UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL MAR DEL PLATA
ARQUITECTURA DE SISTEMAS OPERATIVOS
1er Año – 1er Cuatrimestre

Trabajo Práctico:
Comunicación y Sincronización entre Procesos
RESOLUCIÓN

1. ¿Qué se entiende por deadlock y livelock? De un ejemplo de cada uno fuera de los sistemas computacionales y analícelo.
   …

2. En un sistema conviven 3 procesos y 2 recursos. Uno de los recursos (R2) es de uso exclusivo y el otro (R1) puede ser compartido por hasta dos procesos.

   a. ¿Puede haber deadlock o livelock?
      En este caso, es posible que se produzca un deadlock.
   b. ¿Y si ahora R1 puede ser compartido por hasta 3 procesos?
      En este otro caso, si los tres procesos intentan adquirir R1 al mismo tiempo, todos podrían obtenerlo sin bloquearse entre sí. La posibilidad de deadlock disminuye porque hay más flexibilidad en la asignación de recursos compartidos.

3. Considere un sistema con 4 recursos del mismo tipo compartidos por 3 procesos. Cada recurso puede ser requerido a lo sumo por 2 procesos, y cada proceso puede requerir hasta 2 recursos. ¿En qué estado se encuentra el sistema? ¿Por qué?
   Si todos los procesos solicitan recursos simultáneamente, podrían cumplir sus solicitudes sin entrar en conflicto, ya que hay suficientes recursos disponibles.

4. En un sistema hay tres procesos (P1, P2 y P3) y tres recursos (R1, R2 y R3). Los tres recursos son de uso exclusivo. Se sabe que P1 requiere los tres recursos, P2 requiere de R1 y R2 y P3 solo requiere R3.

   a. ¿El sistema está libre de deadlock?

Si todos los procesos solicitan sus recursos al mismo tiempo, se podría llegar a un estado de deadlock, ya que P1 tiene todos los recursos que P2 necesita, y P2 tiene un recurso que P1 necesita. Además, P3 podría no obtener su recurso (R3) si está siendo utilizado por P1 o P2. Por lo tanto, el sistema no está libre de deadlock.

b. ¿P3 influye en que el sistema esté o no libre deadlock?
NO, P3 no influiye en la posibilidad de deadlock. Ya que al solicitar solo un recurso si lo toma en algún momento lo va a liberar

c. Si me aseguro que P2 no podrá pedir ningún recurso hasta que P1 haya liberado todos sus recursos. ¿El sistema está libre de deadlock? ¿Por qué?
SI, ya quue de esa forma alguno de los proceso podría tomar los recursos y liberarlos.

5. Considere el siguiente programa. Note que el scheduler irá ejecutando estos procesos de manera concurrente, mezclando la ejecución de P1 y P2.

int x=10;	
Proceso 1	Proceso 2

while (true) {
x--;
x++;
if (x != 10)
printf("x is %d",x);
}
	while (true) {
	x--;
	x++;
	if (x != 10)
	printf("x is %d",x);
	}

a. Muestre una secuencia de ejecución (indicando el trace de programa/instrucción) en que imprima “x is 9”.

Proceso 1	Proceso 2

while (true) { □	
x - -; // x = 9	
x ++; // x = 10	
	while (true) { □
	x - -; // x = 9
	x ++; // x = 10
if(x != 10) □	
while (true) { □	
x - -; // x = 9	
	if(x != 10) □
	printf("x is %d",x); // x is 9

b. Muestre una secuencia de ejecución (indicando el trace de programa/instrucción) en que imprima “x is 10”.

Proceso 1	Proceso 2
while (true) { □	
x - -; // x = 9	
	while (true) { □
	x - -; // x = 8
x ++; // x = 9	
if(x != 10) □	
	x ++; // x = 10
	if(x != 10) □
printf("x is %d",x); // x is 10	

6. Considera que tiene un programa con dos procesos. Un proceso se encarga de imprimir y el otro se encarga de introducir ficheros para su impresión.

<table>
<tr><td colspan="2">int peticiones = 0;</td></tr>
<tr><td>Proceso Impresora</td><td>Proceso Imprime Fichero</td></tr>
<tr><td>int R1;
While (true) {
If (peticiones > 0){
extraerColaPeticion();
R1 = peticiones;
R1 = R1 - 1;
peticiones = R1;
}
}</td><td>int R2;
While (true) {
insertarFicheroCola();
R2 = peticiones;
R2 = R2 + 1;
peticiones = R2;
}</td></tr>
</table>

Nota:

- La función extraerColaPeticion() extrae de la cola de peticiones un fichero y lo manda a imprimir. *(peticiones - 1).*
- La función insertarFicheroCola() inserta en la cola de peticiones un archivo para su impresión. *(peticiones + 1).*

a. ¿Presenta algún problema el código anterior si se ejecuta de manera concurrente? Analice el código.
Si ambos procesos intentan modificar peticiones al mismo tiempo, puede ocurrir una condición de carrera, lo que significa que los resultados de las operaciones pueden depender del orden en que se ejecuten los procesos. Además, nos encontramos con que dicha variable puede incrementarse/decrementarse en varias unidades sin sentido dentro del programa, posiblemente generando pérdida de información.

b. Si la respuesta es negativa, explique el porqué. Si la respuesta es positiva, muestre la secuencia donde se genera el problema (Trace del programa).

