# Computación Concurrente—Tarea 1
# Manipulación de hilos y procesos en C

Jorge Alejandro Ramírez Bondi

29 de octubre de 2020

## 1. Introducción

En la presente tarea, se abordarán los conceptos vistos en clase para el manejo de hilos y procesos en C y Python.

## 2. Comandos y conceptos vistos en clase

### 2.1. Procesos

#### 2.1.1. fork

Permite crear procesos hijo desde el proceso actual. La función devuelve un valor dependiendo del resultado de la instrucción:

- mayor a 0 : ha ocurrido un error al generar el proceso
- igual a 0 : proceso hijo
- menor a 0 : id de proceso del hijo para el proceso padre. Es decir, se ejecutará el proceso padre.

#### 2.1.2. getpid()

Nos permite conocer el id de proceso del proceso actual.

#### 2.1.3. wait

Pone al proceso padre en reposo hasta que el proceso hijo referenciado termine. Posteriormente, continúa la ejecución.

#### 2.1.4. pipe

Función utilizada para pasar información de un proceso a otro. Dado que es una función unidireccional, se pueden generar dos elementos de este tipo; uno para la lectura y el segundo para escritura.

## 2.2. Hilos

### 2.2.1. pthread_create

Permite crear un nuevo hilo dentro del proceso actual. Los atributos estarán definidos dentro de la función *pthread_attr_init()*.

### 2.2.2. pthread_join

Permite que el hilo o proceso que llama a la función espere a que termine el hilo puesto en el *targe*. Si la operación de espera es correcta, la función devuelve un 0; si no, devuelve $-1$ y muestra el error.

### 2.2.3. pthread_exit

Permite terminar el hilo actual al momento de llamar a esta función. Generalmente se pone dicho comando al finalizar todas las operaciones que se buscaban operar en un hilo separado.

En el caso específico de que los hilos se busquen ejecutar a pesar de que *main* ya haya terminado, se pone al final de la función principal el comando en cuestión.

# 3. Investigación

## 3.1. *Global Interpreter Lock—GIL*

El *Global Interpreter Lock—GIL*—es un candado, más precisamente un *mutex*, que permite que solamente un hilo mantenga el control y operación del intérprete de Python. Es decir, indica que solamente un hilo estará ejecutándose.

Su uso debe realizarse con cuidado, dado que puede generar un cuello de botella en la ejecución de un programa ya que se desaprovechan recursos del procesador. Sin embargo, es importante preguntarse por qué existe esta herramienta. Resulta que en Python, todos los objetos tienen un identificador numérico que contiene el número de referencias que utilizan dicho objeto.

Por lo anterior, se pueden generar condiciones en las que múltiples hilos deseen actualizar una variable de manera simultánea, lo que puede provocar errores de memoria que alteren el funcionamiento del programa, provoquen problemas de seguridad y no permitan la ejecución de instrucciones. A pesar de que se podrían utilizar múltiples candados compartidos entre los hilos, esto también generaría condiciones de bloqueo o *deadlocks* que provocarían problemas adicionales.

Así, se generó un solo candado que permite controlar las condiciones de actualización simultánea de variables. Además de dar un control seguro en Python, también ha permitido que se aumente la disponibilidad de bibliotecas escritas en C, las cuales requieren un manejo de memoria mucho más preciso.

[Aji]

### 3.1.1. Ley de Amdahl

La ley de Amdahl es una expresión que permite encontrar la mejora que habrá sobre la totalidad de un sistema cuando solamente una parte de este se mejorará. Por ello, es muy útil para calcular—en computación paralela—la mejora teórica de un programa cuando se paralelice parte de éste.

