CC: Tarea 1

Martínez Ostoa Néstor Iván

29 de octubre del 2020

1 Resumen funciones en C

1.1 *fork*()

Función que crea un nuevo proceso mediante la duplicación del proceso que lo llama. Es decir, a partir del proceso A, al llamar a $fork()$, el sistema operativo crea un nuevo proceso B copiado de A y a B se le conocerá, desde ahora, como el proceso hijo y a A como el proceso padre. Los dos procesos se ejecutarán en espacios diferentes de memoria y al hacer el $fork()$, ambos procesos tendrán el mismo contenido. Sin embargo, difieren en algunos aspectos como los siguientes:

- El proceso hijo tendrá su propio ID que será único entre todos los procesos de la computadora
- El ID del padre del proceso hijo será idéntico al ID del proceso que llamó a $fork()$
- El proceso hijo no hereda ninguno de los candados ni semáforos del proceso padre

Valor de regreso.
Esta función devuelve, de ser exitosa, el PID del proceso hijo en el proceso padre y regresa 0 en el proceso hijo. Si falla, la función regresa -1.
Headers.

```
#include <sys/types.h>
#include <unistd.h>
```

1.2 *getpid*()

Función que regresa el ID del proceso (PID) del proceso que la manda a llamar. Para usar esta función, necesitamos incluir los headers de la función $fork()$

1.3 *getppid*()

Función que regresa el PID del proceso padre. Para usar esta función, necesitamos incluir los headers de la función $fork()$.

1.4 *wait*()

Función empleada para suspender la ejecución del proceso padre hasta que la ejecución de uno de sus procesos hijos cambie de estado. Estos cambios de estado pueden ser los siguientes: el proceso hijo ha terminado, fue detenido por una señal o resume su ejecución por medio de una señal. Si el proceso hijo ha terminado, al realizar un *wait* permite al sistema operativo liberar los recursos asociados con el hijo. Si no se realiza un *wait*, el proceso hijo permanecerá en estado de *zombie*.
Headers.

```
#include <sys/types.h>
#include <sys/wait.h>
```

1.5 *perror*()

Escribe un mensaje de error hacia **stderr**.

1.6 *pthread_create*()

Esta función inicia un nuevo hilo en el proceso que lo manda a llamar y guarda el ID del nuevo hilo en un buffer llamado **thread**. El nuevo hilo empieza su ejecución al invocar a la función *start_routine*(). Este nuevo hilo se termina de alguna de las siguientes formas:

- Se manda a llamar a *pthread_exit*() especificando un status de salida
- Regresa de *start_routine*()
- Se cancela usando *pthread_cancel*()
- Mandando a llamar a *exit*()

Header

```
#include <thread.h>
```

1.7 *pthread_join*()

Esta función espera a que el hilo especificado por el buffer **thread** termine. Si el hilo ya terminó, esta función regresa inmediatamente. Esta función acepta un parámetro **void **retval** que si no es **NULL**, entonces la función copia el estado de exit (exit status) en la ubicación de **retval**. Si la ejecución es exitosa, esta función regresa un 0.

Header

```
#include <thread.h>
```

1.8 *pthread_exit*()

Termina el hilo que la manda a llamar y regresa un valor a través del parámetro **retval** que es disponible para otro hilo en el mismo proceso que llame a *pthread_join*(). Cuando un hilo es terminado, los recursos compartidos del proceso (mutexes, variables compartidas, semáforos y descriptores de archivos) no se liberan.

Header

```
#include <thread.h>
```

1.9 *pipe*()

Esta función crea un canal unidireccional de datos que se puede usar para la comunicación entre procesos. La creación de un **pipe** tiene asociado la creación de dos descriptores de archivos:

- **pipefd[0]**: usado para leer desde el pipe
- **pipedf[1]**: usado para escribir en el pipe

La información escrita en el pipe se va a un buffer y se queda ahí hasta ser leída por el canal de lectura del pipe.