Proceso Impresora	Proceso Imprime Fichero
While (true) {	
	While (true) {
If (peticiones > 0){ □	
	insertarFicheroCola(); // p = 1
While (true) {	
	R2 = peticiones; // R2 = 1
If (peticiones > 0){ □	
	R2 = R2 + 1; // R2 = 2
extraerColaPeticion(); // p = 0	
	peticiones = R2; // p = 2 □
R1 = peticiones; // R1 = 2	
	While (true) {
R1 = R1 - 1; // R1 = 1	
	insertarFicheroCola(); // p = 3
peticiones = R1; // p = 1 □	

7. Se tienen 3 procesos A, B y C. Implementar en máquina:
   a. El código con semáforos de manera tal que la secuencia sea ABC, ABC,ABC…

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

sem_t sem_A, sem_B, sem_C;

void* proceso_A(void* arg) {
    while (1) {
        sem_wait(&sem_B);
        printf("A");
        sem_post(&sem_C);
    }
    return NULL;
}

void* proceso_B(void* arg) {
    while (1) {
        sem_wait(&sem_C);
        printf("B");
        sem_post(&sem_A);
    }
    return NULL;
}

void* proceso_C(void* arg) {
    while (1) {
        sem_wait(&sem_A);
        printf("C\n");
        sem_post(&sem_B);
    }
    return NULL;
}

int main() {
    sem_init(&sem_A, 0, 1);
    sem_init(&sem_B, 0, 0);
    sem_init(&sem_C, 0, 0);
```

```
    pthread_t hilo_A, hilo_B, hilo_C;

    pthread_create(&hilo_A, NULL, proceso_A, NULL);
    pthread_create(&hilo_B, NULL, proceso_B, NULL);
    pthread_create(&hilo_C, NULL, proceso_C, NULL);

    pthread_join(hilo_A, NULL);
    pthread_join(hilo_B, NULL);
    pthread_join(hilo_C, NULL);

    sem_destroy(&sem_A);
    sem_destroy(&sem_B);
    sem_destroy(&sem_C);

    return 0;
}
```

b. ¿Y si quiero que la secuencia sea BCA, BCA, BCA…?

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

sem_t sem_A, sem_B, sem_C;

void* proceso_A(void* arg) {
    while (1) {
        sem_wait(&sem_B);
        printf("A\n");
        sem_post(&sem_C);
    }
    return NULL;
}

void* proceso_B(void* arg) {
    while (1) {
        sem_post(&sem_A);
        printf("B");
        sem_wait(&sem_C);
```

```
    }
    return NULL;
}

void* proceso_C(void* arg) {
    while (1) {
        sem_wait(&sem_A);
        printf("C");
        sem_post(&sem_B);
    }
    return NULL;
}

int main() {
    sem_init(&sem_A, 0, 1);
    sem_init(&sem_B, 0, 0);
    sem_init(&sem_C, 0, 0);

    pthread_t hilo_A, hilo_B, hilo_C;

    pthread_create(&hilo_A, NULL, proceso_A, NULL);
    pthread_create(&hilo_B, NULL, proceso_B, NULL);
    pthread_create(&hilo_C, NULL, proceso_C, NULL);

    pthread_join(hilo_A, NULL);
    pthread_join(hilo_B, NULL);
    pthread_join(hilo_C, NULL);

    sem_destroy(&sem_A);
    sem_destroy(&sem_B);
    sem_destroy(&sem_C);

    return 0;
}
```

8. ¿Tiene algún problema el siguiente programa? ¿Por qué?

P1	**P2**

sem_wait(Sem1);
sem_wait(Sem2);
// Sección Crítica //
sem_post(Sem2);
sem_post(Sem1);
	sem_wait(Sem2);
	sem_wait(Sem1);
	// Sección Crítica //
	sem_post(Sem1);
	sem_post(Sem2);

Sí, al estar los sem_wait() cruzados en cada proceso se podrñia producir que ambos queden bloqueados, llevando al sistema a un deadlock.

9. Implementar en máquina la sincronización de los procesos A y B de tal manera que, siendo X una variable global:

A	B

x=199;
x=x+1;
Print(x);
	x=500;
	x=X/10;
	Print(x);

a. Siempre el resultado de la ejecución sea 200 y 50.

