Práctico 3 - Concurrencia

Concurrencia

Ejercicio 1. Dados estos 3 procesos en paralelo:

Pre: $x = 0$		
$P_0 : a_0 = x$ $;a_0 = a_0 + 1$ $;x = a_0$	$P_1 : x = x + 1$ $;x = x + 1$	$P_2 : a_2 = x$ $;a_2 = a_2 + 1$ $;x = a_2$

(a) ¿Qué valores finales puede tomar x ?

(b) Muestre para cada uno de los valores un escenario de ejecución que los produzca. Es decir, numere las sentencias y construya la secuencia en base a la numeración.

(c) ¿Cuántos escenarios de ejecución hay? * ¿Cuántos para cada valor final de x ?

Respuesta

(a)(b) Tomando $P_0[i]$ como la línea i del proceso P_0, $P_1[i]$ como la línea i del proceso P_1 y $P_2[i]$ como la línea i del proceso P_2, voy a plantear los distintos escenarios posibles para x:

- x **no puede ser cero:** en los tres procesos necesariamente incrementan al menos una vez el valor de x.
- x **no puede ser mayor a 4:** no hay mas de 4 incrementos en total.
- x **puede ser uno:** en el siguiente escenario $P_0[1] - P_1[1] - P_1[2] - P_2[1] - P_2[2] - P_2[3] - P_0[2] - P_0[3]$.
- x **puede ser dos:** en el siguiente escenario $P_0[1] - P_1[1] - P_1[2] - P_0[2] - P_0[3] - P_2[1] - P_2[2] - P_2[3]$.
- x **puede ser tres:** en el siguiente escenario $P_0[1] - P_2[1] - P_2[2] - P_2[3] - P_0[2] - P_0[3] - P_1[1] - P_1[2]$
- x **puede ser cuatro:** si se ejecutan secuencialmente los procesos: $P_0[1] - P_0[2] - P_0[3] - P_1[1] - P_1[2] - P_2[1] - P_2[2] - P_2[3]$.

(c) Hay 8! permutaciones de todas las sentencias, luego hay que sacarles las (3!2!3!) permutaciones de los 3 programas que estan fuera de orden:

$$\frac{8!}{3!2!3!} = 560$$

Ejercicio 2. Dados estos 2 procesos en paralelo:

Pre: $x = 0$	
$P_0 : while(1)\ \{$ $x = x + 1$ $; x = x - 1$ $\}$	$P_1 : while(1)\ \{$ $x = x + 1$ $; x = x - 1$ $\}$

(a) ¿El multiprograma termina?

(b) ¿Qué valores puede tomar x?

Respuesta

(a) Son dos ciclos infinitos, independientemente de que esten ejecutados en paralelo, nunca van a terminar, el planificador va a saltar entre uno y otro.

(b) Los programas nos permiten formar cualquier valor en el caso de que las sentencias **no sean atómicas**, es decir x puede tomar cualquier valor entero. Si son atómicas puede tomar el valor de 0,1,2.

Ejercicio 3. Considere los procesos:

Pre: cont ∧ x= 1 ∧ y= 2	

`P0 : while (cont && x<20) {`
`x = x * y;`
`}`
	`P1 : y = y + 2;`
	`cont = false;`

Post: ¬cont ∧ x=? ∧ y=?	

(a) Calcule los posibles valores finales de x e y.

(b) Si en P1 se cambia la instrucción `y = y + 2;` por `y = y + 1; y = y + 1;` en dos líneas distintas. ¿Cambia esto los posibles valores finales? Justifique.

Respuesta

(a) Y solo va a poder tomar de valor final 2, luego x va a poder tomar los valores : 1,2,4,8,16, 32,64.

(b) Al eliminarse la atomicidad de sumarle 2 y dividirla en 2 sumas de 1, estamos permitiendo que la variable y tome mas valores, por lo que ahora el rango de x es distinto: $\{1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 32, 36, 48, 54, 64, 72\}$.

Ejercicio 4. Considere los procesos P0 y P1:

(a) ¿A cuánto pueden diferir como máximo m y n durante la ejecución?

