Licenciatura en Ciencia de Datos
Computación Concurrente

29 de octubre de 2020

Tarea 1

`Nombre`: Hernández Lozano Juan Pablo
`Equipo`: Mosqueda García Raúl Isaid Ruiz
Puga Ingrid Pamela
Veleros Vega Luis Alfonso
`Github`: github.com/IsaidMosqueda/LCD-CC-2021-I.

`Repositorio de Overleaf:` https://www.overleaf.com/1767152681dfzkcdcrbcyv

Actividad 2

`Resume de forma individual las funciones que fueron utilizadas en` C `para crear, comunicar y manipular procesos e hilos en` C

- **fork()**: Crea un procesos hijo idéntico al padre pero con su propio identificador y pone en espera al padre hasta concluir con el proceso hijo.
- **exit()**: Termina con la llamada al proceso
- **perror()**: Imprime un mensaje de descripción del error
- **wait()**: bloquea la llamada al proceso hasta que uno de sus procesos hijos termine.
- **sleep()**: pone en pausa la ejecución de un procesos dado un parámetro de tiempo.
- **getpid()**: Obtiene el PID del proceso hijo.
- **getppid()**: Obtiene el PID del proceso padrea.
- **getuid()**: Obtiene el identificador del usuario.
- **pipe()**: Crea un canal de comunicación entre procesos donde el fd[0] se usa para lectura y fd[1] para escritura
- **pthread_t**: Inicia un hilo
- **pthread_create**: Inicia la ejecución de un hilo
- **pthread_join**: Espera la terminación de un hilo para poder empezar con otro
- **pthread_exit**: Termina la llamada a un hilo

Actividad 3

`Investiga de forma individual los siguientes conceptos`

- *Global Interpreter Lock* `(GIL) en el contexto del multiprocesamiento en python. Resume los conceptos más importantes.`

El Global Interpreter Lock es un mecanismo usado en intérpretes de lenguajes computacionales para sincronizar la ejecución de hilos de tal forma que solo una hilo nativo pueda ser ejecutado a la vez. Un intérprete que usa GIL siempre permite que un hilo sea ejecutado a la vez, incluso si se tiene un procesador de varios núcleos. Cada hilo debe esperar a que el GIL sea liberado por otro hilo.

Python usa conteo de referencias para el maneo de la memoria. Los objetos creados en python tienen una variable de conteo de referencia que mantiene el registro del numero de referencias a las cuales apunta el objeto. Cuando el conteo alcanza el cero, la memoria ocupada por el objeto se libera. EL problema es que esta variable de conteo de referencias necesita protección de condiciones en las cuales dos hilos incrementan o decrementan su valor simultáneamente. Si esto pasa puede ocasionar que la memoria se libere incorrectamente cuando la referencia a un objeto aún existe, lo cual puede ocasionar errores y bugs.

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.

- *Ley de Amdahal*: `Concepto, terminología relacionada, fórmula de la ley.`

La ley de Amdahl es un modelo matemático que describe la relación entre la aceleración esperada de la implementación paralela de un algoritmo y la implementación serial del mismo algoritmo.

La Ley de Amdahal establece que: *La mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente.* El incremento de velocidad de un programa utilizando múltiples procesadores en computación distribuida está limitada por la fracción secuencial del programa

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

Donde:

- T_m: tiempo de ejecución mejorado.
- T_a: tiempo de ejecución antiguo.
- 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.

Usando la definición del incremento de la velocidad $A = \frac{T_a}{T_m}$, donde A es la aceleración o ganancia en velocidad conseguida en el sistema completo debido a la mejora de uno de sus subsistemas. podemos reescribir la fórmula como:

$$A = \frac{1}{(1 - F_m) + \frac{F_m}{A_m}} \tag{2}$$

La Ley de Amdahl nos da una visión pesimista de las ventajas de la paralelizacion ya que nos dice que tenemos limitada la escalabilidad, y que este límite depende de la fracción de código no paralelizable

- *Multiporcessing*: `Resume la descripción y uso del este módulo de python, enumera y describe los métodos o funciones más importantes del módulo, elige al menos 5 métodos y enlista los argumentos que reciben.`

El paquete de multiprocesamiento `multiprocessing` perimite la creación y manejo de los subprocesos en Python, también ofrece concurrencia local y remota.
El multiprocesamiento funciona creando un objeto del `process` y luego llamando a su método `start()` como se muestra a continuación:

```
from multiprocessing import Process


def greeting():
    print 'hello world'

if __name__ == '__main__':
    p = Process(target=greeting)
    p.start()
    p.join()
```

El multiprocesamiento admite dos tipos de canales de comunicación entre procesos: Tubos (pipes) que se utilizan principalmente para comunicación entre procesos, y colas(queue) se utilizan para pasar datos entre procesos:

```
#ejemplo cola (queue)
def is_even(numbers, q):
    for n in numbers:
        if n % 2 == 0:
            q.put(n)

if __name__ == "__main__":

    q = multiprocessing.Queue()
    p = multiprocessing.Process(target=is_even, args=(range(20), q))

    p.start()
    p.join()

    while q:
        print(q.get())

#ejemplo tubo (pipe)
def f(conn):
    conn.send(['hello world'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print parent_conn.recv()
    p.join()
```

También podemos asegurarnos de que solo un proceso de ejecute en un momento dado con el método `Lock()` o aplicar el método `release` para terminar el Lock.
Entre las principales funciones de esta biblioteca están :

- Dividir el trabajo entre múltiples procesos
- Empezar, esperar y terminar procesos
- Creación de procesos demonio
- Pasar mensajes a procesos
- Control de acceso concurrente a los recursos
- Controlar el acceso a los recursos

A continuación se listan algunos métodos y objetos de la clase *multiprocessing* y sus utilidades:

- `Process`: objetos que representan la actividad ejecutada en un proceso separado.
- `run()` método que representa la actividad del proceso
- `Start()` empieza la actividad del proceso
- `is_alive()` regresa si el proceso está en ejecución
- `pid` regresa el id del proceso
- `terminate()` termina el proceso
- `Pipe()` pasar un mensaje entre dos procesos
- `join()` espera hasta que un proceso haya completado su trabajo y termine

Actividad 4

`Investiga de forma individual la manera de crear procesos como un objeto que hereda de la clase multiprocessing.process y muestra un ejemplo.`

multiprocessing es un paquete que admite procesos de generación, ofrece concurrencia tanto local como remota, esquivando el término Global Interpreter Lock mediante el uso de subprocesos en lugar de hilos (threads). Debido a esto, el módulo multiprocessing le permite al programador aprovechar al máximo múltiples procesadores en una máquina determinada. Se ejecuta tanto en Unix como en Windows.

La forma más sencilla de generar un segundo proceso es crear una instancia del objeto `Process` con una función de destino y llamar `start()` para que comience a trabajar. Por lo general, es más útil poder generar un proceso con argumentos para decirle qué trabajo hacer. Para pasar argumentos a un `multiprocessing Process`, los argumentos deben poder ser serializados usando pickle. Este ejemplo pasa a cada trabajador un número para imprimir.

```
import multiprocessing


def worker(num):
    """thread worker function"""
    print('Worker:', num)


if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=worker, args=(i,))
        jobs.append(p)
        p.start()
```