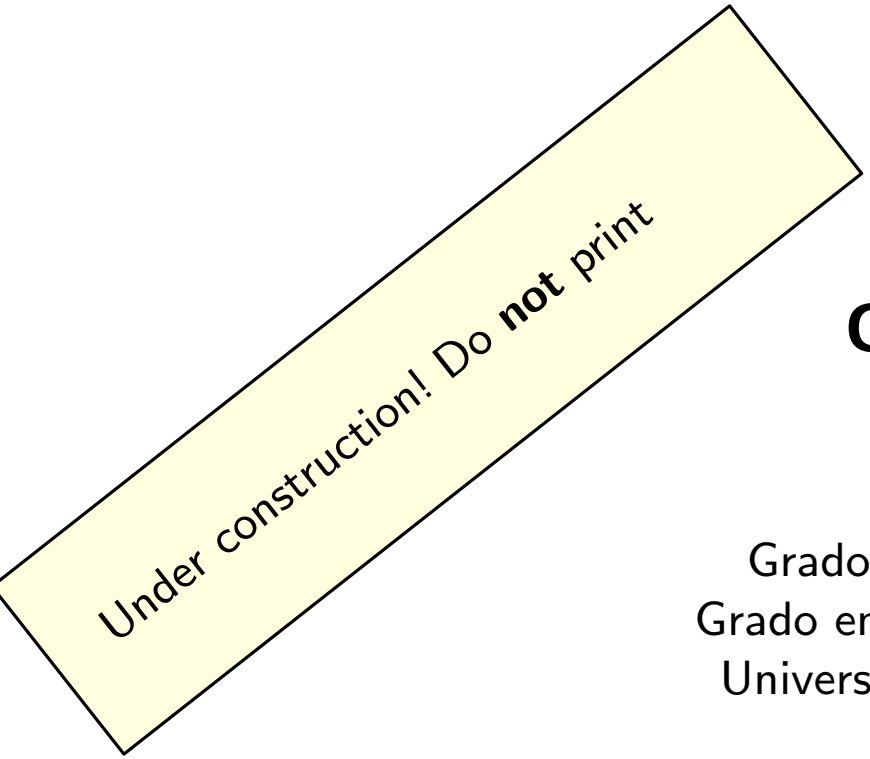
L7: Busy-Wait



César Sánchez

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

Wed, 25 Feb 2015

Mapa Conceptual

$$\text{Concurrency} = \text{Simultaneous} + \text{Nondeterminism} + \text{Interaction}$$

$$\text{Interaction} = \text{Communication} \mid \text{Synchronization}$$

$$\text{Synchronization} = \text{Mutual Exclusion} \mid \begin{array}{l}\text{Conditional} \\ \text{Synchronization}\end{array}$$

- Terminology:

atomic	race condition
interleaving	busy-wait
mutual exclusion	critical section
deadlock	livelock
liveness	

Critical Section and Mutual exclusion

Critical Section:

a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution. A critical section will usually terminate in fixed time, and a thread will have to wait for a fixed time to enter it.

Mutual exclusion:

the requirement of ensuring that no two concurrent processes are in their critical section at the same time; it is a basic requirement in concurrency control, to prevent race conditions.

Busy0.java

```
static class Incrementador2 extends Thread {
   public void run() {
      for (int i = 0; i < N_OPS; i++) {
         // CRITICAL_BEGIN:
         en_sc_inc = true;
         cont++;
         // intentamos detectar violaciones de mutex
         if (en_sc_inc && en_sc_dec) {
            System.out.print("CRAAAAAAAAAASH!!!!!!!\n");
         }
         en_sc_inc = false;
         // CRITICAL_END
      }
   }
}
```

Busy0.java

Problems?

```
static class Incrementador2 extends Thread {
   public void run() {
      for (int i = 0; i < N_OPS; i++) {
         // CRITICAL_BEGIN:
         en_sc_inc = true;
         cont++;
         // intentamos detectar violaciones de mutex
         if (en_sc_inc && en_sc_dec) {
            System.out.print("CRAAAAAAAAAASH!!!!!!!\n");
         }
         en_sc_inc = false;
         // CRITICAL_END
      }
   }
}
```

BusyAE.java

```
static class Incrementador2 extends Thread {
  public void run() {
     for (int i = 0; i < N_OPS; i++) {
        while (turno != INC) {};
        en_sc_inc = true;
        cont++;
        if (en_sc_inc && en_sc_dec) {
           System.out.print("CRAAAAAAAAAASH!!!!!!!\n");
        }
        if (ultima_op == INC) {
           probable_alternancia_estricta = false;
        }
        ultima_op = INC;
        en_sc_inc = false;
        turno = DEC;
     }
  }
}
```