(b) ¿En cuántas iteraciones termina? Indicar mínimo y máximo.

(c) ¿Qué valores pueden tomar n y m en la Post? Justifique de manera rigurosa.

Pre: n= 0 ∧ m= 0	

P0 : while (n<100) {
n = n*2;
m = n;
}
	P1 : while (n<100) {
	n = n+1;
	m = n;
	}

Post: n=? ∧ m=?	

Respuesta

- $n = 2 \cdot (m + 1)$, esto ocurre en el siguiente escenario:

Proceso 0	Proceso 1	Valor de n	Valor de m
-	n = n+1	n+1	-
-	m = n	n+1	n+1
n = n*2	-	2*(m+1)	n+1
m = n	-	2*(m+1)	2*(m+1)

(b) **No hay un máximo definido** y el mínimo es cuando el crecimiento es el máximpo posible, este es cuando n tome los valores 1,2,4,8,16,32,64,128. Es decir, **el mínimo de iteraciones es 8**.

(c) Se pueden tomar todos los valores entre 100 y 200:

1. **No puede ser menor que 100:** puesto que si fuese menor a 100, seguiría cumpliendose la guarda dando lugar a que el valor se siga incrementando.
2. **No puede ser mayor que 200:** puesto que para que llegue a valer 200, quiere decir que P1 me soló la variable compartida con su valor en 100 y luego P0 me la duplicó, y ya al salir de ahi y concretarse ambos procesos, ninguna de las guardas se va a cumplir.

Locks

Ejercicio 5. La modificación en el punto (b) del Ejercicio 3 introduce cambios en los posibles valores finales, utilice locks para que vuelvan a devolver los mismos valores del punto (a).

Respuesta

```
P0: while (cont && x<20){
    lock(mutex);
    x = x*y;
    unlock(mutex);
}
P1: lock(mutex);
    y = y+1;
    y = y+1;
    unlock(mutex);
    cont = false;
```

Ejercicio 6. Dar una secuencia de ejecución (escenario de ejecución) de las sentencias de dos procesos P_0 y P_1 que corren el código de Simple Flag donde ambos entran a la región crítica.

```c
typedef struct __lock_t { int flag; } lock_t;
void init(lock_t *mutex) {
  mutex->flag = 0;
}

void lock(lock_t *mutex) {
  while (mutex->flag == 1)
    ;
  mutex->flag = 1;
}

void unlock(lock_t *mutex) {
  mutex->flag = 0;
}
```

Respuesta

Es una solución que se basa en un flag que indica si un candado esta ocupado o no, acá surge el problema de que dos hilos pueden leer el flag al mismo tiempo y ambos ver que el candado está libre, por lo que ambos intentarán entrar a la sección crítica. Se produce una condición de carrera.

Ejercicio 7. Hacer una matriz de entradas booleanas para comparar todos los algoritmos de exclusión mútua respecto a características importantes.

- Algoritmos: CLI/STI, Simple Flag, Test-And-Set, Dekker, Peterson, Compare-And-Swap, LL-SC, Fetch-And-Add, TS-With-Yield, TS-With-Park.
- Características: ¿Correcto?, ¿Justo?, Desempeño, ¿Espera Ocupada?, ¿Soporte HW?, ¿Multicore?, ¿Más de 2 procesos?

Ejercicio 8. El siguiente programa asegura exlusión mutua en las regiones críticas:

$$t = 0 \land \neg c0 \land \neg c1$$

```
P0:                              P1:
1:  while (1) {                  A:  while (1) {
2:      {Región no crítica}      B:      {Región no crítica}
3:      (c0,t) = (true,1)        C:      (c1,t) = (true,0)
4:      while (t!=0 && c1);      D:      while (t!=1 && c0);
5:      {Región crítica}         E:      {Región crítica}
6:      c0 = false               F:      c1 = false
7:  }                            G:  }
```

Las sentencias 3 y C son asignaciones múltiples que se realizan de manera atómica. Por ejemplo, para el caso de la sentencia 3, las asignaciones **c0** = true y t = 1 se realizarían en un solo paso de ejecución.

