LABORATORIO DE SISTEMAS OPERATIVOS

PERÍODO ACADÉMICO: 2024 – A **EQUIPO:**

PROFESOR: Marco Sánchez PhD.

TIPO DE INSTRUMENTO: Guía de Laboratorio

TEMA: USO DE SEMÁFOROS

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

1. OBJETIVOS

1.1. Implementar el uso de semáforos y mutex en C.
1.2. Asimilar los conceptos teóricos revisados en clase.

2. MARCO TEÓRICO

Problemas de Acceso Simultáneo
Permitir acceso simultáneo a recursos compartidos puede llevar a problemas de sincronización, condiciones de carrera y otras inconsistencias si no se manejan correctamente. Aquí se explican estos problemas y cómo se gestionan:

1. **Condiciones de Carrera (Race Conditions):**
   - Ocurren cuando dos o más hilos ***acceden y modifican*** un recurso compartido al mismo tiempo ***sin la adecuada sincronización***, lo que puede llevar a resultados inconsistentes.
   - **Ejemplo**: Dos hilos incrementando una variable compartida sin protección puede resultar en un valor incorrecto.
2. **Interbloqueos (Deadlocks):**
   - Sucede cuando dos o más hilos se bloquean mutuamente esperando por recursos que los otros hilos poseen.
   - **Ejemplo**: Hilo A tiene el recurso 1 y espera por el recurso 2, mientras que el Hilo B tiene el recurso 2 y espera por el recurso 1.
3. **Inanición (Starvation):**
   - Ocurre cuando un hilo nunca obtiene acceso a los recursos necesarios porque otros hilos lo están acaparando constantemente.
   - **Ejemplo**: Un hilo de baja prioridad esperando indefinidamente porque los hilos de alta prioridad siempre acaparan los recursos.

Gestión del Acceso Simultáneo
Para evitar estos problemas, se utilizan mecanismos de sincronización adecuados. A continuación, se explican los mecanismos de sincronización como semáforos y mutex, y cómo se utilizan para gestionar el acceso simultáneo de forma segura.

Semáforo
Definición:
Un semáforo es un mecanismo de sincronización utilizado para controlar el acceso a un recurso común mediante múltiples hilos (threads) en un sistema concurrente, como un sistema operativo multitarea. Es un tipo de datos abstracto o variable que puede ser incrementado y decrementado de manera atómica.

¿Qué significa de manera atómica?
Atómico: En el contexto de la programación concurrente, una operación atómica es una operación que se completa en un único paso sin interrupciones. Esto

significa que una vez que una operación atómica comienza, no puede ser interrumpida hasta que haya terminado, asegurando que ningún otro hilo o proceso pueda ver un estado intermedio.

Tipos de Semáforos:

- **Semáforo Binario:** Solo tiene dos estados, 0 y 1. Permite el acceso exclusivo a un recurso por un solo proceso o hilo a la vez.
- **Semáforo Contable (Counting Semaphore):** Mantiene un contador que representa el número de recursos disponibles, permitiendo que un número limitado de procesos o hilos accedan a la sección crítica simultáneamente.

Garantía de Sincronización con Semáforos Contables:

- Aunque un semáforo contable permite que múltiples hilos accedan a un recurso simultáneamente, no garantiza por sí mismo la protección contra condiciones de carrera. Los problemas de sincronización pueden ser gestionados adecuadamente mediante el diseño de la sección crítica para evitar la modificación concurrente de datos compartidos.
- Por ejemplo, si los hilos solo leen datos o si cada hilo trabaja en su propia copia de los datos, no habrá problemas de sincronización. Si los hilos necesitan modificar datos compartidos, deben utilizar mecanismos adicionales (como mutex) para proteger esas modificaciones.

Funciones Utilizadas en C:

- **Inicializar el semáforo:** La función `sem_init` se utiliza para inicializar un semáforo.
  int sem_init(sem_t *sem, int pshared, unsigned int value);

- **sem:** Puntero al semáforo a inicializar.
  - Es una referencia al semáforo que se va a inicializar. Este puntero debe apuntar a una variable de tipo sem_t.
- **pshared:** Especifica si el semáforo se comparte entre hilos o procesos.
  - Si pshared es 0, el semáforo es compartido entre los hilos del mismo proceso.
  - Si pshared es distinto de 0, el semáforo puede ser compartido entre procesos. En este caso, el semáforo debe estar ubicado en una región de memoria compartida, accesible por todos los procesos que lo usen.
- **value:** Valor inicial del semáforo.
  - Define el número inicial de recursos disponibles o el estado inicial del semáforo. Para un semáforo binario, el valor inicial debe ser 0 o 1. Para un semáforo contable, puede ser cualquier valor positivo que represente el número de recursos disponibles.