BusyAE.java

```
static class Incrementador2 extends Thread {
  public void run() {
    for (int i = 0; i < N_OPS; i++) {
      while (turno != INC) {};
      en_sc_inc = true;
      cont++;
      if (en_sc_inc && en_sc_dec) {
        System.out.print("CRAAAAAAAAAASH!!!!!!!\n");
      }
      if (ultima_op == INC) {
        probable_alternancia_estricta = false;
      }
      ultima_op = INC;
      en_sc_inc = false;

    }
  }
}
```

Protocolo Entrada

Protocolo Salida

BusyAE.java

```
static class Incrementador2 extends Thread {
  public void run() {
    for (int i = 0; i < N_OPS; i++) {
      while (turno != INC) {};
      en_sc_inc = true;
      cont++;
      if (en_sc_inc && en_sc_dec) {
        System.out.print("CRAAAAAAAAAASH!!!!!!!\n");
      }
      if (ultima_op == INC) {
        probable_alternancia_estricta = false;
      }
      ultima_op = INC;
      en_sc_inc = false;

    }
  }
}
```

Protocolo Entrada

Critical Section

Protocolo Salida

Busy-wait

Busy-waiting:

> *busy-waiting or spinning is a technique in which a process repeatedly checks to see if a condition is true, such as whether keyboard input or a lock is available.*

Busy-wait

Busy-waiting:

busy-waiting or spinning is a technique in which a process repeatedly checks to see if a condition is true, such as whether keyboard input or a lock is available.

Busy-wait

Busy-waiting:

> *busy-waiting or spinning is a technique in which a process repeatedly checks to see if a condition is true, such as whether keyboard input or a lock is available.*

```
    static class Incrementador2 extends Thread {
      public void run() {
         //...
         while (turno != INC) {};
         //...
   }
}
```

Strict Alternation

Strict Alternation:

Mutual exclusion can be achieved by a thread going from non critical region to the critical region by checking the value of turn, and switching turns at the end of the critical section.

```
static class Incrementador2
extends Thread {
    public void run() {
        //..
        while (turno != INC) {};
        //...
        turno = DEC;
        }
    }
}
```

```
static class Decrementador2
extends Thread {
    public void run() {
        //..
        while (turno != DEC) {};
        //...
        turno = INC;
        }
    }
}
```

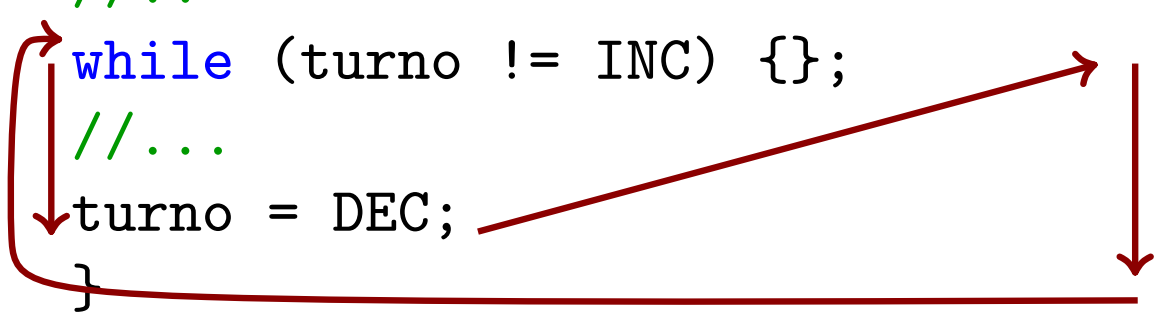
Strict Alternation

Strict Alternation:

Mutual exclusion can be achieved by a thread going from non critical region to the critical region by checking the value of turn, and switching turns at the end of the critical section.

```
static class Incrementador2
extends Thread {
    public void run() {
        //..
        while (turno != INC) {};
        //...
        turno = DEC;
        }
    }
}
```

```
static class Decrementador2
extends Thread {
    public void run() {
        //..
        while (turno != DEC) {};
        //...
        turno = INC;
        }
    }
}
```



BusyNM.java

```
    static class Incrementador2 extends Thread {
     public void run() {
        for (int i = 0; i < N_OPS; i++) {
           while (en_sc_dec) { };
           // avisamos de que estamos dentro
           en_sc_inc = true;
           // seccion critica
           cont++;
           // intentamos detectar violaciones de mutex
           if (en_sc_inc && en_sc_dec) {
              System.out.print("CRAAAAAAAAAASH!!!!!!!\n");
           }
           // protocolo de salida
           en_sc_inc = false;
        }
     }
  }
```

