LISTA 01

16/08/2021

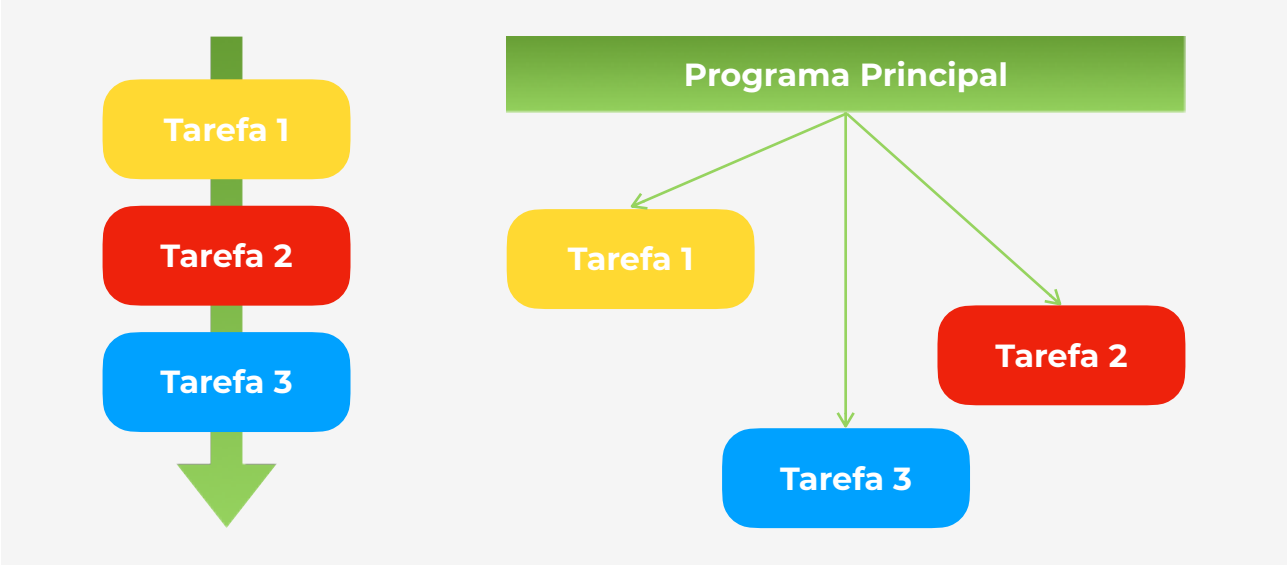
UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
CIÊNCIA DA COMPUTAÇÃO
COMPUTAÇÃO CONCORRENTE (2021.1)

PROFESSORA: SILVANA ROSSETTO
ALUNO: IAGO RAFAEL L. MARTINS | DRE: 119181254

Questão 01 (2,5 pts) Responda as questões abaixo, **justificando todas as respostas**:

a) O que caracteriza que um programa é concorrente e não sequencial?

Um programa é concorrente quando a tarefa única que ele executa pode ser dividida em tarefas menores distribuídas entre *threads* que as realizarão de forma simultânea e independente, o que inclui diferentes linhas de controle. Logo, um programa é concorrente e não sequencial quando contêm mais de um contexto de execução ativo ao mesmo tempo, com o objetivo de executar uma tarefa única (ao fim das pequenas tarefas feitas concomitantemente). Para exemplificar desenhei esse esquema abaixo; à esquerda, observa-se a representação de um programa sequencial e, à direita, de um concorrente.



b) Para quais tipos de problema a programação concorrente é indicada?

É aconselhável que se use a programação concorrente quando há a necessidade de fluxos de execução independentes realizando tarefas diferentes, com uma linha de controle para cada, em processadores com mais de um núcleo. A concorrência também se faz necessária na computação de programas que precisam ser mais otimizados e ter maior performance. Por último, é recomendada ao lidar com dispositivos independentes pela necessidade de se ter um contexto de execução para preservar tarefas que forem eventualmente interrompidas com o sistema operacional.

c) Qual será a *aceleração máxima* de uma aplicação que possui 5 tarefas que consomem o mesmo tempo de processamento, das quais 3 poderão ser executadas de forma concorrente e 2 precisarão continuar sendo executadas de forma sequencial?

Aceleração máxima = $\frac{T_{sequencial}}{T_{concorrente}}$, t = tempo de execução de uma tarefa.

Pensando de forma sequencial, ao rodarmos cinco tarefas, teremos $5t$.

De forma concorrente, as três tarefas serão feitas em $1t$ (por serem feitas ao mesmo tempo e de forma independente uma da outra) e as outras duas sequenciais em $2t$ (por serem dependentes uma da outra, a execução da segunda exige que a primeira já tenha sido realizada).

