Práctico 3 SO - Concurrencia

Ejercicio 1. Dados estos 3 procesos en paralelo[1]:

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.

Nombremos las sentencias de cada proceso, para lograr representatividad.

- En P_0 tenemos 1, 2, 3.
- En P_1 tenemos A, B.
- En P_2 tenemos a, b y c.

Valor final variable x	Escenario de ejecución
1	1 → A, B → a, b, c → 2, 3
2	1, 2, 3 → a → A, B → b, c
3	1, 2 → a, b, c → 3 → A, B
4	$P_0 \rightarrow P_1 \rightarrow P_2$

c. ¿Cuántos escenarios de ejecución hay?¿Cuántos para cada valor final de x?

No hay escenario de ejecución que de menor/igual a cero o mayor/igual que 5 por lo antes mostrado, el máximo valor posible es 4 (cuando hay secuencialidad entre los 3 procesos) y el menor resultado posible es 1 (debido a que en cada proceso existe una sentencia que aumenta en 1 al valor de a_i o x). La cantidad de escenarios sería:

- Para que x = 1 tenemos al menos 2 por la simetría de los procesos.
- Para que x = 2 tenemos al menos 2 por la simetría de los procesos.
- Para x = 3 tenemos al menos 3.
- Para x = 4 tenemos 2 por la simetría de los procesos.

Veamos la cantidad de escenarios posibles exactos, tenemos 8 sentencias en total pero tenemos que tener en cuenta que no existen permutaciones posibles entre cada sentencia particular de cada proceso, puesto que el scheduler se encarga de ello (es decir, no puede ocurrir que en P_1 se ejecute primero B que A). Luego:

→ **8! / (3! * 2! * 3!) = 560 escenarios de ejecución.**

→ De aquí tendríamos que ver, a qué valores de x convergen.

→ Separamos en casos de ejecución.

→ Este cálculo se deja a cargo del lector.

Práctico 3 SO - Concurrencia

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?

Definición: un multiprograma, es decir, un programa compuesto por otros ordenados de manera secuencial que se ejecutan en paralelo termina si y sólo si, todas sus componentes terminan.
Por lo antes mencionado, es fácil de ver que P_0 no termina nunca así como también P_1. Finalmente, **el multiprograma no termina.**

b. ¿Qué valores puede tomar x?

Atomicidad en línea:

Descartemos que x no puede ser negativo, ya que antes de una resta siempre hay una suma. Ahora supongamos que cada línea es atómica, luego x puede valer 0, 1 o 2.

→ Ya vimos que x no puede ser menor a cero.

→ x <= 2, puesto que hay dos sumas seguidas de dos restas.

→ **Por corrección y completitud, 0 ... 1 ... 2 son los valores posibles.**

Sin atomicidad en línea:

Aquí cambia, ya que como las líneas pueden ser interrumpidas, se puede dar el caso de que el número sea cualquier positivo o negativo.

→ El invariante sería {True}.

→ No podemos asegurar nada sobre x.

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.

La precondición restringe los valores en ambas variables.

→ x debe valer 1 e y debe valer 2, por lo que en principio no pueden tomar valores menores a esos.

→ El cuerpo del ciclo se ejecuta mientras x sea menor a 20 y cont = true.

→ x puede tomar cualquier valor en {2, 4, 8, 16, 32, 64}.
→ y puede tomar cualquier valor desde {2, 4}.

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.

Si cambia los valores debido a que puede haber interrupciones entre ambas líneas, generando así distintos valores de x debido a que y ahora podría valer 3.
→ x puede tomar cualquier valor anterior y {1,3,6,9,12,18,24,27,36,48,54,72}.
→ y puede tomar cualquier valor desde {2, 3, 4}.

Ejercicio 4. Considere los procesos P0 y P1 a continuación

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=?	

