instrucciones básicas

- Obviamente tenemos que asumir que ciertas acciones de un proceso se pueden realizar correctamente independientemente de las acciones de los demás procesos.
- Dichas acciones se llaman (también) atómicas (porque son indivisibles) y se garantizan por hardware.
- Normalmente, asumimos que podemos acceder a variables de cierto tipo (p.ej. enteros) de forma atómica con lectura y escritura (`load` y `store`)

Implementamos varios protocolos de exclusión mutua (aquí solamente para dos procesos) que solamente están basadas en instrucciones simples.

Un posible protocolo (asimétrico)

```
  P0                                  P1
a: loop                               loop
b:  non-critical section               non-critical section
c:  set v0 to true                     set v1 to true
d:  wait until v1 equals false         while v0 equals true
e:                                      set v1 to false
f:                                      wait until v0 equals false
g:                                      set v1 to true
h:  critical section                   critical section
i:  set v0 to false                    set v1 to false
j: endloop                            endloop
```

principio (teorema) de la bandera

El principio de la bandera es un teorema que podemos usar para comprobar si está garantizado la exclusión mutua para dos procesos en un código concreto.

- Si dos procesos primero levantan sus banderas
- y después miran al otro lado
- por lo menos uno de los procesos ve la bandera del otro levantado.

comprobación con contradicción

- asumimos P0 era el último en mirar
- entonces la bandera de P0 está levantada
- asumimos que P0 no ha visto la bandera de P1
- entonces P1 ha levantado la bandera después de la mirada de P0
- pero P1 mira después de haber levantado la bandera
- entonces P0 no era el último en mirar

propiedades de interés del protocolo

Normalmente, un protocolo de entrada y salida debe cumplir con las siguientes condiciones:

- sólo un proceso debe obtener acceso a la sección crítica (garantía del acceso con exclusión mutua)
- por lo menos un proceso debe obtener acceso a la sección crítica después de un tiempo de espera *finito*.
- Obviamente se asume que ningún proceso ocupa la sección crítica durante un tiempo infinito.

propiedades de interés del protocolo

La propiedad de espera finita se puede analizar según los siguientes criterios:

justicia:
: hasta que medida influyen las peticiones de los demás procesos en el tiempo de espera de un proceso

espera:
: hasta que medida influyen los protocolos de los demás procesos en el tiempo de espera de un proceso

tolerancia a fallos:
: hasta que medida influyen posibles errores de los demás procesos en el tiempo de espera de un proceso.

análisis del protocolo asimétrico

Analizamos el protocolo de antes respecto a dichos criterios:

- ¿Está garantizado la exclusión mutua?
- ¿Influye el estado de uno (sin acceso) en el acceso del otro?
- ¿Quién gana en caso de peticiones simultáneas?
- ¿Qué pasa en caso de error?

soporte hardware

- Dependiendo de las capacidades del hardware la implementación de los protocolos de entrada y salida es más fácil o más difícil, además las soluciones pueden ser más o menos eficientes.
- Vimos que se pueden realizar protocolos seguros solamente con las instrucciones `load` y `store` de un procesador.
- Las soluciones no suelen ser muy eficientes, especialmente si muchos procesos compiten por la sección crítica. *Pero: su desarrollo y la presentación de la solución ayuda en entender el problema principal.*
- A veces no hay otra opción disponible.
- Todos los microprocesadores modernos proporcionan instrucciones básicas que permiten realizar los protocolos de forma más directo y en muchas ocaciones más eficiente.

protocolo simple: primer intento

Usamos una variable `v` que nos indicará cual de los dos procesos tiene su turno.

```
  P0                                P1
a: loop                              loop
b:  wait until v equals P0            wait until v equals P1
c:  critical section                  critical section
d:  set v to P1                       set v to P0
e:  non-critical section              non-critical section
f: endloop                           endloop
```

primer intento: propiedades

- Está garantizada la exlusión mutua porque un proceso llega a su línea `c:` solamente si el valor de `v` corresponde a su identificación (que asumimos siendo única).
- Obviamente, los procesos pueden acceder al recurso solamente alternativamente, que puede ser inconveniente porque acopla los procesos fuertemente.
- Un proceso no puede entrar más de una vez seguido en la sección crítica.
- Si un proceso termina el programa o no llega más por alguna razón a su línea `d:`, el otro proceso puede resultar bloqueado.
- La solución se puede ampliar fácilmente a más de dos procesos.

primer intento en Java (comenzar es fácil)

El protocolo del primer intento para implementar un ping pong.

```
public class PingPong {
  volatile static int turn;  // It's v.

  public static void main(String[] args) {
    Thread ping1 = new Player(1);
    Thread ping2 = new Player(2);

    ping1.start();
    ping2.start();

    turn=1;
  }
```

primer intento en Java (comenzar es fácil)

```
class Player extends Thread {
  private int id;
  public Player(int id) { this.id=id; }
  public void run() {
    while(true) {                          // a:
      while(
        id!=PingPong.turn                  // b:
      );
      System.out.print("ping"+id+" ");// c:
      PingPong.turn=id%2+1;                // d:
    }                                      // f:
  }
}
```

