Universidad de Concepción
Facultad de Ingeniería
Departamento de Ing. Informática y
Cs. de la Computación

Informe proyecto 2

Sistemas Operativos
Profesora Cecilia Hernández

Alumnos:

Nicolás Araya

Martina Cádiz

Fecha:

29/11/21

Introducción

El presente proyecto tiene como objetivo entender y conocer sobre problemas asociados al uso de concurrencia y los mecanismos de sincronización para evitar estos problemas. El proyecto se divide en 3 partes con códigos escritos en lenguaje C.

Se busca implementar un programa que se divida en N hebras, donde la ejecución viene dada por M etapas. Para la elaboración del código se utilizó la librería pthreads.h de C, y se utilizó las barreras propias de la librería. Además, se implementaron así como una implementación basadas en elementos básicos como semáforos, mutex y variables de condición.

Primera parte

En esta sección se creó un programa que sea capaz de utilizar N hebras para sincronizar su ejecución usando M etapas. Para lograr la sincronización se usó una barrera.

Para este ejercicio, la librería que aportó a la solución fue pthread.h. Además, se utilizaron las librerías comunes de C para apoyar a la solución

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

Las variables globales a utilizar son dos barreras de tipo pthread_barrier_t, llamadas ciclo y barrera.

El problema se puede solucionar utilizando una sola barrera, pero se utilizan 2 para siempre imprimir algo para diferenciar el inicio de cada etapa.

También, se creó una estructura llamada arg_struct, que contiene el valor entero del hilo y el valor de las etapas totales.

```c
pthread_barrier_t ciclo;
pthread_barrier_t barrera;


struct arg_struct {
    int thread;
    int m_value;
};
```

El proceso que tendrá que trabajar cada hilo, corresponde a la función fun. En cada etapa, calcula un número aleatorio “a” y luego hace uso de las barreras declaradas previamente.

```c
void* fun(void* arguments){
    struct arg_struct *args = (struct arg_struct*) arguments;
    for(int i = 0; i < args->m_value; i++){
        int a = rand()%100 + 10;
        sleep(rand()%5+2);
        pthread_barrier_wait(&barrera);
        pthread_barrier_wait(&ciclo);
        printf("thread:  %d, value = %d\n", args->thread, a);
    }
}
```

Por último, la función main solicita como parámetros el número de hilos y de etapas. Inicializa las barreras, con N+1 ya que se considera el hilo del proceso principal. Luego, el ciclo for crea los N hilos utilizando la función pthread_create y se le entrega como argumentos el vector h y la estructura asociada al hilo.

```c
int main(int argc, char const *argv[]){
    srand(time(NULL));
    if(argc != 3) return 0;

    int N = atoi(argv[1]); //argumento hebras
    int M = atoi(argv[2]); //argumento etapas

    struct arg_struct args[N];
    pthread_t h[N]; //vector con hebras

    pthread_barrier_init(&barrera, NULL, N+1); //para las N hebras y la hebra main
    pthread_barrier_init(&ciclo, NULL, N+1);

    for(int i = 0; i < N; i++){
        args[i].thread = i;  //valor hebra
        args[i].m_value = M; //valor etapas totales
        if(pthread_create(&h[i], NULL,  &fun, (void*)&args[i]) != 0) perror("Error 1"); //creación del hilo
    }

    for(int i = 0; i < M; i++){
        pthread_barrier_wait(&barrera); //espera a que se realicen todas las hebras
        printf("\nEtapa %d Completa \n", i+1);
        pthread_barrier_wait(&ciclo);  //imprime todo junto
    }

    for (int i = 0; i < N; i++) {
        if (pthread_join(h[i], NULL) != 0) {
            perror("Failed to join thread");
        }
    }

    return 0;
}
```

Por último, el segundo ciclo for se encarga de llevar a cabo la función de las barreras. Sincronizando los hilos por etapa.

El último ciclo for es para verificar que no haya habido algún error con los hilos

Segunda parte

1)

Una posible implementación de una barrera reutilizable con semáforos podría ser la siguiente

```
    sem_wait(&barrier->mutex);
    barrier->count = barrier->count + 1;
    sem_post(&barrier->mutex);

    if (barrier->count == barrier->n) {
        sem_post(&barrier->s1);
    }


    sem_wait(&barrier->s1);
    sem_post(&barrier->s1);

    sem_wait(&barrier->mutex);
    barrier->count = barrier->count -1;
    sem_post(&barrier->mutex);


    if(barrier->count==0){
        sem_wait(&barrier->s1);
    }
```

Esta implementación, si bien es funcional, puede fallar en ciertos casos.