Este protocolo es demasiado exigente en el sentido de que requiere la ejecución de múltiples asignaciones en un sólo paso de ejecución (¡se necesitaría implementar un mecanismo de exclusión mutua en sí mismo para administrar esta atomicidad!). Analice cuál de las 4 posibles realizaciones de este protocolo de exclusión mutua - en el cual las asignaciones ya no son atómicas y por lo tanto hay que darle un orden determinado- es correcta.

Respuesta

Las cuatro posibilidades son las siguientes:
Forma 1: Esta es la manera correcta de realizar las asignaciones para que haya exclusión mutua en las regiones críticas.

Tabla 1. Implementaciones del Algoritmo de la Sección Crítica

Solución a la CS	¿Exclusión Mútua?	¿Es justa?	Desempeño	¿MultiCore?	¿Requiere soporte especial del μP?
Simple Flag	NO	NO	Depende de la CS	SÍ	NO
Spinlock Test and Set	SÍ	NO	Depende de la CS y cores	SÍ	SÍ
Dekker ('68)	SÍ	NO	Idem	Dual Core	NO
Peterson ('81)	SÍ	NO	Idem	Dual Core	NO
Test and Set con Yield	SÍ	NO	Está pensado para una CS grande	SI	NO
Test and Set con Yield, Park, unpark	SÍ	NO, ya que puede haber deadlock	Está pensado para una CS grande	SI	NO

```
P0: while(true)
    {no CS}
    c0 := true
    t := 1
    while (t != 0 && c1) {}
    {CS}
    c0 := false
P1: while(true)
    {no CS}
    c1 := true
    t := 0
    while (t != 1 && c0) {}
    {CS}
    c1 := false
```

Forma 2: Hay un escenario de ejecución en el que ambos quedan en la zona crítica por lo que no se puede asegurar la exclusión mutua. (123ABCDEG457)

```
P0: while(true)
    {no CS}
    t := 1
    c0 := true
    while (t != 0 && c1) {}
    {CS}
```

```
        c0 := false
    P1: while(true)
        {no CS}
        c1 := true
        t := 0
        while (t != 1 && c0) {}
        {CS}
        c1 := false
```

Forma 3: Hay un escenario de ejecución en el que ambos quedan en la zona crítica por lo que no se puede asegurar la exclusión mutua. (123ABCDEG457)

```
    P0: while(true)
        {no CS}
        t := 1
        c0 := true
        while (t != 0 && c1) {}
        {CS}
        c0 := false
    P1: while(true)
        {no CS}
        t := 0
        c1 := true
        while (t != 1 && c0) {}
        {CS}
        c1 := false
```

Forma 4: Hay un escenario de ejecución en el que ambos quedan en la zona crítica por lo que no se puede asegurar la exclusión mutua. (ABC123457DEG)

```
    P0: while(true)
        {no CS}
        c0 := true
        t := 1
        while (t != 0 && c1) {}
        {CS}
        c0 := false
    P1: while(true)
        {no CS}
        t := 0
        c1 := true
        while (t != 1 && c0) {}
        {CS}
        c1 := false
```

Variables de Condición

Ejercicio 9. Para el siguiente código que intenta implementar productor/consumidor, buscar una falla instanciando dos consumidores y un productor C_1, C_2, P_1. Dar una secuencia de líneas que provoca una condición no-deseada.

```c
int loops;
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);
    if (count == 1)
      pthread_cond_wait(&empty, &mutex);
    put(i);
    pthread_cond_signal(&fill);
    pthread_mutex_unlock(&mutex);
  }
}

void *consumer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);
    if (count == 0)
      pthread_cond_wait(&fill, &mutex);
    int tmp = get();
    pthread_cond_signal(&empty);
    pthread_mutex_unlock(&mutex);
    printf("%d\n", tmp);
  }
}
```

