Computación Concurrente

Tarea 01. Introducción a Concurrencia.

Lara Moreno Yeudielu

Tarea 01. Introducción a Concurrencia

Lara Moreno Yeudiel

24 de septiembre de 2021

1. Funciones utilizadas en C para crear comunicar y manipular procesos

Funciones que vimos en clase para comunicar y manipular procesos.

- **<sys/types.h>** Permite definir tipos y estructuras de datos.
- **<unistd.h>** Archivo que nos permite trabajar con POSIX.
- **int** dato de tipo entero.
- **main()** inicializa la función principal.
- **fork()** LLama al sistema para crear un proceso. Retorna el id del proceso hijo al padre como valor entero. Si no logra crear el proceso hijo retorna -1.
- **exit()** Finalizar el proceso que se esta ejecutando.
- **wait()** Espera a la terminación del proceso hijo. Si recibe como argumento *NULL* espera a terminar el proceso, si recibe *&status* retorna el id del proceso hijo cuando termine su proceso.
- **sleep()** Detiene el proceso durante una cantidad dada de segundos.
- **getpid()** Identificador del proceso actual.
- **getppid()** Identificador del padre actual.
- **getuid()** Devuelve el identificador de usuario final.
- **pipe()** Archivo en la memoria principal sobre el cual pueden escribir y/o leer los diferentes procesos que se ejecuten siempre y cuando estén entrelazados.
- **write()** Recibe como argumentos $\texttt{fd}, *\texttt{bd}, \texttt{count}$. Provoca que los primeros bytes del buffer sean escritos en el archivo que se le asocia con el descriptor de fd.
- **read()** Recibe como argumentos $\texttt{fd}, *\texttt{bd}, \texttt{count}$. Lee un descriptor del archivo y nos indica en donde se guardara.

- **close()** Cierra un descriptor de archivo.
- **gets()** Lee una línea de entrada completa hasta la terminación con EOF caracter ASCII especial.
- **usleep()** Suspende la ejecución para intervalos de microsegundos
- **fflush()** Limpia el buffer de escritura.
- **‹pthread.h›** Archivo que proporciona tipos, contantes y funciones para el manejo de threads.
- **pthread create()** Construye un hilo, pasando como argumento *thread, un apuntador donde guardará el hilo.
- **pthread join()** La función espera a que termine el hilo especificado por thread. Si ese hilo termina, entonces la función regresa inmediatamente. El hilo especificado por thread (debe poder unirse).

2. Conceptos Importantes

2.1. Global Interpreter Lock (GIL)

Python Global Interpreter Lock o GIL , en palabras simples, es un mutex (o un bloqueo) que permite que solo un hilo mantenga el control del intérprete de Python.

En consecuencia solo un subproceso puede estar en estado de ejecución en cualquier momento. El impacto de GIL no es visible para los desarrolladores que ejecutan programas de un solo subproceso, pero puede ser un cuello de botella en el rendimiento en el código vinculado a la CPU y de varios subprocesos.

Dado que GIL permite que solo se ejecute un subproceso a la vez, incluso en una arquitectura de subprocesos múltiples con más de un núcleo de CPU, GIL se ha ganado la reputación de ser una característica ïnfame"de Python.

2.1.1. ¿Qué problema resolvió el GIL para Python?

Python usa el recuento de referencias para la gestión de la memoria . Significa que los objetos creados en Python tienen una variable de recuento de referencias que realiza un seguimiento del número de referencias que apuntan al objeto. Cuando este recuento llega a cero, se libera la memoria ocupada por el objeto.

El problema era que esta variable de recuento de referencia necesitaba protección contra condiciones de carrera en las que dos subprocesos aumentan o disminuyen su valor simultáneamente. Si esto sucede, puede causar una pérdida de memoria que nunca se libera o, lo que es peor, liberar la memoria incorrectamente mientras todavía existe una referencia a ese objeto. Esto puede causar bloqueos u otros errores .[ex]traños.[en] sus programas de Python.

El GIL fue la solución al problema pues es un bloqueo único en el propio intérprete que agrega una regla de que la ejecución de cualquier código de bytes de Python requiere adquirir el bloqueo del intérprete. Esto evita los interbloqueos (ya que solo hay un bloqueo) y no introduce mucha sobrecarga de rendimiento. Pero efectivamente hace que cualquier programa de Python vinculado a la CPU sea de un solo subproceso.

2.1.2. Eliminando el GIL

Deshacerse del GIL es un tema ocasional en la lista de correo de python-dev. Nadie lo ha logrado todavía. Las siguientes propiedades son todas muy deseables para cualquier posible reemplazo de GIL; algunos son requisitos estrictos.