Logo, $1t + 2t = 3t$ e Aceleração máxima = $\frac{5t}{3t} = \frac{5}{3}$.

d) O que é *seção crítica* do código em um programa concorrente?

Seção crítica é o termo utilizado para se referir à área do código de um programa concorrente que é obrigada a ser rodada sequencialmente (não podendo ser acedida de forma concorrente por mais de um fluxo de execução) por acessar um recurso de memória compartilhada.

e) Como funciona a *sincronização por exclusão mútua*?

Sincronização por exclusão mútua tem como objetivo assegurar o acesso exclusivo de leitura e escrita a um determinado recurso que é compartilhado por uma ou mais *threads*; e serve para previnir o problema da disputa entre *threads* em regiões críticas. Ela funciona basicamente a partir de uma espécie de fechadura (chamada `lock()`) que garante que a próxima *thread* espere para acessar essa região crítica (só quando a fechadura for aberta).

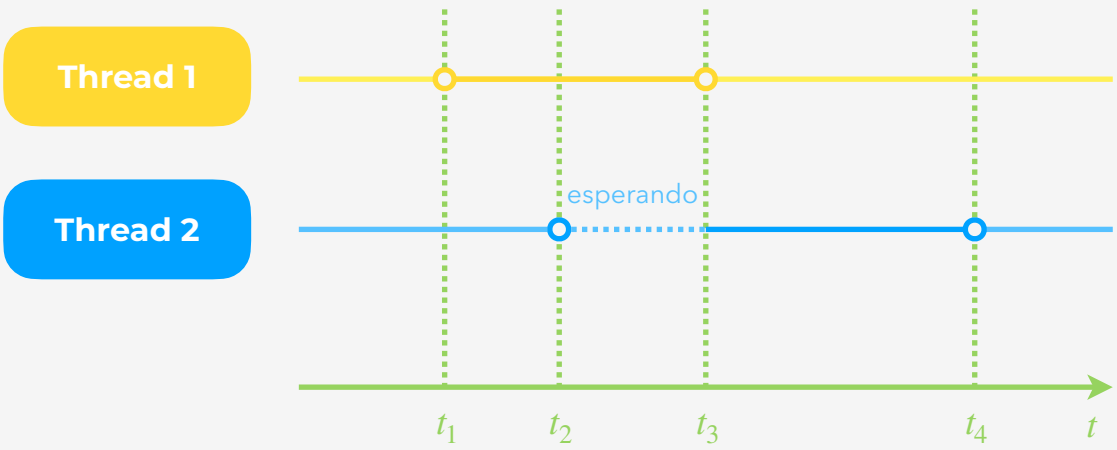
Exemplificando:

t_1 **Thread 1** entra na seção crítica (chama `lock()`);

t_2 **Thread 2** tenta entrar na seção crítica;

t_3 **Thread 1** sai da seção crítica (chama `unlock()`);
Thread 2 entra na seção crítica;

t_4 **Thread 2** sai da seção crítica.



Questão 02 (2,5 pts) Implemente um algoritmo concorrente (apenas a função que será executada por todas as *threads*) para encontrar o valor de π usando a série:

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots\right)$$

Considere que o número de *threads* e o valor de n serão passados como argumento para o algoritmo. **Justifique suas decisões e comente os passos principais do algoritmo.**

```
void *calculaPi(void *arg) {
      //Struct responsável por armazenar os argumentos que forem inseridos na função
      tArgs *args = (tArgs *)arg;
      int nthreads = args -> nthreads; //Número de threads
      int n = args -> n; //Valor de n
      int id = args -> id; //Identificador da thread

      //Inicializa um espaço de memória que armazenará a soma
      double *somaParcial = (double *)malloc(sizeof(double));
      *somaParcial = 0;

      for(int i = id; i <= n; i += nthreads) {
            //Aplicação da série
            *somaParcial += 4 * pow(-1, i) / (2*i + 1); //Calcula a função a
cada rodada do for e soma tudo na variável somaParcial
      }

      pthread_exit((void *)somaParcial); //Retorna a soma parcial de π
}
```

Questão 03 (2,5 pts) Uma aplicação dispara três *threads* ($T1$, $T2$ e $T3$) para execução (códigos mostrados abaixo). Verifique se os vetores $-1, 0, 2, -2, 3, -3, 4$ podem ser impressos na saída padrão quando essa aplicação é executada. Em caso afirmativo, mostre uma sequência de execução das threads que gere valor correspondente.

```
int x=0; //variavel global
(0)      T1:                  T2:              T3:
(1)      x = x-1;             x = x+1;         x = x+1;
(2)      x = x+1;             x = x-1;         if(x == 1)
(3)      x = x-1;                                 printf("%d",x);
(4)      if (x == -1)
(5)         printf("%d",x);
(6)
```

Legenda: Caso afirmativo = ✓; Caso negativo = ✗; $Tx_{(y)} \longrightarrow Tz_{(w)}$ = o primeiro é interrompido pelo segundo (condição de corrida); $Tx_{(y)\rightarrow(z)}$ = as linhas x à z da Tx fazem parte da sequência de execução.

