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?

Solución:
(a) x puede tomar los valores tales que $x \in (1, 2, 3, 4)$
(b)

Valor de x	Escenario de ejecución
4	ABC12abc ABCabc12 abc12ABC 12ABCabc ...
2	A1aB2bCc ABCa12bc abcA12BC ...
1	A12abcBC a12ABCbc A1aB2bcC a1Ab2BCc ...
3	12ABabCc 12aABCbc 12ABabcC 12aABCbc

	...
5?	No se puede dar ya que si ejecutamos secuencialmente cada código el máximo aumento que se le puede hacer a x es 4 y la variable está inicializada en 0.
0	No puede dar porque todas las variables terminan valiendo > 0 — Al menos un incremento se ejecuta

(c) número de permutaciones / los casos que están mal
(3+2+3)! / (3!*2!*3!) = 40320 / 72 = 560

Ejercicio 2.
Dados estos 2 procesos en paralelo

Pre: x = 0	

P_0 : while(1) {
A: x = x + 1
B: ; x = x - 1
}
	P_1 : while(1) {
	a: x = x + 1
	b: ; x = x - 1
	}

(a) ¿El multiprograma termina?
(b) ¿Qué valores puede tomar x?

Solución:

(a) No termina, pero igual cada componente es infinito, el scheduler va a ir y venir entre uno y el otro pero no terminarían ambas componentes igual. Es un while(1), no debería terminar.
(b) Cualquier valor, la idea es descomponer las instrucciones a nivel “assembler” para observar qué podemos evitar un aumento o decremento de la variable. Por ejemplo si ejecutamos AB (a0=1), abc (x = 1), C(x = 1) cuando en este punto x debería de valer 2. Lo mismo sucede con los decrementos.

Pre: x = 0	

P_0 : while(1) {
A: a0 = x;
B: a0 = a0+1;
C: x = a0;
 	
D: a0 = x;
	P_1 : while(1) {
	a: a1 = x;
	b: a1 = a1+1;
	c: x = a1;
	
	d: a1 = x;

E: a0 = a0-1;
F: x = a0;
}
	e: a1 = a1-1;
	f: x = a1;
	}

Ejercicio 3.

Considere los procesos:

Pre: cont ∧ x = 1 ∧ y = 2	

P_0 : while(cont && x < 20) {
x = x * y;
}
	P_1 : 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.

Solución:

(a)

Valores de x	Valores de y	Escenarios de ejec (P0:A-B; P1:1-2)
32	4	ABABABABAB12
1	4	12A
64	4	1ABABAB2 ó ABABABABA1B2
2	4	AB12
4	4	ABAB12
8	4	ABABAB12
16	4	ABABABAB12

Para mi x en {1, 2, 4, 8, 16, 32, 64}

En gral, si p1 se ejecuta en medio de las iteraciones de p0 => el bucle se interrumpe al incumplirse la guarda, por lo tanto x ={1,2,4,8,16,32,64} e y = {4}, pues independientemente de la planificación, P1 va a ejecutarse, e incrementar y en 2.

b) Al eliminarse la atomicidad de sumarle 2 y dividirla en 2 sumas de 1, más cc's pueden darse, en medio de la suma de y, por lo tanto y = {3,4} y x = {1,2,3,4,6,8,9,12,16,18,24,32,36,48...}

Ejercicio 4.

Considere los procesos P0 y P1 a continuación

Pre: $n = 0 \land m = 0$	

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

Post: $n = ? \land m = ?$	

(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.

Solución:

(a) Como mínimo puede diferir en 2 hasta que m sea asignado.
Como máximo puede diferir 100 ya que podemos ejecutar hasta 99 incrementos de n, luego hacer context switch con m valiendo 98. Por el otro lado multiplicamos 99*2 = 198. En este punto n = 198 y m = 98, difieren en 100.

(b) min = 8;
Tenemos una iteración en P1 para n = 1, y luego 7 iteraciones más hasta llegar a n = 128. No creo que convenga intercalar entre P0 y P1 ya que es un solo incremento y no cambia tanto el ascenso de n, si no que nos genera más iteraciones. El camino más corto es seguir multiplicando por 2.

max = 100;
Cuando hacemos 99 incrementos y luego un *2 a n.

(c) Para que ambos programas terminen, n debe ser mayor estricto que 100. Esto nos dice que para superar esa cifra multiplicando de a dos. Por lo menos, n tiene que llegar a 50. Entonces todos los valores posibles de n son n = x * 2 en donde $50 \leq x \leq 99$

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).