Headers

```
#include <unistd.h>
```

1.10 *write*()

La ocupamos para escribir sobre un descriptor de archivo el mensaje que queramos enviar hacía otro proceso.

Headers

```
#include <unistd.h>
```

1.11 *close*()

Se emplea para cerrar el descriptor de archivo para ya no haga referencia a ningún arhchivo y así evitar las lecturas sobre el pipe.

Headers

```
#include <unistd.h>
```

2 Global Interpreter Lock (GIL)

Es un mutex que protege el acceso a los objetos de Python y previene que múltiples hilos ejecuten una porción de código al miso tiempo. Un *mutex* es un candado exclusivo y es necesario porque la administración de memoria no es segura a nivel de hilo. El GIL permite que solo un hilo controle el intérprete de Python por lo que solo permite que se ejecute un hilo a la vez incluso en un programa que uso múltiples hilos. Este GIl surge como respuesta al contador de referencias que usa Python y necesita de ser protegido de fenómenos como condiciones de carrera.

El contador de referencias es un contador que indica la cantidad de referencias hacía un mismo objeto y deben de ser protegidas de condiciones de carrera o posibles problemas por la concurrencia.

3 Ley de Amdahal

Esta ley establece que el mejoramiento en el desempeño de un sistema compuesto por varios componentes está limitado a la fracción de tiempo que ocupa un componente dado. Es decir, si tenemos un sistema S con n componentes y cada uno de ellos ocupa una fracción de tiempo diferente para realizar la tarea del sistema S; si quisiéramos mejorar S, entiéndase por mejorar disminuir el tiempo que le toma realizar la tarea, lo que tendríamos que hacer es optimizar lo más que se pueda la componente s_i que tome la mayor cantidad de tiempo, porque, tan solo una optimización en s_i, reducirá mucho más el tiempo de procesamiento de S que si optimizáramos una componente s_j que tomara mucho menos en realizarse. En la siguiente figura es ejemplifica lo anterior:

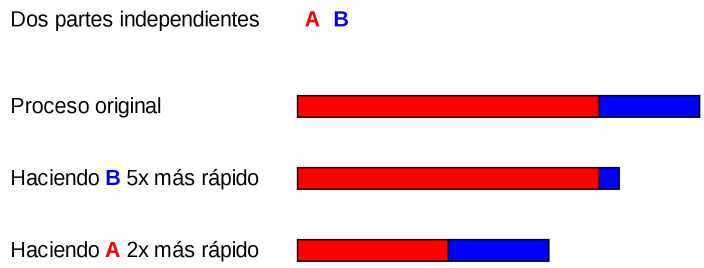

Figure 1: Optimización de diferentes tareas de un sistema

Esta ley tiene la siguiente ecuación:

$$T_m = T_a((1 - F_m) + \frac{F_m}{A_m}) \tag{1}$$

Donde:

- F_m: fracción de tiempo que el sistema utiliza el subsistema mejorado
- A_m: factor de mejora que se ha introducido en el subsistema mejorado
- T_a: tiempo de ejecución anterior
- T_m: tiempo de ejecución mejorado

Finalmente, esta ley la podemos aplicar a la cantidad de procesadores dentro de un CPU de una computadora y veremos que un CPU con menos procesadores y un porcentaje más elevado de paralelización es mucho mejor que un CPU sin paralelización y muchos procesadores. Lo anterior se refleja en la siguiente figura:

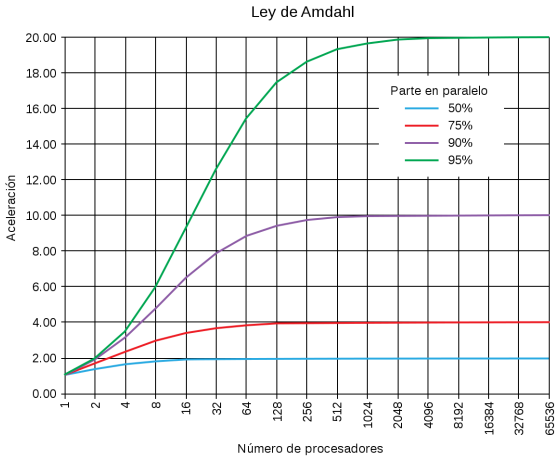

Figure 2: Ley de Amdahl en el número de procesadores con % de paralelización vs. la aceleración

