CC-Tarea1

Benito Vicente Franco López

October 2020

1 Concurrencia en C

1.1 Funciones y tipos:

1.1.1 fork

Cuando se llama la función fork , esta genera un duplicado del proceso actual. El duplicado comparte los valores actuales de todas las variables, ficheros y otras estructuras de datos. La llamada a fork retorna al proceso padre el identificador del proceso hijo y retorna un cero al proceso hijo.

1.1.2 exit

En programación C disponemos de la función exit que permite finalizar de forma controlada un programa. La forma de uso habitual será: exit (-1); Donde -1 es un valor devuelto por la función al ambiente de ejecución (y podrá ser éste u otro; normalmente se usa -1 para indicar una detención del programa por un problema detectado).

1.1.3 perror

sintaxis:
int perror(char *cadena);
La función perror transforma el número de error en la expresión entera de errno a un mensaje de error. Escribe una secuencia de caracteres al stream estándar de errores

1.1.4 wait

sintaxis:
$\text{pid}_t wait(int * status)$;

La función wait suspende la ejecución del proceso actual haste que un proceso hijo haya terminado, o hasta que se produce una señal cuya acción es terminar el proceso actual o llamar a la función manejadora de la señal. Si un hijo ha salido cuando se produce la llamada (lo que se entiende por proceso "zombie"),

la función vuelve inmediatamente. Todos los recursos del sistema reservados por el hijo son liberados

1.1.5 waitpid

sintaxis:
$\text{pid}_t waitpid(pid_t pid, int * status, int options)$;

La función waitpid suspende la ejecución del proceso en curso hasta que un hijo especificado por el argumento pid ha terminado, o hasta que se produce una señal cuya acción es finalizar el proceso actual o llamar a la función manejadora de la señal.

1.1.6 getpid

Es una llamada para el control de procesos que retorna el pid del proceso que la invoco. Esta función no recibe ningún argumento y retorna un entero del tipo pid_t

1.1.7 getppid

getppid devuelve el identificador de proceso del padre del proceso actual.

1.1.8 pid-t

el tipo pid-t que no es más que un identificador de proceso linux PID.

1.1.9 pthread-t

pthread-t es el tipo de datos que se utiliza para identificar de forma exclusiva un hilo. Es devuelto por pthread-create () y usado por la aplicación en llamadas a funciones que requieren un identificador de hilo.

1.1.10 pthread-create

La función que nos permite crear un un nuevo hilo de ejcución es pthread-create() que admite cuatro parámetros:

pthread-t * es un puntero a un identificador de thread. La función nos devolverá este valor relleno, de forma que luego podamos referenciar al hilo para "hacerle cosas", como matarlo, esperar por él, etc.

1.1.11 pthread-join

La función pthread-join sirve para esperar a otro hilo. Recibe un identificador de hilo (parámetro thread) al que debemos esperar. El segundo parámetro es el valor de terminación del hilo.

1.1.12 pthread-exit

Es la función que nos permite terminar con el hilo que habiamos creado previamente

2 Multiprocesamiento Python

2.1 Global Interpreter Lock o GIL

El mecanismo que impide a la implementación en C de Python (a la que nos referiremos siempre como CPython a partir de ahora) la ejecución de bytecode por varios hilos a la vez se llama Global Interpreter Lock o GIL es un mecanismo utilizado en CPython para impedir que múltiples threads modifiquen los objetos de Python a la vez en una aplicación multihilo. El GIL es un bloqueo a nivel de intérprete. Este bloqueo previene la ejecución de múltiples hilos a la vez en un mismo intérprete de Python. Cada hilo debe esperar a que el GIL sea liberado por otro hilo.

https://www.genbeta.com/desarrollo/multiprocesamiento-en-python-global-interpreter-lock-gil

2.2 Ley de Amdahal

La ley de Amdahl es utilizada para averiguar la mejora máxima de un sistema de información cuando solo una parte de éste es mejorado.

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».

La fórmula original de la ley de Amdahl es la siguiente:

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

Figure 1: ley amdahal

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.

