



UNIVERSIDAD SANTO TOMÁS
PRIMER CLAUSTRO UNIVERSITARIO DE COLOMBIA



UNIVERSIDAD SANTO TOMÁS
PRIMER CLAUSTRO UNIVERSITARIO DE COLOMBIA

SISTEMAS DIGITALES III

SINCRONIZACIÓN DE PROCESOS 2018-I

Carolina Higuera Arias



Hasta el momento ...

Acceso concurrente a datos por procesos o hilos paralelos puede resultar en **condiciones de carrera**

Proceso productor – (proceso padre)

```
while (true) {  
    /*produce an item and put in nextProduced */  
    while (counter == BUFFER_SIZE); // do nothing  
    buffer [in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

Proceso consumidor – (proceso hijo)

```
while (true) {  
    while (counter == 0); // do nothing  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
    /*consume the item in nextConsumed*/  
}
```

¿Cuál es la condición de carrera?

Ambos procesos comparten la variable `counter`

```
register1 = counter
```

```
register1 = register1 +/- 1
```

```
counter = register1
```

¿Cuál es la condición de carrera?

Considere la siguiente ejecución alternada si `counter=5`

- S0: producer: `register1 = counter` {`register1 = 5`}
- S1: producer: `register1 = register1+1` {`register1 = 6`}
- S2: consumer: `register2 = counter` {`register2 = 5`}
- S3: consumer: `register2 = register2-1` {`register2 = 4`}
- S4: producer: `counter = register1` {`count = 6`}
- S5: consumer: `counter = register2` {`count = 4`}

Critical Section Problem

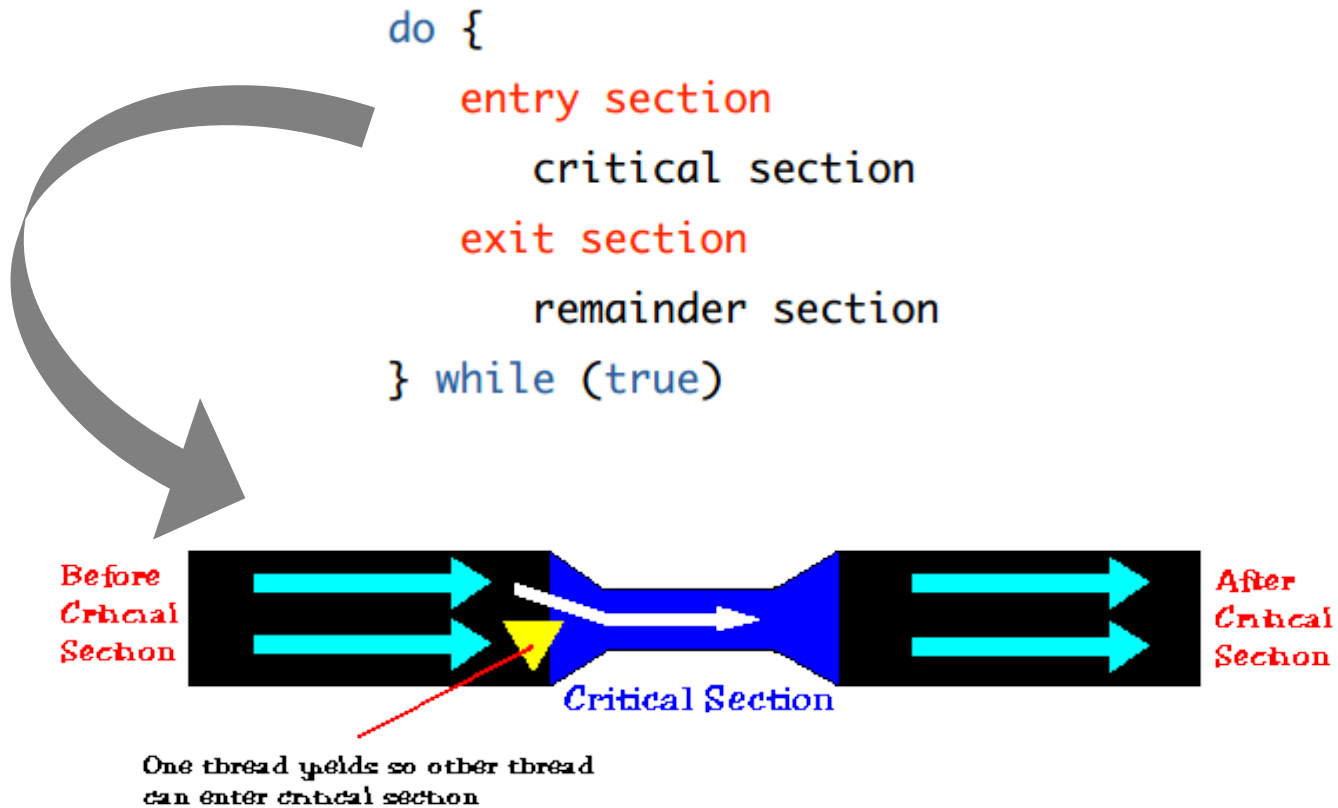
Considere n procesos $\{p_0, p_1, \dots, p_{n-1}\}$

Donde cada proceso tiene una sección de código bajo la figura de *critical section*

Solo **un proceso a la vez** puede estar en su critical section:

- Cuando un proceso ejecuta su *critical section*, ningún otro tiene acceso
- Todo proceso debe pedir aval para acceder a su *critical section*
- El aval debe ser relevado una vez se termina la sección

Critical Section Problem



Solución al problema de la sección crítica

Toda solución debe satisfacer 3 requerimientos:

Exclusión mutua: solo un proceso a la vez puede ejecutar su sección crítica

Progreso: todo proceso debe pasar por su CS. Solo los proceso en espera deciden quién entra a su CS.

Espera acotada: un proceso no puede permanecer accediendo a su CS si hay otros procesos esperando.

Solución por software: Algoritmo de Peterson

• P_0 :

