

Procesos: Sincronización

[SGG05]: capítulo 6

Resumen

- Problema
- Definiciones básicas
- Problema de sección crítica
- Características de la solución
- Solución monoprocesador
- Soporte hw a la solución: swap y test&set
- Mutex
- Semáforos

Problema

Acceso concurrente a variable compartida

`r0=@i` `i=0`

Proceso 1: `i=i+1`

Proceso 2: `i=i-1`

Proceso 1

```
ldr r1, [r0]
add r1, r1, #1
str r1, [r0]
```

Proceso 2

```
ldr r1, [r0]
sub r1, r1, #1
str r1, [r0]
```

Ejemplo práctico: Número de enlaces de un fichero en unix

Problema generalizable a:

- Varios procesos (n)
- Multiprocesadores

Posibles resultados

$r0=@i$ $i=0$

Proceso 1	Proceso 2	Proceso 2	Proceso 2
			ldr r1, [r0]
ldr r1, [r0]			
	ldr r1, [r0]		sub r1, r1, #1
add r1, r1, #1			
	sub r1, r1, #1		str r1, [r0]
str r1, [r0]			
	str r1, [r0]	ldr r1, [r0]	
		sub r1, r1, #1	
		str r1, [r0]	



$i=-1$

$i=0$

$i=1$

tiempo

NO

OK

NO

Definiciones básicas

- **Problema:** El acceso concurrente a datos/recursos compartidos puede llevar a inconsistencias en los datos/recursos
- **Condición de carrera** (race condition): Situación como la anterior cuyo resultado depende del orden concreto de ejecución de las instrucciones
- **Sección crítica:** Zona de código donde un proceso accede a recursos compartidos con otros procesos
- **Procesos cooperantes:** Aquellos cuyo resultado puede afectar o verse afectado por la ejecución de otros procesos.

Problema de sección crítica

- Sistema con n procesos P_1, P_2, \dots, P_n
- Recursos compartidos (variables, ficheros, etc.)
- Cada proceso tiene una sección crítica (SC) en el que quiere acceder a los recursos compartidos
- Diseñar un protocolo que permita a los procesos compartir los recursos sin incurrir en inconsistencias (condiciones de carrera)

Solución al problema de SC

Debe satisfacer 3 propiedades:

- **Exclusión mutua:** Que no entren dos procesos en sus SCs de forma simultánea
- **Progreso:** Si ningún proceso está en su SC y un subconjunto de procesos quiere entrar a sus SCs, la decisión se toma entre ellos y sin postponerla indefinidamente
- **Espera acotada:** Si un proceso P_i quiere entrar en su SC, hay una cota o límite en el número de veces que otros procesos P_j ($i \neq j$) pueden entrar en sus SCs antes de que lo haga P_i

Exclusión mutua → proporcionada por HW

Progreso + espera acotada → proporcionada por SW

Esquema de los procesos

do {

Sección de entrada

Sección crítica

Sección de salida

resto

} while (TRUE)

Sección de entrada: Obtener llave para entrar a la SC

Sección de salida: Liberar llave

Resto: Zona de código local del proceso

Solución para un solo procesador

- **Inhabilitar interrupciones** durante el acceso a la variable compartida
 - Asegura el acceso en exclusión mutua
- **Problema:** Ineficiente en multiprocesadores
- **Lo importante: Asegurar a un proceso el acceso y modificación de la variable compartida sin interrupción por otro proceso**
- **Operación atómica:** Conjunto de instrucciones que se ejecutan sin interrupción.
 - O se ejecutan todas o ninguna
 - Una vez comenzada la operación atómica debe terminar sin interrupción

Soporte hw a la exclusión mutua

- Instrucciones habituales

Swap: Intercambio **atómico** $\text{var} \leftrightarrow \text{var}$ o $\text{reg} \leftrightarrow \text{var}$

En esencia es 2 ó 1 ldr+str atómico

Test&set: Lectura y modificación **atómico** de var

En esencia es un ldr+str atómico

- En ARM v4

swp: Intercambio **atómico** $\text{reg} \leftrightarrow \text{mem}$ 32 bits

`swp r0, r0, [r1]` ; intercambio atómico $r0 \leftrightarrow \text{Mem32}[r1]$

swpb: Intercambio **atómico** $\text{reg} \leftrightarrow \text{mem}$ 8 bits

`swpb r0, r0, [r1]` ; intercambio atómico $r0[7:0] \leftrightarrow \text{Mem8}[r1]$

Mutex

- Variable booleana S
- Operaciones **atómicas**:
 - **Lock**: Coger llave
 - **Unlock**: Dejar llave

Proceso 1

```
lock(S) ;  
    //SC  
unlock(S) ;
```

Proceso 2

```
lock(S) ;  
    //SC  
unlock(S) ;
```

```
void lock(S) {  
    while (S!=0) esperar;  
    S=1;  
}
```

```
void unlock(S) {  
    S=0;  
}
```

- Problema: **Espera activa** de los procesos que quieran coger la llave y no puedan

Mutex bajo nivel (ARM v4)

Lock (*S)

`; r0=@S`

`mov r1, #1`

`whi swp r1, r1, [r0] ; S \leftrightarrow 1 atómico`

`cmp r1, #0`

`bne whi`

Unlock (*S)

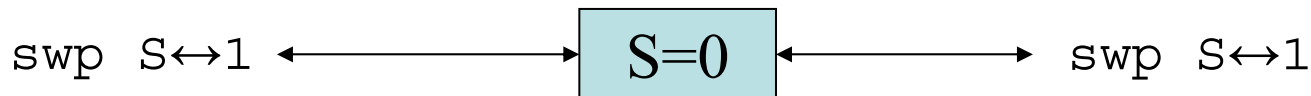
`; r0=@S`

`mov r1, #0`

`str r1, [r0]`

Proceso 1

Proceso 2



Sólo el proceso que ejecuta primero swp toma la llave
Los otros procesos quedan en espera activa

Semáforos

- Generalización del mutex a varias llaves.
- Variable `int S`
- Operaciones **atómicas**:
 - **Wait**: Coger llave
 - **Signal**: Dejar llave

Proceso 1

```
wait(S);  
    //SC  
signal(S);
```

Proceso 2

```
wait(S);  
    //SC  
signal(S);
```

```
void wait(S) {  
    while (S ≤ 0) esperar;  
    S = S - 1;  
}
```

```
void signal(S) {  
    S = S + 1;  
}
```

- Problemas: **Espera activa + malos usos**

Semáforos

- Reducir la **espera activa** (ahora muy corta)
- Se le añade al semáforo una cola de procesos bloqueados

```
void wait(S) {  
    S.valor--;  
    if (S.valor<0) {  
        añadir_proceso(S.cola);  
        bloquear_proceso();  
    }  
}
```

```
void signal(S) {  
    S.valor++;  
    if (S.valor<=0) {  
        P=sacar_proceso(S.cola);  
        desbloquear(P);  
    }  
}
```

- Si $S.valor > 0$ Número de procesos que pueden pasar el semáforo
- Si $S.valor < 0$ Número de procesos bloqueados → **espera inactiva**
- Sigue habiendo **espera activa** en el acceso en exclusión mutua a las variables del semáforo ($S.valor$ y $S.cola$), pero es muy corta
- Semáforos normalmente implementados en el kernel, lo que facilita su uso

Semáforos

- Ejercicio: Implementar el wait y signal en ARM v4 con swp eliminando condiciones de carrera