  Ejemplo:

```
sem_t semaforo;
sem_init(&semaforo, 0, 1);  // Inicializar semáforo binario con valor 1
```

Decrementar el valor del semáforo (wait): int sem_wait(sem_t *sem);

- Decrementa el valor del semáforo. Si el valor es menor o igual a 0, el proceso se bloquea hasta que el semáforo sea positivo.

Incrementar el valor del semáforo (signal): int sem_post(sem_t *sem);

- Incrementa el valor del semáforo. Si hay procesos bloqueados, uno de ellos se desbloquea y puede continuar.

Librería a Utilizar:

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

Ejemplo de Uso del Semáforo Contable:
Este programa demuestra cómo usar un semáforo contable para permitir el acceso simultáneo de hasta tres hilos a un recurso compartido.

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

sem_t semaforo;

void* proceso(void* arg) {
    sem_wait(&semaforo);  // P operation
    printf("Hilo %d está en la sección crítica.\n", *(int*)arg);
    sleep(1); // Simula trabajo en la sección crítica
    sem_post(&semaforo);  // V operation
    return NULL;
}

int main() {
    pthread_t hilos[5];
    int ids[5] = {1, 2, 3, 4, 5};

    sem_init(&semaforo, 0, 3);  // Inicializar semáforo contable con valor 3

    for (int i = 0; i < 5; i++) {
        pthread_create(&hilos[i], NULL, proceso, &ids[i]);
    }

    for (int i = 0; i < 5; i++) {
        pthread_join(hilos[i], NULL);
```

```
    }

    sem_destroy(&semaforo);
    return 0;
}
```

Cuando inicializas un semáforo contable con un valor de 3 usando sem_init(&semaforo, 0, 3);, significa que hasta tres hilos pueden estar en la sección crítica simultáneamente. Sin embargo, no significa que solo tres hilos pueden acceder a la función proceso. Lo que significa es que solo tres hilos pueden estar dentro de la sección crítica (el bloque de código protegido por el semáforo) al mismo tiempo. Los otros hilos tendrán que esperar hasta que uno de los hilos dentro de la sección crítica salga y libere el semáforo.

1. **Declaración del Semáforo:** sem_t semaforo; declara una variable de tipo semáforo.
2. **Inicialización del Semáforo:** sem_init(&semaforo, 0, 3); inicializa el semáforo con un valor de 3.
    - **sem**: Puntero a la variable de tipo sem_t que representa el semáforo.
    - pshared: 0, indicando que el semáforo se compartirá entre los hilos del mismo proceso.
    - **value: 3,** indicando que hasta tres hilos pueden acceder simultáneamente a la sección crítica.
3. **Función proceso:** Cada hilo ejecuta esta función. Usa sem_wait para decrementar el valor del semáforo antes de entrar en la sección crítica. Si el valor del semáforo es menor o igual a 0, el hilo se bloquea. Una vez dentro de la sección crítica, el hilo realiza su trabajo simulado (usando sleep(1)) y luego usa sem_post para incrementar el valor del semáforo, permitiendo que otro hilo pueda entrar en la sección crítica.
4. **Creación de Hilos:** Se crean 5 hilos, cada uno ejecutando la función proceso.
5. **Esperar a que Terminen los Hilos:** pthread_join asegura que el programa principal espera a que todos los hilos terminen su ejecución.
6. **Destrucción del Semáforo:** sem_destroy libera los recursos del semáforo.

Detalles sobre el Acceso a la Sección Crítica:

- **Hilos Concurrentes:** Cuando el programa comienza, se crean 5 hilos. Todos estos hilos intentan ejecutar la función proceso.
- **Acceso a la Sección Crítica:** Los primeros tres hilos que ejecutan sem_wait(&semaforo); decrementan el valor del semáforo de 3 a 2, luego de 2 a 1, y finalmente de 1 a 0. Estos tres hilos pueden entrar en la sección crítica simultáneamente.
- **Bloqueo de Hilos Adicionales:** Cuando el cuarto hilo ejecuta sem_wait(&semaforo);, el valor del semáforo es 0. Esto bloquea el cuarto hilo hasta que uno de los hilos en la sección crítica llama a sem_post(&semaforo);, incrementando el valor del semáforo y permitiendo que el cuarto hilo entre.

- **Secuencia de Acceso:** Este proceso asegura que solo tres hilos estén en la sección crítica simultáneamente. Una vez que uno de los hilos sale de la sección crítica (llamando a sem_post), otro hilo bloqueado puede entrar.

Ejemplo de Uso del Semáforo Binario:
Este programa demuestra cómo usar un semáforo binario para asegurar que solo un hilo acceda a la sección crítica a la vez.

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

sem_t semaforo;

void* proceso(void* arg) {
    sem_wait(&semaforo);  // P operation
    printf("Hilo %d está utilizando la impresora.\n", *(int*)arg);
    sleep(1); // Simula uso de la impresora
    sem_post(&semaforo);  // V operation
    return NULL;
}

int main() {
    pthread_t hilos[5];
    int ids[5] = {1, 2, 3, 4, 5};

    sem_init(&semaforo, 0, 1);  // Inicializar semáforo binario con valor 1

    for (int i = 0; i < 5; i++) {
        pthread_create(&hilos[i], NULL, proceso, &ids[i]);
    }

    for (int i = 0; i < 5; i++) {
        pthread_join(hilos[i], NULL);
    }

    sem_destroy(&semaforo);
    return 0;
}
```