4 *multiprocessing*

- **Descripción y uso.** Es una librería que soporta la generación de procesos y ofrece soporte para concurrencia local como remota y evita el Global Interpreter Lock usando subprocesos en lugar de hilos. Dentro de esta librería podemos controlar la mayor cantidad de operaciones de concurrencia, como creación, sincronización y comunicación entre procesos.
- **Métodos/funciones más importantes.**
  - **Creación de procesos.** La creación de procesos es crucial a la hora de trabajar la concurrencia y para *multiprocessing* tenemos tres métodos para crear un proceso:
    - *spawn*: el proceso padre empezará un nuevo proceso de intérprete de Python y el proceso hijo solo heredará los recursos necesarios para ejecutar el método *run()*.
    - *fork*: el proceso padre utiliza este método para hacer un fork del intérprete de Python. El proceso hijo será ideńtico al proceso padre y todos los recursos del padre serán heredados al proceso hijo.

    * *forkserver*: este método inicia un proceso servidor lo que permite que cuando se necesite un nuevo proceso, el proceso padre se conecta al servidor y pide por un fork de un nuevo proceso.

  - **Sincronización de procesos.** Para esta funcionalidad, necesitamos utilizar un candado como base para asegurar que solo un proceso se ejecutará en un tiempo dado. Necesitamos usar candados para evitar que la ejecución de los procesos se intercale.

```
from multiprocessing import Process, Lock
def f(lock, i):
    lock.acquire()
    try:
        print('Hello world', i)
    finally:
        lock.release()
if __name__ == '__main__':
    lock = Lock()
    for n in range(10):
        Process(target=f, args=(lock, n)).start()
```

  - **Compartiendo el estado entre procesos.** Recordemos que al hablar de concurrencia, necesitamos evitar el mayor tiempo posible que los hilos que creemos no compartan recursos porque nos puede llevar a condiciones de carrera. Sin embargo, para las ocasiones en las que queramos compartir recursos entre procesos o hilos, podemos emplear la clase *Value* o *Array*.

```
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print(num.value)
    print(arr[:])
```

- **Procesos servidores.** Un objeto *Manager()* controla un proceso servidor que contiene objetos dePython y permite que otros procesos los manipulen usando proxies.
- **Usando un pool de trabajadores.** La clase *Pool* representa un conjunto de procesos trabajadores que nos permiten que un proceso jefe delege tareas a uno o más procesos trabajadores en diferentes maneras.
- **5 métodos con argumentos**
  - *run()*: método que representa la actividad de un proceso.
  - *start()*: método que inicia la actividad de un proceso.
  - *join(timeout=None)*: método que bloquea la ejecución del proceso hasta que el método llamado por *join()* termine. *timeout* es un parámetro que nos indica los segundos que debemos esperar a que un proceso termine, el valor por defecto de *timeout* es *None*.

- *is_alive()*: método que regresa un *Bool* indicando si un proceso está vivo.
- *terminate()*: método que termina la ejecución del proceso que la manda a llamar.

5 Creación de procesos

En el siguiente código veremos la manera de crear procesos como un objeto que hereda de la clase **multiprocessing.Process**:

```
from multiprocessing import Process

def f1(name):
    print("Hello, my name is", name)

if __name__ == '__main__':
    p1 = Process(target=f1, args=(('P1',)))
    p1.start()
    p1.join()
```

6 Referencias

Ley de Amdahl

GIL

Módulo multiprocessing en Python