1;
  }
}
```

Synchronized objects

An alternative way is to declare an object as synchronized, making the entire object, rather than a method, mutually exclusive:

```
class SynchedObject extends Thread {
  Counter c;

  public SynchedObject(Counter c) {
    this.c = c;
  }

  public void run () {
    for (int i = 0; i < 5; i++) {
      synchronized(c) {
        c.increment();
      }
    }
  }
  public static void main(String [] args) {
    Counter c = new Counter();
    SynchedObject c1 = new SynchedObject(c);
    SynchedObject c2 = new SynchedObject(c);
    c1.start();
    c2.start();
    try {
      c1.join();
      c2.join();
    }
    catch (InterruptedException e) {}
    System.out.println(c.value);
  }
}
```

The disadvantage of this is that it requires the user of the shared object to lock the object, rather than placing this inside the shared object and encapsulating it. If a user fails to lock the object correctly, race conditions can occur.

Monitors

Monitors are language features that help with providing mutual exclusion to shared data.

In Java, a monitor is an object that encapsulates some (private) data, with access to the data *only* via synchronized methods. However, a monitor is more than just a collection of synchronized methods. It manages the blocking and unblocking of processes that vie for access.

Monitors

Monitors are language features that help with providing mutual exclusion to shared data.

In Java, a monitor is an object that encapsulates some (private) data, with access to the data *only* via synchronized methods. However, a monitor is more than just a collection of synchronized methods. It manages the blocking and unblocking of processes that vie for access.

On the next slides we consider a bank account that is shared between a parent and child. The parent deposits money into the account, and the child withdraws. The child attempts to withdraw funds whenever possible, but will only succeed if enough funds are available.

A naive bank account system

```
class Account {
  int balance = 0;

  public synchronized void withdraw(int amount) {
    int temp = balance;
    try { Thread.sleep(1); }
    catch (InterruptedException e) {}
    balance = temp - amount;
  }

  public synchronized void deposit(int amount) {
    int temp = balance;
    try { Thread.sleep(1); }
    catch (InterruptedException e) {}
    balance = temp + amount;
  }
}
```

```java
class Parent extends Thread {
  Account a;
  java.util.Random r = new java.util.Random();

  Parent(Account a) {
    this.a = a;
  }

  public void run() {
    for (int i = 0; i < 10; i++) {
      a.deposit(100);
      System.out.print("deposit 100; ");
      System.out.println("balance = " + a.balance);
      int s = r.nextInt(10);
      try { Thread.sleep(s); }
      catch (InterruptedException e) {}
    }
  }
}
```

```
class Child extends Thread {
  Account a;

  Child(Account a) {
    this.a = a;
  }

  public void run() {
    for (int i = 0; i < 10; i++) {
      a.withdraw(100);
      System.out.print("withdraw 100; ");
      System.out.println("balance = " + a.balance);
    }
  }
}
```

A naive bank account system

The above naive bank account class allows withdraws and deposits, but does not check that there are enough funds for a withdrawal. The `Father` class deposits funds at random intervals, while the `Child` class withdraws funds constantly. This leads to the account balance being negative for periods.

A less naive bank account system

One solution to this would be to let the child monitor the bank account balance to ensure that there are always enough funds. We could implement this by inserting `while (a.balance < 100);` before the call to `withdraw(100)` in `Child.run()`.

A less naive bank account system

One solution to this would be to let the child monitor the bank account balance to ensure that there are always enough funds. We could implement this by inserting `while (a.balance < 100);` before the call to `withdraw(100)` in `Child.run()`.

However, this may not work as planned. The problem is that some compilers (or virtual machines) may optimise the code by loading the value of `a.balance` into a register, and then checking that the value is less than 100. When the parent updates the value, the new value is not reloaded. Therefore, the loop above executes infinitely, the thread becomes blocked, and withdrawals are not made. Good news for the parent, but not for the child!

Volatile variables

The way around this is to declare the `Account.balance` variable as volatile:

