L5: Race Conditions
(condiciones de carrera)

César Sánchez

Grado en Ingeniería Informática
Grado en Matemáticas e Informática
Universidad Politécnica de Madrid

Wed, 18-Feb-2015

Este texto se distribuye bajo los términos de la Creative Commons License
Mapa Conceptual

Concurrency = Simultaneous + Nondeterminism + Interaction

Interaction = Communication | Synchronization

Synchronization = Mutual Exclusion | Conditional Synchronization

- Terminology:
  - atomic
  - interleaving
  - mutual exclusion
  - deadlock
  - liveness
  - race condition
  - cache
  - context switch
Short HW2: Provocar una condición de carrera

Homework:
- HW1: Creación de threads en Java
- HW2: Provocar una condición de carrera
- HW3: Garantizar la exclusión mutua con espera activa
- HW4: Garantizar la exclusión mutua con semáforos

Fecha de Cierre:
Viernes 20-Febrero-2015 11am

Entrega online:
http://lml.ls.fi.upm.es/~entrega
Today’s topic: Race Condition

Race Condition
Today’s topic: Race Condition

Race Condition

A race condition or race hazard is the behavior of an electronic or software system where the output is dependent on the sequence or timing of other uncontrollable events.

It becomes a bug when events do not happen in the order the programmer intended.

The term originates with the idea of two signals racing each other to influence the output first.
Famous pitfalls

Therac 25
Famous pitfalls

Northeast 2003 blackout
Process vs Thread

Process

Thread
Process vs Thread

Process

- Protect resources (including memory)
- Context switch: slow
- Support from OS
- High overhead
- a.k.a Heavy process

Thread
Process vs Thread

Process

- **Protect** resources (including memory)
- Context switch: slow
- Support from OS
- High overhead
- a.k.a *Heavy process*

Thread

- **Shared** resources (including memory)
- Context switch: fast
- Support from OS or library
- Low overhead
- a.k.a *Lightweight process*
Race Conditions

In a nutshell:

Hard to find
Hard to repeat a trace  Puzzling behaviors
Hard to debug
Race Conditions

In a nutshell:

- Hard to find
- Hard to repeat a trace
- Hard to debug
- Puzzling behaviors

Only solution:

Prevent them!
Types of Race Conditions

Main types of race conditions:
Types of Race Conditions

Main types of race conditions:

1. Check-then-act

```java
public Singleton getInstance(){
    if(_instance == null){ //
        _instance = new Singleton();
    }
}
```
Types of Race Conditions

Main types of race conditions:

1. Check-then-act

```java
public Singleton getInstance(){
    if(_instance == null){ //
        _instance = new Singleton();
    }
}
```

2. Read-modify-write

```java
if(!hashtable.contains(key)){
    hashtable.put(key,value);
}
```
Types of Race Conditions

Main types of race conditions:

1. Check-then-act

```java
public Singleton getInstance(){
    if(_instance == null){ //
        _instance = new Singleton();
    }
}
```

2. Read-modify-write

```java
if(!hashtable.contains(key)){
    hashtable.put(key,value);
}
```

Root causes of race conditions:
- Shared data (among threads)
- Mutation of the shared data
Races: Sentence Level

```c
int a[3] = {3, 4, 5};

Thread 0
a[1] = a[0] + a[1];

Thread 1
```
Races: Sentence Level

```c
int a[3] = {3, 4, 5};
```

Thread 0
```
a[1] = a[0] + a[1];
```

Thread 1
```
```

```
a = { 3, ?, ?, ? }
```
Races: Sentence Level

```c
int a[3] = {3, 4, 5};
```

Thread 0
```
a[1] = a[0] + a[1];
```

Thread 1
```
```

```
a = {3, ?, ?, ?}
```

Thread 0 ; Thread 1 ➔ a = {3, 7, 12}
Races: Sentence Level

```c
int a[3] = {3, 4, 5};

Thread 0
a[1] = a[0] + a[1];

Thread 1

a = {3, 7, 9}
```

