



Sistemas Operativos 2021-2022

TEMA 3

Coordinación y sincronización de procesos

3.1. Introducción

3.2 El problema de la exclusión mutua

3.3. Soluciones hardware a la exclusión mutua

3.4. Semáforos

3.1.1. Principios de concurrencia

3.1.2. Problemas de la concurrencia

3.1.3. Clasificación de interacción entre procesos

3.1.4. El problema de la exclusión mutua



Introducción

Principios de concurrencia

- **Procesos concurrentes**

- Aquellos que **intercalan y/o superponen** sus ejecuciones en el tiempo

- **Procesos paralelos**

- Aquellos que **superponen** sus ejecuciones en el tiempo.



Introducción

Principios de concurrencia

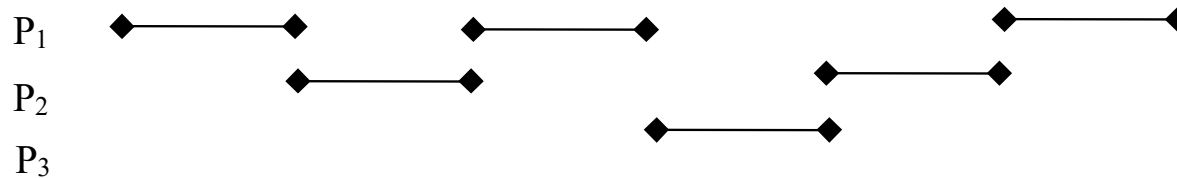
- **¿Si hay concurrencia hay paralelismo?**
 - No necesariamente
 - Si la concurrencia es por intercalación de instrucciones la respuesta es NO.
 - Si la concurrencia es por superposición de instrucciones la respuesta es SÍ
- **¿Si hay paralelismo hay concurrencia?**
 - La respuesta es SÍ, SIEMPRE

Introducción

Principios de concurrencia

- Concurrencia en los sistemas operativos:

- Multiprogramación (monoprocesador)

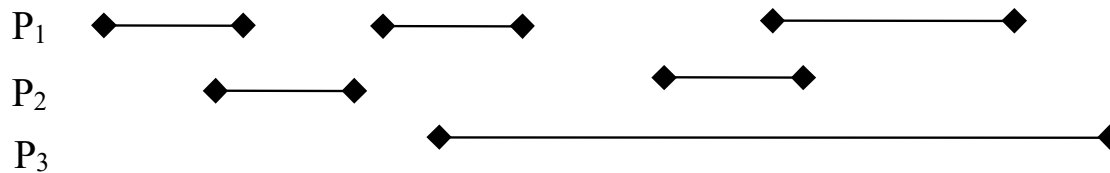


- Intercalación de instrucciones en el procesador
 - **No hay paralelismo pero SÍ hay concurrencia**
 - Tiempo compartido (monoprocesador)
 - Idem que antes porque si hay tiempo compartido hay multiprogramación

Introducción

Principios de concurrencia

- Concurrencia en los sistemas operativos: (continua)
 - Multiprocesamiento y distribuidos (varios núcleos o CPUs)



- Intercalación y superposición de instrucciones
- **Hay Paralelismo y por tanto también Concurrencia**



Introducción

Principios de concurrencia

- **Conclusión:** en los sistemas operativos actuales SIEMPRE hay concurrencia
 - Se presentan nuevos problemas
 - Que son independientes del número de procesos y CPUs
 - **Las soluciones a dichos problemas deben garantizar la corrección de la ejecución de los procesos concurrentes bajo cualquier secuencia de intercalado y/o superposición de instrucciones**



Introducción

Problemas de la concurrencia

- **Compartición de recursos **globales****

```
Global x; // Variable global compartida
Procedimiento echo ()
    Local sal;
    Entrada(x); // leer del teclado
    sal:=x;
    Salida(sal); // mostrar por pantalla
End.
```

- Varios procesos concurrentes invocan *echo()*
- Puede ocurrir que un proceso haya leído por la entrada un valor y que cuando imprima en la pantalla dicho valor salga otro diferente ☹

Introducción

Problemas de la concurrencia

■ Sincronización entre procesos

Proceso P1

i_1

.....

i_x

Sincronización

Resto proceso P1

Proceso P2

i_1

...

