Computación Concurrente: Tarea 1

Rodolfo Figueroa Soriano

5 de noviembre de 2020

Parte 2

Las definiciones de esta sección provienen de las `man pages` de linux. Una copia en línea puede accesarse en [1].

- `fork()`: Crea un nuevo proceso, llamado *proceso hijo* que corre de manera concurrente con el proceso que hizo la llamada a `fork` (conocido como *proceso padre*). Regresa un entero p con los siguientes posibles valores:
  - $p < 0$: El proceso hijo falló dn ser creado.
  - $p = 0$: Se regresó al proceso padre.
  - $p > 0$: Se regresó al proceso hijo.
- `exit(int s)`: Termina el proceso que lo llamó. Regresa el valor `s` al proceso padre.
- `perror(const char *s)`: Imprime el mensaje de error `str` a `stderr`.
- `wait(&status)`: Detiene la ejecución del proceso hasta que uno de sus procesos hijos haya acabado. Llena `status` con el código de salida del proceso hijo.
- `getpid()`: Regresa el id del proceso que lo llamó.
- `getppid()`: Regresa el id del proceso padre del proceso que lo llamó.
- `getuid()`: Regresa el id de usuario real del proceso que lo llamó.

- `pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start)(void *), void *arg)`: Crea un nuevo hilo, con atributo s especificados por `attr`; si este es `NULL`, usa los atributos por defecto. Si `thread` no es `NULL`, el id del hilo creado se guarda en él. Al momento de ser creado, el hilo ejecuta `start` con `arg` como argumento.

- `pthread_join(pthread_t thread, void **retval)`: Espera a que el hilo especificado por `thread` acabe. Si `retval` no es `NULL`, copia el estado de salida del hilo a él.

- `pthread_exit(void *retval)`: Termina el hilo que lo llamó y regresa el valor `retval`.

- `pipe(int pipefd[2])`: Crea un *pipe*, un canal de datos unidireccional que puede usarse para que dos procesos se comuniquen entre sí. El arreglo `pipefd` se usa para regresar dos descriptores de archivo referentes a los extremos del *pipe*: `pipefd[0]` es el extremo de lectura, y `pipefd[1] es el extremo de escritura`. Cualquier dato escrito en el de escritura estará disponible en el de lectura.

- `close(int fd)`: Cierra un descriptor de archivo. Si `fd` corresponde al descriptor de un *pipe*, bloquea dicho *pipe*, impidiendo la lectura y escritura.

Parte 3

Global Interpreter Lock

El mayor obstáculo al momento de implementar métodos concurrentes en Python es el *Global Interpreter Lock* (GIL). Esta es una restricción que otorga todo el control del interpretador de Python a un único hilo; es decir, solo un hilo puede estar ejecutándose en cualquier momento.[2]

El GIL existe debido a que Python administra la memoria mediante el llamado *conteo de referencias*[3]: cada objeto en Python lleva un conteo interno de cuántas referencias apuntan hacia él, y cuando este número llega a cero, Python sabe que es seguro limpiar la memoria ocupada por el objeto. Sin embargo, si hay más de un hilo ejecutando el mismo código, varios hilos pueden intentar operar sobre el mismo objeto al mismo tiempo, lo cual puede

causar una fuga de memoria, o peor aún, hacer que el objeto se borre cuando aún hay referencias hacia él.

En principio, esto puede solucionarse poniéndole un *lock* a cada uno de los objetos compartidos entre hilos; empero, esto puede dar lugar a *deadlocks*. Además, el poner y quitar *locks* varias veces para cada objeto es computacionalmente costoso, e incrementaría el tiempo de ejecución.

Con el GIL no se tiene este problema: al tener un único *lock* permanente que opera sobre todos los objetos, se evaden todas estas complicaciones. Además, como no hay que preocuparse por reescribir código que dependa de concurrencia, muchas librerías de C pueden ser importadas a Python sin tener que hacer ningún cambio.

Sin embargo, si se quiere ejecutar código concurrente, el GIL representa una gran restricción, aún más si el tiempo de ejecución depende principalmente del CPU. Es por esto que la comunidad de Python ha creado varias maneras de sobrepasarlo sin causar problemas de compatibilidad, tales como el fork de CPython Gilectomy, o la paquetería `multiprocessing`.

Ley de Amdahl

La ley de Amdahl describe qué tanto en teoría puede optimizarse la latencia de ejecución de una tarea si se mejoran los recursos del sistema que la está ejecutando [4].