Thread 0 ; Thread 1 → a = {3, 7, 12}

Thread 1 ; Thread 0 → a = {3, 7, 9}
Races: Instruction Level

```
int count = 10

Thread 0          ||          Thread 1
count++;          ||          count--;
```
Races: Instruction Level

```java
int count = 10

Thread 0
count++;

Thread 1
count--;

count = ?
```
Races: Instruction Level

```plaintext
int count = 10

Thread 0
=count++;

LOAD Reg, count
ADD #1
STORE Reg, count

Thread 1
=count--;

LOAD Reg, count
SUB #1
STORE Reg, count

count = ?
```
Races: Instruction Level

```c
int count = 10

Thread 0
    count++;

Thread 1
    count--;
```

Instructions:

<table>
<thead>
<tr>
<th>Inst</th>
<th>Reg</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>ADD</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>STORE</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inst</th>
<th>Reg</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>SUB</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>STORE</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>
Races: Instruction Level

```java
int count = 10

Thread 0

count++;

LOAD Reg, count
ADD #1
STORE Reg, count

Thread 1

count--;

LOAD Reg, count
SUB #1
STORE Reg, count

count = 11

<table>
<thead>
<tr>
<th>Inst</th>
<th>Reg</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>ADD</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>STORE</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inst</th>
<th>Reg</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>SUB</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>STORE</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>
```
Races: Instruction Level (2)

Consider an 8 bit architecture

```c
int x=0
```

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=271</td>
<td>x=1</td>
</tr>
</tbody>
</table>

```
x = ?
```
Races: Instruction Level (2)

Consider an 8 bit architecture

```c
int x=0

Thread 0 || Thread 1
x=271    ||    x=1
```


```c
x = 271, 1, 257, 15
```
Races: Instruction Level (2)

Consider an 8 bit architecture

```c
int x=0

Thread 0                      ||                      Thread 1
x=271                           ||                           x=1
```

\[ x = 271, 1, 257, 15 \]

Why??
Races: Instruction Level (2)

Consider an 8 bit architecture

```c
int x = 0
```

Thread 0:
- `x = 271`
- `x_high = 0x01`
- `x_low = 0x0f`

Thread 1:
- `x = 1`
- `x_high = 0x00`
- `x_low = 0x01`

$x = 271, 1, 257, 15$

**Why??**
Races: Instruction Level (2)

Consider an 8 bit architecture

```
int x=0
```

Thread 0

```
x=271
x_high = 0x01
x_low = 0x0f
```

Thread 1

```
x=1
x_high = 0x00
x_low = 0x01
```

```
x = 271, 1, 257, 15
```

0x010f

0x0001

0x000f

0x0101
Races: Hardware “reordering”

Consider a modern processor with a cache

```plaintext
int x=0
int y=0
```

Thread 0

x=1
print y

Thread 1

y=1
print x

prints:
Races: Hardware “reordering”

Consider a modern processor with a cache

```
int x=0
int y=0
```

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=1</td>
<td>y=1</td>
</tr>
<tr>
<td>print y</td>
<td>print x</td>
</tr>
</tbody>
</table>

prints:  

```
"1 1"
"0 1"
"1 0"
"0 0" Why??
```
Races: Hardware “reordering”

Consider a modern processor with a cache

```
int x=0
int y=0
```

```
Thread 0
  x=1
  print y
```

```
Thread 1
  y=1
  print x
```

prints:

```
"1 1"
"0 1"
"1 0"
"0 0"
```

Why??

Caches!
How can a race happen?

Two cases:
How can a race happen?

Two cases:

- **static Variables:**
  - Two threads modify the same static variable
- **Steps:**
  - The code of two threads defined to access (read and write) the same verb+static+
  - Build two or more thread instances
  - *During execution, these threads access concurrently the same memory*
How can a race happen?

