Python - impasse de thread

Um deadlock pode ser descrito como um modo de falha de simultaneidade. É uma situação em um programa onde um ou mais threads esperam por uma condição que nunca ocorre. Como resultado, os threads não conseguem progredir e o programa fica travado ou congelado e deve ser encerrado manualmente.

A situação de impasse pode surgir de várias maneiras em seu programa simultâneo. Os impasses nunca são desenvolvidos intencionalmente; em vez disso, são na verdade um efeito colateral ou bug no código.

As causas comuns de deadlocks de thread estão listadas abaixo -

- Um thread que tenta adquirir o mesmo bloqueio mutex duas vezes.
- Threads que esperam um no outro (por exemplo, A espera em B, B espera em A).
- Quando um thread que não consegue liberar um recurso como bloqueio, semáforo, condição, evento, etc.
- Threads que adquirem bloqueios mutex em ordens diferentes (por exemplo, falham ao executar a ordem de bloqueio).

Se mais de um thread em um aplicativo multithread tentar obter acesso ao mesmo recurso, como executar uma operação de leitura/gravação no mesmo arquivo, isso poderá causar inconsistência de dados. Portanto, é importante que o tratamento simultâneo seja sincronizado para que seja bloqueado para outros threads quando um thread estiver usando o recurso.

O módulo de threading fornecido com Python inclui um mecanismo de bloqueio simples de implementar que permite sincronizar threads. Um novo bloqueio é criado chamando o método Lock(), que retorna o novo bloqueio.

O objeto de bloqueio

Um objeto da classe Lock possui dois estados possíveis - bloqueado ou desbloqueado, inicialmente no estado desbloqueado quando criado pela primeira vez. Um bloqueio não pertence a nenhum thread específico.

A classe Lock define os métodos adquirir() e release().

O método adquirir()

Quando o estado é desbloqueado, este método altera o estado para bloqueado e retorna imediatamente. O método recebe um argumento de bloqueio opcional.

Sintaxe

```
Lock.acquire(blocking, timeout)
```

Parâmetros

- **bloqueio** - Se definido como False, significa não bloquear. Se uma chamada com bloqueio definido como True for bloqueada, retorne False imediatamente; caso contrário, defina o bloqueio como bloqueado e retorne True.

O valor de retorno deste método será True se o bloqueio for adquirido com sucesso; Falso se não.

O método release()

Quando o estado está bloqueado, este método em outro thread o altera para desbloqueado. Isso pode ser chamado de qualquer thread, não apenas do thread que adquiriu o bloqueio

Sintaxe

```
Lock.release()
```

O método release() só deve ser chamado no estado bloqueado. Se for feita uma tentativa de liberar um bloqueio desbloqueado, um RuntimeError será gerado.

Quando a fechadura estiver bloqueada, redefina-a para desbloqueada e retorne. Se algum outro thread estiver bloqueado aguardando o desbloqueio do bloqueio, permita que exatamente um deles continue. Não há valor de retorno deste método.

Exemplo

No programa a seguir, dois threads tentam chamar o método sincronizado(). Um deles adquire o bloqueio e ganha o acesso enquanto o outro espera. Quando o método run() é concluído para o primeiro thread, o bloqueio é liberado e o método sincronizado fica disponível para o segundo thread.

Quando ambos os threads se juntam, o programa chega ao fim.

```
from threading import Thread, Lock
import time
```

```
lock=Lock()
threads=[]

class myThread(Thread):
   def __init__(self,name):
      Thread.__init__(self)
      self.name=name
   def run(self):
      lock.acquire()
      synchronized(self.name)
      lock.release()

def synchronized(threadName):
   print ("{} has acquired lock and is running synchronized method".format(threadName
   counter=5
   while counter:
      print ('**', end='')
      time.sleep(2)
      counter=counter-1
   print('\nlock released for', threadName)

t1=myThread('Thread1')
t2=myThread('Thread2')

t1.start()
threads.append(t1)

t2.start()
threads.append(t2)

for t in threads:
   t.join()
print ("end of main thread")
```

Ele produzirá a seguinte **saída** -

```
Thread1 has acquired lock and is running synchronized method
**********
lock released for Thread1
Thread2 has acquired lock and is running synchronized method
**********
lock released for Thread2
end of main thread
```

O objeto semáforo

Python suporta sincronização de threads com outro mecanismo chamado **semáforo** . É uma das técnicas de sincronização mais antigas inventadas por um conhecido cientista da computação, Edsger W. Dijkstra.

O conceito básico do semáforo é usar um contador interno que é decrementado a cada chamada de aquisição() e incrementado a cada chamada de liberação(). O contador nunca pode ficar abaixo de zero; quando adquire() descobre que é zero, ele bloqueia, aguardando até que algum outro thread chame release().

A classe Semaphore no módulo threading define os métodos adquire() e release().

O método adquirir()

Se o contador interno for maior que zero na entrada, diminua-o em um e retorne True imediatamente.

Se o contador interno for zero na entrada, bloqueie até ser acordado por uma chamada para release(). Uma vez acordado (e o contador for maior que 0), diminua o contador em 1 e retorne True. Exatamente um thread será despertado por cada chamada para release(). A ordem em que os threads são ativados é arbitrária.

Se o parâmetro de bloqueio estiver definido como False, não bloqueie. Se uma chamada sem argumento for bloqueada, retorne False imediatamente; caso contrário, faça o mesmo que quando chamado sem argumentos e retorne True.

O método release()

Libere um semáforo, incrementando o contador interno em 1. Quando ele for zero na entrada e outros threads estiverem esperando que ele se torne maior que zero novamente, ative n desses threads.

Exemplo

```
from threading import *
import time

# creating thread instance where count = 3
lock = Semaphore(4)

# creating instance
def synchronized(name):

   # calling acquire method
```

```
    lock.acquire()

    for n in range(3):
        print('Hello! ', end = '')
        time.sleep(1)
        print( name)

        # calling release method
        lock.release()

# creating multiple thread
thread_1 = Thread(target = synchronized , args = ('Thread 1',))
thread_2 = Thread(target = synchronized , args = ('Thread 2',))
thread_3 = Thread(target = synchronized , args = ('Thread 3',))

# calling the threads
thread_1.start()
thread_2.start()
thread_3.start()
```

Ele produzirá a seguinte **saída** -

```
Hello! Hello! Hello! Thread 1
Hello! Thread 2
Thread 3
Hello! Hello! Thread 1
Hello! Thread 3
Thread 2
Hello! Hello! Thread 1
Thread 3
Thread 2
```