ISCTE IUL
Instituto Universitário de Lisboa

Synchronization

Programação Concorrente e Distribuída

Parallel and Distributed Programming

2013-2014

After this class you will be able to…

- Understand the notion of race condition.
- Interference between threads
- Memory inconsistency errors
- Synchronization and atomic actions
- Intrinsic locks and Synchronization
- Synchronized methods / Synchronized blocks

Threads Synchronization

- Threads share memory so communication can be based on shared references.

- This is a very effective way to communicate but is prone to two types of errors:
  - Interference between threads
  - Memory inconsistency errors

Sharing data between Threads

- Can be problematic due to "*race condition*" between threads accessing the same data at the same time;
- A *race condition* is **a problem** when the order of execution of two or more threads may affect the value of some variable or the outcome of the program.

Stack of integers

```
class Pilha {
     private int pos = 0;
     private int pilha[]= new int[100];

     public void push(int i) {
          pos++;
          pilha[pos]=i;
     }

     public int pop() {
          if (pos>0){
               return(pilha[pos--]);
          } else
               return 0;
     }

     public int peak() {
          return pilha[pos];
     }
}
```

Non atomic operation (3 steps)

1. Memory read of pos
2. Increment of pos
3. Memory store of pos

Stack of integers

```
class Pilha {
     private int pos = 0;
     private int pilha[]= new int[100];

     public void push(int i) {
          pos++;
          pilha[pos]=i;
     }

  ...
}
```

If two threads (A and B) both do a push on the same stack at the same time. We could have:

A) pos++ (pos = 1)

B) pos++ (pos = 2)

A) pilha[pos]=7

B) pilha[pos]=4

The incoherent result would be:

- In position 1 there is a meaningless value.
- In position 2 the value 4 is stort.ed.
- The value 7 from thread A is los

Thread Interference

- Interference can happen when the actions of two or more threads on the same data are not atomic.
- Non-atomic actions can be divided into a set of atomic actions. This means that atomic actions can be interleaved or parallelized and can therefore access the data in an intermediate / incomplete state.

Memory inconsistency errors

- Happens when two or more threads have different versions of the same data.

Thread A)	Thread B)
pos++	a=pos;

- What are the possible outcomes?

Memory inconsistency errors

Thread A)	Thread B)
pos++	a=pos;

- The result depends on the order of the access to the data (pos).
- It's important to establish an order in the execution of different actions of different threads…

Synchronization and atomic actions

Atomic actions: Instructions that can be executed by a single thread at any time only.

Critical section: A sequence of instructions that have to be executed atomically.

Synchronized: A method or a section of Java code that should be treated as a critical section and therefore should be executed atomically.

How should the synchronized mechanism work?

When entering a synchronized method, all other threads are automatically suspended...

This solution is not efficient because a method can be complex and take a long time, and even threads that would never interfere would also be suspended.

How should the synchronized mechanism work?

When entering a synchronized method, only threads that could interfere are automatically suspended...

This solution is very complex. It is not not easy to predict which threads will do what in the future. It is therefore not easy to find out which threads should be suspended.

How should the synchronized mechanism work?

Protect critical sections with a lock. Before entering a critical section, the thread has to check if it lock is open. If so the thread locks the critical section and proceeds. If the lock is closed it has to wait while the tread that is running the critical section finishes.

This is the actual solution that Java uses. It is efficient because it only causes threads that would otherwise interfere to be suspended.

Mutex: mutual exclusion

- A basic locking mechanism in multithreaded programming
- A *mutex*-lock has two basic operations:
  - Lock
  - Unlock
- Only one thread can own the lock at any time.
- If a thread tries to lock a mutex that is already locked by another thread, it will have to wait until the lock becomes free (unlocked).
- If multiple threads are waiting for a mutex to become unlocked, only one thread will be able to lock the mutex when it happens.

Intrinsic locks and synchronization

- In Java every object has an intrinsic lock
- By convention a thread that needs exclusive and consistent access to an object's data must acquire the object's intrinsic lock before accessing its data. It should release the intrinsic lock when it is done. The thread owns the intrinsic lock between the act of acquiring and releasing.
- It is the programmer's responsibility to define the methods or blocks that should be subject to the owning of the lock. This is done in Java using the “synchronized” keyword.

Synchronized methods

```
class Pilha {
    private int pos = 0;
    private int pilha[]= new int[100];

    public synchronized void push(int i) {
        pos ++;
        pilha[pos]=i;
    }

    public synchronized int pop() {
        if (pos >0){
            return(pilha[pos --]);
        } else return 0;
    }

    public synchronized int peak() {
        return pilha[pos];
    }
}
```

Request Lock

Free Lock

Synchronized Blocks

In synchronized blocks we must specify the object from which the lock will be used.

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}

public void addEmploye(String name){
    synchronized(boss) {
        employeCount++;
    }
    nameList.add(name);
}
```

Request this lock

Free this lock

Request boss lock

Free boss lock

Synchronized Blocks

```
public class MultiCounter{
     private int c1 = 0;
     private int c2 = 0;
     private Object lock1 = new Object();
     private Object lock2 = new Object();

     public void incrementConter1() {
          synchronized(lock1) {
               c1++;
          }
      }

     public void incrementConter2() {
          synchronized(lock2) {
               c2++;
          }
     }
}
```

Explicit Locking - Lock Interface

void lock()	Acquires the lock.
boolean tryLock()	Acquires the lock only if it is free at the time of invocation.
void unlock()	Releases the lock.

```
Lock l = ...;
l.lock();
try {
   // access the resource protected by
      this lock
} finally {
   l.unlock();
}
```

Exercício 5.1

Incremento e decremento concorrente

Programe uma classe contador que contenha os seguintes métodos:

- `void incrementar()`
- `void decrementar()`
- `int consulta()`

Programe também uma classe (thread) onde cada instância incrementa o valor do contador e mostra na consola o nome do processo e o valor após o incremento. O processo deve repetir esta acção 20 vezes após o qual termina.

Teste o seu programa com quatro processos ligeiros e um contador.

To read...

- Object's API
- Recommended bibliography.
- Proposed exercises.

Bibliography

- JAVA tutorial:

http://download.oracle.com/javase/tutorial/essential/concurrency/sync.html

- JAVA Threads, O'Reilly, chapter 3
- Multithreaded Programming, Lewis & Berg, chapters 5-6

Summary

Threads Synchronization:

- Interference and memory consistency errors.
- Atomic instructions and actions
- Mutex locks
- Synchronized methods
- Synchronized blocks