Se define mejora a la expresión que nos permite conocer la reducción en tiempo de ejecución cuando se realizan cambios en el sistema. Si definimos a $s$ como la mejora, $t_o$ como el tiempo antes del cambio y $t_c$ como el tiempo de ejecución después de los cambios: $s = \frac{t_o}{t_c}$

Entonces, siguiendo la Ley de Amdahl, definimos $f_i$ como el factor de mejora y $f_e$ como el factor de cambio, con lo que tenemos: $s = \frac{1}{(1-f_e)+(\frac{f_e}{f_i})}$

[Flo]

### 3.1.2. *multiprocessing*

Es una biblioteca de Python que permite generar procesos concurrentes tanto en sistemas operativos *UNIX-like* o Windows. La biblioteca permite generar las funciones adaptando la GIL, sin quitarle libertad al programador. Para crear un proceso, es necesario generar un nuevo objeto de la clase *Process* y llamar al método *start()*.

Dentro de los métodos de manejo de procesos más importantes de la biblioteca tenemos:

**Creación de procesos**

- fork—el proceso padre genera un nuevo intérprete de Python y el hijo hereda todo exactamente igual al padre. Solo está disponible en Unix y es el método utilizado por defecto.
- forkserver—se genera un proceso de tipo servidor para el proceso hijo. Cada vez que el proceso padre requiera algo de éste, se conectará al servidor.
- spawn—el proceso padre genera un nuevo intérprete de Python dentro del nuevo proceso. Para ello, solo se heredan los valores esenciales para el funcionamiento del nuevo proceso. En macOS y Windows son el método por defecto. Está disponible en plataformas Unix que permitan el paso de parámetros mediante *pipes*.

**Paso de parámetros entre procesos**

Se utiliza la clase de *Value* y *Array* para guardar los parámetros que deseamos compartir entre los distintos procesos. Es importante tener cuidado al utilizar esta funcionalidad dado que pueden generarse condiciones de carrera.

**Sincronización**

¿Qué sucede si deseamos sincronizar dos procesos que se están ejecutando? Lo anterior es útil cuando deseamos que ambos accedan a una misma variable u objeto. Es vital el uso de un objeto de tipo *Lock* para asegurarse que los recursos compartidos están disponibles.

El siguiente ejemplo obtenido de la documentación de Python, nos permite mostrar la funcionalidad:

```
from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    try:
        print('hello world', i)
    finally:
        l.release()

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()
```

De la misma manera, podemos listar 5 de los métodos más importantes a utilizar junto con la biblioteca *multiprocessing*:

- *start()* ← permite iniciar un proceso.
- *run()* ← método que representa la actividad de un proceso
- *join([timeout])* ← si el parámetro *timeout* es *None* se bloquea la ejecución hasta que el proceso termina. Si se pone un entero como parámetro, se bloquea la ejecución a lo más el número de segundos dado.
- $is_alive()$← *método que devuelve si el proceso sigue existiendo o no.*
- terminate() ← *termina el proceso en referencia. Es importante aclarar que no termina los procesos hijo, cambia su estado a huérfano.*
- close() ← *libera todos los elementos o variables asociados al proceso en cuestión.*

*Los parámetros entre paréntesis, son los que recibe el método.*

# 4. Implementación de *multiprocessing.process*

Para heredar de la clase *process*, es necesario implementar el código utilizando el siguiente esquema—tomado de la documentación oficial de Python 3:

```
from multiprocessing import Process

def f(name):
    print('hello', name)
```

```
if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

[Fou]

## 5. Referencias

## Referencias


[Aji] Abhinav Ajitsaria. *What Is the Python Global Interpreter Lock (GIL)?* URL: https://realpython.com/python-gil/. (accessed: 29.10.2020).

[Flo] CS University of Florida. *Amdahl's Law Tutorial.* URL: https://www.cise.ufl.edu/~mssz/CompOrg/CDA3101-S16-AmdahlsLaw-TEXTSUMMARY.pdf. (accessed: 29.10.2020).

[Fou] Python Software Foundation. *multiprocessing — Process-based parallelism.* URL: https://docs.python.org/3/library/multiprocessing.html#module-multiprocessing. (accessed: 29.10.2020).