Explicación del Programa de Semáforo Binario:
1. **Declaración del Semáforo:** sem_t semaforo; declara una variable de tipo semáforo.

2. **Inicialización del Semáforo:** sem_init(&semaforo, 0, 1); inicializa el semáforo con un valor de 1, permitiendo que solo un hilo acceda a la sección crítica a la vez.
    - **sem**: Puntero a la variable de tipo sem_t que representa el semáforo.
    - **pshared: 0,** indicando que el semáforo se compartirá entre los hilos del mismo proceso.
    - **value: 1,** indicando que solo un hilo puede acceder a la sección crítica a la vez.
3. **Función proceso:** Cada hilo ejecuta esta función. Usa sem_wait para decrementar el valor del semáforo antes de entrar en la sección crítica. Si el valor del semáforo es menor o igual a 0, el hilo se bloquea. Una vez dentro de la sección crítica, el hilo realiza su trabajo simulado (usando sleep(1)) y luego usa sem_post para incrementar el valor del semáforo, permitiendo que otro hilo pueda entrar en la sección crítica.
4. **Creación de Hilos:** Se crean 5 hilos, cada uno ejecutando la función proceso.
5. **Esperar a que Terminen los Hilos**: pthread_join asegura que el programa principal espera a que todos los hilos terminen su ejecución.
6. **Destrucción del Semáforo**: sem_destroy libera los recursos del semáforo.

Diferencias Clave entre Semáforo Contable y Semáforo Binario

- **Semáforo Contable:** Permite que un número limitado de hilos accedan simultáneamente a la sección crítica. Es útil para controlar el acceso a recursos limitados, como conexiones de base de datos, donde se puede permitir un número específico de accesos concurrentes.
- **Semáforo Binario:** Permite que solo un hilo acceda a la sección crítica a la vez. Es útil para proteger secciones críticas donde solo un hilo debe estar presente en cualquier momento para evitar condiciones de carrera.

Conclusión

Comprender y utilizar correctamente los mecanismos de sincronización como semáforos contables y semáforos binarios es fundamental para la programación concurrente y el diseño eficiente de sistemas operativos. Estos mecanismos previenen problemas de concurrencia y aseguran que los recursos compartidos sean utilizados de manera ordenada y sin conflictos, garantizando la estabilidad y fiabilidad del sistema.

3. PROCEDIMIENTO

3.1 Código sin semáforos

Ejecutar varias ocasiones el código Sin semáforos con la variable MAX igual

a 1000, 10000, 100000, 1000000.
Completar la tabla 1 con el valor de la variable a que se imprime en pantalla.
Comentar por qué se obtienen los resultados.

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


//Variable compartida
int a=0,MAX=10;

void * funcion_hilo1(void *arg);
void * funcion_hilo2(void *arg);

int main (void)
{
        pthread_t hilo1, hilo2;
        pthread_create(&hilo1, NULL, *funcion_hilo1, NULL);
        pthread_create(&hilo2, NULL, *funcion_hilo2, NULL);
        //Para esperar que terminen los hilos
        pthread_join(hilo1, NULL);
        pthread_join(hilo2, NULL);
        printf ("El valor de a es %d",a);
        return 0;
}

void * funcion_hilo1(void *arg)
{
        for(int i=0; i<MAX; i++)
        {
                a +=1;
        }
}