```
volatile int balance = 0;
```

Declaring a variable `volatile` directs the virtual machine to re-load the value of a variable every time it needs to refer to it. This means that `a.balance` will be re-loaded for each loop, and the thread will not become blocked waiting for the balance to become high enough.

Back to monitors

The problem with the previous solution is that the responsibility to wait is left up to the user; in this case, the `Child` object. This has three drawbacks:

1. The user code has to continually poll the variable and check whether there are enough funds, which wastes CPU cycles. It would be better if the process somehow just waited.
2. The code will have to be replicated in many places if there is more than one user.
3. If one user incorrectly implements the waiting code, interference can occur.

Monitors are a way to achieve the same outcome, but with the encapsulating class doing the hard work.

Monitors: a definition

A monitor is an encapsulated piece of data, with operations/methods, that maintain a queue of processes wanting to access the data. First, we'll look at monitors in Java.

All objects in Java have monitors. The `Object` class in Java, from which all other classes inherit, contains the following three methods relevant to monitors:

- `void wait()`: Causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object.
- `void notify()`: Wakes up a single thread that is waiting on this object's lock (the choice of thread that is awoken is arbitrary).
- `void notifyAll()`: Wakes up all threads that are waiting on this object's lock.

Using monitors

Back to our bank account example. Now, instead of the `Child` class implementing the check on the balance, our bank account class does:

```
class MonitorAccount extends Account {
  public synchronized void withdraw(int amount) {
    while (balance < amount) {
      try {
        wait();
      }
      catch (InterruptedException e) {}
    }
    super.withdraw(amount);
  }
  public synchronized void deposit(int amount) {
    super.deposit(amount);
    notify();
  }
}
```

Using monitors

In the `withdraw` method of this new class, there is now a condition that says: "while there are not enough funds, make this process (the calling process, an instance of `Child`), wait until notified by another process that something has changed. This blocks the *Child* process.

The new implementation of `deposit` now updates the balance and notifies any waiting processes.

This is different to the solution in which we simply add `while (balance < amount)` to `withdraw`. With a monitor, the calling process is blocked, and it releases its lock, allowing other threads to execute the synchronized method.

Using monitors

If the `Child` object tries to withdraw when there are not enough funds, `wait()` is executed. The virtual machine suspends the `Child` process, placing it in a set of processes that are waiting. If another process calls `withdraw`, it will also be suspended.

Finally, a deposit is made, and `notify()` is called. The virtual machine wakes an arbitrary waiting process.

The execution of `withdraw` will continue, looping back to `while (balance < amount)`. If there are enough funds, withdrawal will continue. If not, the process is suspended again by calling `wait()`.

Java has lightweight monitors

A lock is associated with every object. To execute a `synchronized` method, a process must first acquire the lock for that object. It releases the lock upon return.

Java monitors do not guarantee fairness.

A process P that holds the lock on object `o` can choose to give it up, by invoking `wait()`. It is then waiting on `o`.

Another process Q may execute `o.notify()`, thereby changing some waiting process's state to locking (P, say).

Java has lightweight monitors

The process Q that executes `o.notify()` obviously has o's lock (and is running).

Some programming languages (and the original monitor proposals) stipulate that, at that point, the notified process P is given priority over other locking processes, and even the notifying process Q. (This makes sense, because Q has presumably changed some condition that made notification of P pertinent.)

Java does not do this. Instead we get the typical pattern of wrapping a `while` loop around a `wait()`.

The notifier will hold on to the lock (until the end of the synchronised method or code block).

Java has lightweight monitors

Some of the original monitor concepts allowed for many separate wait sets—each waiting for some specific condition to hold. Java does not have that.

