Laboratorio 1 - Sistemas Operativos

Stephano Wurttele

Octubre 2020

Ejercicio 1

Para este ejercicio, elaboramos un programa que simule la técnica de Monte Carlo, que consiste en crear puntos aleatoriamente en el área de un cuadrado, y calcular cuantos de estos fueron creados dentro de un circulo inscrito en el cuadrado, donde el círculo es de radio 1. Para ello usamos una lógica matemática simple para encontrar dichos puntos, lógica que será encargada a una thread. El programa será descrito a continuación:

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

#define MAX 1000000
#define FACTOR ( MAX/2 )
#define N 100000000

volatile int points_in_circle;
```

Inicialmente,incluimos las librerías estándar, la librería de "math.h" para poder usar operaciones de exponenciación, y "pthread.h", para poder hacer uso de la interfaz de threads implementada por POSIX.

Además, definimos *MAX*, el número de dígitos de la primera parte de la aleatoriedad de la creación de los números que se usará en las coordenadas de los puntos; "FACTOR", que será utilizado para estandarizar los números a un máximo de 1; y "N", usado para el número de puntos generados. Finalmente, declaramos nuestra variable volátil entera *points_in_circle*, que guardará el número de puntos dentro del círculo.

```
void* generate_random(void *v){
  double x = 0;
  double y = 0;
  double temp = 0;
  srand (time(NULL));
  for(int i = 0; i < N ; ++i ){
    int num = (rand() % MAX) + 1.0;
    x = (double) (num - FACTOR) / FACTOR;
    num = (rand() % MAX) + 1.0;
```

```
    y = (double) (num - FACTOR) / FACTOR;
    temp = y < 0 ? y*(-1) : y;
    if(temp <= sqrt(1-pow(x,2))) ++points_in_circle;
  }
}
```

A continuación, definimos la función que sera ejecutada por la thread que se usará para crear los N puntos. Esta función determina números aleatorios de *MAX* dígitos, al cual luego resta la mitad, que vendría a ser equivalente al *FACTOR*, y se le divide entre el mismo número para normalizarlo a 1. Se crean dos números por iteración, pues estos serían los valores de las coordenadas x y y. Finalmente, se compara si está dentro del círculo y, de ser así, se suma a la variable *points_in_circle*. Todo este proceso se repite N veces.

```
int main(){
  pthread_t thread;
  pthread_create(&thread, NULL, generate_random, NULL);
  pthread_join(thread, NULL);
  printf("%f", 4.0 * points_in_circle / N);
  return 0;
}
```

Por último, el main. Aquí se crea el thread, al cuál se le asigna la función *generate_random* definida previamente. Luego se llama a *pthread_join*, que se encargará de hacer que la thread principal espere a que la función del argumento de la creación del thread secundario, que es la que calcula los puntos, termine. De esta forma, la siguiente línea que imprime el número de puntos dentro del círculo, multiplicado por 4 y dividido con el número de puntos totales, que vendría a ser la aproximación de π, se imprima correctamente.

Ejercicio 2

Este ejercicio es casi idéntico al anterior, por lo que solo se remarcaran las diferencias. Escencialmente, se busca que en lugar de usar un solo thread, se puedan usar N threads en la generación de puntos, y que el resultado siga siendo igual de coherente, sin que las threads se afecten entre ellas.

```
#define N_THREADS 4
pthread_mutex_t lock;
```

El primer cambio es la inclusión de la variable *N_THREADS*, la cual indica el número de threads a utilizar. Además, definimos el *pthread_mutex_t lock*, que será el encargado de mantener la exclusión mutua.

```
pthread_mutex_lock(&lock);
++points_in_circle;
pthread_mutex_unlock(&lock);
}
```

Otro cambio, posiblemente el más importante, es la inclusión del lock y unlock sobre el mutex previamente definido. Estos se colocan en la entrada

y salida de la sección crítica, donde habría un problema si más de un thread modificara la variable.