Excurso: primer intento en Java (terminar no es tan fácil)

un método para abreviar:

```
  static void Wait(int us) {
    try {
      Thread.sleep(us);
    } catch(InterruptedException e) {
      System.out.println("sleeping interrupted");
    }
  }
}
```

Excurso: primer intento en Java (terminar no es tan fácil)

```
public class PingPong {
  volatile static int turn;  // It's v.
  public static void main(String[] args) {
    Thread ping1 = new Player(1);
    Thread ping2 = new Player(2);
    ping1.start();
    ping2.start();
    System.out.println("playing some seconds");
    turn=1;
    Wait(2000);
    System.out.println("waiting for players");
    turn=3;  // Try to stop :-)
    try {
      ping1.join();
      ping2.join();
    }
    catch(InterruptedException e) {
      System.out.println("got interrupted");
    }
    System.out.println("finished");
  }
```

primer intento en Java (comenzar es fácil)

```
class Player extends Thread {
  private int id;
  public Player(int id) { this.id=id; }
  public void run() {
    while(true) {                          // a:
      while(
        id!=PingPong.turn                  // b:
      );
      System.out.print("ping"+id+" ");// c:
      PingPong.turn=id%2+1;                // d:
    }                                      // f:
  }
}
```

Excurso: primer intento en Java (terminar no tanto)

```
class Player extends Thread {
  private int id;
  public Player(int id) { this.id=id; }
  public void run() {
    while(PingPong.turn!=3) {
      while(
        id!=PingPong.turn &&
        PingPong.turn!=3
      );
      System.out.print("ping"+id+" ");
      if(PingPong.turn==id) {
        PingPong.turn=id%2+1;
      }
    }
  }
}
```

Excurso: primer intento en Java (terminar no tanto)

```
class Player extends Thread {
  private int id;
  public Player(int id) { this.id=id; }
  public void run() {
    while(PingPong.turn!=3) {
      while(
        id!=PingPong.turn &&
        PingPong.turn!=3
      );
      System.out.print("ping"+id+" ");
      if(PingPong.turn==id) {
        PingPong.Wait(100);        // And blocking!!!
        PingPong.turn=id%2+1;
      }
    }
  }
}
```

segundo intento

Intentamos evitar la alternancia. Usamos para cada proceso una variable, `v0` para `P0` y `v1` para `P1` respectivamente, que indican si el correspondiente proceso está usando el recurso.

```
  P0                                  P1
a: loop                                loop
b:  wait until v1 equals false          wait until v0 equals false
c:  set v0 to true                      set v1 to true
d:  critical section                    critical section
e:  set v0 to false                     set v1 to false
f:  non-critical section                non-critical section
g: endloop                             endloop
```

segundo intento: propiedades

- Ya no existe la situación de la alternancia.
- Sin embargo: el algoritmo no es correcto, porque los dos procesos pueden alcanzar sus secciones críticas simultáneamente.

- El problema está escondido en el uso de las variables de control. `v0` se debe cambiar a verdadero solamente si `v1` sigue siendo falso.
- ¿Cuál es la intercalación maligna?

tercer intento

Cambiamos el lugar donde se modifica la variable de control:

```
  P0                                P1
a: loop                              loop
b:  set v0 to true                    set v1 to true
c:  wait until v1 equals false        wait until v0 equals false
d:  critical section                  critical section
e:  set v0 to false                   set v1 to false
f:  non-critical section              non-critical section
g: endloop                           endloop
```

tercer intento: propiedades

- Está garantizado que ambos procesos no entren al mismo tiempo en sus secciones críticas.
- Pero se bloquean mutuamente en caso que lo intenten simultáneamente que resultaría en una espera infinita.

- ¿Cuál es la intercalación maligna?

cuarto intento

Modificamos la instrucción `c:` para dar la oportunidad que el otro proceso encuentre su variable a favor.

```
  P0                              P1
a: loop                           loop
b:  set v0 to true                 set v1 to true
c:  repeat                         repeat
d:   set v0 to false                set v1 to false
e:   set v0 to true                 set v1 to true
f:  until v1 equals false          until v0 equals false
g:  critical section               critical section
h:  set v0 to false                set v1 to false
i:  non-critical section           non-critical section
j: endloop                        endloop
```

cuarto intento: propiedades

- Está garantizado la exclusión mutua.
- Se puede producir una variante de bloqueo: los procesos hacen algo pero no llegan a hacer algo útil (*livelock*)

- ¿Cuál es la intercalación maligna?