```
synchronized methodX() {
    if (x == 0)
        wait();
```

```
synchronized methodY() {
    if (y == 0)
        wait();
```

Some third process may update x and y. If it changes x, it would want to notify whoever waits for x. If it changes y, it would want to notify whoever waits for y.

However, Java falls back on the `notifyAll()` solution: Release all processes waiting on the affected object (as opposed to the condition).

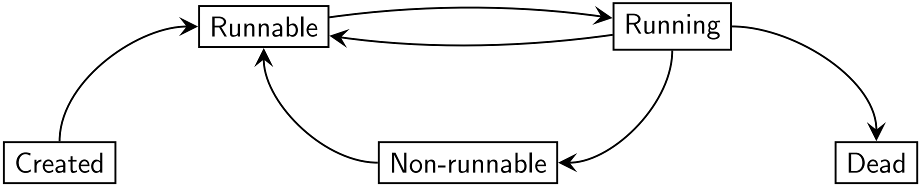
Implementing Java monitors

For a class to meet the requirements of a monitor:

- all attributes should be `private`
- all methods that access these attributes should be `synchronized`

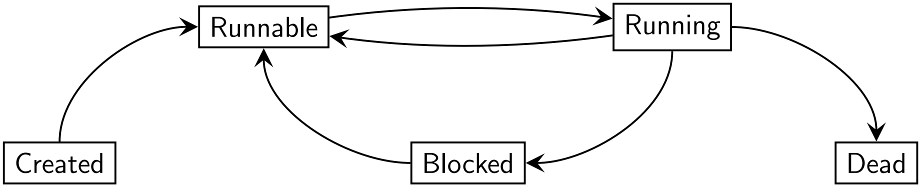
This ensures that all methods will be treated as atomic events.

Process states again



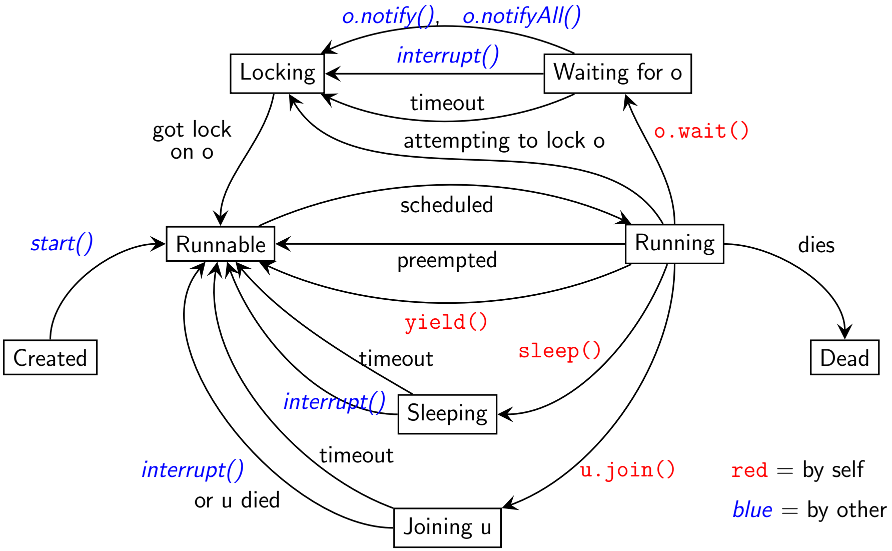
We previously discussed how running threads could pause their execution by yielding (to return to runnable) and sleeping (to temporarily become non-runnable).

Process states again



Various synchronization constructs, such as monitors, can result in a different type of non-runnable state, in which the process is blocked. A blocked process relies on other processes to unblock it, after which it is again runnable.

Java thread states in more detail



References

- M. Ben-Ari, *Principles of Concurrent and Distributed Programming*, Prentice Hall, 2nd edition, 2006.
- B. Goetz, *Java: Concurrency in Practice*, Addison-Wesley, 2006.
- D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 2nd edition, 2000.
- P. Sestoft, *Java Precisely*, MIT Press, 2nd edition, 2005.