✓ Vetor -1 : Sequência de execução $T1$;

✓ Vetor 0 : Sequência de execução $T1_{(1)\rightarrow(4)}$, $T2_{(1)}$ e $T1_{(5)}$;

✓ Vetor 2 : Sequência de execução $T3_{(1)\rightarrow(2)}$, $T2_{(1)}$ e $T3_{(3)}$;

✓ Vetor -2 : Sequência de execução $T1_{(1)}$, $T2_{(1)} \longrightarrow T1_{(2)}$, $T1_{(3)\rightarrow(4)}$, $T2_{(2)}$ e $T1_{(5)}$;

✓ Vetor 3 : Sequência de execução $T3_{(1)\rightarrow(2)}$, $T2_{(1)} \longrightarrow T1_{(1)}$, $T1_{(2)}$, $T3_{(3)}$;

✗ Vetor -3;

✗ Vetor 4.

Questão 04 (2,5 pts) O algoritmo abaixo apresenta uma proposta de implementação de exclusão mútua com espera ocupada (as threads ficam presas em um *loop* até conseguirem acessar a seção crítica, sem usar *locks*). A solução proposta prevê que a aplicação terá apenas duas *threads* ($T0$ e $T1$). Analise o código e responda as questões abaixo **justificando todas as respostas.**

a) O que acontecerá se a *thread* $T0$ começar a executar primeiro que *thread* $T1$? Ela conseguirá executar a seção crítica?

Com a thread $T0$ começando primeiro, existem dois casos possíveis:

1) Começando bem antes que a thread $T1$, a $T0$ conseguirá passar pelo *while* por satisfazer todas as condições para entrar nele, executar a seção crítica e atualizar o valor da variável queroEntrar_0 para *false*;
2) Começando somente com "uma linha de distância" de $T1$, a $T0$ não conseguirá entrar no *while* antes de $T1$ sobrescrever a variável TURN, o que, além de impossibilitar a $T0$ de entrar no laço, possibilitará a $T1$ satisfazer todas as condições para entrar no *while*, executar a seção crítica e atualizar o valor da variável queroEntrar_1 para *false*.

b) O que acontecerá se a *thread* $T1$ começar a executar primeiro que *thread* $T0$? Ela conseguirá executar a seção crítica?

Com a *thread* $T1$ começando primeiro, existem os mesmos dois casos possíveis da **letra a)** (só que espelhados, ou seja, onde está $T0$ é $T1$ e vice-versa).

c) O que acontecerá se $T0$ e $T1$ executarem juntas?

Ao executarem juntas, uma delas entrará no *while* o que impossibilitará a outra de fazer o mesmo (como melhor explicado na **letra d)**), tornando, assim, esse problema uma exclusão mútua sem necessidade de *locks*.

d) Essa implementação garante exclusão mútua?

Sim, $T0$ e $T1$ nunca executarão a seção crítica ao mesmo tempo: se $T0$ executar, então queroEntrar_0 é *true*. Além disso, ou a variável queroEntrar_1 é *false* (significando que $T1$ executou a seção crítica), ou TURN = 0 (significando que $T1$ está tentando executar a seção crítica, mas está esperando para que a $T0$ acabe de executá-la) ou $T1$ está tentando executar,

depois de atualizar a variável queroEntrar_1 para true, mas antes de atualizar TURN para 0 e esperar pela $T1$ finalizar a execução da seção crítica. Então, se ambos $T0$ e $T1$ estiverem executando a seção crítica, logo conclui-se que queroEntrar_0 = *true* e queroEntrar_1 = *true* e TURN = 0 e TURN = 1. Não existe situação que tanto TURN = 0, quanto TURN = 1, logo não existe situação na qual ambos $T0$ e $T1$ estejam executando a seção crítica.

```
boolean queroEntrar_0 = false, queroEntrar_1 = false; int TURN;
```

T0

```
while(true) {
(1)  queroEntrar_0 = true;
(2)  TURN = 1;
(3)  while(queroEntrar_1 &&
                TURN == 1) { ; }
(4)  //executa a seção crítica
(5)  queroEntrar_0 = false;
(6)  //executa fora da seção crítica
}
```

T1

```
while(true) {
(1)  queroEntrar_1 = true;
(2)  TURN = 0;
(3)  while(queroEntrar_0 &&
                TURN == 0) { ; }
(4)  //executa a seção crítica
(5)  queroEntrar_1 = false;
(6)  //executa fora da seção crítica
}
```