- Sencillez
- Simultaneidad
- Velocidad
- Compatibilidad con API
- Destrucción inmediata
- Destrucción ordenada

2.2. Ley de Amdahal

La ley de Amdahl, que lleva el nombre de un arquitecto informático llamado Gene Amdahl y su trabajo en la década de 1960, es una ley que muestra cuánta latencia se puede eliminar de una tarea de rendimiento mediante la introducción de la computación concurrente.

En computación concurrente, la ley de Amdahl se usa principalmente para predecir la aceleración máxima teórica para el procesamiento de programas usando múltiples procesadores.

La definición de esta ley establece que: *La mejora obtenida en el rendimiento de un sistema debido a la alternación de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente.*

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

Donde:

- 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.

Esta fórmula se puede reescribir usando la definición del incremento de la velocidad que viene dado por $A = \frac{T_a}{T_m}$, por lo que la fórmula anterior se puede reescribir como:

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

Donde:

- A es la aceleración o ganancia en velocidad conseguida en el sistema completo debido a la mejora de uno de sus subsistemas.
- A_m es el factor de mejora que se ha introducido en el subsistema mejorado.
- F_m es la fracción de tiempo que el sistema utiliza el subsistema mejorado.

2.3. *Multiprocessing*

multiprocessing es un paquete que admite procesos de generación mediante una API similar al módulo de threading. El multiprocessing es un paquete que ofrece simultaneidad local y remota, evitando, efectivamente el bloqueo global de intérprete mediante el uso de subprocesos en lugar de subprocesos. Debido a esto, el módulo multiprocessing permite al programador aprovechar al máximo de múltiples procesadores en una máquina determinada. Funciona tanto en Unix como en Windows.

El módulo multiprocessing también introduce API's que no tienen análogos en el módulo de threading. Un buen ejemplo de esto es el objeto Pool que ofrece un medio conveniente para paralelizar la ejecución de una función a través de múltiples valores de entrada, distribuyendo los datos de entrada entre los procesos (paralelismo de datos).

Algunos métodos importantes del módulo son:

Método	**Argumentos**	**Breve Descripción**
Process	group es None por default, target función objetivo, name, args() argumentos que recibe la función, kwargs=, *, daemon=None	Crea un proceso con su propio espacio de memoria
Queues	No tiene argumentos	Construye una estructura de datos tipo cola.
Pipes	No tiene argumentos	Esta función devuelve un par de objetos de conexión conectados por una tubería que por defecto es dúplex (bidireccional).
run()	No tiene argumentos	Método que representa la actividad del proceso. Puede anular este método en una subclase. El run() método estándar invoca el objeto invocable pasado al constructor del objeto como argumento de destino, si lo hay, con argumentos secuenciales y de palabra clave tomados de los argumentos args y kwargs , respectivamente.
start()	No tiene argumentos	Inicie la actividad del proceso. Esto se debe llamar como máximo una vez por objeto de proceso. Organiza que el run()método del objeto se invoque en un proceso separado.
join()	No tiene argumentos	Si el tiempo de espera del argumento opcional es None(el predeterminado), el método se bloquea hasta que join()finaliza el proceso cuyo método se llama. Si el tiempo de espera es un número positivo, bloquea como máximo los segundos de tiempo de espera .

Cuadro 1: Métodos y funciones importantes de *multiprocessing*

3. Creación de procesos como un objeto que hereda de la clase multiprocessing.process

3.1. Ejemplo

Utilizamos la creación de clases en python y pasamos un proceso como argumento para poder realidar lo que se nos solicita.

```
import multiprocessing
import os

class MyProccess(multiprocessing.Process):
  #Creamos un subproceso.
  def __init__( self , name ) :
    super(MyProccess,self).__init__()
    self.name = name

  #Creamos un atributo para imprimir el ide del proceso.
  def run(self):
    print("Proceso: {}, tengo ID:  {}".format(self.name, os.getpid()))


if __name__ == '__main__':
  print("Proceso padre con ID: {}".format(os.getpid()))
  hijo = MyProccess("hijo")
  hijo.start()
  hijo.join()
  print("Fin de proceso padre.")
```

Obtenemos como salida:

```
Proceso padre con ID: 65
Proceso: hijo, tengo ID:  147
Fin de proceso padre.
```

Referencias

[1] *GlobalInterpreterLock - Python Wiki.* (2020, 22 diciembre). Python.org. https://wiki.python.org/moin/GlobalInterpreterLock

[2] *multiprocessing — Process-based parallelism — Python 3.9.7 documentation.* (2021, 23 septiembre). Python.org. https://docs.python.org/3/library/multiprocessing.html

[3] Nguyen, Q. (2018). *Mastering Concurrency in Python.* Packt Publishing.

[4] Real Python. (2021, 15 mayo). *What Is the Python Global Interpreter Lock (GIL)?* https://realpython.com/python-gil/

[5] Techopedia. (2020, 30 junio). *Amdahl's Law*. Techopedia.Com. https://www.techopedia.com/definition/17035/amdahls-law