Universidad Nacional Autónoma de México

Instituto de Investigaciones en Matemáticas Aplicadas y Sistemas

Tarea 1

Computación Concurrente

Autor:
Yañez Espindola Jose Marcos

29 Octubre 2020

Índice

1. Funciones en C para procesos e hilos

1.1. Funciones para procesos

fork()

Esta función crea un proceso hijo, el cual correrá concurrentemente con el proceso padre, las instrucciones posteriores al fork().
No recibe parámetros, y regresa:
- Un valor negativo en caso de que la creación del proceso hijo fallo.
- Un 0 para el proceso hijo
- Un valor positivo (El ID del proceso hijo) para el proceso padre

wait()

Pausa la ejecución del proceso padre, hasta que alguno de sus procesos hijos termine. Una vez que el proceso hijo termina su ejecución, el proceso padre continua con la ejecución de las instrucciones posteriores al wait().
Recibe como parámetro un apuntador de tipo entero, el cual usara para almacenar el estado de salida de un proceso hijo.
Regresa el identificador del proceso hijo cuya ejecución termino, si el proceso desde el cual es llamado el wait() no tiene ningún proceso hijo, regresa un -1 inmediatamente.

exit()

Termina inmediatamente el proceso que la llama.
Recibe un entero correspondiente al estado con el que se termina ese proceso, normalmente 0, para indicar que finalizo con éxito.

pipe()

Es usada para transmitir información de un proceso a otro, este es un canal unidireccional, sin embargo, se pueden configurar dos pipes, una para cada dirección.
Recibe un apuntador a la localidad de memoria que será usada como canal.
Regresa un 0 si se creó exitosamente, o un -1 en caso contrario.

1.2. Funciones para hilos

pthread_create()

Inicia un nuevo hilo en el proceso que la llama. El nuevo hilo inicia su ejecución, invocando a la start_routine().
Recibe como parámetros la localidad de memoria donde se almacenara el hilo, la start_routine(), que es una función, con el código que va a ejecutar el hilo y

los argumentos que serán pasados a esta funcion.
Regresa un 0 si la creación del hilo se hizo exitosamente, o un error en caso contrario.

pthread_join()

Espera a que la ejecución de un hilo en específico, que es recibido como parámetro, termine.
Recibe como parámetro el hilo que se desea esperar, y un apuntador doble el cual se usara para escribir el valor recibido vía retval que envía pthread_exit().
Regresa un 0 en caso de éxito, o un -1 en caso contrario.

pthread_exit()

Termina inmediatamente la ejecución del hilo que llama esta función y regresa un valor vía retval, el cual estará disponible para cualquier otro hilo que llame a la funcion pthread_join().
Recibe como parámetro, el valor que será regresado al terminar la ejecución del hilo.

2. Investigación de conceptos

2.1. Global Interpreter Lock (GIL)

Python Global Interpreter Lock (GIL) es un tipo de bloqueo de proceso que Python utiliza siempre que se ocupa de procesos. Generalmente, Python solo usa un hilo para ejecutar el conjunto de declaraciones escritas. Esto significa que en Python solo se ejecutará un hilo a la vez. El rendimiento del proceso de un solo subproceso y el proceso de múltiples subprocesos será el mismo en Python y esto se debe a GIL en Python. No podemos lograr subprocesos múltiples en Python porque tenemos un bloqueo de intérprete global que restringe los subprocesos y funciona como un solo subproceso.
Python tiene algo que ningún otro lenguaje tiene que es un contador de referencias. Con la ayuda del contador de referencias, podemos contar el número total de referencias que se hacen internamente en Python para asignar un valor a un objeto de datos. Debido a este contador, podemos contar las referencias y cuando este conteo llegue a cero, la variable u objeto de datos se liberará automáticamente. Por ejemplo:

```
# Python program showing
# use of reference counter
import sys

geek_var = "Geek"
print(sys.getrefcount(geek_var))
```

```
string_gfg = geek_var
print(sys.getrefcount(string_gfg))
```

Esta variable de contador de referencia necesitaba ser protegida, porque a veces dos subprocesos aumentan o disminuyen su valor simultáneamente al hacerlo, puede provocar una fuga de memoria, por lo que para proteger el subproceso agregamos bloqueos a todas las estructuras de datos que se comparten entre subprocesos, pero a veces agregando bloqueos existe un bloqueo múltiple que conduce a otro problema que es el interbloqueo. Para evitar la pérdida de memoria y el problema de interbloqueos, utilizamos un solo bloqueo en el intérprete que es Global Interpreter Lock (GIL).

