Universidad de Concepción
Ingeniería civil informática

DEPARTAMENTO
INGENIERÍA INFORMÁTICA
Y CIENCIAS DE LA COMPUTACIÓN
FACULTAD DE INGENIERÍA UNIVERSIDAD DE CONCEPCIÓN

Proyecto 2

Integrantes:
Leonardo Aravena C.
Ivonne Flores R.
Felipe Henríquez R.
Patricio Inostroza A.
Profesora: Dra. Cecilia Hernández R.

Asignatura: Sistemas operativos.
Fecha: 29/11/21

Desarrollo

El proyecto consiste en realizar una implementación de barrera reutilizable, para N hebras que realicen M etapas, utilizando monitor, semáforos y pthreads.

Parte 1:
Implementar una aplicación simple usando pthreads o threads de c++11 o superior con la implementación de barreras disponibles para cada caso.

Esta implementación es realizada en lenguaje C:

Primero creamos las variables globales

```
pthread_barrier_t mibarrerabonita;  // Barrera
int M;                              // Etapas
```

Luego en el main, recibimos por argumento el número de hebras (N) y el número de etapas (M), para así crear las N hebras después de haber inicializado la barrera.

```
pthread_barrier_init(&mibarrerabonita, NULL, N+1);

for(i = 0; i < N ; i++){

        myids[i] = i;
        pthread_create(&ids[i] , NULL , f , &myids[i] );
}
```

Al crear la hebra se le pasa la función f, y un arreglo de enteros que contiene un número para identificar la hebra en ejecución. La función recibe el id de la hebra y debe entrar a un ciclo que itera M veces, en cada iteración la hebra realiza un “trabajo” que es llamar a la función sleep() con un valor aleatorio y lo simula, seguido a esto se llama a pthread_barrier_wait(&mibarrerabonita), en donde le pasamos la barrera creada como variable global y esta queda dormida hasta que todas las hebras lleguen a este punto.

```
void* f(void * id){
        int id_h = *(int*)id;
        int espera;
        for(int i = 0 ; i < M ; i++){
                espera = 1 + rand() % 3;
                printf("M = %d, Hebra %d espera por %d seg\n", i+1 , id_h , espera);
                sleep(espera); // "TRABAJO DE LA HEBRA"
                pthread_barrier_wait(&mibarrerabonita);    // Hebra espera
        }

        return NULL;
}
```

Finalmente, en un ciclo, la barrera espera que se completen las M etapas, sincronizando las hebras correctamente durante dichas etapas. La hebra main espera por las N hebras (join) y se destruye la barrera utilizada.

```
//Hebra main espera
for ( int j = 0; j < M; j++){
        printf("espero..\n");
        pthread_barrier_wait(&mibarrerabonita);
}

//Hebra main espera por las N hebras
for (int i=0; i < N; i++) {
        pthread_join(ids[i], NULL);
}

printf("main() termina\n");

// Destruimos barrera
pthread_barrier_destroy(&mibarrerabonita);
```

Parte 2:

Defina algoritmos e implementaciones de una barrera reutilizable usando semáforos.

Parte 2.1

Proporcione un algoritmo que expone algún problema de sincronización explicando claramente en qué consisten tales problemas. Defina escenarios donde los problemas se pueden producir.

Se crean 2 semáforos, s y mutex, con valor 0 y 1 respectivamente, un contador de hebras llamado *cont* con valor 0 y *N* que es el total de hebras.

```
s  = 0;       // Semáforo
mutex = 1 // Semáforo
cont = 0;   // Contador de hebras
N = núm. de hebras

barrera_mala(){
        wait(mutex)
        cont++;

        if(cont == N){
```

```
            signal(s);
            cont = 0;
        }

        signal(mutex);

        wait(s);
        signal(s);

        }
}
```

Para las N hebras que se ejecuten en la primera etapa el algoritmo funcionará correctamente, el problema ocurre en la segunda etapa, el valor del semáforo s será afectado y no comienza con valor 0 sino con valor 1. Al comenzar la siguiente etapa con valor 1 la hebra entrará al wait pero no quedará en la cola, quedará con valor 0 y seguido a esto hara un signal que hará que el valor quede nuevamente en 1, así las hebras no quedarán en espera y se ejecutarán sin esperar que la última hebra finalice perdiendo la sincronización.

Parte 2.2 *Proporcione un algoritmo correcto, explique claramente cómo usa cada uno de los semáforos utilizados y use su implementación para sincronizar N hebras en M etapas.*

Primero, se crean los semáforos y variables globales

```
sem_t mutex;
sem_t b;
int cont; // Contador de hebras
int N,M;  // N: Hebras M: Etapas
```

En el main, se inicializan los semáforos, como se ve a continuación:

```
sem_init(&mutex, 1, 1);
sem_init(&b, 1, 0);
```

Luego, en el main igualmente, se crean las N hebras, y se inicializan.

```
for (i = 0; i < N; i++){
        myids[i] = i;
        pthread_create(&ids[i], NULL, f, &myids[i]);

}
```

