Sistemas Operativos

Problema de sección crítica

Departamento de Ingeniería en Sistemas y Computación
Universidad Católica del Norte, Antofagasta.

Problema de sección crítica

Conceptos importantes

- Concurrencia: Existencia simultánea de varios procesos en ejecución (existencia simultánea no implica ejecución simultánea)
- Necesidad de sincronización y comunicación
  - Comunicación: Necesidad de transmisión de información entre procesos concurrentes
  - Sincronización: Necesidad de que las ejecuciones de los procesos concurrentes se produzcan según una secuencia temporal conocida y establecida entre los propios procesos

Problema de sección crítica

Conceptos importantes

- Concurrencia: Existencia simultánea de varios procesos en ejecución (existencia simultánea no implica ejecución simultánea)
- Necesidad de sincronización y comunicación
  - Comunicación: Necesidad de transmisión de información entre procesos concurrentes
  - Sincronización: Necesidad de que las ejecuciones de los procesos concurrentes se produzcan según una secuencia temporal conocida y establecida entre los propios procesos

Problema de sección crítica

Conceptos importantes

- Concurrencia: Existencia simultánea de varios procesos en ejecución (existencia simultánea no implica ejecución simultánea)
- Necesidad de sincronización y comunicación
  - Comunicación: Necesidad de transmisión de información entre procesos concurrentes
  - Sincronización: Necesidad de que las ejecuciones de los procesos concurrentes se produzcan según una secuencia temporal conocida y establecida entre los propios procesos

Problema de sección crítica

Conceptos importantes

- Concurrencia: Existencia simultánea de varios procesos en ejecución (existencia simultánea no implica ejecución simultánea)
- Necesidad de sincronización y comunicación
  - Comunicación: Necesidad de transmisión de información entre procesos concurrentes
  - Sincronización: Necesidad de que las ejecuciones de los procesos concurrentes se produzcan según una secuencia temporal conocida y establecida entre los propios procesos

Exclusión mutua

- El acceso a ciertos recursos debe ser exclusivo de un proceso cada vez
- A la parte del programa que los utiliza se le llama sección crítica

Sección crítica

- Cada proceso tiene un segmento de código llamado sección crítica
- No está permitido que varios procesos estén simultáneamente en su sección crítica
- La forma de entrar y salir de la sección crítica es regida por algún protocolo

Sección crítica

- Cada proceso tiene un segmento de código llamado sección crítica
- No está permitido que varios procesos estén simultáneamente en su sección crítica
- La forma de entrar y salir de la sección crítica es regida por algún protocolo

Sección crítica

- Cada proceso tiene un segmento de código llamado sección crítica
- No está permitido que varios procesos estén simultáneamente en su sección crítica
- La forma de entrar y salir de la sección crítica es regida por algún protocolo

Sección crítica

Cualquier solución al problema de la sección crítica debe satisfacer:

- Exclusión mutua: Sólo un proceso ejecuta simultáneamente su sección crítica
- Progreso: Cuando ningún proceso ejecuta su sección crítica, algún proceso que lo solicite puede entrar, impidiendo la entrada simultánea de varios procesos
- Espera limitada: Ningún proceso debe esperar ilimitadamente la entrada en la sección crítica

Sección crítica

Cualquier solución al problema de la sección crítica debe satisfacer:

- Exclusión mutua: Sólo un proceso ejecuta simultáneamente su sección crítica
- Progreso: Cuando ningún proceso ejecuta su sección crítica, algún proceso que lo solicite puede entrar, impidiendo la entrada simultánea de varios procesos
- Espera limitada: Ningún proceso debe esperar ilimitadamente la entrada en la sección crítica

Sección crítica

Cualquier solución al problema de la sección crítica debe satisfacer:

- Exclusión mutua: Sólo un proceso ejecuta simultáneamente su sección crítica
- Progreso: Cuando ningún proceso ejecuta su sección crítica, algún proceso que lo solicite puede entrar, impidiendo la entrada simultánea de varios procesos
- Espera limitada: Ningún proceso debe esperar ilimitadamente la entrada en la sección crítica

Semáforos

- Mecanismo cómodo y efectivo para lograr la sincronización
- Es una variable entera, `S`
- Utiliza primitivas `wait()` y `signal()`

Semáforos

- Mecanismo cómodo y efectivo para lograr la sincronización
- Es una variable entera, `S`
- Utiliza primitivas `wait()` y `signal()`

