Computación Concurrente - Tarea1

Andrés Urbao Guillermo Gerardo

23 de Septiembre del 2021

Documentación de las funciones básicas de POSIX para crear procesos y threads en lenguaje C

<**sys/types.h**> Encabezado que contiene definiciones de varios tipos, uno de ellos será `pid_t`.

<**unistd.h**>.Es el archivo de cabecera que proporciona acceso a la API del sistema operativo POSIX.

pid_t. Se utiliza para ID de proceso e ID de grupo de proceso.

fork(). Crea un nuevo proceso duplicando el proceso que lo llamo. El nuevo proceso se denomina proceso hijo. El proceso de llamada se conoce como proceso padre. El proceso hijo y el proceso padre se ejecutan en espacios de memoria separados. En el momento de 'fork()' ambos espacios de memoria tienen el mismo contenido. En caso de éxito, el PID del proceso hijo se devuelve en el padre y se devuelve 0 en el hijo. En caso de error, se devuelve -1 en el padre, no se crea ningún proceso hijo.

exit(estado). La función de biblioteca exit finaliza un proceso, haciendo que todos los recursos (memoria, descriptores de archivos abiertos, etc.) utilizados por el proceso disponible para posterior reasignación por parte del kernel. El argumento de estado es un número entero que determina el estado de terminación del proceso. Usando la llamada al sistema wait(), el padre puede recuperar este estado.

wait(estado). La llamada al sistema wait(estado) tiene dos propósitos. Primero, si un hijo de este proceso aún no ha terminado llamando a exit(), entonces wait() suspende la ejecución del proceso hasta que uno de sus hijos haya terminado. En segundo lugar, el estado de terminación del hijo se devuelve en el argumento de estado de wait().

getpid(). Devuelve el identificador de proceso del proceso actual

getppid(). Devuelve el identificador de proceso del padre del proceso actual.

getuid(). Devuelve el identificador de usuario actual.

pipe(pipefd). Crea una tubería, un canal de datos unidireccional que se puede utilizar para la comunicación entre procesos. La matriz pipefd se utiliza para devolver dos descriptores de archivo que se refieren a los extremos de la tubería.

- pipefd[0] se refiere al extremo leído de la tubería.
- pipefd[1] se refiere al extremo de escritura de la tubería. Los datos escritos en el extremo de escritura de la tubería son almacenados en búfer por el kernel hasta que se leen desde el extremo de lectura de la tubería.

write(fd, *buf, count) . La llamada al sistema write hace que los primeros bytes del buffer sean escritos en el archivo asociado con el descriptor de archivos `fd`.

read(fd, *buf, count). Lee de un descriptor de archivo indicando el numero de bytes y un buffer donde se guardara la lectura.

close(). Cierra un descriptor de archivo, de modo que ya no hace referencia a ningún archivo y se puede reutilizar.

gets(s). Lee una línea desde la entrada estadandar en el búfer al que apunta `s` hasta que termina una nueva línea o EOF.

fflush() Es una función para la limpieza de buffer de escritura, escritura de ficheros o escritura en la pantalla.

usleep() Suspender la ejecución para intervalos de microsegundos.

<**pthread.h**>.Es el archivo de cabecera que proporciona tipos, contantes y funciones para la manipulacion de `threads`.

pthread_create(*thread) Crea un nuevo hilo pasando como argumento un apuntador `thread` donde se guardará.

pthread_join(thread). La función espera a que termine el hilo especificado por `thread`. Si ese hilo ya ha terminado, entonces `pthread_join()` regresa inmediatamente. El hilo especificado por `thread` debe poder unirse.

Global Interpreter Lock

El GIL es un mutex que protege el acceso a los objetos de Python, evitando que múltiples hilos ejecuten código de Python a la vez. Este bloqueo es necesario principalmente porque la gestión de memoria de CPython no es segura para multithreading. Esta limitacion no aplica a Jython y otras distribuciones de python. CPython utiliza el recuento de referencias para implementar su gestión de memoria. Esto da como resultado el hecho de que varios hilos pueden acceder y ejecutar código de Python simultáneamente; esta situación es indeseable, ya que puede provocar un manejo incorrecto de los datos, y decimos que este tipo de manejo de memoria no es seguro para hilos.

Para abordar este problema, el GIL es un candado que permite que solo un hilo acceda al código y a los objetos de Python. Sin embargo, esto también significa que, para implementar programas multithreading, los desarrolladores deben conocer el GIL y solucionarlo. Es por eso que muchos tienen problemas al implementar sistemas concurrentes en Python.
El GIL es bastante popular en la comunidad de programación concurrente de Python. Diseñado como un bloqueo que solo permitirá que un hilo acceda y controle el intérprete de Python en un momento dado, el GIL en Python a menudo se conoce como el infame GIL que evita que los programas multithreading alcancen su velocidad totalmente optimizada.

A pesar de que GIL evita que los programas de multihilos aprovechen al máximo los sistemas de multiprocesador en determinadas situaciones, la mayoría de las operaciones de bloqueo o de ejecución prolongada, como E / S, procesamiento de imágenes y procesamiento de números NumPy, ocurren fuera del GIL. Por lo tanto, el GIL solo se convierte en un posible cuello de botella para los programas multihilos que pasan mucho tiempo dentro del GIL.

El multithreading es solo una forma de programación concurrente y, si bien el GIL plantea algunos desafíos para los programas multithreading que permiten que más de un hilo acceda a recursos compartidos, otras formas de programación concurrente no tienen este problema . Por ejemplo, las aplicaciones de multiprocessing que no comparten ningún recurso común entre procesos, como E / S, procesamiento de imágenes o procesamiento de números NumPy, pueden funcionar sin problemas con GIL.