2.2. Ley de Amdahal

Permite obtener una medida de como influye la mejora de un componente o varios (los cuales se utilizan en un porcentaje del tiempo de ejecución) en la mejora global del rendimiento de una computadora. Partiendo de que, si mejoramos x veces un componente, el componente será x veces mas rápido, esto quiere decir que hará el mismo trabajo en x veces menos tiempo. Si este componente se utiliza durante una fracción del tiempo total de la ejecución, entonces, ¿Cuál será el tiempo de ejecución después de hacer la mejora?

La ley de Amdahal establece que:

$$\text{Tiempo de ejecucion con mejora} = \frac{\text{Tiempo de ejecucion sin mejora} * F}{x + \text{Tiempo de ejecucion sin mejora} * (1 - F)}$$

Donde $1 - F$ representa la fraccion de tiempo en la cual no hay mejora.
Si calculamos el speedup, el cual se define como:

$$S = \frac{\text{Tiempo de ejecucion sin mejora}}{\text{Tiempo de ejecucion con mejora}}$$

Obtenemos que:

$$S = \frac{x}{F + x(1 - F)}$$

2.3. Multiprocessing

Es un modulo de python que permite la generacion y manipulacion de procesos. Multiprocessing ofrece una manera efectiva de hacer concurrencia local y remota, evitando el GIL usando subprocesos en lugar de threads. Debido a esto, ofrece la libertad de usar los multiples procesadores de una computadora.
La clase mas importante de este modulo es la clase Process, de la cual los metodos mas importantes son los siguientes:

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

Es el constructor de la clase, y recibe como parametros: - group: el cual siempre debe ser None.
- target: es la funcion o metodo que sera llamada por el metodo run().
- name: es el nombre del proceso.
- args: recibe los argumentos con los que sera invocada la funcion en target.
- kwargs: es un diccionario para los argumentos.
- daemon: es una bandera que cuando es True, establece el proceso como un proceso demonio.

start()

Inicia la actividad del proceso. Se debe llamar como maximo una vez por objeto de proceso. Hace que el metodo run() sea ejecutado en un proceso separado.

join()

Espera a que la ejecucion del proceso a partir del cual se invoca el metodo finalize. Si el parametro tiemout que es opcional es un numero positivo, esperara al menos timeout segundos.
Este metodo regresa None si termina el tiempo de espera o si termina la ejecucion del proceso.
Un proceso puede ser "juntado"varias veces, sin embargo no puede ser juntado consigo mismo.

is_alive()

Devuelve verdadero si el proceso aun sigue vivo. Un proceso permanece vivo desde que se ejecuta el start(), hasta que este termina o es juntado con otro.

terminate()

Termina el proceso, sin embargo los procesos decendientes de este NO terminaran, simplemente quedaran huerfanos.
Si este metodo es usado mientras se usa un tuberia o cola, estas pueden corromperse volverse inutilizables por otro proceso.

3. Clase multiprocessing.Process y herencia

La clase Process del modulo multiprocessing, puede heredar a una subclase. Esto se hace creando una nueva clase que reciba como parámetro la clase Process, y sobrescribiremos los métodos __init__(self) y run(self), además dentro de esta subclase podemos declarar nuevos métodos o atributos, dependiendo del uso que se le pretenda dar.

A continuación, un ejemplo que ilustra lo mencionado.

```
# worker_thread_subclass.py
import random
import multiprocessing
import time
class WorkerProcess(multiprocessing.Process):
    def __init__(self, name):
        multiprocessing.Process.__init__(self)
        self.name = name
    def run(self):
        """
        Run the thread
        """
        worker(self.name)
def worker(name: str) -> None:
    print(f'Started worker {name}')
    worker_time = random.choice(range(1, 5))
    time.sleep(worker_time)
    print(f'{name} worker finished in {worker_time} seconds')
if __name__ == '__main__':
    processes = []
    for i in range(5):
        process = WorkerProcess(name=f'computer_{i}')
        processes.append(process)
        process.start()
    for process in processes:
        process.join()
```

4. Referencias

- https://www.geeksforgeeks.org/wait-system-call-c/
- https://www.geeksforgeeks.org/fork-system-call/?ref=lbp
- https://www.geeksforgeeks.org/c-program-demonstrate-fork-and-pipe/?ref=lbp
- https://man7.org/linux/man-pages/man3/pthread_create.3.html
- https://man7.org/linux/man-pages/man3/pthread_join.3.html
- https://man7.org/linux/man-pages/man3/pthread_exit.3.html
- https://www.geeksforgeeks.org/what-is-the-python-global-interpreter-lock-gil/
- https://www.cartagena99.com/recursos/alumnos/apuntes/ININF1_M10_U1_T4_MT.pdf

- https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Process.run
- https://dzone.com/articles/python-101-creating-multiple-processes