```
for(int i = 0; i < N_THREADS; ++i){
    pthread_create(&threads[i], NULL, generate_random, NULL);
  }
  for(int i = 0; i < N_THREADS; ++i){
    pthread_join(threads[i], NULL);
  }
```

Por último, en el main también se cambian unos detalles. En lugar de crear una sola thread, se crean N threads, y se le hace join a cada una de esas para esperar a que termine de generar los puntos cada una de ellas.

Lo importante a reconocer en este ejercicio es la sección crítica, la cuál debe ser protegido con mecanismos de exclusión mutua que, en este caso, es el mutex de pthread.

Ejercicio 3

Para el tercer ejercicio el enfoque cambiamos el enfoque a la secuencia de Fibonacci. Básicamente, queremos encontrar N elementos de la secuencia usando una thread que los calcule para que luego, una vez esta thread haya terminado, la thread principal pueda imprimir los resultados. Nuevamente, el programa será descrito a continuación:

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

int size;
int* array;
```

Inicialmente,incluimos las librerías estándar, y la librería "pthread.h", para poder hacer uso de la interfaz de threads implementada por POSIX.

Además, definimos declaramos la variable size, que será el tamaño del array que contendrá a la secuencia de Fibonacci (y por lo tanto el número de elementos que se quiere generar); y el arreglo mismo que será un puntero a enteros, de forma que luego se le podrá asignar el espacio de forma dinámica.

```
int recursive_fib( int pos ){
  if(pos <= 1){
    *(array+pos) = pos;
    return pos;
  }
  *(array+pos) = recursive_fib(pos-1) + recursive_fib(pos-2);
  return *(array+pos);
}

void* generate_fib(void *args){
  recursive_fib(size);
}
```

Para esta primera implementación, se usa un fibonacci recursivo. La función *generate_fib* es la que se le asignará a la thread que será declarada en el main. Adicional a eso, poco hay que decir, pues es un fibonacci tradicional que guarda los valores en el arreglo utilizando las posiciones de memoria desde el array para asignar enteros a las posiciones siguientes al puntero.

```
int main(int argc, char *argv[]){
  int i = 0;
  size = atoi(argv[1]);
  pthread_t thread;
  array = calloc(size, sizeof(int));
  pthread_create(&thread, NULL, generate_fib, NULL);
  pthread_join(thread, NULL);
  for ( i = 0; i < size+1; i++ )
    printf("Element %d = %d\n", i+1, *(array+i) );
  free(array);
  return 0;
}
```

Por último, el main. Aquí se lee el número enviado como parámetro y se asigna como cantidad de números a la secuencia de fibonacci. Con este tamaño se define el tamaño del arreglo. Luego, se crea el thread, al cuál se le asigna la función *generate_fib* definida previamente. Luego se llama a *pthread_join*, que se encargará de hacer que la thread principal espere a que la función del argumento de la creación del thread secundario, que es la que genera los valores, termine. De esta forma, el loop de la siguiente linea podrá imprimir los valores de la serie fibonacci almacenados en el arreglo correctamente.

Ejercicio 4

Para el último ejercicio se usa también la serie de fibonacci, pero con mayor concurrencia utilizando semáforos y futex de forma que para cada elemento producido, se lea al mismo tiempo en lugar de esperar a que todos sean producidos. Para esta implementación, tal vez un poco más de análisis sería oportuno. Podemos ver el proceso de creación de uno de los elementos de fibonacci como un **requisito** para que la thread principal pueda leer, igualmente, uno de estos valores. Esto tiene una similitud con el problema de **productor consumidor**, donde el productor es el thread de los elementos de fibonacci y el consumidor es la thread principal. Siguiendo este principio, podemos utilizar una solución que alterne con los semáforos o con el futex, siempre y cuando limite el consumo del recurso antes de que sea producido. Dado que el código de este ejercicio es casi idéntico al anterior, sólo se remarcaran las diferencias.

Con semáforos

```
#include<semaphore.h>
sem_t semaphore;
sem_t semaphore2;
```

Para esta implementación, debemos agregar la librería de semáforos para acceder a ellos.
Además, se definen los dos semaforos que se usarán para limitar la obtención y producción de recursos, con el objetivo que sea concurrente y alterne correctamente.

```
void* generate_fib(void *args){
  int init = 0;
  for(init; init < size; ++init){
    if (init <= 1){
      *(array+init) = init;
    }
    else{
      *(array+init) = *(array+init-1) + *(array+init-2);
    }
    printf("Adding element\n");
    sem_post(&semaphore);
    sem_wait(&semaphore2);
  }
}
```

