UA

ProgramaciónConcurrente

DanielAsensiRochDNI : **48776120C**

14 de diciembre de 2022

Índice

1. Ventajas y inconvenientes de pthread_mutex_t y pthread_cond_t

pthread_mutex_t y pthread_cond_t son dos tipos de variables que se utilizan en la programación de hilos (threads) en C. Un pthread_mutex_t se utiliza para proteger una sección crítica de código de modo que sólo un hilo pueda acceder a ella a la vez, mientras que un pthread_cond_t se utiliza para sincronizar diferentes hilos.

Las principales ventajas de utilizar pthread_mutex_t y pthread_cond_t son:

1. Permiten implementar la exclusión mutua, lo que es esencial para evitar conflictos y errores en el acceso a recursos compartidos entre diferentes hilos.
2. Ayudan a mejorar el rendimiento del programa al permitir que varios hilos ejecuten tareas en paralelo.
3. Proporcionan una manera fácil y eficiente de sincronizar diferentes hilos de manera que se asegure que ciertas tareas se ejecuten en un orden determinado.

Sin embargo, también hay algunos inconvenientes que se deben tener en cuenta al utilizar pthread_mutex_t y pthread_cond_t:

1. Uno de los principales inconvenientes de los mutexes es que pueden causar una interbloqueo, también conocido como "deadlock". Esto ocurre cuando dos o más hilos están esperando por el mismo mutex y ninguno de ellos puede continuar. Como resultado, el programa queda atrapado en un estado en el que ningún hilo puede avanzar.
2. Otro inconveniente es que el uso incorrecto de mutexes y variables de condición puede llevar a errores difíciles de detectar y depurar. Por ejemplo, si un hilo no desbloquea un mutex después de adquirirlo, otros hilos pueden quedarse esperando de manera indefinida, lo que puede causar también una interbloqueo.
3. Además, el uso excesivo de mutexes y variables de condición puede ralentizar el rendimiento del programa, ya que cada operación de bloqueo y desbloqueo implica una pequeña sobrecarga en el sistema. Por lo tanto, es importante utilizar estas herramientas de manera eficiente y evitar el uso innecesario.

2. Problema de los Filosofos

2.1. Descripción del problema

El problema de la cena de los filósofos es un problema clásico en la teoría de la sincronización de procesos. El problema plantea un escenario en el que cinco filósofos se sientan en una mesa redonda y tienen que comer. Cada filósofo tiene un tenedor a su derecha y a su izquierda, y para poder comer, un filósofo necesita tener ambos tenedores disponibles.

El problema surge cuando dos o más filósofos intentan tomar los mismos tenedores al mismo tiempo, lo que puede causar un deadlock (un bloqueo) en el que ningún filósofo puede comer. La solución al problema implica encontrar una manera de asignar tenedores a los filósofos de manera que se eviten estos bloqueos.

2.2. Resolución del problema

2.2.1. Estructura de Comida.h

Lo primero que deberemos realizar para resolución de nuestro problema, es definir los métodos y estructuras que podrán realizar nuestros filósofos para interactuar con la comida.

En primer lugar definiremos **la estructura de la comida**, para controlar la comida de los filosofos.

```
    #ifndef _COMIDA_H_
#define _COMIDA_H_
#include <sys/types.h>
#include <pthread.h>
```

```
#define CANT_FILOSOFOS 5

//Estructura del monitor que controla la comida de filosofos
typedef struct comida
{
    // nº de cubiertos disponibles de cada filosofo, de 0 a 2.
    int cubiertos[CANT_FILOSOFOS];
    // semaforo para acceder a la sección crítica
    pthread_mutex_t candadoSC;
    // variable de condicion por cada filosofo
    pthread_cond_t puedeComer[CANT_FILOSOFOS];
} comidaStruct
```

Una vez definida la estructura, definiremos el inicializador, el cual le dirá a todos los filosofos que pueden comer, ya que los cubiertos disponibles para cada uno de ellos estarán a 2 que son los necesarios para poder comer:

```
    // Inicialmente los filosofos pueden comer y los cubiertoss estan disponibles = 2
#define EMPIEZA_LA_COMIDA                                              \
    {                                                                  \
        {[0 ...(CANT_FILOSOFOS - 1)] = 2},                             \
            PTHREAD_MUTEX_INITIALIZER,                                 \
        {                                                              \
            [0 ...(CANT_FILOSOFOS - 1)] = PTHREAD_COND_INITIALIZER     \
        }                                                              \
    }
```

Una vez definidos los métodos inicializadores, definiremos los métodos que podrá usar cada filósofo para interactuar con la Sección crítica y la comida. Las funciones cubierto derecho y izquierdo seleccionarán los cubiertos correspondientes para cada uno de los filósofos.

```
//El filosofo del id devuelve el cubierto derecho
int cubiertoDerecho(int id);
//El filosofo del id devuelve el cubierto izquierdo
int cubiertoIzquierdo(int id);
//El filosofo del id coge los cubierto
void cogerCubierto(comidaStruct *c, int id);
//El filosofo del id deja los cubiertos
void dejarCubierto(comidaStruct *c, int id);
```