Semáforos

- Mecanismo cómodo y efectivo para lograr la sincronización
- Es una variable entera, `S`
- Utiliza primitivas `wait()` y `signal()`

Semáforos

wait(S):

```
while(S ≤ 0){
  }
  S = S - 1
```

Semáforos

signal(S):

```
S = S + 1
```

- Estas operaciones se realizan sin interrupción

Ejemplo

- Problema sección crítica con N procesos (sin espera limitada):
  - Semáforo común hola inicialmente a 1.

mientras verdadero hacer

espera(hola)

sección crítica

señal(hola)

sección restante

- Sincronización E_2 y E_1, código de dos procesos.
  - Semáforo común con valor inicial 0.

Proceso P_1:	Proceso P_2:
E_1	*espera(hola)*
señal(hola)	E_2

Bloqueo mutuo

- Un conjunto de procesos está en estado de bloqueo mutuo cuando cada uno esta esperando un suceso que sólo puede producir otro proceso del conjunto.
- Ejemplo de bloqueo mutuo por uso de semáforos.
  - Q y S semáforos, sin espera activa inicialmente = 1)

Proceso P0	**Proceso P1**
espera(S)	espera(Q)
espera(Q)	espera(S)
. . .	. . .
señal(S)	señal(Q)
señal(Q)	señal(S)

Ejemplo

Se tienen 3 procesos cuyo código es (para $i = 1, 2, 3$):

```
Pi()
{
printf("Soy el proceso %d\n", i);
}
```

Que semáforos (con que inicialización?) deberíamos insertar para sincronizar los procesos, generando la salida en el orden Proceso 1, Proceso 2 y Proceso 3 ?

Ejemplo

semaforo dos, tres = 0

```
P1()
{
printf("Soy %d\n", 1);
signal(dos);
}
```

```
P2
{
wait(dos)
printf("Soy %d\n", 2);
signal(tres);
}
```

```
P3()
{
wait(tres);
printf("Soy %d\n", 3);
}
```

Problema productor-consumidor

- Valores iniciales: *vacio = n , lleno = 0 , mutex = 1.*
- **Proceso productor:**

```
mientras verdadero hacer
        ...   producir un elemento en productor_siguiente
        espera(vacio)
        espera(mutex)
        ...        añadir productor_siguiente a buffer
        señal(mutex)
        señal(lleno)
```

- **Proceso consumidor:**

```
mientras verdadero hacer
        espera(lleno)
        espera(mutex)
        ...        pasar un elemento de buffer a csiguiente
        señal(mutex)
        señal(vacio)
        ... consumir elemento en consumidor_siguiente
```

Ejercicio

Un conjunto de procesos comparten una región de memoria que está organizada en *M* bloques, todos del mismo tamaño. Los bloques pueden estar libres u ocupados. Para registrar los bloques libres se dispone de un Stack que almacena las direcciones de los bloques libres. Los procesos obtienen y devuelven bloques usando las funciones `getspace` y `release` respectivamente:

```
getspace()
{
 a=Stack[top]; //a es una dirección
 top=top-1;
 return a;
}
```

```
release(b)  //b es una dirección
{
 top=top+1;
 Stack[top]=b;
}
```

- Defina e inicialice las variables tipo semáforo
- Reescriba las funciones incluyendo los semáforos definidos

Ejercicio

- Semaforo mutex = 1; Semáforo para excluir mutuamente a los procesos que modifican el Stack
- Semaforo Bloques = M; Para sincronizar a los procesos con la disponibilidad de bloques

```
getspace()
{
wait(Bloques);
wait(mutex);
a=Stack[top]; //a es una dirección
 top=top-1;
signal(mutex);
 return a;
}
```

```
release(b)  //b es una dirección
{
wait(mutex);
top=top+1;
Stack[top]=b;
signal(mutex);
signal(Bloques);
}
```

Ejercicio

Un proceso genera los Threads A y B parte de su código es:

semaforo alfa=0, beta=0;	
Thread A	**Thread B**
1. print 'a1 \n'; 2. signal(alfa); 3. wait(beta); 4. print 'a2 \n';	1. print 'b1 \n'; 2. signal(beta); 3. wait(alfa); 4. print 'b2 \n';

- Indique todas las salidas posibles que puede generar este proceso.

Ejercicio

a1	a1	b1	b1
b1	b1	a1	a1
b2	a2	b2	a2
a2	b2	a2	b2