En general, la ley puede formularse de la siguiente manera:

$$S = \frac{1}{(1-p) + \frac{p}{s}}$$

Donde:

- S es cuánto en teoría se optimiza la tarea.
- s es la optimización de la parte de la tarea que se beneficia de los recursos mejorados.
- p es la proporción del tiempo de ejecución que la parte que se beneficia de los recursos mejorados ocupaba.

Para ilustrarla, consideremos un programa que se lleva 10 horas en ejecutar en un solo hilo. En principio, si lo ejecutamos en 10 hilos, terminaría de ejecutarse en tan sólo una hora. Sin embargo, si el programa tiene una

subrutina de una hora que no puede ser paralelizada, esto significa que solo las 9 horas restantes ($p = 0{,}9$) pueden ser optimizadas, y el tiempo total de ejecución nunca podrá ser menor a una hora. Por lo tanto, la optimización máxima está limitada a 10 veces del rendimiento de un solo hilo:

$$\frac{1}{1-p} = 10$$

multiprocessing

`multiprocessing` es un paquete de Python que permite ejecutar código de manera simultánea, efectivamente ignorando el GIL. La paquetería logra esto usando subprocesos en vez de hilos para manejar la ejecución concurrente. El problema con esto es que la comunicación entre procesos separados se vuelve complicada, ya que cada uno debe de tener su propia instancia de CPython ejecutándose. Sin embargo, `multiprocessing` mitiga un poco este efecto, ya que ofrece muchas funciones auxiliares para facilitar la administración de estos procesos [5].

La piedra angular de `multiprocessing` es la clase de `Process`. Cualquier objeto con esta clase representa actividad que está siendo ejecutada en un proceso separado. Para realizar alguna acción sobre dicho proceso (inicialziación, interrupción, etc.), pueden ejecutarse funciones sobre su correspondiente objeto `Process`, y Python se encargará del resto. Algunas de los métodos más importantes en `multiprocessing` son:

- `Process(group=None, target=None, name=None, args=(), kwargs=, *, daemon=None)`: Constructor del objeto. `group` siempre debe de ser `None`(solo existe para compatibilidad con `threading`). `target` es la función que se ejecutará al iniciar el proceso, con argumentos `args` y `kwargs`. `name` es el nombre que se le dará al proceso, y `daemon` indica si debe de correrse como un *daemon*.

- `start()`: Inicia la actividad del proceso representado por el objeto.

- `join([timeout])`: Si `timeout` es `None`, el método se bloquea hasta que el proceso que lo llamó termina. Si `timeout` es un número positivo, este bloqueo dura a lo mucho este número de segundos.

- `terminate()`: Termina el proceso. Hay que tener mucho cuidado al usar este método, ya que si hay *pipes* siendo usados por el proceso,

estos pueden corromperse, y si el objeto tiene *locks* o semáforos, otros procesos pueden entrar en *deadlock*.

- `is_alive()`: Regresa si el proceso está vivo o no.
- `Pipe([duplex])`: Regresa un par `(c1,c2)` de objetos `Connection` que representan los extremos de un *pipe*. Si `duplex` es verdadero, el *pipe* es bidireccional; de otra manera, es unidireccional.

Parte 4

Como se mencionó anteriormente, podemos usar el constructor `Process` para crear un objeto representativo de un proceso. Posteriormente, se inicia el proceso utilizando el método `start()`.

Por ejemplo, el siguiente código crea cuatro procesos, los cuales generan un número entre `i` y `pid` + 1, donde `pid` es el id del proceso:

```
import multiprocessing as mp
import random
import os

def rand_num():
  p = os.getpid()
  num = random.random() + p
  print("PID:", p, "\nRand:", num,"\n---")

processes = [mp.Process(target=rand_num) for i in range(4)]

for p in processes:
  p.start()
  p.join()

```

Out:

```
PID: 420564
Rand: 420564.03614360985
---
PID: 420585
Rand: 420585.5549975002
---
PID: 420606
Rand: 420606.47376360255
---
```

```
PID: 420627
Rand: 420627.29933302687
---
```

Referencias

[1] "Linux man pages online." https://man7.org/linux/man-pages/. Accesado: 2020-10-29.

[2] "Globalinterpreterlock." https://wiki.python.org/moin/GlobalInterpreterLock. Accesado: 2020-10-29.

[3] "What is the python global interpreter lock?." https://realpython.com/python-gil/. Accesado: 2020-10-29.

[4] "Amdahl's law." http://tutorials.jenkov.com/java-concurrency/amdahls-law.html. Accesado: 2020-10-29.

[5] "multiprocessing." https://docs.python.org/3/library/multiprocessing.html. Accesado: 2020-10-29.