Sincronización

i_y

Resto proceso P2

- El proceso P2 no debe ejecutar la instrucción i_y hasta que P1 no haya finalizado de ejecutar la instrucción i_x
→ ¿Cómo cumplir esto?



Introducción

Problemas de la concurrencia

- Dificultad de asignar los recursos de forma óptima
 - Posible uso ineficiente de los recursos
 - Ejemplo
 - Proceso A consigue un recurso y se bloquea por alguna razón
 - Si un proceso B lo solicita, tendrá que esperar
- Dificultad para localizar errores de programación
 - Las situaciones no son repetibles → Distintas ejecuciones dan lugar a distinto intercalado de instrucciones

Clasificación de interacción entre procesos

- Vamos a ver TODAS las formas diferentes en que un proceso interactúa con otro
- De cada forma vamos a analizar si se da algún problema de concurrencia.
- Nos va a interesar saber quién debe solucionar el problema de concurrencia:
 - El Sistema operativo
 - El Usuario programador

Clasificación de interacción entre procesos

1. Competencia entre procesos por recursos

- Procesos independientes que necesitan acceder a un recurso durante su ejecución (por ejemplo disco)
 - No se deben ver afectadas sus ejecuciones
 - Cada proceso debe dejar el recurso tal y como estaba
- **Se presenta el problema de compartición de recursos globales**
- **El control de esta competencia es responsabilidad del SO no del programador**
 - Es el que asigna recursos y resuelve conflictos

Clasificación de interacción entre procesos

2. Cooperación entre procesos por compartición

- Ciertos procesos **cooperan** entre sí
 - Gracias al uso de variables y/o ficheros compartidos
 - Son conscientes de que otros procesos pueden usarlos
- **Se presenta el problema de compartición de recursos globales**
- **El control de la cooperación es responsabilidad del programador no del S.O.**
- El SO debe proporcionar mecanismos que ayuden a programar dicho control → Llamadas al sistema

Clasificación de interacción entre procesos

2. Cooperación entre procesos por compartición (*cont.*)

- Ejemplo: Transparencia 8

```
Global x;  
Procedimiento echo ()  
    Local sal;  
    Entrada(x);    // (*)  
    sal:=x;  
    Salida(sal);  
End.
```

- Dos procesos P1 y P2 concurrentes invocan *echo()*
- ¿Que pasa si P1 sale de la CPU después de ejecutar la instrucción (*) y a continuación entra P2 y ejecuta también hasta la instrucción (*) incluida?

Clasificación de interacción entre procesos

3. Cooperación entre procesos por comunicación

- Los procesos se envían mensajes
- **Se presenta el problema de sincronización entre procesos**
 - El proceso receptor debe esperar a que el emisor envíe un mensaje
- **El control de la comunicación es responsabilidad del programador no del S.O.**
- El SO o el lenguaje de programación (mas habitualmente) proporcionarán servicios de envío y recepción



Introducción

Conclusiones

- El programador debe saber solucionar dos tipos de problemas en la programación concurrente:
 1. Compartición de recursos globales (normalmente variables)
 2. Sincronización de procesos
- Para solucionar el primer tipo será suficiente con **garantizar el acceso exclusivo** a los recursos globales compartidos
- Para solucionar el segundo tipo deberemos tener mecanismos para que los procesos puedan comunicarse y esperar por dicha comunicación



El problema de la exclusion mutua

Introducción

- El problema de compartir recursos globales se soluciona garantizando acceso exclusivo a los mismos
- Ahora el problema es cómo garantizar ese acceso exclusivo → *Exclusión mutua* de dichos recursos
- Vamos a ver ahora como lograr esa *exclusión mutua* a nivel teórico → Sin ver ninguna herramienta en concreto
- Los pasos descritos a continuación habrá que hacerlos SIEMPRE independientemente de la herramienta o mecanismos que se utilice



El problema de la exclusion mutua

Secciones críticas del código

- **Primer paso:** El programador escribe el programa (Código) sin tener en cuenta los problemas derivados de la concurrencia → Como si éstos no existieran
- **Segundo paso:** El programador identifica las **secciones críticas del Código**
- ¿Qué es una sección crítica?
 - Lo vemos a continuación

El problema de la exclusion mutua

Secciones críticas del código

- Sea un sistema en el que existen n procesos $\{P_1, \dots, P_n\}$ que ejecutan el siguiente código para el uso de recursos globales $\{R_1, \dots, R_m\}$
- Código de P_i

Repetir

...

