

Curso Superior de Engenharia da Computação

Relatório de Laboratório 3

Roteiro: Comunicação Interprocessos

Turma: 2018.1

Aluno: Erick Spinelli Pimentel

Data: 10/09/2019

1. **Dê uma descrição do que faz cada um desses tipos ou funções.**
   1. A classe Thread: representa uma atividade que é executada em uma Thread de controle separado. Há duas maneiras de especificar a atividade:
      1. Passando um objeto que pode ser chamado ao construtor
      2. Substituindo o método run () em uma subclasse.
   2. thread.start(): Inicia a atividade da Thread.
      1. thread.start() só pode ser chamado uma vez pelo objeto da classe Thread (a classe Thread instanciada). Esse método gera um RuntimeError se chamado mais de uma vez no mesmo objeto da classe Thread (a classe Thread instanciada)
   3. thread.join(): Aguarda até a Thread terminar (uma thread é iniciada, e acessa o espaço critico, o método thread.join() impede que outra thread acesse o espaço crítico, colocando a thread que está tentando acessar o espaço crítico para dormir/sleep)
      1. Quando o argumento timeout não estiver presente (ou for None) a operação que estiver tentando acessar o espaço crítico será bloqueada até o término do processamento da Thread.
      2. thread.join gera um RuntimeError se for feita uma tentativa de ingressar em uma Thread atual.
2. **Rode o programa. O que ele faz? Explique cada linha do resultado da saída do programa.**

O programa cria uma lista chamada “threads”, cria e starta 6 Threads (em um for), que tem como argumento a função “the Works” (cujo objetivo é gerar 1000000 de números aleatórios de 0 á 50 e soma-los).

Será printado o index de cada Thread e a soma dos números do randint.

1. **Por que as threads do programa não terminam na ordem em que são criadas?**

Cada Thread recebe uma fatia de tempo do escalonador para realizar o processamento, mas não termina de ser executada pois acaba a fatia de tempo e o escalonador escolhe outra Thread para fornecer o intervalo de tempo para ser processada, e por isso não se pode prever a ordem de saída do programa. (thread.join() está em um for separado, por isso não consegue segurar o fluxo de execução, para que outra Thread não acesse o mesmo espaço crítico da Thread que está em execução)

1. **O que acontece se você retirar as linhas 21 e 22 do arquivo joinEx.py. O programa precisa dessas linhas para terminar com o resultado esperado?**

Não, o programa continua com a mesma saída. (as Threads ainda não são printadas de forma ordenada) (para as Threads serem printadas de forma ordenada a linha threads[i].join() tem que estar no primeiro for, após threads[i].start()).

1. **O que a função sys.exit() faz nesse caso?**

A função sys.exit() termina a execução da Thread que está sendo executada no momento, sem ser a principal, por isso o programa continua executando as outras

1. **Dê uma descrição do que faz cada um desses tipos ou funções.**
   1. a classe Lock: implementa objetos de bloqueio primitivos. Quando uma Thread é bloqueada, as tentativas de acessá-la são bloqueadas, até que a Thread seja liberada. Qualquer Thread pode liberá-la.
   2. Lock.acquire(): altera o estado para bloqueado (ou seja entra na região crítica e impede outras threads de entrarem).
   3. Lock.release(): libera a região crítica para que outras threads do processo possa acessar.
2. **Rode o programa. O que ele faz?**

O programa cria uma lista chamada “threads”, cria e starta 5 Threads (em um for), que tem como argumento a função “add1000” (cujo o objetivo é somar 1000 à variável global chamada “shared\_data”).

A primeira Thread entra na região crítica pelo mutex.acquite() (se a fatia de tempo da Thread acabar, o mutex bloqueia qualquer outra Thread de acessar essa região crítica), a variável “j” recebe a variável global “shared\_data”, depois a Thread dorme(sleep) por 2 segundos, ai só a partir da linha que contém o mutex.release() que o mutex libera a região crítica da Thread que estava sendo executada, para outra Thread ser executada (por tanto, a fatia de tempo pode acabar, e outra Thread pode usar o espaço crítico), depois é printado "1000 added!" e a Thread é encerrada.

1. **Retire as diretivas de sincronização "acquire()" e "release()". O que acontece com o programa? Você saberia explicar esse comportamento?**

Se tirar o acquire() e o release(), as Threads poderão acessar o espaço crítico, por tanto a variável global “shared\_data” ainda não vai ter sido atualizada (com o +1000).

Como todas as Threads podem acessar o espaço crítico, quando uma Thread atualiza o valor de “shared\_data” as outras Threads já vão ter iniciado “share\_data” como 0 e posteriormente vão somar +1000, por esse motivo é printado “Shared data = 1000”.

1. **Desenvolva um programa em Python que modela o problema do produtor-consumidor usando threads e as diretivas de sincronização de Python. Seu programa deve ter as seguintes características:**

* O tamanho do buffer deve estar definido em uma constante declarada logo após os imports. Eu quero poder mudar o tamanho do buffer e o seu programa deve continuar funcionando corretamente.
* Sempre que o produtor produzir um item X (que deve ser um inteiro), ele deve escrever na tela "Produced X"
* Sempre que o consumidor consumir um item X (que deve ser um inteiro), ele deve escrever na tela "Consumed X"
* Faça uma função de delay para podermos ver o que acontece no terminal. Caso contrário a tela vai rolar muito rápido e não teremos como ver.
* Se o buffer estiver vazio e for a vez do consumidor executar, ele deve dormir
* Se o buffer estiver cheio e for a vez do produtor executar, ele deve dormir
* Verifique na documentação as diretivas de wait() e notifyAll() da classe Condition

1. **Adicione seu código corrigido ao relatório.**

from threading import Thread, Lock

import time

import random

from threading import Condition

buffer = []

lock = Lock()

MAX\_NUM = 5

condition = Condition()

class ConsumidorThread(Thread):

def run(self):

global buffer

while True:

condition.acquire() #entrar na região crítica

if not buffer: #se não tiver nada no buffer

print("Nada no Buffer, consumidor em espera")

condition.wait() #espera até que notify() seja chamado

print("Produtor adicionou algo ao buffer, e liberou o consumidor")

num = buffer.pop(0) #tira o valor da posição 0

print("Consumido", num)

condition.notify()

condition.release() #sair da região crítica

time.sleep(5)

class ProdutorThread(Thread):

def run(self):

nums = range(MAX\_NUM)

global buffer

while True:

condition.acquire()

if len(buffer) == MAX\_NUM:

print ("Buffer cheio, produtor esperando")

condition.wait()

print ("Espaço liberdo, produtor iniciado")

num = random.choice(nums) #pega um valor aleatorio

buffer.append(num) #adiciona ao buffer

print("Produzido", num)

print(len(buffer))

condition.notify()

condition.release()

time.sleep(2)

ProdutorThread().start()

ConsumidorThread().start()