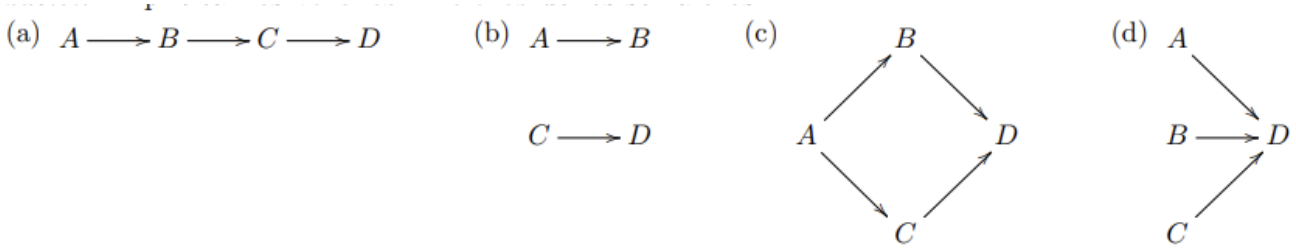
Respuesta

En la columna de consumidor, o productor, se muestran las líneas de ejecución tomando como referencia el código del enunciado, luego se reporta cada acción en la parte de nota.

C1	Estado	C2	Estado	P1	Estado	Nota
20	Running	-	Ready	-	Ready	lock(mutex)
21	Running	-	Ready	-	Ready	condición se cumple
22	Running	-	Ready	-	Ready	unlock(mutex)
-	Duerme (fill)	20	Running	-	Ready	lock(mutex)
-	Duerme (fill)	21	Running	-	Ready	condición se cumple
-	Duerme (fill)	22	Running	-	Ready	unlock(mutex)
-	Duerme (fill)	-	Duerme (fill)	8	Running	lock(mutex)
-	Duerme (fill)	-	Duerme (fill)	9	Running	condición no se cumple
-	Duerme (fill)	-	Duerme (fill)	11	Running	cont = 1
-	Duerme (fill)	-	Duerme (fill)	12	Running	signal(fill)
-	Duerme (fill)	-	Duerme (fill)	13	Running	unlock(mutex)
-	Duerme (fill)	-	Duerme (fill)	8	Running	lock(mutex)
-	Duerme (fill)	-	Duerme (fill)	9	Running	condición se cumple
-	Duerme (fill)	-	Duerme (fill)	10	Running	unlock(mutex)
23	Running	-	Ready	-	Duerme (wait)	cont = 0
24	Running	-	Ready	-	Duerme (wait)	signal(empty)
25	Running	-	Ready	-	Duerme (wait)	unlock(mutex)
-	Duerme (fill)	23	Running	13	Duerme (wait)	PRE(get) no se cumple

Semáforos

Ejercicio 10. Utilize semáforos para sincronizar los procesos como lo indican los grafos de sincronización. Explicitar los valores iniciales de los semáforos.



Respuesta

Primera sincronización

```
sem_init(&s1, 0, 1);
sem_init(&s2, 0, 0);
sem_init(&s3, 0, 0);

A: sem_up(&s1);
B: sem_down(&s1);
   sem_up(&s2);
C: sem_down(&s2);
   sem_up(&s3);
D: sem_down(&s3);
```

Segunda sincronización

```
sem_init(&1, 0, 0);
sem_init(&2, 0, 0);

A: sem_up(&s1);
B: sem_down(&s1);

C: sem_up(&s2);
D: sem_down(&s2);
```

Tercera sincronización

```
sem_init(&1, 0, 0);
A: sem_up(&s1);
B: sem_up(&s1);
C: sem_up(&s1);
D: sem_down(&s1);
```

Ejercicio 11. Agregar semáforos para sincronizar multiprogramas anteriores.

(a) Modifique el programa del Ejercicio 1 agregando semáforos para que el resultado del multiprograma sea determinista (es decir, que no dependa del planificador) y devuelva el mínimo valor posible.

(b) Sincronice los procesos del Ejercicio 4 con semáforos de manera que se alternen entre P0 y P1 en cada iteración hasta el final de sus ejecuciones. ¿Qué valores toman n y m al finalizar?