2.2.1 En palabras coloquiales:

Esta ley lo que nos dice es que la mejora de rendimiento de un sistema (entendiéndose por sistema un conjunto de piezas como puede ser un PC) cuando cambias una única pieza, está limitada por el tiempo que se utilice dicho componente.

Dicho de otra manera con un ejemplo: la mejora de rendimiento de tu PC cuando le cambias la memoria RAM está limitada por el tiempo que vayas a utilizar dicho componente Nos permite: Ver gráficamente si el mejorar el

rendimiento de uno u otro componente merecerá o no la pena, y hasta qué punto.

2.3 Multiprocessing

El multiprocessingmódulo permite al programador aprovechar completamente varios procesadores en una máquina determinada.

Ofrece simultaneidad local y remota.

El módulo de multiprocesamiento evita las limitaciones de Global Interpreter Lock (GIL) mediante el uso de subprocesos en lugar de threads. El código multiprocesado no se ejecuta en el mismo orden que el código de serie. No hay garantía de que el primer proceso que se cree sea el primero en completarse. Python tiene tres módulos para la concurrencia: multiprocessing, threading, y asyncio. Cuando las tareas son intensivas en CPU, debemos considerar el multiprocessing módulo.

2.3.1 Clase Process

Hay dos funciones importantes que pertenecen a la clase Process start() y la join() función.
1.-start():
Nos permite iniciar un proceso no recibe argumentos
2.-join():
Nos permite terminar un proceso no recibe argumentos

3.-Process nos permite crear un objeto de la clase procesos, usa la sentencia:
proc = Process(target=print-func, args=(name,))
recibe como argumentos una función en target y los argumentos de esa función como una tupla en args

2.3.2 Clase lock

La tarea de la clase Lock es bastante simple. Permite que el código reclame el bloqueo para que ningún otro proceso pueda ejecutar el código similar hasta que se libere el bloqueo. Entonces, la tarea de la clase Lock es principalmente dos. Una es reclamar la cerradura y la otra es liberarla. Para reclamar el bloqueo, acquire() se usa la función y para liberar el bloqueo release().
4.-acquire():
Reclama el bloqueo en un proceso, no recibe argumentos
5.-release():
Libera el bloqueo en un proceso, no recibe argumentos

2.3.3 Colas

Los objetos de cola son una estructura de datos FIFO que son seguros para procesos y subprocesos, lo que los hace perfectos para pasar datos entre diferentes procesos sin dañarlos potencialmente. Usarlos es relativamente simple, podemos expandir.
6.-Queue():
Nos permite crear un objeto tipo cola, no recibe argumentos
7.-put():
Es una función con la cual podemos insertar datos para luego ponerlos en cola.
8.-get():
Nos permite obtener elementos de las colas

3 Maneras de crear procesos como un objeto que hereda de la clase multiprocessing.process

multiprocessing es un paquete que permite crear nuevos procesos utilizando un API similar a la del módulo threading. Debido a que utiliza subprocesos en lugar de hilos (threads), permite llevar a cabo varias operaciones concurrentes sin las limitaciones del Global Interpreter Lock. Corre en sistemas Unix y Windows.

3.1 ¿Como se crea un proceso?

En la clase multiprocessing se pasa una función para ser ejecutada en otro hilo (target) junto con sus argumentos (args). Luego, se llama al método start() para iniciar la ejecución y join() para esperar a que finalice
Basicamente hay dos formas de crearlo:
Como lo vimos en clase y como un un objeto que heredado de la clase multiprocessing:

```
from multiprocessing import Process

def say_hello(name):
    print("Hello, %s!" % name)

if __name__ == '__main__':
    p = Process(target=say_hello, args=("world",))
    p.start()
    p.join()
```

```
from multiprocessing import Process

class ConcurrentProcess(Process):

    def say_hello(self, name):
        print("Hello, %s!" % name)

    def run(self):
        self.say_hello("world")

if __name__ == "__main__":
    p = ConcurrentProcess()
    p.start()
    p.join()
```