Pre: init(l), cont ∧ x = 1 ∧ y = 2	

P_0 : while(cont && x < 20) {
mutex_lock(l);
x = x * y;
mutex_unlock(l);
}
	P_1:mutex_lock(l);
	y=y+1;
	y=y+1;
	mutex_unlock(l);
	cont=false;

Post: ¬cont ∧ x = ? ∧ y = ?	

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)
7    >> SpinLockwhile (TS(&(mutex->flag),1) == 1)
8      ;
9    //mutex->flag=1;
10 }
11
12 void unlock(lock_t *mutex) {
13   mutex->flag = 0;
```

```
14 }
```

Hay un race condition.
P0: 78 (context switch)
P1: 789 (context switch)
P0: 9
Los dos entran en la región crítica.
Si hay varios esperando pueden salir todos a la vez, entonces si hay un caso en el que ambos procesos están en región crítica

Pre: x=0
lock_t *mutex = malloc(sizeof(lock_t)); ¿Está bien hacer malloc?
init(mutex);

P_0:	P_1:

A: lock(mutex);
B: x=x+1;
C: unlock(mutex);
	a: lock(mutex);
	b: x=x+1;
	c: unlock(mutex);

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?

Solución a la CS	¿Exclusión Mútua? (seguridad)	¿Es justa? (progreso)	Desempeño	¿MultiCore?	¿Requiere soporte especial del µP(microprocesador)?
Solución homeopática	No hay mutex	Si, es justa	Si	Si	No
Espera infinita	Si	No	?	Si	No
Deshabilitar las interrupciones	Si	No	Si, solo ocupa dos instrucciones. (Monopoliza en su zona crítica)	No	No
Bandera Simple (espera ocupada) Entero que si es 0 lo tomo y si es 1 espero.	**No**, hay una race condition. Luego de evaluar la guarda del while ocurre un context switch, o dos cores evalúan la misma guarda.	No	Si, si la zona crítica es corta, caso contrario es muy mala porque los demás hilos en el mismo o en otros cores están quemando ciclos de reloj.	Si	No
Spin lock Test&Set (espera ocupada) Instrucción que testea y setea atomicamente el valor del flag.	Si, como es una instrucción atómica si dos cores quieren hacer T&S uno va a recibir 0 y el resto va a recibir 1. (Está en la implementación interna de T&S)	No	Si la CS es corta: -si es monocore: No Pasa todo un quanto -si es multicore: Si Si la CS es larga: No	Si	Si. Requiere Test&Set.
Spin lock Compare&Swap Variante de Test&Set Setea el valor nuevo sii el original es igual al esperado. Es más amigable o eficiente. (64 cores actualizando una zona de memoria todo el tiempo)	Si, instrucción atómica.	No	Idem, salvo que escribe menos a la memoria.	Si	Si
Spin lock LoadLink StoreConditional (Variante de bandera simple) LL(test) = retornar el valor del flag SC(set) = setear el valor si no hubo actualizacion desde el último LL	Si, instrucción atómica.	No	"	Si	Si y medio rarito, tipo RISC.
Spin lock Fetch&Add Parecido a Test&Set Toma el valor viejo, lo aumenta en 1 y devuelve el valor viejo. lock() lo que hace es tomar el ticket actual, si es igual al turno actual lo toma. Si no hay que esperar a que alguien aumente el turno con unlock().	Si, instrucción atómica.	Si	Es medio caro, siempre escribe en memoria. Pero hace justicia.	Si	Si
Peterson ('81)	Si	No	"	Dual-core	No, solo con instrucciones estándar se arregla
Dekker ('68)	Si	No	"	Dual-core	No, solo con instrucciones estándar se arregla.
Panadero ('74)	Si	Si	"	Ilimitado	Funciona en arquitecturas donde el store no es atómico.
Yield() La idea es no esperar al vicio, poner a dormir a aquellos hilos/procesos que quieren tomar un lock y no pueden.	Si	No	Está pensado para una sección crítica grande.	Ilimitado	No
Espera inactiva usando park/unpark Ponerlos a dormir y además agregarlos a una cola de espera FIFO. Se agrega otro lock llamado guard ya que se necesita modificar la estructura de datos del lock de forma atómica. Se implementa espera ocupada de guard porque se sabe que la zona crítica (modificar el lock) es muy corta.	Si	No! Pero por otra razón: deadlock! Puede esperar dormido infinitamente. Deadlock porque lock hace el park luego de soltar guard. Se agrega una syscall setpark() que anuncia el próximo park.	"	Ilimitado	No
Espera inactiva usando var. de condición	Si	Si, internamente la cond.var implementa una FIFO	"	Ilimitado	No

---

Ejercicio 8.

El siguiente programa asegura exclusión mutua en las regiones críticas:

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

(c0, t) = (true, 1)	c0 = true; t = 1;	t = 1; c0 = true;
(c1, t) = (true, 0)	c1 = true; t = 0;	t = 0; c1 = true;

t=0 ∧ ¬c0 ∧ ¬c1	
P0: 1: while (1) { 2: {Región no crítica} 3: c0=true; 4: t=1; 5: while (t!=0 && c1) SKIP; 6: >>>{Región crítica} 7: c0 = false 8: }	P1: A: while (1) { B: {Región no crítica} C: c1=true; D: t=0; E: while (t!=1 && c0) SKIP; F: >>>{Región crítica} G: c1 = false H: }

La única forma de llegar a las líneas 5 y E es pasar por las instrucciones anteriores, entonces solo queda que t sea una cosa u otra. Por lo tanto solo un While se cumple ergo solo uno de los procesos entra a la región crítica

no queda otra, ¿funciona? elijo creer. Y efectivamente, es algoritmo de Peterson. C0 y C1 son las “intenciones de entrar” a la región crítica de p0 y p1, respectivamente.

P0	P1	diagnóstico
3: c0 = true 4: t = 1 5: while	C: c1 = true D: t = 0 E: while	Anda bien
3: t=1 4: c0=true 5: while	C: t=0 D: c1=true E: while	34CDE5 3CDEF45
3: t=1 4: c0=true 5: while	C: c1 = true D: t = 0 E: while	3CDE45 Ambos entran en la región crítica.
3: c0 = true 4: t = 1 5: while	C: t = 0 D: c1 = true E: while	C345DE Ambos entran en la región crítica.

Variables de Condición

Ejercicio 9.
Para el siguiente código que intenta implementar productor/consumidor, buscar una falla instanciando dos consumidores y un productor C1 , C2 , P1 . Dar una secuencia de líneas que provoca una condición no-deseada.