Acceso al recurso R_k ;

...

Acceso al recurso R_c ;

...

indefinidamente

Líneas de código que acceden a un recurso



El problema de la exclusion mutua

Secciones críticas del código

- Cada proceso ejecuta trozos de código en el que accede a recursos globales comunes
 - **Secciones críticas del código**
- Código de P_i

Repetir

...

Código que no es sección crítica;

Sección crítica de P_i para acceder a $R_{k,i}$

Código que no es sección crítica;

...

Código que no es sección crítica;

Sección crítica de P_i para acceder a $R_{c,i}$

Código que no es sección crítica;

...

Secciones críticas

El problema de la exclusion mutua

Secciones críticas del código

■ Ejemplo: Transparencia 8

```
Global x;  
Procedimiento echo ()  
    Local sal;  
    Entrada(x);  
    sal:=x;  
    Salida(sal);  
End.
```

← ***Seccion crítica***

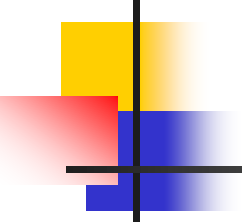
- Varios procesos concurrentes invocan *echo()*
- ¿Qué secciones críticas hay?
 - Una sección crítica



El problema de la exclusion mutua

Secciones críticas del código

- Garantizar acceso exclusivo a los recursos →
 - Garantizar acceso exclusivo a las secciones críticas →
 - Solucionado el problema de la exclusión mutua
- **Tercer paso:** El programador añade un código a TODAS las secciones críticas:
 - Inmediatamente antes de la sección crítica (*sección de entrada*)
 - Inmediatamente después de la sección crítica (*sección de salida*)
- ¿Qué son y para qué sirven las secciones de entrada y salida
 - Lo vemos a continuación



El problema de la exclusion mutua

Secciones de entrada y salida

- Es necesario que los procesos sigan un protocolo en el acceso a las secciones críticas (SC)

Sección de entrada

Sección crítica para usar R_k

Sección de salida

Resto de código

- ***Sección de entrada***

- **El proceso solicita entrar en la SC**

- Si lo consigue, ejecuta las instrucciones de la SC
- Cualquier otro proceso que lo intente, debe esperar

- ***Sección de salida***

- **El proceso abandona la SC, e indica que queda libre**



El problema de la exclusion mutua

Mecanismos y herramientas de programación

- El hardware debe SIEMPRE proporcionar algún mecanismo de exclusión
- El SO nos debe proporcionar llamadas al sistema para poder implementar las secciones de entrada y salida → Que se basan en el mecanismo de exclusión del hardware
- Los lenguajes de programación pueden proporcionarnos mecanismos (librerías) más abstractas → Que se basan en los servicios del SO anteriores
- Vamos a ver a continuación distintos mecanismos para solucionar la exclusión mutua



Soluciones hardware a la exclusión mutua

Índice

3.2.1. Deshabilitación de interrupciones

3.2.2. Instrucciones máquina especiales



Soluciones hardware a la exclusión mutua

Deshabilitación de interrupciones

- En un ordenador monoprocesador los procesos concurrentes sólo pueden ser intercalados
 - Un proceso continúa usando el procesador **hasta que llega una interrupción**
- Se puede garantizar la exclusión mutua impidiendo la interrupción del proceso mientras está en una SC

Repeat

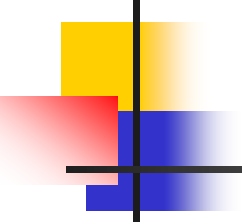
...