algoritmo de Dekker: quinto intento

```
Initially:  v0,v1 are equal to false, v is equal to P0 o P1

  P0                              P1
a: loop                           loop
b:  set v0 to true                 set v1 to true
c:  loop                           loop
d:   if v1 equals false exit        if v0 equals false exit
e:   if v equals P1                 if v equals P0
f:    set v0 to false                set v1 to false
g:    wait until v equals P0         wait until v equals P1
h:    set v0 to true                 set v1 to true
i:   fi                             fi
j:  endloop                        endloop
k:  critical section               critical section
l:  set v0 to false                set v1 to false
m:  set v to P1                    set v to P0
n:  non-critical section           non-critical section
o: endloop                        endloop
```

quinto intento: propiedades

El algoritmo de Dekker resuelve el problema de exclusión mutua en el caso de dos procesos, donde se asume que la lectura y la escritura de un valor íntegro de un registro se puede realizar de forma atómica.

otros algoritmos para obtener exclusión mutua

Existen otros algoritmos que solamente con operaciones atómicas de `load` y `store` consiguen un acceso con exclusión mutua a secciones críticas. Algunos históricos que resuelven para n-procesos:

- algoritmo de Peterson
- algoritmo de Lamport
- algoritmo de Eisenberg–McGuire

Operaciones en la memoria

- Existen instrucciones más potentes (que los simples `load` y `store`) en los microprocesadores actuales para la realización la exclusión mutua más fácil.
- (Casi) todos los procesadores implementan varios tipos de instrucciones atómicas que realizan algún cambio en la memoria al mismo tiempo que devuelve el contenido anterior de la memoria.
- La idea principal es implementar ciclos de *read-modify-write* de forma atómica directamente en memoria compartida.
- El hecho que tienen que actuar finalmente en memoria compartida suelen tener una eficiencia reducida ya que no pueden aprovechar tanto de la jerarquia de memoria (cachés).

TAS

La instrucción `test-and-set` (TAS) implementa

- una comprobación a cero del contenido de una variable en la memoria
- al mismo tiempo que varía su contenido (normal a 1 en hardware)
- en caso que la comprobación se realizó con el resultado verdadero.
- https://en.wikipedia.org/wiki/Test-and-set

CAS

- La instrucción `compare-and-swap` (CAS) es una generalización de la instrucción `test-and-set`.
- La instrucción trabaja con dos variables, les llamamos `C` (de *compare*) y `S` (de *swap*).
- Se intercambia el valor en la memoria por `S` si el valor en la memoria es igual que `C`.
- Se devuelve un booleano con un valor dependiendo si se ha realizado el intercambio o no.
- https://en.wikipedia.org/wiki/Compare-and-swap
- Es la operación que se usa por ejemplo en los procesadores de Intel y es la base para facilitar la concurrencia en la máquina virtual de Java 1.5 para dicha familia de procesadores.

otras posibilidades

Existen otras operaciones hardware para conseguir sincronización entre procesos en la memoria, ejemplos son:

- EXCH (atomic exchange)
- F&A (fetch-and-add)
- DCAS (double compare-and-swap)
- LL/SC (link load / store conditional)
- y muchos más...

Uso de memoria común

- Como hemos visto todos los protocolos necesitan variables con acceso atómico en memoría compartida entre procesos.
- Tal hecho puede resultar en pérdidas de rendimiento, si existe una jerarquía de memoria con diferentes niveles de cachés.
- Muchos compiladores ofrecen acceso directo a las instrucciones de nivel bajo, por ejemplo la gama de compiladores GCC con sus *atomic builtins*, (https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html)
- Nuevas versiones ya lo hacen diferente (https://gcc.gnu.org/onlinedocs/gcc-8.3.0/gcc/)
- Mirad por ejemplo: https://gcc.gnu.org/onlinedocs/gcc-8.3.0/libstdc++/manual/
- C++ y OpenMP modernos ofrecen accesos atómicos (tanto de instrucciones como de secciones) a nivel alto del lenguaje.

ABA problem

no se transmite information

- Los protocolos de entrada y salida como implementados hasta ahora no transmiten información de un hilo al otro,
- en el sentido que si un hilo entra en su sección crítica dos veces,
- no puede averiguar si otro hilo ha (o otros hilos han) entrado mientras tanto entre sus dos entradas.
- Este problema se conoce como ABA-problema (la secuencia *primero A, luego B, después A*, no se puede distinguir del simple hecho *solo A*).
- Un ejemplo, donde este problema puede ser relevante es, si se compara solo punteros o referencias para averiguar posibles cambios de estado,
- por ejemplo en acciones sobre listas compartidas (secuencias de borrar e insertar),
- puede ser que los punteros no se modificaron (ya que están protegidos con exclusión mutua) pero el lugar a donde apuntan sí.