void * funcion_hilo2(void *arg)
{
        for(int i=0; i<MAX; i++)
        {
                a -=1;
        }
}
```

Figura 1. Código sin semáforos

Para ejecutar utilizamos: gcc -pthread figura1.c -std=c99 -o figura1

Tabla1.

MAX	a
1000	
10000	
100000	
1000000	

3.2 Código con semáforos

Ejecutar varias ocasiones el código con semáforos con la variable MAX igual a 1000000, 1e9, 1e12.
Completar la tabla 2 con el valor de la variable a que se imprime en pantalla.
Comentar por qué se obtienen los resultados.

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

//Variable compartida
int a=0;
long MAX=1000000000;

void * funcion_hilo1(void *arg);
void * funcion_hilo2(void *arg);

//Declarar el semáforo
sem_t s;

int main (void)
{
        pthread_t hilo1, hilo2;
        //Inicializa el semáforo s, 0 porque no es compartido entre procesos sino ente hilos de un mismo proceso, valor inicial
        sem_init(&s,0,1);
        pthread_create(&hilo1, NULL, *funcion_hilo1, NULL);
        pthread_create(&hilo2, NULL, *funcion_hilo2, NULL);
        //Para esperar que terminen los hilos
        pthread_join(hilo1, NULL);
        pthread_join(hilo2, NULL);
        printf ("El valor de a es %d \n",a);
        return 0;
}

void * funcion_hilo1(void *arg)
{
        for(int i=0; i<MAX; i++)
        {
                //Bloqueo la variable compartida con sem_wait
                sem_wait(&s);
                a +=1;
                //Incremento el valor del semáforo
                sem_post(&s);
        }
}

void * funcion_hilo2(void *arg)
{
        for(int i=0; i<MAX; i++)
        {
                sem_wait(&s);
                a -=1;
                sem_post(&s);
        }
}
```

Figura 2. Código con semáforos

Tabla 2

MAX	a
1e6	
1e9	
1e12	

3.3 Código con 3 hilos e impresión de semáforo

Para el código de la Figura 2, añadir un tercer hilo e imprimir en pantalla el valor del semáforo durante cada ejecución. Colocar el código propuesto en el informe con comentarios en las instrucciones más relevantes.

3.4 Código con semáforo. Variable color de texto.

Tomar como base el código de la Figura 3 propuesto y añadir un semáforo

que permita compartir la variable que define el color del texto.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void * rojo(void *id)
{
#define A "\x1b[31m"
printf (A "Este texto es ROJO! \n");
}

void * verde(void *id)
{
#define B "\x1b[32m"
printf(B  "Este texto es VERDE! \n");
}

int main()
{
pthread_t hilo_rojo, hilo_verde;
pthread_create(&hilo_rojo, NULL,*rojo, NULL);
pthread_create(&hilo_verde, NULL,*verde, NULL);
pthread_join(hilo_rojo, NULL);
pthread_join(hilo_verde, NULL);
#define C "\x1b[34m"
printf (C   "Este texto es AZUL \n");
return 0;
}
```

Figura 3. Script de ejemplo para observar la variable "Color de texto"

3.5 Código con mutex.

Tomar como base los códigos de las Figuras 3 y 4 y verificar que un mutex permita compartir la variable que define el color del texto.

```c
#include <pthread.h>
#include <stdio.h>
#define MAX 10000000

int a=0;

void * funcion_hilo1(void *arg)
{

        pthread_mutex_t * mutex=arg;
        pthread_mutex_lock(mutex);
        for(int i=0; i<MAX; i++)
        {
                a+=1;
        }

        pthread_mutex_unlock(mutex);
}
void * funcion_hilo2(void *arg)
{
        pthread_mutex_t * mutex=arg;
        pthread_mutex_lock(mutex);
        for(int i=0; i<MAX; i++)
        {

                a-=1;
        }

        pthread_mutex_unlock(mutex);
}

int main()
{
        pthread_t hilo1, hilo2;
        pthread_mutex_t mutex;
        pthread_mutex_init(&mutex,NULL);
        pthread_create(&hilo1, NULL,funcion_hilo1, &mutex);
        pthread_create(&hilo2, NULL,funcion_hilo2, &mutex);
        pthread_join(hilo1, NULL);
        pthread_join(hilo2, NULL);
        pthread_mutex_destroy(&mutex);
        printf ("El valor de a es %d \n",a);
        return 0;
}
```

Figura 4.Código con mutex

4. INFORME

Resolver todos los problemas propuestos en las preguntas de la sección 3.

5. CONCLUSIONES Y RECOMENDACIONES