BusyNM.java

```
static class Incrementador2 extends Thread {
 public void run() {
    for (int i = 0; i < N_OPS; i++) {
       while (en_sc_dec) { };
       // avisamos de que estamos dentro
       en_sc_inc = true;
       // seccion critica
       cont++;
       // intentamos detectar violaciones de mutex
       if (en_sc_inc && en_sc_dec) {
          System.out.print("CRAAAAAAAAAASH!!!!!!!\n");
       }
       // protocolo de salida
       en_sc_inc = false;
    }
  }
}
```

BusyLL.java

```
static class Incrementador2 extends Thread {
  public void run() {
     for (int i = 0; i < N_OPS; i++) {
        incrementador_quiere = true; // PROTOCOL IN
        while (decrementador_quiere) {
           busy_inc = true;
           if (busy_inc && busy_dec) {
              System.out.print("INC: la he liao parda!!\n");
           }
        }
        busy_inc = false;
        critical_section();   //CRITICAL
        incrementador_quiere = false; //PROTOCOL OUT
     }
  }
}
```

BusyLL.java

```
static class Incrementador2 extends Thread {
  public void run() {
     for (int i = 0; i < N_OPS; i++) {
        incrementador_quiere = true; // PROTOCOL IN
        while (decrementador_quiere) {
           busy_inc = true;
           if (busy_inc && busy_dec) {
              System.out.print("INC: la he liao parda!!\n");
           }
        }
        busy_inc = false;
        critical_section();   //CRITICAL
        incrementador_quiere = false; //PROTOCOL OUT
     }
  }
}
```

Deadlock

Deadlock:

> *a deadlock is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does.*

Deadlock

Deadlock:

> *a deadlock is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does.*

"When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

Statute passed by the Kansas State Legislature, early in the 20th century.

Livelock

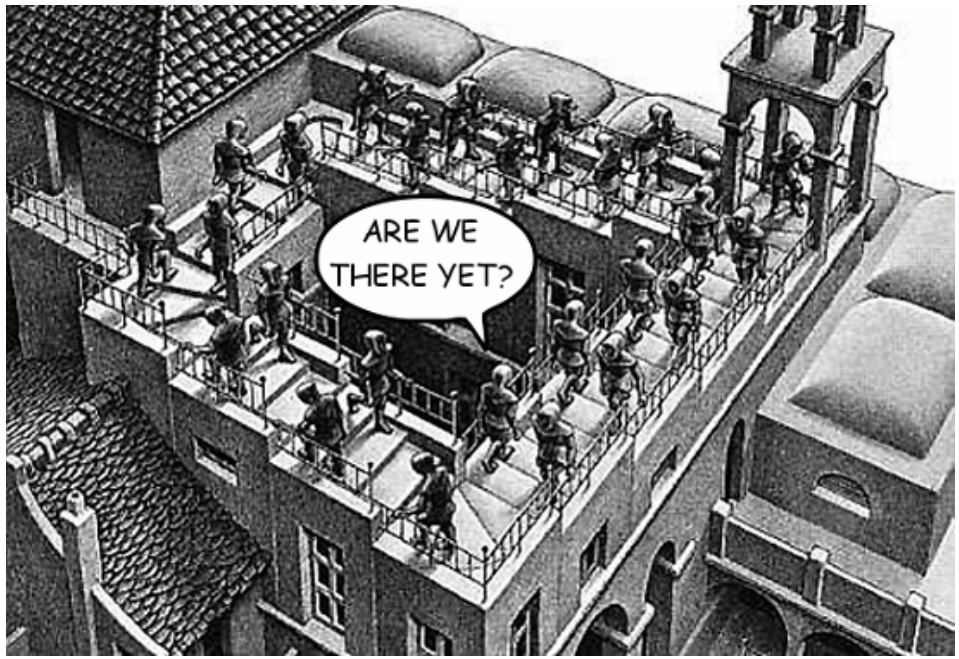
Livelock:

A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing.

Livelock

Livelock:

A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing.



BusyST.java

```
static class Incrementador2 extends Thread {
  public void run() {
     for (int i = 0; i < N_OPS; i++) {
        incrementador_quiere = true;    //PROTOCOL IN
        while (decrementador_quiere) {
           incrementador_quiere = false;
           incrementador_quiere = true;
        }
        critical_section();             //CRITICAL SECTION
        incrementador_quiere = false;   //PROTOCOL OUT
     }
  }
}
```

BusyST.java

```
static class Incrementador2 extends Thread {
  public void run() {
     for (int i = 0; i < N_OPS; i++) {
        incrementador_quiere = true;     //PROTOCOL IN
        while (decrementador_quiere) {
           incrementador_quiere = false;
           incrementador_quiere = true;
        }
        critical_section();              //CRITICAL SECTION
        incrementador_quiere = false;    //PROTOCOL OUT
     }
  }
}
```

Starvation

Starvation:

> *Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads.*