Two cases:

- **static Variables:**
  - Two threads modify the same static variable
  - **Steps:**
    - The code of two threads defined to access (read and write) the same verb+static+
    - Build two or more thread instances
    - **During execution, these threads access concurrently the same memory**

- **Passing objects by reference:**
  - Java objects (unlike basic types) are passed by reference
  - **Steps:**
    - The code of two threads defined to access (read and write) the same object during execution
    - Build two or more thread instances
    - **During execution, these threads access concurrently the same instance**
How can a race happen?

Two cases:

- **static Variables:**
  - Two threads modify the same static variable
  - Steps:
    - The code of two threads defined to access (read and write) the same variable
    - Build two or more thread instances
    - During execution, these threads access concurrently the same memory

- **Passing objects by reference:**
  - Java objects (unlike basic types) are passed by reference
  - Steps:
    - The code of two threads defined to access (read and write) the same object during execution
    - Build two or more thread instances
    - During execution, these threads access concurrently the same instance

**LESSON:**
Unless we control these cases, race conditions can happen.
Volatile variables

A *volatile* variable forces the thread to sync to main memory, a prevents hardware reorderings
Volatile variables

A volatile variable forces the thread to sync to main memory, a prevents hardware reorderings
Volatile variables

A **volatile** variable forces the thread to sync to main memory, a prevents hardware reorderings

- **Visibility is synchronized** (even for long and double)
- Writes **propagate to** all threads before continuing
- Compiler cannot reorder

**It only affects to reads and writes of a single volatile variable**
- r.g. ++ over a volatile is **NOT** atomic

Performance:
- Volatile is more expensive (because accesses to main memory, and less compiler optimization)
Chapter 17 of the Java Language Specification:

If one action *happens-before* another, then the first is visible to and ordered before the second.
If one action *happens-before* another, then the first is visible to and ordered before the second.

- If x and y occur in the same thread and x comes before y in program order then $hb(x, y)$,
- $hb$ is transitive
- A write to a volatile happens-before every subsequent read.
- A call to start() on a thread happens-before any actions in the started thread.
- All actions in a thread happen-before any other thread successfully returns from a join() on that thread.
Dekker with volatile

Consider a modern processor with a cache

```c
volatile int x=0
volatile int y=0
```

Thread 0
```
x=1
print y
```

Thread 1
```
y=1
print x
```

prints:
```
"1 1"
"0 1"
"1 0"
"0 0"
```

prints: 
```
Example of volatile

In the following code (that does not use volatile), assume:
- one thread runs one() and
- many threads run two()

```java
class Test {
    static int i = 0, j = 0;
    static void one() { i++; j++ }
    static void two() {
        System.out.println("i=");
    }
}
```
Example of volatile

In the following code (that does not use volatile), assume:
- one thread runs one() and
- many threads run two()

```java
class Test {
    static int i = 0, j = 0;
    static void one() { i++; j++ }
    static void two() {
        System.out.println("i=\" + i + " j=\" + j);
    }
}
```

The value of j printed may be higher than i
Example of volatile

In the following code (that does uses volatile), assume:
- one thread runs one() and
- many threads run two()

```java
class Test {
    static volatile int i = 0, j = 0;
    static void one() { i++; j++ }
    static void two() {
        System.out.println("i=", + i + " j=", + j);
    }
}
```
Example of volatile

In the following code (that does uses volatile), assume:
- one thread runs `one()` and
- many threads run `two()`

```java
class Test {
    static volatile int i = 0, j = 0;
    static void one() { i++; j++ }
    static void two() {
        System.out.println("i=" + i + " j=" + j);
    }
}
```

The value of j printed is never higher than i
Race Conditions. Conclusions

Race conditions are hard to find, debug and test.

Race conditions must be avoided by proper synchronization.
Race Conditions. Conclusions

Race conditions are hard to find, debug and test.

Race conditions must be avoided by proper synchronization.