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

Imaginemos que P_0 entra en el ciclo con n = 0, pero luego ocurre un cambio de contexto y ejecutamos P_1 hasta que n = 100 y m = 99. Luego, continuamos la ejecución dentro del bucle de P_0 donde n = 2 * 100 y m = n.
→ P_1 genera n = 100, m = 99.
→ P_0 genera n = 200 y m = 99.
→ 200 - 99 = 101
→ **101 es la máxima diferencia entre n y m.**

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

Veamos cómo obtenemos el mínimo de iteraciones, para ello necesitamos que se ejecute una sola vez P_1 en primer lugar y luego veamos cuántas iteraciones logramos en P_0 antes de salir del ciclo.
→ En P_0 podemos entrar con un valor de n = 0 + 1 = 1.
→ 2 * (1) ⇒ n ← 2
…
→ 2 * (64) ⇒ n ← 128
(Fin del ciclo).
Finalmente, sumamos las iteraciones en total.
→ P_1 tuvo una iteración.
→ P_2 tuvo 7 iteraciones.
→ **El mínimo de iteraciones es 7 + 1 = 8.**
Por otro lado, el máximo de iteraciones lo logramos al ejecutar solamente P_1.
→ **Es claro que el ciclo tendría 100 ejecuciones como máximo.**
→ {x = 0 … x = 99}.

c. ¿Qué valores pueden tomar n y m en la Post? Justifique de manera rigurosa.
→ X está entre {100, …, 200}

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).

P_1:
```
lock(&mutex);
y = y + 1;          //Critical Section
y = y + 1;          //Critical Section
unlock(&mutex);
cont = false;
```

Ejercicio 6. Dar una secuencia de ejecución (escenario de ejecución) de las sentencias de dos procesos P0 y P1 que corren el código de Simple Flag donde ambos entran a la región crítica.

```
1  typedef struct __lock_t { int flag; } lock_t;
2  void init(lock_t *mutex) {
3    mutex->flag = 0;
4  }
5
6  void lock(lock_t *mutex) {
7    while (mutex->flag == 1)
8      ;
9    mutex->flag = 1;
10 }
11
12 void unlock(lock_t *mutex) {
13   mutex->flag = 0;
14 }
```

P_0	P_1

fun_p₁
{
1 **while**(mutex→flag == 1)
2 ;
3 mutex→flag = 1;
…
4 **CS0**
…
5 mutex->flag = 0;
}
	fun_p₂
	{
	A **while**(mutex→flag == 1)
	B ;
	C mutex→flag = 1;
	…
	D **CS1**
	…
	6 mutex->flag = 0;
	}

Aclaración: existe atomicidad línea a línea.

Es super fácil ver que el algoritmo de “flag simple” es pésimo, lo demostramos mediante este escenario de ejecución donde ambos procesos acceden a la sección crítica.

1 → A → 3 → C → **4** → **D** → 5 → 6

Ejercicio 7. Hacer una matriz de entradas booleanas para comparar todos los algoritmos de exclusión mutua 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?

Algoritmo	Correcto	Justo (no hay starvation)	Requiere soporte de hardware	Es multicore	Cantidad de procesos
CLI/STI	Si	No	No	No	∞
Simple Flag	No	No	No	-	-
Test-And-Set	Si	No	Si	Si	∞
Dekker, Peterson	Si	No	No	Si	2
Compare-And-Swap	Si	No	Si	Si	∞
LL-SC	Si	No	Si	Si	∞
Fetch-And-Add	Si	Si	Si	Si	∞

Ejercicio 8. El siguiente programa asegura exclusión 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 realizaron en un solo paso de ejecución. Este protocolo es demasiado exigente en el sentido de que requiere la ejecución de múltiples asignaciones en un solo 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.