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

sem_t mutex;
int x = 0;

void* proceso_A(void* arg) {
    x = 199;
    x = x + 1;
```

```
    sem_post(&mutex);
    sem_wait(&mutex);
    printf("%d\n", x);
    return NULL;
}

void* proceso_B(void* arg) {
    x = 500;
    x = x / 10;
    sem_post(&mutex);
    sem_wait(&mutex);
    printf("%d\n", x);
    return NULL;
}

int main() {
    sem_init(&mutex, 0, 0);

    pthread_t hilo_A, hilo_B;

    pthread_create(&hilo_A, NULL, proceso_A, NULL);
    pthread_create(&hilo_B, NULL, proceso_B, NULL);

    pthread_join(hilo_A, NULL);
    pthread_join(hilo_B, NULL);

    sem_destroy(&mutex);

    return 0;
}
```

b. Siempre el resultado de la ejecución sea 50 y 200.

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

sem_t mutex;
int x = 0;
```

```
void* proceso_A(void* arg) {
    sem_wait(&mutex);
    x = 199;
    x = x + 1;
    printf("%d\n", x);
    sem_post(&mutex);
    return NULL;
}

void* proceso_B(void* arg) {
    sem_post(&mutex);
    x = 500;
    x = x / 10;
    printf("%d\n", x);
    sem_wait(&mutex);
    return NULL;
}

int main() {
    sem_init(&mutex, 0, 0);

    pthread_t hilo_A, hilo_B;

    pthread_create(&hilo_A, NULL, proceso_A, NULL);
    pthread_create(&hilo_B, NULL, proceso_B, NULL);

    pthread_join(hilo_A, NULL);
    pthread_join(hilo_B, NULL);

    sem_destroy(&mutex);

    return 0;
}
```

10. Dado el problema del Productor-Consumidor y los siguientes algoritmos:

sem_t espacioLibre, manipulaciónDelBuffer, elementoDisponible;	
Productor	**Consumidor**

While(true) {
producirUnElemento;
sem_wait(espacioLibre);
sem_wait(manipulaciónDelBuffer);
depositarElementoEnBuffer;
sem_post(manipulaciónDelBuffer);
sem_post(elementoDisponible);
}
	While(true) {
	sem_wait(manipulaciónDelBuffer);
	sem_wait(elementoDisponible);
	sacarElementoDelBuffer;
	sem_post(manipulaciónDelBuffer);
	sem_post(espacioLibre);
	consumirElemento;
	}

a. Inicialice los semáforos según corresponda. ¿Es válida la solución anterior? ¿Por qué?

```
sem_init(&espacio_libre, 0, TamañoDelBuffer);
sem_init(&manipulacion_buffer, 0, 1);
sem_init(&elemento_disponible, 0, 0);
```

En el código del Productor, después de realizar la manipulación del buffer, se debería hacer el sem_post() primero de "elementoDisponible" antes de "manipulaciónDelBuffer". Del mismo modo, en el código del Consumidor, después de realizar la manipulación del buffer, se debería hacer el sem_post() primero de "espacioLibre" antes de señalizar "manipulaciónDelBuffer".

Código de ejemplo:

```
#include <stdio.h>
```

```
#include <pthread.h>
#include <semaphore.h>

#define TamañoDelBuffer 5

// Inicialización de semáforos
sem_t espacio_libre, manipulacion_buffer, elemento_disponible;

void* productor(void* arg) {
    while (1) {
        // Simular la producción de un elemento
        int elemento_producido = 1;

        // Esperar hasta que haya espacio libre en el buffer
        sem_wait(&espacio_libre);

        // Esperar hasta que sea seguro manipular el buffer
        sem_wait(&manipulacion_buffer);

        // Depositar un elemento en el buffer
        printf("Productor deposita: %d\n", elemento_producido);

        // Señalizar que se ha manipulado el buffer
        sem_post(&manipulacion_buffer);

        // Señalizar que hay un elemento disponible
        sem_post(&elemento_disponible);
    }

    return NULL;
}

void* consumidor(void* arg) {
    while (1) {
        int elemento_consumido;

        // Esperar hasta que sea seguro manipular el buffer
        sem_wait(&manipulacion_buffer);

        // Esperar hasta que haya un elemento disponible
        sem_wait(&elemento_disponible);
```

```
        // Sacar un elemento del buffer
        printf("Consumidor consume: %d\n", elemento_consumido);

        // Señalizar que se ha manipulado el buffer
        sem_post(&manipulacion_buffer);

        // Señalizar que hay espacio libre en el buffer
        sem_post(&espacio_libre);
    }

    return NULL;
}

int main() {
    // Inicialización de los semáforos con valores adecuados
    sem_init(&espacio_libre, 0, TamañoDelBuffer);
    sem_init(&manipulacion_buffer, 0, 1);  // Inicializado a 1 para permitir
acceso exclusivo al buffer
    sem_init(&elemento_disponible, 0, 0);

    // Crear hilos
    pthread_t hilo_productor, hilo_consumidor;

    // Iniciar los hilos
    pthread_create(&hilo_productor, NULL, productor, NULL);
    pthread_create(&hilo_consumidor, NULL, consumidor, NULL);

    // Esperar a que los hilos terminen (esto no sucederá en este caso, ya
que son ciclos infinitos)
    pthread_join(hilo_productor, NULL);
    pthread_join(hilo_consumidor, NULL);

    // Destruir los semáforos
    sem_destroy(&espacio_libre);
    sem_destroy(&manipulacion_buffer);
    sem_destroy(&elemento_disponible);

    return 0;
}
```