

# Informe 3

## *Threads*

Rodrigo Hernández Gonzales  
201730036-1

November 23, 2020

## 1 Threads

### 1.1 Comienzo sincrónico

La idea del código está en la función *secant* donde si un thread termina su ejecución, bloquea la estructura de datos utilizada con el `lock(&estado)`. Con esto cuando un thread alcance esa sección del código y sea el "ganador", bloqueará la entrada a los siguientes threads y envía su id mediante `thread_exit()`. Si otro thread llega y al momento de encontrarse la estructura bloqueada, el thread retornará un -1 y se determinará que no ganó esta vuelta.

```
/*Si la estructura no esta bloqueada, la bloquea y retorna el id del
if (estado.used == false){
    lock(&estado);
    thread_exit(id);
}
else if (estado.used == true){
    thread_exit(-1);
}
```

Figure 1: función `secant()`

### 1.2 Main e inicialización de threads

En el main se definen mis variables auxiliares junto con los threads (4 en el código) mediante la función `thread create`. Así se crean el número de threads que se quieran y necesiten. Los thread ganadores se guardan en el array *ganadores* para luego imprimirlos en pantalla. Se puede apreciar igualmente para que el usuario pueda ingresar la cantidad de vueltas a la carrera, se agrega un `argv`. Por lo que cuando se llame al programa, se le deberá indicar la cantidad de vueltas con un espacio.

```

int main(int argc, char **argv){
    /*argv para que el usuario ingrese la cantidad de vueltas*/
    int max_vueltas = atoi(argv[1]);
    estado.used = false;
    estado.current_lap = 0;
    int ganadores[max_vueltas];
    for (int z = 0; z < max_vueltas; z++){
        /*Se crean los threads que se quieran*/
        for (int i = 0; i < nro_hilos; i++){
            thread_create(&hilos[i], &secant, i);
        }
    }
}

```

Figure 2: main()

Para la sincronizacion de threads, se hace un thread join y se espera a que todos los threads terminen. Cuando todos hayan terminado, se define al ganador, se guarda y se comienza otra vuelta

```

        for (int i = 0; i < nro_hilos; i++){
            int ganador = thread_join(hilos[i]);
            if(ganador != -1){
                ganadores[z] = ganador;
                estado.current_lap = estado.current_lap + 1;
            }
        }
        unlock(&estado);
    }
}

```

Figure 3: main()

Para la estructura de lock y unlock, se modificó un poco de la función original entregada y se agregaron variables globales para su mas expedito uso en el código. Estas variables definen la cantidad de threads y la estructura que contendrá los locks. Así, si se quiere probar con distintos valores, solamente hace falta que se cambien estas variables. Se aprecia igualmente la función matemática implementada en el código.

```

/*-----datos-----*/
#define nro_hilos 4
static thread_t hilos[nro_hilos];
Data estado;

void lock(Data *info){
    info -> used = true;
}

void unlock(Data *info){
    info -> used = false;
}

double fun(double x){
    double resultado = 5 * x*x*x*x + 2*x*x - 7*x + 1;
    return resultado;
}

```

Figure 4: Datos de la estructura

Se probó en wsl para Windows. En mi caso, siempre terminaba el primer thread antes que los demás, pero después de investigar y por lo leído en el discord, se asume que esto es normal en la plataforma. Se agrega su Makefile respectivo

## 2 Referencias

- <https://www.classes.cs.uchicago.edu/archive/2018/spring/12300-1/lab6.html>  
Se vio como es la estructura y uso de las funciones de pthreads. Se utilizó muchísimo en el código
- <https://www.tutorialspoint.com/print-numbers-in-sequence-using-thread-synchronization-in-c-program>  
Aclaró bastantes puntos.