Inhabilitar interrupciones; // Sección entrada
SECCIÓN CRÍTICA;

Habilitar interrupciones; // Sección salida

...

Until falso;

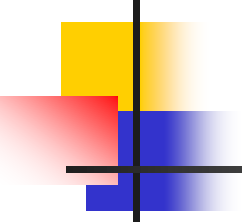


Soluciones hardware a la exclusión mutua

Deshabilitación de interrupciones

- Inconvenientes

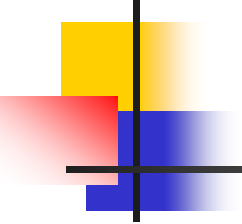
- La eficiencia de la ejecución puede verse degradada, dado que el procesador se ve limitado en su capacidad para intercalar procesos → SO no puede intervenir durante la ejecución de la SC
- Esta solución **no sirve con multiprocesamiento**
 - Es posible que más de un proceso esté simultáneamente en ejecución en diferentes procesadores
 - Varios de ellos podrían estar en SC para el uso del mismo recurso ☹



Soluciones hardware a la exclusión mutua

Instrucciones máquina especiales

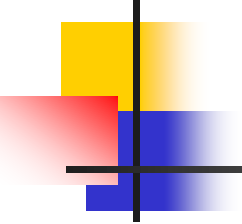
- En una arquitectura multiprocesador, el acceso a memoria es exclusivo para un procesador en un momento dado (lo garantiza el hardware)
 - Es posible, entonces, construir procesadores con instrucciones máquina especiales que accedan a una posición de memoria para leer o escribir en un solo ciclo de instrucción
 - De esta manera, no es posible la interferencia con otras instrucciones → Exclusión mutua → Acceso exclusivo a memoria RAM
 - Si el hardware no garantizara esto sería IMPOSIBLE solucionar el problema de la exclusion mutua con multiprocesamiento.



Soluciones hardware a la exclusión mutua

Instrucciones máquina especiales

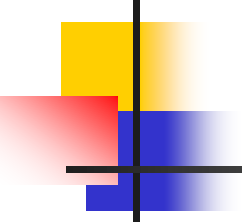
- Las instrucciones máquina especiales se basan en la propiedad del hardware anterior para garantizar acceso exclusivo a una posición de memoria
 - Las instrucciones máquina especiales permitirán leer y establecer un valor en dicha posición de memoria sin verse afectado por otros procesos ni el SO



Soluciones hardware a la exclusión mutua

Instrucciones máquina especiales

- Instrucciones máquina especiales
 - Test&Set
 - Swap



Soluciones hardware a la exclusión mutua

Instrucciones máquina especiales

- Instrucción **Test&Set**

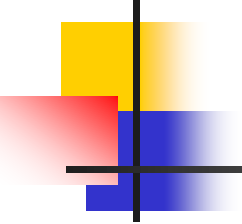
```
boolean Test&Set(boolean &origen) {  
    boolean aux = origen;  
    if !origen  
        origen=true  
    return aux;  
}
```



Soluciones hardware a la exclusión mutua

Instrucciones máquina especiales

- Instrucción **Test&Set** (*continuación*)
 - Permite simultáneamente obtener el valor booleano actual de una posición de memoria (*origen*) y **posteriormente** cambiarla a Cierto si valía Falso
 - La instrucción se ejecuta atómicamente
 - No hay posibilidad de que sea interrumpida
 - Además, la CPU que ejecuta esta instrucción bloquea el bus de memoria para impedir a otras CPUs acceder a ella
 - Podemos usar esta instrucción para garantizar el acceso exclusivo a un recurso global compartido → Implementar secciones de entrada y salida

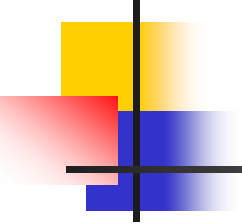


Soluciones hardware a la exclusión mutua

Instrucciones máquina especiales

- Instrucción **Test&Set** (*continuación*)
 - Por cada recurso global compartido usaremos una posición de memoria (un booleano *lock*) para saber si podemos acceder o no a dicho recurso (si vale Cierto no podemos entrar)
 - Las secciones de entrada y salida de las secciones críticas de dicho recurso serían:

