CC-Tarea1

Aguilar Rodríguez José Antonio

October 29, 2020

1 Funciones utilizadas en C

Dividiremos las funciones en dos secciones, la primera en donde se encuentren las utilizadas para crear, comunicar y manipular procesos y posteriormente las que fueron utilziadas para crear, comunicar y manipular hilos.

1.1 Procesos

1. **fork()**. Es la función que es utilizada para la creación de un nuevo proceso hijo.

2. **wait()**. Es la función que indica que se debe esperar a la terminación de un proceso para comenzar otro. Devuelve el pid del proceso por el cual se esperó. Además, como argumento puede recibir la dirección en memoria de alguna variable que se empleó en el proceso por el cual se esperó para que sea utilizada por el proceso que esperó.

3. **pipe()**. El propósito de esta función es crear un canal de comunicación entre dos procesos. Recibe como argumento un arrelgo con dos entradas, una de estas entradas será usada como canal de entrada (escritura) y la otra como canal de salida (lectura). Tiene asociadas las siguientes tres funciones.

4. **close()**. Cierra el pipe en algún canal, recibe como argumento la posición del arreglo que fue utlizado para crear el pipe.

5. **write()**. Sirve para escribir en el canal de entrada, recibe la posición del arreglo destinada para la entrada, el dato que se escribirá y el tamaño de dicho dato

6. **read()**. Funciona para leer el canal de salida, recibe la posición del arreglo destinada para la salida, la variable en donde se guardará el dato que se leyó y el tamaño de la variable.

1.2 Hilos

1. **pthread_create()**. Esta función es utilizada para la creación de un hilo. Recibe como argumentos un objeto tipo pthread_t, el método que llevará a cabo el hilo y el argumento que utilizará para dicho método.

2. **pthread_join()**. Es el análogo a la función **wait()** en los procesos, espera la terminación de un hilo. Recibe como argumento el hilo por el cual se desea esperar.

3. **pthread_exit()**. Termina la llamada del hilo.

2 Definición de conceptos

2.1 Global Interpreter Lock (GIL)

Para empezar es un mutex, un algoritmo de exlclusión mutua que se usa en programación concurrente para evitar que entre más de un proceso a la vez en la sección crítica (fragmento de código donde puede modificarse un recurso compartido). Este mutex en Python proteje el acceso a objetos de Python e impide que diferentes hilos ejecuten bytecodes de Python. El GIL es necesario debido al manejo de memoria de Python. Es un tema muy controversial ya que impide que programas de **CPython** con hilos exploten por completo a los sistemas multiprocesador. Es básicamente un cuello de botella para los programas con hilos.

- **CPython** es un intérprete de bytecode. Tiene una interfaz de funciones foráneas para varios lenguajes (incluyendo C, C++ y Fortran) con el que se pueden codificar bindings para bibliotecas escritas en lenguajes diferentes a Python.

2.2 Ley de Amdahl

La definición de esta ley establece que:

"La mejora obtenida en el rendimiento de un sistema debido a la alternación de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente"

$$T_m = T_a \left((1 - F_m) + \frac{F_m}{A_m} \right)$$
$$\implies A = \frac{1}{(1 - F_m) + \frac{F_m}{A_m}}$$

Donde:
- F_m es la francción de tiempo que el sistema utiliza el subsistema mejorado.
- A_m es el factor de mejora que se ha introducido en el sistema.

- T_a es el tiempo de ejecución antiguo.
- T_m es el tiempo de ejecución mejorado.
- A es la ganancia de velocidad. $\left(A = \frac{T_a}{T_m}\right)$

2.3 Multiprocessing

El paquete multiprocessing ofrece concurrencia tanto local como remota, esquivando el término Global Interpreter Lock mediante el uso de subprocesos en lugar de hilos. Debido a esto, el módulo multiprocessing le permite al programador aprovechar al máximo múltiples procesadores en una máquina determinada. A continuación enlistaremos algunas funciones importantes del módulo:

1. **start()** Comienza la actividad del proceso.
2. **join()** Si el argumento opcional *timeout* es None (el valor predeterminado), el método se bloquea hasta que el proceso cuyo método **join()** se llama termina. Si timeout es un número positivo, bloquea como máximo *timeout* segundos.
3. **Pipe()** Retorna un par de objetos *(conn1, conn2)* de la Connection que representan los extremos de una tubería (pipe). Si *duplex* es True (el valor predeterminado), entonces la tubería (pipe) es bidireccional. Si duplex es False, entonces la tubería es unidireccional: *conn1* solo se puede usar para recibir mensajes y *conn2* solo se puede usar para enviar mensajes.
4. **send()** Envía un objeto al otro extremo de la conexión que debe leerse usando **recv()**.
5. **recv()** Retorna un objeto enviado desde el otro extremo de la conexión usando **send()**. Se bloquea hasta que haya algo para recibir. Se lanza EOFError si no queda nada por recibir y el otro extremo está cerrado.

3 Crear procesos como un objeto que hereda de la clase *multiprocessing.process*

La forma de crear un proceso como un objeto de la clase *multiprocessing.process* es mediante el siguiente comando:

multiprocessing.Process(group=None, target=None, name=None, args=(), kwargs=, *, daemon=None)

El constructor siempre debe llamarse con argumentos de palabras clave. **group** siempre debe ser None; existe únicamente por compatibilidad con ***threading.Thread***. **target** es el objeto invocable a ser llamado por el método **run()** . El valor predeterminado es None, lo que significa que nada es llamado. **name**

es el nombre del proceso. **args** es la tupla de argumento para la invocación de destino. **kwargs** es un diccionario de argumentos de palabras clave para la invocación de destino. Si se proporciona, el argumento **daemon** solo de palabra clave establece el proceso daemon en True o False. Si None (el valor predeterminado), este indicador se heredará del proceso de creación.

Un breve ejemplo de su aplicación es como sigue:

```
from multiprocessing import Process
def f(name):
print('hello', name)
if __name__ == '__main__':
p = Process(target=f, args=('bob',))
p.start()
p.join()
```

4 Referencias

- *https://wiki.python.org/moin/GlobalInterpreterLock*
- *https://man7.org/*
- *https://es.linkfang.org/wiki/CPython*
- *http://ict.udlap.mx/people/oleg/docencia/PARALELO/sec/LEY%20DE%20AMDAHL.doc*
- *https://hardzone.es/reportajes/que-es/ley-de-amdahl*
- *https://python-docs-es.readthedocs.io/es/3.8/library/multiprocessing.html*