Respuesta

Ejercicio 1

```
{- PRE: x = 0 -}
sem_init(&s1, 0, 0);
sem_init(&s2, 0, 0);
// Procesos distintos
P1: a0 := x;
    sem_up(&s1);
    sem_up(&s1);
    x := a0 + 1;
    sem_down(&s2);
    sem_down(&s2);
    x := a0
P2: sem_down(&s1);
    x :=  x+1
    x := x+1
    sem_up(&s2);
P3: sem_down(&s1);
    a2 := x
    a2 := a2 + 1
    x  := a2
    sem_up(&s2);
{- POS: x = 1 -}
```

Ejercicio 4

```
{- PRE: n = 0 && m = 0 -}
sem_init(&s1, 0, 0);
sem_init(&s2, 0, 1);
// Procesos distintos
P0: while(n < 100){
    sem_down(&s1);
    n = n*2;
    m = n;
    sem_up(&s2);
}
P1: while(n < 100){
    sem_down(&s2);
    n = n+1;
    m = n;
    sem_up(&s1);
}
{- POS: n,m 126,127 -}
```

Deadlock

Ejercicio 12. Explicar que hace este programa para cada una de las siguientes combinaciones de valores iniciales de los semáforos: $(E, F) = \{(0,0), (0,1), (1,0), (1,1)\}$.

```
1  sem_t empty;
2  sem_t full;
3
4  void *ping(void *arg) {
5    int i;
6    for (i = 0; i < loops; i++) {
7      sem_wait(&empty);
8      printf("Ping\n");
9      sem_post(&full);
10   }
11 }
12
13 void *pong(void *arg) {
14   int i;
15   for (i = 0; i < loops; i++) {
16     sem_wait(&full);
17     printf("Pong\n");
18     sem_post(&empty);
19   }
20 }
21
22 int main(int argc, char *argv[]) {
23   // ...
24   sem_init(&empty, 0, E);
25   sem_init(&full, 0, F);
26   // ...
27 }
```

Respuesta

- **Caso (0,0):** Ambos procesos se quedan esperando el semáforo del otro, por lo que se produce un deadlock.
- **Caso (0,1):** Se ejecuta la secuencia pong, ping, pong, ping, ...
- **Caso (1,0):** Se ejecuta la secuencia ping, pong, ping, pong, ...
- **Caso (1,1):** Se ejecutan de manera no determinista, pero siempre se ejecutan de manera alternada.

Ejercicio 13. ¿Qué primitiva de sincronización implementa el código de abajo con una variable de condición y un mutex?

Respuesta

Implementa un semáforo las funciones son:

- **unknown_0** es **sem_init**,
- **unknown_A** es **sem_up**,
- **unknown_B** es **sem_down**.

```c
typedef struct __unknown_t {
  int a;
  pthread_cond_t b;
  pthread_mutex_t c;
} unknown_t;

void unknown_0(unknown_t *u, int a) {
  u->a = a;
  cond_init(&u->b);
  mutex_init(&u->c);
}

void unknown_A(unknown_t *u) {
  mutex_lock(&u->c);
  u->a++;
  cond_signal(&u->b);
  mutex_unlock(&u->c);
}

void unknown_B(unknown_t *u) {
  mutex_lock(&u->c);
  while (u->a <= 0)
    cond_wait(&u->b, &u->c);
  u->a--;
  mutex_unlock(&u->c);
}
```

Ejercicio 14. Considere los siguientes tres procesos que se ejecutan concurrentemente:

P_0 : lock(printer) ;lock(disk) ;unlock(disk) ;unlock(printer)	P_1 : lock(printer) ;unlock(printer) ;lock(cd) ;lock(disk) ;unlock(disk) ;unlock(cd)	P_2 : lock(cd) ;unlock(cd) ;lock(printer) ;lock(disk) ;lock(cd) ;unlock(cd) ;unlock(disk) ;unlock(printer)

(a) Dé la planificación que lleva a un estado de deadlock.

(b) Agregue semáforos de manera de evitar que los procesos entren en deadlock. Trate de maximizar la concurrencia.