1. Si algún hilo es interrumpido luego de realizar el mutex inicial podría causar un error ya que provocaría que se llame a post en un momento que no corresponde.
2. Un hilo podría pasar por el segundo mutex, lo que provocaría que este una etapa adelantado que el resto de los hilos

La solución propuesta se basa en usar 2 semáforos, se usó la librería semaphore.h y pthread.h. También se utilizaron las dos variables globales que hacen referencia a las barreras creadas.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>
```

Y se crearon 2 estructuras. Una llamada barrier, que contiene 3 semáforos y dos contadores. La otra estructura es la misma mencionada en la parte 1.

```c
struct arg_struct {
    int thread;
    int m_value;
};

typedef struct {
    int n;
    int count;
    sem_t mutex;
    sem_t s1;
    sem_t s2;
}barrier;
```

Para inicializar las barreras, se usó la función barr_init.

```c
void barr_init(barrier * b, int n){
    b->n = n;
    b->count = 0;
    sem_init(&b->mutex, 0, 1);
    sem_init(&b->s1, 0, 0);
    sem_init(&b->s2, 0, 1);
}
```

Por otro lado, la función que soluciona el error declarado anteriormente es wait_barr. De esta función fue a la que se le cambió la implementación.

```
void wait_barr(barrier *b){
    sem_wait(&b->mutex);
    b->count++;

    if (b->count == b->n) {
        sem_wait(&b->s2);
        sem_post(&b->s1);
    }
    sem_post(&b->mutex);

    sem_wait(&b->s1);
    sem_post(&b->s1);

    sem_wait(&b->mutex);
    b->count = b->count -1;

    if(b->count==0){
        sem_wait(&b->s1);
        sem_post(&b->s2);
    }
    sem_post(&b->mutex);

    sem_wait(&b->s2);
    sem_post(&b->s2);
}


barrier b1, b2;
```

Básicamente, lo que ocurre es que se inicializa la barrera con un mutex en 1, el primer semáforo se encuentra bloqueado, mientras que el segundo semáforo se encuentra abierto. La idea es que cuando todos los hilos lleguen al primer semáforo este se desbloquee y se bloquee el segundo semáforo, y luego ocurre lo contrario, cuando llegamos al segundo semáforo se desbloquea, y se bloquea el primero, evitando asi que algun hilo pase de etapa antes que los demás.

También ahora dentro de los mutex se encuentra el condicional if, evitando así que si un hilo se cae durante la ejecución haga post.

Tercera parte

1) Una posible implementación de una barrera utilizando monitor podría ser la siguiente

```
typedef struct{
    int n;
    int count;
    pthread_mutex_t mimutex;
    pthread_cond_t condicion;
}Monitor;
```

```
void wait_monitor(Monitor* m){
    pthread_mutex_lock(&m->mimutex);
    m->count++;
    if(m->count == m->n){
        m->count=0;
        pthread_cond_broadcast(&m->condicion); //notifica a todas las hebras que esperan la condicion
    }
    pthread_mutex_unlock(&m->mimutex);
}
```

En este caso no se está asignando la condición a cada hebra que llama a wait, por lo que cuando se realice broadcast las hebras no serán notificadas y no continuarán de ejecutarse.

2. La solución es asignar a cada hebra la condición, para que estas sean notificadas cuando se cumpla la condición. Lo cual ocurre cuando se llama N veces a la función wait y se realiza el broadcast.

```
void init_monitor(Monitor* m, int num){
    m->n = num;
    m->count = 0;
}

void wait_monitor(Monitor* m){
    pthread_mutex_lock(&m->mimutex);
    m->count++;
    if(m->count == m->n){
        m->count=0;
        pthread_cond_broadcast(&m->condicion); //notifica a todas las hebras que esperan la condicion
    }
    else{
        pthread_cond_wait(&m->condicion, &m->mimutex); //asigna a la hebra que espera hasta que se cumpla la condicion

    }
    pthread_mutex_unlock(&m->mimutex);
}
```