Para estas dos implementaciones, se usa un fibonacci iterativo. En este fibonacci se trabaja con los semáforos, donde se hace un *sem_post*, equivalente a *semaphore_up* en teoría, en la variable *semaphore* donde se incrementará este semáforo que representa el número de recursos producidos, habilitando el mismo semáforo que quedó bloqueado cuando se intenta consumir el recurso en el thread principal. Luego se hace *sem_wait* en el semáforo2, equivalente a hacer *semaphore_down*, el cuál controla que no se produzca más de uno de los números de la serie y que alterne correctamente. Este semáforo sería incrementado cuando se consume/lee uno de los valores, que se verá en el código del main.

```
int main(int argc, char *argv[]){
  int i = 0;
  size = atoi(argv[1]);
  pthread_t thread;
  array = calloc(size, sizeof(int));
  sem_init(&semaphore, 0, 0);
  sem_init(&semaphore2, 0, 0);
  pthread_create(&thread, NULL, generate_fib, NULL);
  for(i; i < size; ++i){
    sem_wait(&semaphore);
    printf("Element %d = %d\n", i+1, *(array+i) );
    sem_post(&semaphore2);
  }
  pthread_join(thread, NULL);
  free(array);
  return 0;
}
```

En el main, escencialmente inicia de la misma forma. Sin embargo, eventualmente cambia cuando se empiezan a inicializar los semáforos. Luego, tenemos el for donde se van leyendo los valores del arreglo de la serie, el cuál hará un *sem_wait* en semaforo que evitará que se lean recursos no producidos, y al final se hace un *sem_post* que permitirá al producer seguir creando.

El algoritmo básicamente sigue la lógica de decrecer el **semáforo** cuando no hayan recursos creados, y decrecer **semaforo2** cuando los recursos sean creados. De esta forma, se crea un tope y un mínimo en ambas acciones, permitiendo que solo se pueda leer si se ha producido y que solo se pueda producir si ya se leyó. Esto es conocido como un semaforo binario, y es una implementación ideal para solucionar el problema del productor consumidor.

Con futex

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include <sys/syscall.h>
#include <linux/futex.h>
#include <sys/syscall.h>
#include <unistd.h>

int futex_wait(void *addr, int val1) {
  return syscall(SYS_futex, addr, FUTEX_WAIT,
                 val1, NULL, NULL, 0);
}

int futex_wake(void *addr, int n) {
    return syscall(SYS_futex, addr, FUTEX_WAKE,
                   n, NULL, NULL, 0);
}

int size;
int* array;
int flag = 0;
int* fut = &flag;
```

Para esta implementación, debemos agregar la librería de syscalls y de futex para acceder a ellos. Además, creamos las funciones de futex donde se llama al syscall respectivo recibiendo el puntero a fut, y el valor que se usará para comparación. Uno de ellos se encargará de enviar la thread a dormir mientras la otra la despertar.
También se definen las nuevas variables globales flag y fut, donde flag es un entero y fut es un puntero al mismo, el cual nos servirá como contenedor para nuestro futex.

```
void* generate_fib(void *args){
  int init = 0;
  for(init; init < size; ++init){
    futex_wait(fut, 1);
    if (init <= 1){
      *(array+init) = init;
    }
    else{
      *(array+init) = *(array+init-1) + *(array+init-2);
    }
    printf("Adding element\n");
    flag = 1;
    futex_wake(fut, 1);
```

```
  }
}
```

En este fibonacci se trabaja con el flag y el futex. Para ello, usamos el *flag* como señal de haber insertado un elemento, que se pone como 1 al final, de tal forma que al inicio del for se bloquea si esque dicho flag esta en 1, pues se llama a la funcion wait con el valor de 1. De esta forma, bloqueamos el ingreso de valores seguidos, permitiendo que se alterne la creación con el consumo del recurso. Este valor solo cambiará cuando se lea en el main, sección que analizaremos a continuación.

```
for(i; i < size; ++i){
    futex_wait(fut, 0);
    printf("Element %d = %d\n", i+1, *(array+i) );
    flag = 0;
    futex_wake(fut, 1);
  }
```

Finalmente, en el main, todo es igual excepto el recorrido de creación de datos. Aquí, empezamos con un futex que debe esperar si el valor no esta en 0, implicando que no se ha creado o producido ningún valor de la secuencia. Luego, cuando este valor es cambiado en el thread cuando se crea un recurso, este se desbloquea, lee el recurso, hasta que llama nuevamente a la funcion futex wake que permite al thread de producción continuar, y regresa el flag a 0 para que no se consuman más de los que deba.

El algoritmo básicamente sigue la lógica de bloqueo y desbloqueo condicionado por la variable específica en el futex. Este se encarga de detenerlo sin que sea necesario hacer syscalls a menos que sea completamente necesario.