(c) Como solución alternativa, modifique mínimamente el órden de los pedidos y liberaciones para que no haya riesgo de deadlock.

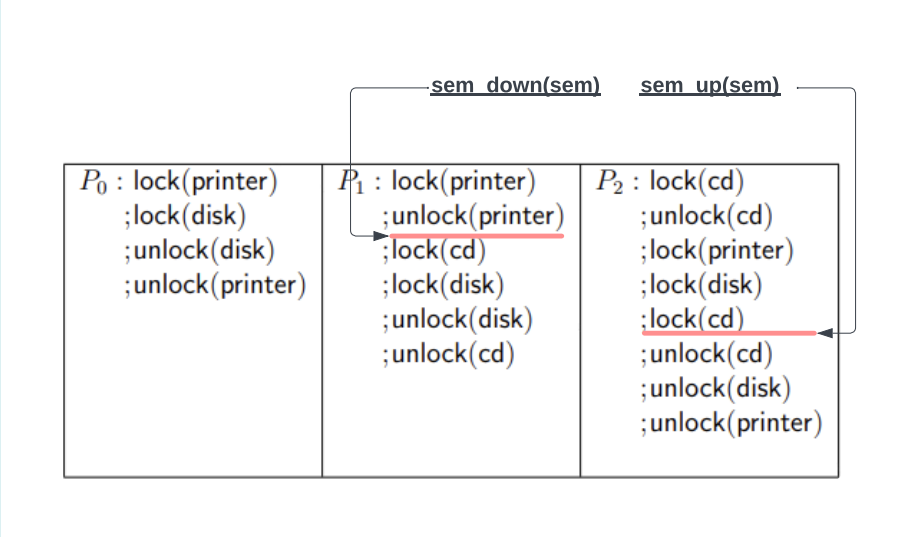
Respuesta

(a) **Planificación que lleva a un deadlock:**

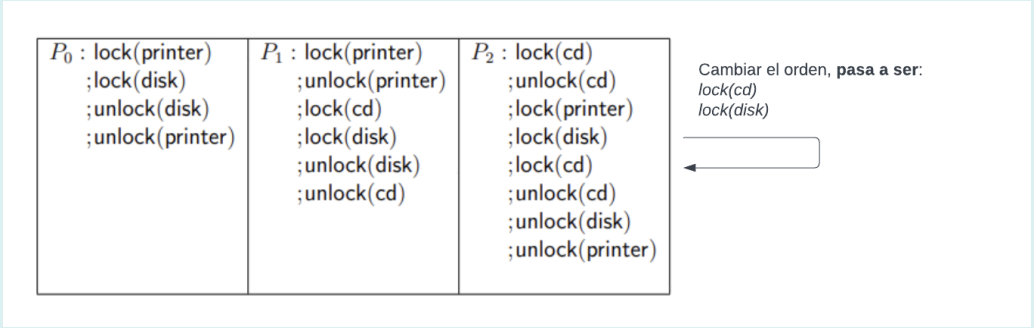
Proceso 0	Proceso 1	Proceso 2	Locks tomados
-	lock(printer)	-	printer
-	unlock(printer)	-	-
-	-	lock(cd)	cd
-	-	unlock(cd)	-
-	-	lock(printer)	printer
-	-	lock(disk)	printer,disk
-	lock(cd)	-	print,disk,cd
-	lock(disk)	-	print,disk,cd (P1 wait disk)
-	-	lock(cd)	print,disk,cd (P2,P1 wait cd,disk)
lock(printer)	-	-	print,disk,cd (P2,P1,P0 wait cd,disk,printer)

Todos se quedan esperando un recurso que otro tiene.

(b) **Solución con semáforos:**



(c) **Solución alternativa:**



Ejercicio 15. Asuma un sistema operativo donde periódicamente se mata algún proceso al azar. ¿Puede haber deadlock en este contexto?

Respuesta

El deadlock es simplemente un ciclo de espera infinita entre locks, este enfoque no me asegura

> que no se caiga en ese ciclo, por lo que es probable que se reduzca la probabilida de que ocurra, pero no se elimina.