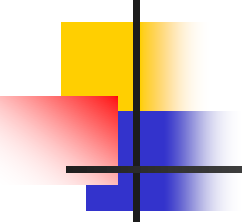
```
...  
while (Test&Set(&lock)) do nada; //sección entrada  
SECCIÓN CRÍTICA;  
lock=false; //sección salida  
...
```



Soluciones hardware a la exclusión mutua

Instrucciones máquina especiales

- Instrucción **Test&Set** (*continuación*)
 - Garantiza la exclusión mutua
 - Sirve para cualquier número de procesos y multiCPUs
 - Sirve para múltiples recursos: cada r_i con su $lock_i$
 - Se emplea espera activa ☹ ☹

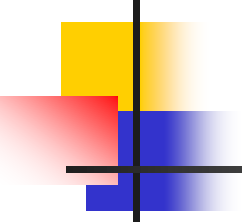


Soluciones hardware a la exclusión mutua

Instrucciones máquina especiales

- Instrucción **Swap**

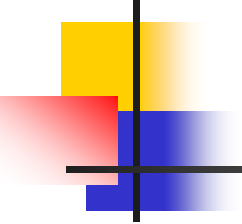
```
void Swap(int &registro, int &memoria) {  
    int aux;  
    aux=memoria;  
    memoria=registro;  
    registro=aux;  
}
```



Soluciones hardware a la exclusión mutua

Instrucciones máquina especiales

- Instrucción **Swap** (*continuación*)
 - Permite simultáneamente obtener el valor actual de una posición de memoria (*memoria*) e intercambiarlo con el valor de un registro
 - El registro quedará con lo que había antes en dicha posición y dicha posición tendrá lo que había antes en el registro
 - La instrucción se ejecuta atómicamente
 - No hay posibilidad de que sea interrumpida
 - Además, la CPU que ejecuta esta instrucción bloquea el bus de memoria para impedir a otras CPUs acceder a ella
 - Podemos usar esta instrucción para garantizar el acceso exclusivo a un recurso global compartido → Implementar secciones de entrada y salida



Soluciones hardware a la exclusión mutua

Instrucciones máquina especiales

- Instrucción **Swap** (*continuación*)
 - Por cada recurso global compartido usaremos una variable local al proceso (*key*) y una posición de memoria (un booleano *lock*) para saber si podemos acceder o no a dicho recurso (si vale Cierto no podemos entrar)
 - Las secciones de entrada y salida de las secciones críticas de dicho recurso serían:

...

```
key=true; // sección entrada
```

```
do swap(lock,key) while (key==true); // s. entrada
```

```
SECCIÓN CRÍTICA;
```

```
lock=false; //sección salida
```

...

- 3.3.1. Definición de semáforo
- 3.3.2. Implementación de semáforos
- 3.3.3. Tipos de semáforos
- 3.3.4. Implementación de semáforos binarios
- 3.3.5. Solución al problema de la exclusión mutua
- 3.3.6. Solución a problemas de sincronización
- 3.3.7. Problemas derivados del mal uso de los semáforos
- 3.3.8. Casos especiales de *Secciones críticas*



Semáforos

Definición de semáforo

- Los semáforos pueden ser vistos como objetos que tienen dos campos:
 1. Un **contador** (de tipo entero)
 2. Una cola **colaPCB** de procesos bloqueados en el semáforo



Semáforos

Definición de semáforo

- (*continuación*) y tres operaciones:
 - 1. Init** .- Iniciar el *contador* del semáforo, con un valor no negativo y vacía la cola *colaPCB*
 - 2. P** .- Disminuye el *contador* del semáforo en uno
 - Si el valor pasa a negativo, el proceso que ejecuta P se bloquea y se añade a la cola del semáforo
 - 3. V** .- Incrementa el *contador* del semáforo en uno
 - Si el valor no es positivo, se elimina un proceso de la cola *colaPCB* del semáforo y se desbloquea