```
1   int loops;
2   cond_t empty, fill;
3   mutex_t mutex;
4
5   void *producer(void *arg) {
6       int i;
7       for (i = 0; i < loops; i++) {
8               pthread_mutex_lock(&mutex);
9               if (count == 1)
10                      pthread_cond_wait(&empty, &mutex);
11              put(i);
12              pthread_cond_signal(&fill);
13              pthread_mutex_unlock(&mutex);
14      }
15  }
16
17  void *consumer(void *arg) {
18      int i;
19      for (i = 0; i < loops; i++) {
20              pthread_mutex_lock(&mutex);
21              if (count == 0)
22                      pthread_cond_wait(&fill, &mutex);
23              int tmp = get();
24              pthread_cond_signal(&empty);
25              pthread_mutex_unlock(&mutex);
26              printf("%d\n", tmp);
27      }
28  }
```

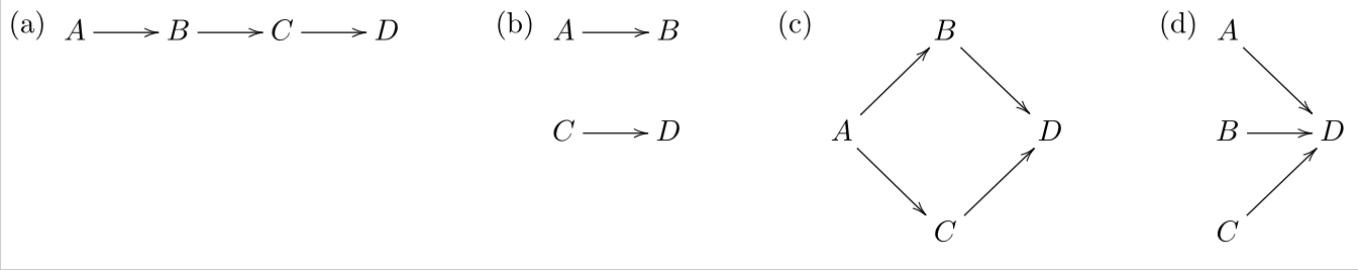
Solución:

Ejecución	1	2	3	4
C1	21, 22 (duermo)			23 get(i)
C2			21 (guarda da false)	23 get(i)
P2		put(i) despierta C1		

Es importante reevaluar la guarda, por eso siempre es preferible un while.

Semáforos

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



Solución:

(a) sem s1,s2,s3
sem_init(s1,0)
sem_init(s2,0)
sem_init(s3,0)

A sem_up(s1)	sem_down(s1) B sem_up(s2)	sem_down(s2) C sem_up(s3)	sem_down(s3) D

(b) sem s1,s2,s3
sem_init(s1,0)
sem_init(s2,0)

A sem_up(s1)	sem_down(s1) B	C sem_up(s2)	sem_down(s2) D

(c) sem s1,s2,s3
sem_init(s1,0)
sem_init(s2,0)
sem_init(s3,0)

A sem_up(s1) sem_up(s1)	sem_down(s1) C sem_up(s2)	sem_down(s1) B sem_up(s3)	sem_down(s3) sem_down(s2) D

(d) sem s1,s2,s3
sem_init(s1,0)
sem_init(s2,0)
sem_init(s3,0)

A sem_up(s1)	C sem_up(s2)	B sem_up(s3)	sem_down(s3) sem_down(s2) sem_down(s1) 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 alternan entre P0 y P1 en cada iteración hasta el final de sus ejecuciones. ¿Qué valores toman n y m al finalizar?

Solución:

(a)