```
do {  
    flag[0] = TRUE;  
    turn = 1;  
    while (flag[1] && turn == 1);  
    critical section  
    flag[0] = FALSE;  
    remainder section  
} while (TRUE);
```

• P_1 :

```
do {  
    flag[1] = TRUE;  
    turn = 0;  
    while (flag[0] && (turn == 0));  
    critical section  
    flag[1] = FALSE;  
    remainder section  
} while (TRUE);
```

Exclusión mutua?

Progreso?

Espera acotada?



UNIVERSIDAD SANTO TOMÁS
PRIMER CLAUSTRO UNIVERSITARIO DE COLOMBIA



Solución por software: Algoritmo de Peterson

• P_0 :

```
do {  
    flag[0] = TRUE;  
    turn = 1;  
    while (flag[1] && turn == 1);  
    critical section  
    flag[0] = FALSE;  
    remainder section  
} while (TRUE);
```

• P_1 :

```
do {  
    flag[1] = TRUE;  
    turn = 0;  
    while (flag[0] && (turn == 0));  
    critical section  
    flag[1] = FALSE;  
    remainder section  
} while (TRUE);
```

Exclusión mutua?

Progreso?

Espera acotada?



Solución por software: Algoritmo de Peterson

Soluciona el problema de sincronización únicamente para 2 procesos

Asume operaciones **atómicas** (cuya ejecución no puede ser interrumpida)

Variables compartidas:

- Turn: indica el turno al proceso que debe entrar a la CS
- flag[2] : si el proceso 0 o 1 está listo para entrar a CS

Solución por OS: Mutex Locks

El proceso debe `acquire()` el lock antes de entrar a su CS

Una vez finalizado, el proceso debe `release()` el lock para que otros procesos puedan entrar a su CS

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (TRUE);
```

Solución por OS: Mutex Locks

Operaciones de `acquire()` y `release()` deben ser ejecutadas de forma atómica

Mutex lock requiere *busy waiting*

También llamado *spinlock*

Solución por OS: Mutex Locks

Ejemplo:

Cree un programa con 4 hilos, en el cual cada hilo que muestre por consola su identificador y la modificación de la variable global id

Thread	19843	1
Thread	19844	2
Thread	19845	3
Thread	19846	4

Solución por OS: Semáforos

Semáforo: entero S que representa cuántas unidades disponibles hay de un recurso en particular


- Solo puede ser actualizado a través de las operaciones atómicas *wait()* y *signal()*

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

```
signal(S) {  
    S++;  
}
```


Solución por OS: Semáforos

Semáforo conteo: dominio no restringido ($s \in \mathbb{Z}$)

Semáforo binario: $s = \begin{cases} 0 \\ 1 \end{cases}$  Similar a
Mutex Locks

Solución por OS: Semáforos

Ejemplo:

Cree un programa en el cual:

- Proceso padre cree proceso hijo y memoria compartida
- Proceso hijo escriba en memoria compartida los números pares de 0 a 20
- Proceso padre lea y muestre en consola el contenido de la memoria compartida
- Proceso padre no haga operación `wait()`

Solución por OS: Semáforos

Ejemplo

Programa multihilo (2) que realice el conteo hasta 2000000

- Los hilos solo pueden afectar una variable global
- Mostrar resultado en consola

Deadlock

- Dos semáforos S y Q
- P_0 bloqueado en espera de Q
- Única forma de desbloquear P_0 :
`signal(Q)` generado por P_1
- Pero ... P_1 bloqueado en espera de S

P_0	P_1
<code>wait(S);</code>	<code>wait(Q);</code>
<code>wait(Q);</code>	<code>wait(S);</code>
<code>.</code>	<code>.</code>
<code>.</code>	<code>.</code>
<code>.</code>	<code>.</code>
<code>signal(S);</code>	<code>signal(Q);</code>
<code>signal(Q);</code>	<code>signal(S);</code>