Bloque 2. Paradigmas de la Programación

Tarea 1: Buscar ejemplos de código donde haya condiciones de carrera.

Condición de carrera: problema que surge al ejecutar un programa y cuyo resultado se ve alterado en función de qué hilo se ejecuta antes (asincronía de los hilos, no sabemos en qué orden se van a ejecutar), producido por el acceso simultáneo y modificación de elementos compartidos por parte de dichos hilos.

Ejemplificando una condición de carrera con un programa en C:

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

#define NUM_HILOS 5

void *hilo(void *pArg) {
  int *p = (int*)pArg;
  int num = *p;
  printf("Hilo número: %d\n", num);
  return 0;
}

int main(void) {
  int i;
  pthread_t tid[NUM_HILOS];

  for(i = 0; i < NUM_HILOS; i++) {
    pthread_create(&tid[i], NULL, hilo, &i);
  }

  for(i = 0; i < NUM_HILOS; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

Al ejecutar el código comprobamos que obtenemos resultados diferentes:

```
./main
Hilo número: 1
Hilo número: 4
Hilo número: 1
Hilo número: 2
Hilo número: 1
```

```
./main
Hilo número: 1
Hilo número: 3
Hilo número: 2
Hilo número: 1
Hilo número: 1
```

Explicación:

Aquí podremos observar un elemento/variable (***i***) que será leída por hilo a la vez que el valor de ***i*** será modificado por otro hilo.

Vamos a suponer un caso, en el que ***i=1*** e hilo empieza a ejecutarse. En este momento, el puntero ***p*** tendrá asignado la dirección de memoria de ***i, (&i).*** La variable ***num*** a su vez, contiene el valor de la variable cuya dirección esté asignada en el puntero ***p*** (en principio, debería corresponder a 1, con lo que resultaría en ***num=1*** y esperaríamos obtener ***Hilo número: 1***). Pero este caso no se llega a dar, ya que otro hilo ha accedido al valor de ***i*** y lo ha incrementado en 1, por lo que ahora su valor es de ***i=2.***

Para evitar las condiciones de carrera, los problemas que nos causan y resolver nuestro problema de forma determinista, tendremos que garantizar el acceso a la variable ***i*** mediante exclusión mutua, de forma que sólo un hilo pueda hacer uso de ella, ya sea mediante métodos, semáforos, cerrojos, etc.

Otro ejemplo, esta vez en Ruby:

```ruby
class DoorLock
 def initialize(locked)
   @locked = locked
 end
  def open?
   !@locked
 end

 def unlock!
   unless open?
     puts "Opening the door!"
     @locked = false
   end
 end
end

door_lock = DoorLock.new(true)
 5.times.map do
 Thread.new do
   unless door_lock.open?
     door_lock.unlock!
   end
 end
end.each(&:join)
```

Algunos resultados que podemos obtener:

`ruby main.rb`
`Opening the door!`
`Opening the door!`
`Opening the door!`
`Opening the door!`
`Opening the door!`
	`ruby main.rb`
	`Opening the door!`

Confirmamos que este código no es determinista, ya que el resultado de su ejecución depende del hilo que llegue y se ejecute antes. También destacaremos que, precisamente por esta carencia de exclusión mutua, este código no es thread-safe.

Código:

Medium. 2020. *Multithreaded Ruby — Synchronization, Race Conditions And Deadlocks*. [online] Available at: <https://medium.com/better-programming/multithreaded-ruby-synchronization-race-conditions-and-deadlocks-f1f1a7cddcea> [Accessed 29 October 2020].