Semáforos

Implementación de semáforos

- Estructura de datos

```
struct sem {  
    int cont;  
    ColaPCB cola;  
}
```

- Operaciones

```
void init(sem s,  
          int valor) {  
    s.cont=valor;  
    cola=NULL;  
}
```

```
void P(sem s) {  
    s.cont--;  
    if (s.cont<0) {  
        insert(proceso,s.cola);  
        bloquear(proceso);  
    }  
}
```

- **Nota:** "proceso" se refiere al proceso que invoca esta operación



Semáforos

Implementación de semáforos

```
void V(sem s) {  
    s.cont++;  
    if (s.cont<=0) {  
        otroproc = extrae(s.cola);  
        desbloquear(otroproc);  
    }  
}
```

- **Nota:** “otroproc” se refiere a algún proceso de la cola de procesos bloqueados en el semáforo
 - Política de la cola
 - FIFO, LIFO, aleatorio, etc.

- Aspecto crítico de implementación
 - Las operaciones de un semáforo se deben ejecutar atómicamente
 - Nótese que se comparte el *contador* y la *cola*
 - Para ello se apoyan en las instrucciones máquina vistas anteriormente



Semáforos

Tipos de semáforos

- **Generales o de cuenta**

- No tiene ninguna limitación en los valores que puede tomar el *contador* del semáforo
- Se corresponde con la implementación vista anteriormente

- **Binarios**

- El *contador* del semáforo sólo pueden tomar los valores 0 y 1
- Es necesario cambiar ligeramente la implementación para asegurar esto



Semáforos

Implementación de semáforos binarios

■ Semáforo binario

```
void P(semBin s) {  
    if (s.cont==1) s.cont=0;  
    else {  
        insert(proceso,s.cola);  
        bloquear(proceso);  
    }  
}  
  
void V(semBin s) {  
    if (s.cola.vacia()) s.cont=1;  
    else {  
        otroproc=extrae(s.cola);  
        desbloquear(otroproc);  
    }  
}
```



Semáforos binarios

Solución al problema de la exclusión mutua

- Los procesos (N) comparten el uso de un semáforo binario (*mutex*) con valor inicial 1
 - **Se usará un semáforo binario por cada recurso**
- Las *secciones de entrada y salida* de una *sección crítica* serían:

...

```
P(mutex) ; // sección de entrada
```

```
SECCIÓN CRÍTICA;
```

```
V(mutex) ; // sección de salida
```

...



Semáforos binarios

Solución a problemas de sincronización

- Dos procesos P_1 y P_2 con conjuntos de sentencias S_1 y S_2 , respectivamente
 - Se desea que la ejecución de S_2 comience cuando ha finalizado la de S_1
 - Los procesos comparten un semáforo binario *sinc* con valor inicial 0

<u>P_1</u>	<u>P_2</u>
...	...
$S_1;$	$P(sinc);$
$V(sinc);$	$S_2;$
...	...

- El proceso que debe esperar hace la operación P y el que debe indicar que ya llegó hace la operación V

Problemas derivados del mal uso de los semáforos

- Interbloqueo (Ejemplo)
 - Dos procesos P_1 y P_2
 - Comparten dos semáforos binarios S y Q con valor inicial 1
 - Se ejecutan $i_{11}, i_{21}, i_{12}, i_{22}$

P_1

$P(S) ; //i_{11}$

$P(Q) ; //i_{12}$

...

$V(S) ;$

$V(Q) ;$

P_2

$P(Q) ; //i_{21}$

$P(S) ; //i_{22}$

...

$V(Q) ;$

$V(S) ;$

Casos especiales de Secciones críticas

- En ciertas situaciones las secciones críticas pueden estar en las siguientes sentencias:
 - Predicado de un IF
 - Predicado de un WHILE
 - En la instrucción RET de una función
- En las situaciones anteriores la implementación de las *secciones de entrada y salida* de la **sección crítica** no es trivial
- Se indica a continuación como garantizar la exclusion mutua en estas situaciones especiales

Secciones de entrada y salida con un IF

- La sección crítica es el propio predicado del IF

```
...  
if (<predicado>)  
{  
    ...  
}
```

- Solución válida pero muy burda ☹

```
...  
P(mutex)  
if (<predicado>)  
{  
    ...  
}  
V(mutex)  
...
```

Secciones de entrada y salida con un IF

- Solución NO válida (aunque se ajusta más a la sección crítica que la solución anterior)

...

