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.

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

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?

Ejercicio 3. Considere los procesos:

Pre: cont $\wedge$ x$= 1 \wedge$ y$= 2$	

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

Post: $\neg$cont $\wedge$ x$=?$ $\wedge$ 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.

Ejercicio 4. Considere los procesos `P0` y `P1` a continuación

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

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

Post: n$=?$ $\wedge$ 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.

[1] En general, y salvo que explícitamente se diga lo contrario, consideraremos la atomicidad a nivel de sentencia.

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).

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.

```
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 }
```

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:

$$\texttt{t} = 0 \ \wedge \ \neg\texttt{c0} \ \wedge \ \neg\texttt{c1}$$

```
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.

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);
  }
}
```

Semáforos

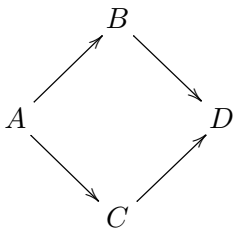
Ejercicio 10. Utilize semáforos para sincronizar los procesos como lo indican los *grafos de sincronización*. Explicitar los **valores iniciales** de los semáforos.

(a) $A \longrightarrow B \longrightarrow C \longrightarrow D$

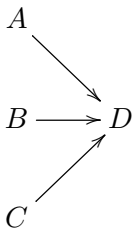
(b) $A \longrightarrow B$

$C \longrightarrow D$

(c)



(d)



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 valor**es** toman `n` y `m` al finalizar?

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

```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);
  // ...
}
```

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

```
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);
}
```

Deadlock

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.

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