Ley de Amdahal

La Ley de Amdahl explica la aceleración teórica de la ejecución de un programa que se puede esperar cuando se usa la concurrencia.

La Ley de Amdahl proporciona una fórmula matemática que calcula la mejora potencial en la velocidad de un programa concurrente al aumentar sus recursos (específicamente, el número de procesadores disponibles).

$$S = \frac{T(1)}{T(j}$$

En la fórmula anterior, $T(j)$ es el tiempo que se tarda en ejecutar el programa cuando se utilizan j procesadores.

- La Ley de Amdahl analiza únicamente la posible aceleración de la latencia resultante de ejecutar una tarea en paralelo.
- El speed de un programa denota el tiempo que tarda el programa en ejecutarse por completo. Esto se puede medir en cualquier incremento de tiempo.
- Speedup es el tiempo que mide el beneficio de ejecutar un cálculo en paralelo. Se define como el tiempo que tarda un programa en ejecutarse en serie (con un procesador), dividido por el tiempo que tarda en ejecutarse en paralelo (con varios procesadores).

Multiprocessing

El multiprocessing es el método de utilizar más de una CPU para ejecutar una tarea.

Si bien hay una serie de usos diferentes del término multiprocessing, en el contexto de la concurrencia y el paralelismo, el multiprocessing se refiere a la ejecución de múltiples procesos concurrentes en un sistema operativo, en el que cada proceso se ejecuta en una CPU separada, a diferencia de un solo proceso que se ejecuta en un momento dado. Por la naturaleza de los procesos, un sistema operativo debe tener dos o más CPU para poder implementar tareas de multiprocessing, ya que debe admitir muchos procesos al mismo tiempo y asignar tareas entre ellos de manera adecuada.

El modulo multiprocessing es uno de los módulos usualmente comunes para la implementación de de programas multiprocesos en Python. Cada objeto `Process` representa una actividad que se ejecuta en un proceso separado.

Podemos pasar los parámetroos `target()` (para la función objetivo) y `args` (para los argumentos de la función objetivo). Tambien podemos sobrescribir el constructor Process() usando la herencia e implementar nuestra propia función `run()`.

La clase Process de multiprocessing proporciona los siguientes métodos y atributos:

- **run()**: este método es ejecutado cuando un nuevo proceso es inicializado y empezado.
- **start()**: este método empieza la inicialización y llama la método `run()`
- **join()**: este método espera a que el objeto `Process` termine la ejecución.
- **isAlive()**: este método devuelve un valor booleano que indica si el el objeto `Process` se está ejecutando actualmente
- **terminar()**: este método termina el objeto `Process` que lo llama.
- **name**: este atributo contiene el nombre del proceso.
- **pid**: este atributo contiene el ID del proceso.

Creación de procesos hererando multiprocessing.Process

Para nuestro ejemplo crearemos 5 procesos que impriman su nombre y su correspondiente ID.

```python
import multiprocessing
import os

class MiProceso(multiprocessing.Process):
    """Creación de un objeto Process personalizado"""

    def __init__(self, name):
        """Inicialización de los atributos del proceso

        Arguments:
        name -- una cadena que representa el nombre del proceso
        """
        Process.__init__(self)
        self.name = name

    def saludar(self):
        """Manda un saludo e imprime el ID del proceso"""
        print("Hola, soy {} y tengo el ID {}".format(self.name, os.getpid()))

    def run(self):
        """Metodo que indicará que tarea hará el proceso"""
        self.saludar()
```

Cuando es llamado el método `start()`, internamente llamará al método `run()` de nuestro objeto `MiProceso` iniciando así la tarea de `saludar()`, una cosa a resaltar es que cuando inicialicemos atributos personalizados de nuestra clase, primero debemos de llamar al inicializador del padre, en este caso la clase `Process` para inicializar los atributos que heredamos y luego inicializar ahora los nuestros.

```python
def main():
    nombres = ["Juan", "Maria", "Pablo", "Jose", "Pedro"]
    procesos = []
    # Arrancamos nuestros procesos
    for i, nombre in enumerate(nombres):
        mi_proceso = MiProceso(nombre)
        procesos.append(mi_proceso)
        procesos[i].start()

    # Esperamos a que termine cada proceso
    for proceso in procesos:
        proceso.join()

if __name__ == "__main__":
    print("Soy el proceso padre con ID {}".format(os.getpid()))
```

```
16      main()
17      print("Fin del proceso padre.")
```

Salida:

```
Soy el proceso padre con ID 22287
Hola, soy Juan y tengo el ID 26053Hola, soy Maria y tengo el ID 26056

Hola, soy Pedro y tengo el ID 26065Hola, soy Pablo y tengo el ID 26061
Hola, soy Jose y tengo el ID 26064


Fin del proceso padre.
```

Referencias

[1] Kerrisk, Michael: *The Linux Programming Interface, No Starch Press.*

[2] Nguyen, Quan: *Mastering concurrency in python, Packt.*

[3] RecursosPython: *Multiprocessing.* https://recursospython.com/guias-y-manuales/multiprocessing-tareas-concurrentes-con-procesos/. Accedido en Septiembre 23 del 2021.

[4] Man7: *Linux.* https://man7.org/linux/man-pages/man2/read.2.html. Accedido en Septiembre 23 del 2021.