2.2.2. Cuerpo de los métodos definidos Comida.c

La selección del cubierto, pasado el id del filósofo se realizará mediante los métodos **cubiertoDerecho y cubiertoIzquierdo**

```
    int cubiertoDerecho(int id)
{
    return (id + CANT_FILOSOFOS - 1) % CANT_FILOSOFOS;
}

int cubiertoIzquierdo(int id)
{
    return (id + 1) % CANT_FILOSOFOS;
}
```

Para coger los cubiertos deberemos interactuar con el puntero a la estructura de la comida. En estos bloquearemos y desbloquearemos el acceso a la sección crítica, además procederemos a la espera si fuera necesario.

```
    /**
 * @brief Coge los cubiertos
 *
 * @param c puntero a la estructura
 * @param id del filosofo que coge los cubiertos
 */
void cogerCubierto(comidaStruct *c, int id)
{

    //Bloqueamos el acceso
    pthread_mutex_lock(&c->candadoSC);
    while (c->cubiertos[id] != 2)
    {
        //Esperamos a que los cubiertos esten disponibles
        pthread_cond_wait(&c->puedeComer[id], &c->candadoSC);
    }

    //Cogemos ambos cubiertos
    c->cubiertos[cubiertoIzquierdo(id)] -= 1;
    c->cubiertos[cubiertoDerecho(id)] -= 1;
    printf("El filosofo %d ha cogido los dos cubiertos\n", id);
    //Desbloqueamos
    pthread_mutex_unlock(&c->candadoSC);
}


/**
 * @brief Deja los cubiertos
 *
 * @param c puntero a la estructura
 * @param id del filosofo que coge los cubiertos
 */
void dejarCubierto(comidaStruct *c, int id)
{

    //Bloqueamos el acceso a la sección crítica
    pthread_mutex_lock(&c->candadoSC);
    printf("El filosofo %d deja los cubiertos\n", id);

    //Le damos un cubierto más a los filosofos de nuestros lados
    c->cubiertos[cubiertoIzquierdo(id)] += 1;
    c->cubiertos[cubiertoDerecho(id)] += 1;

    // si tienen 2 cubiertos disponibles, ya pueden comer.
    if (c->cubiertos[cubiertoIzquierdo(id)] == 2)
    {
        pthread_cond_signal(&c->puedeComer[cubiertoIzquierdo(id)]);
    }
    if (c->cubiertos[cubiertoDerecho(id)] == 2)
    {
        pthread_cond_signal(&c->puedeComer[cubiertoDerecho(id)]);
    }
```

```
    pthread_mutex_unlock(&c->candadoSC);
    // desbloquear acceso a la seccion critica.
}
```

2.2.3. Código ejecutado por los Filósofos

Cada uno de los filósofos se ejecuta en un hilo diferente, su funcionamiento será el siguiente:

```
    // veces que come un filosofo
#define VECES_COME 5

// puntero al monitor
comidaStruct *c;

void pensar(int id)
{
    printf("El filosofo %d se encuentra pensando\n", id);
    sleep(1);
}

void comer(int id)
{
    printf("El filosofo %d se ha puesto a comer\n", id);
    sleep(1);
    printf("El filosofo %d ha llenado la barriga\n", id);
}

void *filosofoHace(void *id)
{

    int i = *(int *)id;

    for (int h = 0; h < VECES_COME; h++)
    {
        // Se pone a pensar
        pensar(i);
        // Coge los cubiertos para comer
        cogerCubierto(c, i);
        // Se pone a comer
        comer(i);
        // Deja los palillos y vuelve a empezar
        dejarCubierto(c, i);
    }
    // Si ya ha comido sus 5 veces nos salimos del hilo
    pthread_exit(id);
}

int main()
{
    pthread_t hiloFilosofo[CANT_FILOSOFOS];
    int filosofos[CANT_FILOSOFOS];

    int h;
    int error;
    int *salida;
```

```
    // inicializar el monitor que controla la comida
    comidaStruct comida = EMPIEZA_LA_COMIDA;
    c = &comida;

    // crear filosofos
    for (h = 0; h < CANT_FILOSOFOS; h++)
    {
        filosofos[h] = h;
        error = pthread_create(&hiloFilosofo[h], NULL, filosofoHace, &filosofos[h]);
        if (error)
        {
            fprintf(stderr, "Error: %d: %s\n", error, strerror(error));
            exit(-1);
        }
    }

    // terminar filosofos
    for (h = 0; h < CANT_FILOSOFOS; h++)
    {
        error = pthread_join(hiloFilosofo[h], (void **)&salida);
        if (error)
        {
            fprintf(stderr, "Error: %d: %s\n", error, strerror(error));
        }
    }

    return 0;
}
```