Este multiprograma funciona:

P_0	P_1
1: while (1) { 2: {Región no crítica} 3: c0 = true 4: t = 1 5: while (t!=0 && c1); 6: {Región crítica} 7: c0 = false 8: }	A: while (1) { B: {Región no crítica} C: c1 = true D: t = 0 E: while (t!=1 && c0); F: {Región crítica} G: c1 = false H: }

P_0	P_1
1: while (1) { 2: {Región no crítica} 3: t = 1 4: c0 = true 5: while (t!=0 && c1); 6: {Región crítica} 7: c0 = false 8: }	A: while (1) { B: {Región no crítica} C: t = 0 D: c1 = true E: while (t!=1 && c0); F: {Región crítica} G: c1 = false H: }

No funciona, con este escenario ambos programas ingresan a la sección crítica

- C → 3 → 4 → 6 (CS) → D → F (CS)

P_0	P_1
1: while (1) { 2: {Región no crítica} 3: c0 = true 4: t = 1 5: while (t!=0 && c1); 6: {Región crítica} 7: c0 = false 8: }	A: while (1) { B: {Región no crítica} C: t = 0 D: c1 = true E: while (t!=1 && c0); F: {Región crítica} G: c1 = false H: }

No funciona, con este escenario ambos ingresan a la sección crítica si es que 3 - 4 - C - D se interrumpen.

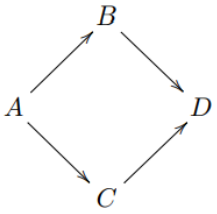
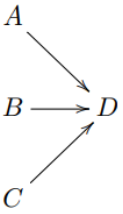
P_0	P_1
1: while (1) { 2: {Región no crítica}	A: while (1) { B: {Región no crítica}

3: t = 1 4: c0 = true 5: while (t!=0 && c1); 6: {Región crítica} 7: c0 = false 8: }	C: c1 = true D: t = 0 E: while (t!=1 && c0); F: {Región crítica} G: c1 = false H: }

No funciona, con este escenario ambos programas ingresan a la sección crítica:

- 3 → C → D → F → 4 → 6

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

(a) $A \longrightarrow B \longrightarrow C \longrightarrow D$ (b) $A \longrightarrow B$, $C \longrightarrow D$ (c)  (d) 

a. Usamos 4 semáforos, imponiendo que cuando quiera empezar algún proceso que no sea el que se tiene que ejecutar se trabe esperando a que el anterior sume uno al semáforo y así habilitarlo.
   → Sem: b, c, d
   → sem_init(b) = 0, sem_init(c) = 0, sem_init(d) = 0

A sem_up(b)	sem_down(b) B sem_up(c)	sem_down(c) C sem_up(d)	sem_down(d) D

b. Usamos 2 semáforos en orden de sincronizar con uno, de a dos procesos.
   → Sem: a, b
   → sem_init (a) = 0, sem_init(b) = 0

A sem_up(a)	sem_down(a) B	C sem_up(b)	sem_down(b) D

c. Usamos dos semáforos.
   → Sem: a, b
   → sem_init(a) = 0, sem_init(b) = 0

A sem_up(a) sem_up(a)	sem_down(a) B sem_up(b)	sem_down(a) C sem_up(b)	sem_down(b) sem_down(b) D

d. Uso un solo semaforo.
   → Sem: a
   → sem_init(a) = 3

A sem_up(a)	B sem_up(a)	C sem_up(a)	sem_down(a) sem_down(a) sem_down(a) D

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?

a. Para obtener el mínimo resultado, x debe valer 1.
→ Propongo utilizar 3 semáforos (quizás se puede hacer con dos).
→ Sem: a, b, c
→ sem_init(a) = 0, sem_init(b) = 0, sem_init(c) = 0

P_0	**P_1**	**P_2**
1: a0 = x ; 2: **sem_up(a)** 3: **sem_down(b)** 4: a0 = a0 + 1 ; 5: x = a0	A: **sem_down(a)** B: x = x + 1; C: x = x + 1; D: **sem_up(c)**	a: **sem_down(c)** b: a2 = x ; c: a2 = a2 + 1 ; d: x = a2 e: **sem_up(b)**

Traza de ejecución (escenario de ejecución):