Al crear la hebra se le pasa la función f, y un arreglo de enteros que contiene un número para identificar la hebra en ejecución. La función recibe el id de la hebra y debe entrar a un ciclo que itera M veces, en cada iteración la hebra realiza un “trabajo” que es llamar a la

función sleep() con un valor aleatorio y lo simula, seguido a esto se llama a barrera(), en donde los semáforos se encargan de sincronizar las hebras en cada una de las M etapas.

```
void* f(void *id){

	int id_h = *(int*)id;

	for (int i = 0; i < M; i++){

		int espera = 1 + rand() % 3;
		printf("M = %d, Hebra: %d espera por %d seg\n",i+1,id_h,espera);
		sleep(espera);
		barrera();

	}

	return NULL;
}
```

En la función barrera, se hace un wait en el semáforo mutex, el cual disminuye a 0 y se aumentan el contador, luego, si el contador es menor a las N hebras, se realiza un signal en el semáforo mutex (permitiendo así que otra hebra pueda entrar y aumentar el contador), y duerme a la hebra con el semáforo b (wait). Por otro lado, si ya se durmieron las N-1 hebras, se realiza un signal en el semáforo mutex, el contador se reinicia a 0, y en un ciclo for, se despiertan las N-1 hebras (con N-1 signal en el semáforo b).

```
void barrera(){

	sem_wait(&mutex);
	cont++;
	if(cont < N){
		sem_post(&mutex);
		sem_wait(&b);
	}

	else{
		sem_post(&mutex);
		cont = 0;
		for(int i = 1; i < N; i++){
			sem_post(&b);
		}
	}

}
```

Finalmente, la hebra main espera por las N hebras (join).

```
for (i=0; i < N; i++) {
        pthread_join(ids[i], NULL);
}
```

Parte 3: *Implementar una barrera reutilizable usando un monitor*

Parte 3.1 *Proporcione un algoritmo que expone algún problema de sincronización explicando claramente en qué consisten tales problemas. Defina escenarios donde los problemas se pueden producir.*

Se crea una clase de nombre *Monitor_malo()*, con métodos de *constructor* y *esperar()*. El constructor recibe por argumento la cantidad de hebras existentes. Luego al crear las hebras, se les entregará el objeto monitor creado, y así las hebras podrán ejecutar el método *esperar().* Las hebras realizarán M etapas, que se simulará en un ciclo for en su interior.

```
Monitor_malo(){
        private:
                int n_hebras, contador;
                condition c;
                mutex m;

        public:
                Monitor(int n_h);
                esperar();
}

Monitor(int n_h){
        n_hebras = n_h;
        contador = 0;   // Contador de hebras
}

esperar(){
        lock(m);
        contador++;
        if(contador < n_hebras){
                wait(c);
                contador--;
```

```
        }
        unlock(m);
        signal(c);
}
```

La etapa M=1 se va a ejecutar correctamente, las N-1 hebras van a esperar y luego la última hebra va a despertar a una de las N-1 hebras, esa hebra se va a encargar de decrementar el valor del contador, libera el mutex y va a despertar a otra hebra que haya quedado dormida.
El problema que se puede apreciar en este código, es que las hebras que fueron dormidas son N-1, y el contador no se va a reiniciar a 0, esto significa que en la siguiente etapa una hebra no va a entrar al if y no va a dormir, sino que despertará de inmediato a otra hebra, por lo que se va a perder la sincronización y no todas las hebras se encontrarán en la misma etapa esperándose.

Parte 3.2 *Proporcione un algoritmo correcto, explique claramente las variables de condición que necesita, y los métodos definidos en su monitor y use su implementación para sincronizar N hebras en M etapas.*

Para implementar el Monitor que permita la sincronización de las N hebras en M etapas se utilizó el lenguaje de programación c++ para crear una clase Monitor con sus atributos privados y métodos públicos.
Se necesita una variable de condición *c* y una variable de mutex *m* para controlar el acceso de la variable entera *contador* que regula la cantidad de hebras que han alcanzado la etapa actual.

El constructor del Monitor recibe como parámetro la cantidad de hebras que lo utilizarán como barrera, inicializando la variable privada *n_hebras* con el valor recibido y el *contador* en cero.

El método público *esperar()* se encarga de mantener a las hebras en espera hasta que todas hayan terminado la etapa. Para esto se utiliza la variable de mutex *m* para asegurar que sólo una hebra modifique el contador cuando haya terminado de ejecutar su etapa. Si el el contador es menor al número de hebras la hebra queda en espera al llamar a la función *wait(c)* de pthread. Si el contador alcanza el número de hebras se llama a la función *mandaSignal()* para despertar a todas las hebras.

El método privado *mandaSignal()* vuelve el *contador* en cero y despierta a todas las hebras que estaban dormidas haciendo *broadcast* sobre la variable de condición *c*.

```
Monitor(){
        private:
                int n_hebras, contador;
                condition c;
                mutex m;
                mandaSignal();

        public:
                Monitor(int n_h);
```

```
                esperar();

        }

Monitor(int n_h){
        n_hebras = n_h;
        contador = 0;
}

esperar(){
        lock(m);
        contador++;
        if(contador < n_hebras){
                wait(c);
        }
        else{
                mandaSignal();
        }
        unlock(m);
}

mandaSignal(){
        contador=0;
        broadcast(c);
}
```