HugoCarlosMoranPeraza-CC-Tarea1

Octubre 29 2020

1 Funciones en C vistas en clase

1.1 Funciones para procesos

fork()

Con este comando podemos crear un proceso hijo, el cual ejecutará concurrentemente las instrucciónes posteriores al mismo comando, con el proceso padre. Esta función no recibe ningún parámetro, y nos regresará un valor negativo si la creación del proceso hijo falló, un cero para el proceso hijo y un número positivo para el proceso padre.

wait()

Como el nombre lo dice, la función detiene la ejecución del proceso padre, hasta que alguno de sus procesos hijos termine. Una vez ocurrido lo anterior el proceso padre continua ejecútando todas las instrucciones escritas después del comando. A diferencia de la función anterior, esta función si recibe un parámetro del tipo entero, el cual es un apuntador para para guardar el estado de salida de uno de sus procesos hijo. Regresa el ID del proceso hijo que concluyó su ejecución.

exit()

Como el nombre lo dice, esta función termina de inmediato el proceso que la llamó.

pipe()

Esta función es usada para transmitir información entre procesos. De parámetro recibe un apuntador a la localidad de memoria que se usará como canal y regresa de output un cero si se creó correctamente, y un menos uno si hubo un fallo.

1.2 Funciones para hilos

pthread_create()

Crea un nuevo hilo, recibe como parámetro la localidad de la memoria donde se almacenará el hilo, también recibe la función start_routine(), con el código que será ejecutado por el hilo. Regresará un cero si todo salió bien, y un error si algo salió mal.

pthread_join()

Espera a que un hilo termine, tal hilo entra como un parámetro, y un apuntador doble. Regresa un cero en caso de éxito, o un menos uno en otro caso

pthread_exit()

Termina de inmediato la ejecución del hilo que lo ha llamado, recibe como parámetro el valor que será regresado al terminar la ejecución del hilo.

2 Conceptos

2.1 Global Interpreter Lock (GIL)

Es el mecanismo utilizado en CPython para impedir que múltiples threads modifiquen los objetos de Python a la vez en una aplicación multi hilo. 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.

Aunque CPython utiliza el soporte nativo del sistema operativo donde se ejecute a la hora de manejar hilos, y la implementación nativa del sistema operativo permite la ejecución de múltiples hilos de forma simultánea, el intérprete CPython fuerza a los hilos a adquirir el GIL antes de permitirles acceder al intérprete, la pila y puedan así modificar los objetos Python en memoria.

En definitiva, el GIL protege la memoria del intérprete que ejecuta nuestras aplicaciones y no a las aplicaciones en sí. El GIL también mantiene el recolector de basura en un correcto y saneado funcionamiento.

2.2 Ley de Amdahl

La ley de Amdahl es, en ciencia de la computación, formulada por Gene Amdahl, utilizada para averiguar la mejora máxima de un sistema de información cuando solo una parte de éste es mejorado. La fórmula original de la ley de Amdahl es la siguiente:

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

siendo:

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 antiguo.
T_m= tiempo de ejecución mejorado.

2.3 Multiprocessing

Es un modulo de python, que nos sirve para manejar procesos, nos ayuda a generar y manipular los procesos requeridos. Este modulo nos da una solución para hacer concurrencia local, evitando lo que vimos anteriormente (GIL). De todas las clases, la más importante es Process, así mostraremos los métodos más importante de esta clase:

Process(group=None, target=None, name=None, args=(), kwargs=, *, daemon=None)
Los objetos de proceso representan la actividad que se ejecuta en un proceso separado. La clase Process tiene equivalentes para todos los métodos threading.Thread.

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 (consulte name para más detalles). 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.

run()

Método que representa la actividad del proceso.

Puede anular este método en una subclase. El método estándar run() invoca el objeto invocable pasado al constructor del objeto como argumento objetivo, si lo hay, con argumentos posicionares y de palabras clave tomados de los argumentos args y kwargs, respectivamente.

start()

Comienza la actividad del proceso.

Esto debe llamarse como máximo una vez por objeto de proceso. Organiza la invocación del método run() del objeto en un proceso separado.

join([timeout])

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. Tenga en cuenta que el método retorna None si su proceso finaliza o si el método agota el tiempo de espera. Verifique el proceso exitcode para determinar si terminó.

Un proceso puede unirse muchas veces.

Un proceso no puede unirse a sí mismo porque esto provocaría un punto muerto. Es un error intentar unirse a un proceso antes de que se haya iniciado.

is_alive()

Retorna si el proceso está vivo.

Aproximadamente, un objeto de proceso está vivo desde el momento en que el método start() retorna hasta que finaliza el proceso hijo.

3 Clase multiprocessing.Process

La clase Process del modulo de Python multiprocessing, puede heredar a una subclase.
Lo anterior quiere decir que podemos crear una nueva clase que reciba como parámetro la clase Process, y posteriormente sobrescribiremos los métodos __init__(self) y run(self), por otro lado también podemos dentro de esta subclase declarar nuevos métodos, dependiendo de lo que tengamos en mente hacer.
Veamos un ejemplo típico del tema.

```
# 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://python-docs-es.readthedocs.io/es/3.8/library/multiprocessing.htmlthe-process-class

https://es.wikipedia.org/wiki/Ley_de_Amdahl

https://www.geeksforgeeks.org/what-is-the-python-global-interpreter-lock-gil/

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

https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Process.run

https://realpython.com/python-gil/