Tarea 1 Computación Concurrente

Guillermo Cota

October 2020

1 Objetivo

Organizar el espacio de trabajo para reunir el material del curso, colocar en el repositorio del curso elconjunto de archivos y programas que se han elaborado a la fecha, desarrollar un sencilla investigaciónsobre multiprocesamiento con Python.

2 Cuenta Github

https://github.com/Equipo-Alfa-Lobo-Dinamita/LCD-CC-2021-I

3 Funciones de C

Algunas de las funciones utilizadas para el lenguaje C son las siguientes:

- `#include <sys/types.h>` Se definen tipos de datos derivado básicos Parte del estandar POSICS.
- `#include <unistd.h>` Nos ayuda a manejar las variables y constantes para hacer la creación de procesos y sincronización de procesos Parte del estandar POSICS.
- `fork()`. Crea un proceso hijo
- `switch()`. Es usado para crear condiciones para ejecutar código. Ya que había tres valores en la creación de fork, nos permitió sincronizar casos. Usando el -1 en caso de que hubiera un error en la creación de otro proceso, 0 para la ejecución del código del hijo y cualquier otro valor para el código del padre.
- `getpid()`. Obtención del ID del proceso actual.
- `getppid()`. Obtención del ID del proceso padre del proceso actual.
- `exit()`. Termina la ejecución de un proceso hijo.

- `wait()`. Método usado por el proceso padre, su función es esperar a que el código hijo termine, para así continuar la ejecución del programa.
- `pipe()`. Comunica a dos procesos a través de una pipa de comunicación, toma como parámetro un arreglo de dos enteros. Tiene dos canales de comunicación, uno de lectura y otro de escritura que permititrán la comunicación entre distintos procesos. Para ello es necesario que cada vez que se lea o escriba sobre el canal, se cierre el otro.
- `close()` Cierra el canal de escritura o lectura dependiendo de qué canal se seleccione (0= lectura, 1=escritura de manera estandar). Toma como parámetro algún canal de una pipa.
- `write()` Escribe en una pipa de comunicación
- `read()`. Realiza la lectura en una pipa de comunicación.
- `#include <pthread.h>`. Librería usada para la implementación de hilos en C.
- `pthread_create()`. Realiza la creación de un hilo.
- `pthread_exit(id)`. Termina la ejecución del hilo con ID=id.
- `pthread_join()`. Método que permite la incorporación del hilo, nuevamente con el proceso principal.

4 Global Interpreter Lock (GIL)

El Global Interpreter Lock, es un bloqueo de hilos de ejecución que tiene integrado Python que impide la ejecución de más de un hilo de manera simultanea. Esto surge como medida preventiva ante una función que tiene Python. La forma en que asigna y desasigna memoria Python, es a través de la función `referencecounter()` que es un contador que se encarga de llevar el número de veces que una variable es referenciada a lo largo del programa, de manera que al llegar este contador a 0, puede liberar memoria ante la falta de necesidad de seguir guardando esa variable. Ante la poca seguridad de la integridad de esta función ante la aparición de distintos hilos de ejecución se implementa el GIL, hecho que ha causado alguna polémica por desarrolladores, pero que al ser quitada, podría traer problemas de compatibilidad. Ante esto, la creación de distintos procesos en vez de hilos de ejecución, funcionan como la alternativa viable para traer la teoría concurrente a Python.

5 Ley de Amdahal

En teoría de computación, la ley de Amdahl es una afirmación que deduce que el mejoramiento de rendimiento que un sistema gana al optimizar solo una parte

de él, está limitado a la fracción de tiempo que esa parte es usada. En otra palabras, la mejora del sistema depende de la fracción de tiempo que la parte que fue mejorada es usada. En general, la fórmula de la ley de Amdahl está dada por:

$$T_i = T_o \times ((1 - F_i) + \frac{F_i}{A_i})$$

Donde:

- T_i: El tiempo de ejecución mejorado.
- T_o: El tiempo de ejecución antiguo.
- F_i: La fracción de tiempo utilizada por la parte mejorada.
- A_i: El factor de mejora que se introfuce al sistema.

6 Multiprocessing en Python

La biblioteca `multiprocessing` de Pytohn, nos permite, al igual que podemos hacer en C, crear procesos hijos que se ejecutaran de manera concurrente a los procesos padres. Algunos métodos importantes son los siguientes:

- `class multiprocessing.Process(group=None, target=None, name=None, args=(), daemon=None)`. Crea un proceso hijo. Argumentos:
  1. group=None. Argumento únicamente usado para tener compatibilidad con el módulo de hilos de Python, siempre debería estar en None
  2. target. Función a la que estará destinado el proceso hijo.
  3. name=None. String, será el nombre del proceso.
  4. args=().Tupla, serán los argumentos que recibirá la función.
  5. daemon=None. Bool, en caso de querer crear un proceso demonio, se pone True.
- `pipe()`. Regresa una tupla de dos elementos. Será la pipa de comunicación entre el proceso padre y el proceso hijo. Tiene asociado los métodos `send()` para escribir, y `recv()` para recibir los mensajes.
- `Pool(processes)`. Crea un grupo de procesos que pueden ser utilizados en distintos trabajos. Recibe el parámetro `processes` que será un número entero y dirá el número de trabajadores a crear. Algunos de sus métodos son `apply()`, `join()`, `exit()`, `terminate()`, `kill()`, `map()`.
- `current_process()`. Regresa el proceso que corresponde al proceso en ejecución.
- `alive_children()` Regresa una lista de procesos hijos del proceso actual.

7 Creación de objetos tipo proceso

En la sección anterior hicimos mención acerca del método `Process`. Ahora, vamos a ahondar un poco más en su uso. El uso del método crea un proceso que espera a la llamada de `start()` para empezar y `join()` para volverse a unir con la ejecución del padre. De igual manera existen otros métodos asociados a este, como el método `terminate(Process.name)` que termina la ejecución de un proceso. La creación de procesos demonios no necesitan ser llamados mediante el método `join()`, pues pueden terminar junto al proceso padre.

```
from multiprocessing import Process
import os

def info():
    print ('El nombre del modulo:', __name__)
    if hasattr(os, 'getppid'):
        print ('Proceso padre PID:', os.getppid())
    print ('PID:', os.getpid())

def greeting(nombre):
    print('\nSoy codigo del proceso creado')
    print (f'hello {nombre}')
    print ('Proceso padre PID:', os.getppid())
    print ('PID:', os.getpid())

if __name__ == '__main__':
    info()
    p = Process(target=greeting, args=('Comunidad de la LCD',))
    p.start()
    p.join()
```

La utilidad de la condicional `if __name__ == '__main__'` radica en que el interprete de python asigna un nombre `__name__` a cada módulo importado, de forma que `'__main__'` corresponde al programa principal. Así al correr el código no habrá problemas no deseados como el comienzo de nuevos procesos sin ser solicitados. También podemos mostrar los PID de los procesos usando el método `getpid()` de la paquetería `os`. si estamos manejando un proceso hijo, podemos recuperar el PID del proceso padre usando `os.getppid()`.

Bibliografía.

- What is the Python Global Interpreter Lock (GIL) - GeeksforGeeks. (2018). Retrieved 28 October 2020, from https://www.geeksforgeeks.org/what-is-the-python-global-interpreter-lock-gil/
- Amdahl's Law - an overview — ScienceDirect Topics. (2017). Retrieved 29 October 2020, from

https://www.sciencedirect.com/topics/computer-science/amdahls-law