P(mutex)

if (<predicado>)

{

V(mutex)

...

}

- No es válida porque si no entró en el IF nunca se hace una operación V y la sección crítica queda ocupada para siempre

Secciones de entrada y salida con un IF

- Solución NO válida

...

P(mutex)

if (<predicado>)

{

V(mutex)

...

}

V(mutex)

...

- No es válida porque si entrá en el IF se hace una operación V DOS VECES y se podrían dejar pasar a DOS procesos/hilos a las secciones críticas del recurso



Semáforos

Secciones de entrada y salida con un IF

■ SOLUCIÓN VÁLIDA FINAL

```
...  
P(mutex)  
if (<predicado>)  
{  
    V(mutex)  
    ...  
}  
else  
{  
    V(mutex)  
    ...  
}  
...
```

Secciones de entrada y salida con un WHILE

- La sección crítica es el propio predicado del WHILE

```
...  
while (<predicado>)  
{  
    ...  
}  
...
```

- Solución válida pero muy burda ☹️

```
...  
P(mutex)  
while (<predicado>)  
{  
    ...  
}  
V(mutex)  
...
```

Secciones de entrada y salida con un WHILE

- Solución NO válida (aunque se ajusta más a la sección crítica que la solución anterior)

...

P(mutex)

while (<predicado>)

{

V(mutex)

...

}

- No es válida porque si no entró en el WHILE nunca se hace una operación V y la sección crítica queda ocupada para siempre

Secciones de entrada y salida con un WHILE

- Solución NO válida

```
...  
P(mutex)  
while (<predicado>)  
{  
    V(mutex)  
    ...  
}  
V(mutex)  
...
```

- No es válida porque si entrá en el WHILE se hace una operación V DOS VECES y se podrían dejar pasar a DOS procesos/hilos a las secciones críticas del recurso

Secciones de entrada y salida con un WHILE

- SOLUCIÓN VÁLIDA FINAL

```
...  
P(mutex)  
while (<predicado>)  
{  
    V(mutex)  
    ...  
    P(mutex)  
}  
V(mutex)  
...
```


Secciones de entrada y salida con un RET

- La sección crítica es la instrucción de retorno de la función

```
int funcion(...)
```

```
{
```

```
...
```

```
ret <expresion>
```

```
}
```

- Solución válida pero muy, muy burda ☹ ☹ ☹
 - Rodear de P(mutex) y V(mutex) cada llamada a la función

Secciones de entrada y salida con un RET

- Solución NO válida (aunque se ajusta más a la sección crítica que la solución anterior)

```
int funcion(...)  
{  
    ...  
    P(mutex)  
    ret <expresion>  
    V(mutex)  
}
```

- No es válida porque nunca se hace la operación V y la sección crítica queda ocupada para siempre

Secciones de entrada y salida con un RET

- SOLUCIÓN VÁLIDA FINAL
- Necesitamos una variable local que calcule, antes de retornar, la expresion a retornar

```
int funcion(...)
{
    ...
    P(mutex)
    var_local = <expresion>
    V(mutex)
    ret var_local
}
```



Lecturas recomendadas

- Stallings, "Sistemas Operativos", 5ª edición
 - Capítulo 5, "Concurrencia. Exclusión mutua y sincronización"
 - Capítulo 6, "Concurrencia. Interbloqueo e inanición"
- Silberschatz, "Fundamentos de Sistemas Operativos", 7ª edición
 - Capítulo 6, "Sincronización de procesos"
 - Capítulo 7, "Interbloqueos"
- Deitel, "Operating Systems", 3rd edition
 - Capítulo 5, "Asynchronous concurrent execution"
 - Capítulo 6, "Concurrent programming"
 - Capítulo 7, "Deadlock and indefinite postponement"