- 1 → 2 **(se habilita** P_1) → A → B → C → D (se habilita P_2) → a → b → c → d → e **(se habilita P_0)** → 3 → 4 → 5

b. Utilizo dos semáforos.
→ Sem: a. b
→ sem_init(a) = 1, sem_init(b) = 0

P_0	**P_1**
1: while(1) { 2: sem_down(a) 3: x = x + 1; 4: x = x − 1; 5: sem_up(b) 6: }	A: while(1) { B: sem_down(b) C: x = x + 1; D: x = x − 1; E: sem_up(a) F: }

Una traza de ejecución (escenario de ejecución), sería algo de la siguiente forma:
1 → A → 2 → 3 → 4 → 5 (alterna) → B → C → D → E (alterna) → 2 →…→ 5 (alterna) →…

Ejercicio 12. Explicar qué 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  }
```

- **Caso (E, F) = (0, 0)**: no hace nada, ya que ambos programas se quedan esperando a que el semáforo aumente en uno, cosa que nunca debido a que al menos algún semáforo debe valer 1.
- **Caso (E, F) = (0, 1)**: imprime “ping” y luego “pong” loops veces.
- **Caso (E, F) = (1, 0)**: imprime “pong” y luego “ping” loops veces.
- **Caso (E, F) = (1, 1)**: impredecible, ya que hay problema de concurrencia. Podría imprimirse en cualquier orden así como también pisarse.

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

```
1   typedef struct __unknown_t {
2     int a;
3     pthread_cond_t b;
4     pthread_mutex_t c;
5   } unknown_t;
6
7   void unknown_0(unknown_t *u, int a) {
8     u->a = a;
9     cond_init(&u->b);
10    mutex_init(&u->c);
11  }
12
13  void unknown_A(unknown_t *u) {
14    mutex_lock(&u->c);
15    u->a++;
16    cond_signal(&u->b);
17    mutex_unlock(&u->c);
18  }
19
20  void unknown_B(unknown_t *u) {
21    mutex_lock(&u->c);
22    while (u->a <= 0)
23      cond_wait(&u->b, &u->c);
24    u->a--;
25    mutex_unlock(&u->c);
26  }
```

Son semáforos.

Práctico 3 SO - Concurrencia

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

P_0	P_1	P_2

1: lock(printer);
2: lock(disk);
3: unlock(disk) ;
4: unlock(printer)
	A: lock(printer);
	B: unlock(printer);
	C: **lock(cd);**
	D: lock(disk);
	E: unlock(disk);
	F: unlock(cd);
		a: lock(cd);
		b: unlock(cd);
		c: lock(printer);
		d: **lock(disk);**
		e: lock(cd);
		f: unlock(cd);
		g: unlock(disk);
		h: unlock(printer);

a. De la planificación que lleva a un estado de deadlock.

Dejamos de lado P_0 pues no lo necesitamos para quedar en un estado de lock. Lo que se busca acá es llegar a un punto en donde no se pueda desbloquear, es decir, en donde las instrucciones posibles restantes intenten seguir tomando el lock.

→ A → B → a → b → c → C → d

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

P_0	P_1	P_2

lock(printer);
lock(disk);
unlock(disk) ;
unlock(printer);
	lock(printer);
	unlock(printer);
	sem_down(a);
	lock(cd);
	lock(disk);
	unlock(disk);
	unlock(cd);
		lock(cd);
		unlock(cd);
		lock(printer);
		lock(disk);
		lock(cd);
		sem_up(a)
		unlock(cd);
		unlock(disk);
		unlock(printer);

c. Como solución alternativa, modifiquen mínimamente el orden de los pedidos y liberaciones para que no haya riesgo de deadlock.

P_0	P_1	P_2

lock(printer);
lock(disk);
unlock(disk) ;
unlock(printer);
	lock(printer);
	unlock(printer);
	lock(cd);
	lock(disk);
	unlock(disk);
	unlock(cd);
		lock(cd);
		unlock(cd);
		lock(printer);
		lock(cd);
		lock(disk);
		unlock(cd);
		unlock(disk);
		unlock(printer);

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

- Si hay n procesos no, debido a que cuando alguno se bloquee eventualmente será matado liberando así el lock.