```
sem s1,s2,s3
init_sem(s1,0)
init_sem(s2,0)
init_sem(s3,0)
```

Pre: x = 0		
P_0 : $a_0 = x$: sem_up(s1) : sem_down(s3) : $a_0 = a_0 + 1$: $x = a_0$	P_1 : sem_down(s1) : x = x + 1 : x = x + 1 : sem_up(s2)	P_2 : sem_down(s2) : $a_2 = x$: $a_2 = a_2 + 1$: $x = a_2$: sem_up(s3)

(b)

```
sem s1,s2
init_sem(s1,1)
init_sem(s2,0)
```

Pre: n = 0 ∧ m = 0	

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)
	}

Post: n = 128 ∧ m = 128	

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)}.

```c
sem_t empty;
sem_t full;

void *ping(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        printf("Ping\n");
        sem_post(&full);
    }
}

void *pong(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);
        printf("Pong\n");
        sem_post(&empty);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, E);
    sem_init(&full, 0, F);
    // ...
}
```

Casos	Explicación
(0,0)	Nada, ninguno avanza.
(0,1)	Imprime PongPing.
(1, 0)	Imprime PingPong.
(1,1)	Indefinito, seguidos Ping o seguidos Pong u orden totalmente random.

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

```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);
}
```

Es un semáforo nombrado.

Deadlock

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)** sem_down(critic) C ;lock(cd) D ;lock(disk) sem_up(critic) E ;unlock(disk) F ;unlock(cd)	**a ;lock(cd)** **b ;unlock(cd)** sem_down(crític) c ;lock(printer) d ;lock(disk) sem_up(critic) e ;lock(cd) f ;unlock(cd) g ;unlock(disk) h ;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.

Solución:

(a) P0 espera a printer
P1 tiene a cd pero espera a disk
P2 tiene a printer y a disk pero espera a cd

abABCcdeD1

b)

P_0 :	P_1 :	P_2 :
1 ;lock(printer) 2 ;lock(disk) 3 ;unlock(disk) 4 ;unlock(printer)	A ;lock(printer) B ;unlock(printer) **sem_down(crític)** C ;lock(cd) D ;lock(disk) **sem_up(critic)** E ;unlock(disk) F ;unlock(cd)	a ;lock(cd) b ;unlock(cd) **sem_down(crític)** c ;lock(printer) d ;lock(disk) **sem_up(critic)** e ;lock(cd) f ;unlock(cd) g ;unlock(disk) h ;unlock(printer)

(c)

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(disk) D ;lock(cd) E ;unlock(cd) F ;unlock(disk)	a ;lock(cd) b ;unlock(cd) c ;lock(printer) d ;lock(disk) e ;lock(cd) f ;unlock(cd) g ;unlock(disk) h ;unlock(printer)

La idea es establecer un orden en el cual adquirimos los locks, en este caso el orden es printer -> disk -> cd.

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

Si se mata un proceso y se liberan los recursos que este estaba usando, el sistema puede continuar sin deadlock siempre y cuando los recursos sean redistribuidos correctamente. El proceso de matar y liberar los recursos efectivamente elimina una de las condiciones necesarias para que un deadlock persista.