un programa concurrente

- asumimos que tengamos un programa concurrente que quiere realizar acciones con recursos:
- si los recursos de los diferentes procesos son diferentes no hay problema,
- si dos (o más procesos) quieren manipular el mismo recurso ¿Qué hacemos?

¿Qué es exclusión mutua?

- Para evitar el acceso concurrente a recursos compartidos hace falta instalar un mecanismo de control
  - que permite la entrada de un proceso si el recurso está disponible y
  - que prohibe la entrada de un proceso si el recurso está ocupado.
- Es importante entender cómo se implementan los protocolos de entrada y salida para realizar la exclusión mutua.
- Obviamente no se puede implementar exclusión mutua usando exclusión mutua: se necesita algo más básico.
- Un método es usar un tipo de protocolo de comunicación basado en las instrucciones básicas disponibles.

estructura general basada en protocolos

Entonces el protocolo para cada uno de los participantes refleja una estructura como sigue:

```
P0                          ... Pi
...\\                           ...\\
entrance protocol               entrance protocol
critical section                critical section
exit protocol                   exit protocol
...\\                           ...\\
```

instrucciones básicas

- obviamente tenemos que asumir que ciertas acciones de un procesos se puede realizar correctamente independientemente de las acciones de los demás procesos
- dichas acciones se llaman “atómicas” (porque son indivisibles) y se garantizan por hardware
- asumimos que podemos acceder a variables de cierto tipo (p.ej. entero) de forma atómica con lectura y escritura (`load` y `store`)

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 de la bandera

Si

- ambos procesos primero levantan sus banderas
- y después miran al otro lado

por lo menos un proceso 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

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)
- un proceso debe obtener acceso a la sección crítica después de un tiempo de espera *finita*

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 simultaneas?
- ¿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 facil o más difícil, además las soluciones pueden ser más o menos eficientes.
- Vimos y veremos 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 problem principal.
- A veces no hay otra opción disponible.
- Todos los microprocesadores modernos proporcionan instrucciones básicas que permiten realizar los protocolos de forma más eficiente.

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.

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 está seguro, 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 no entren ambos procesos al mismo tiempo en sus secciones críticas.
- Pero se bloquean mutuamente en caso que lo intentan simultaneamente 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.

algoritmo de Peterson

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

algoritmo de Lamport

o algoritmo de la panadería:

- cada proceso tira un ticket (que están ordenados en orden ascendente)
- cada proceso espera hasta que su valor del ticket sea el mínimo entre todos los procesos esperando
- el proceso con el valor mínimo accede la sección crítica

algoritmo de Lamport: observaciones

- se necesita un cerrojo (acceso con exclusión mutua) para acceder a los tickets
- el número de tickets no tiene límite
- los procesos tienen que comprobar continuadamente todos los tickets de todos los demás procesos

El algoritmo no es verdaderamente practicable dado que se necesitan infinitos tickets y un número elevado de comprobaciones.
Si se sabe el número máximo de participantes basta con un número fijo de tickets.

otros algoritmos

- Como vimos, el algoritmo de Lamport (algoritmo de la panadería) necesita muchas comparaciones de los tickets para *n* procesos.
- Existe una versión de Peterson que usa solamente variables confinadas a cuatro valores.
- Existe una generalización del algoritmo de Peterson para *n* procesos (filter algorithm).
- Se puede evitar la necesidad de un número infinito de tickets, si se conoce antemano el número máximo de participantes (uso de grafos de precedencia).
- Otra posibilidad es al algoritmo de Eisenberg–McGuire (que garantiza una espera mínima para *n* procesos).

límites

- Se puede comprobar que se necesita por lo menos n campos en la memoria para implementar un algoritmo (con `load` and `store`) que garantiza la exclusión mutua entre n procesos.

Operaciones en la memoria

- Si existen instrucciones más potentes (que los simples `load` y `store`) en el microprocesador se puede realizar la exclusión mutua más fácil.
- Hoy casi todos los procesadores implementan un tipo de instrucción atómica que realiza algún cambio en la memoria al mismo tiempo que devuelve el contenido anterior de la memoria.

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
- en caso que la comprobación se realizó con el resultado verdadero.

TAS

```
Initially:  vi is equal false
            C  is equal true

a: loop
b:   non-critical section
c:   loop
d:     if C equals true                ; atomic
         set C to false and exit
e:   endloop
f:   set vi to true
g:   critical section
h:   set vi to false
i:   set C to true
j: endloop
```

TAS: propiedades

- En caso de un sistema multi-procesador hay que tener cuidado que la operación `test-and-set` esté realizada en la memoria compartida.
- Teniendo solamente una variable para la sincronización de varios procesos el algoritmo arriba no garantiza una espera limitada de todos los procesos participando. ¿Por qué?
- Para conseguir una espera limitada se implementa un protocolo de paso de tal manera que un proceso saliendo de su sección crítica da de forma explícita paso a un proceso esperando (en caso que tal proceso exista).
- ¿Cómo se puede garantizar una espera limitada?

EXCH

La instrucción `exchange` (a veces llamado `read-modify-write`)

- intercambia un registro del procesador
- con el contenido de una dirección de la memoria

en una instrucción atómica.

EXCH

```
Initially:  vi is equal false
            C  is equal true

a: loop
b:   non-critical section
c:   loop
d:     exchange C and vi          ; atomic exchange
e:     if vi equals true exit
f:   endloop
g:   critical section
h:   exchange C and vi
i: endloop
```

EXCH: propiedades

- Se observa lo mismo que en el caso anterior, no se garantiza una espera limitada.
- ¿Cómo se consigue?

F&A

La instrucción `fetch-and-increment`

- aumenta el valor de una variable en la memoria
- y devuelve el resultado

en una instrucción atómica.

- Con dicha instrucción se puede realizar los protocolos de entrada y salida.
- ¿Cómo?
- También existe en la versión `fetch-and-add` que en vez de incrementar suma un valor dado de forma atómica.

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`.
- 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.
- Con CAS se pueden realizar los protocolos de entrada y salida. ¿Cómo?

double CAS

Existen también unas mejoras del `CAS`, llamado *double-compare-and-swap* DCAS (Motorola), que realiza dos CAS normales a la vez, o *double-wide compare-and-swap* (Intel/AMD x86), que opera con dos punteros a la vez para el intercambio, o *single-compare double-swap* (Intel itanium), que compara un valor (puntero) pero excribe dos punteros en memoria adyacente.
El código, expresado a alto nivel, para DCAS sería:

```
if  C1 equal to V1 and C2 equal to V2
  then
    swap S1 and V1
    swap S2 and V2
    return true
  else
    return false
```