Computação Concorrente - UFRJ - 2021.1 - Lista - Módulo 1

Gabriele Jandres Cavalcanti

DRE: 119159948

Questão 01

a No caso da programação sequencial, existe somente um fluxo de execução. Isto implica que um programa sequencial seja formado por várias instruções que são executadas uma após a outra, de modo que não há chance de que uma instrução seja executada sem que a anterior tenha terminado. Em resumo, a cada instante somente uma instrução é executada.

Já no caso da programação concorrente, existem vários fluxos de execução. Isso implica que um programa concorrente seja dividido em partes que podem ser executadas simultaneamente sem que haja prejuízo de resultado de execução. Cada uma dessas partes é formada por diversas instruções que são executadas de forma concorrente em processos distintos, que cooperam para a execução da tarefa principal. Por essa característica de possibilidade de simultaneidade de tarefas, os programas concorrentes tendem a ganhar em questões de desempenho quando comparados às soluções sequenciais.

b A programação concorrente é indicada para casos em que o resultado final esperado pode ser alcançado independentemente da ordem de execução das diferentes tarefas. Isto é, se tivermos uma situação em que é possível executar diferentes passos simultaneamente, de forma a adiantar a obtenção do resultado, a concorrência é indicada, mesmo que seja o caso de somente uma parte da execução poder ocorrer de forma concorrente. Um exemplo disso seria algum processamento sequencial que tivesse que ser feito com os dados lidos de um arquivo. A parte da leitura poderia ser realizada de forma concorrente porque para o resultado final dessa tarefa (os dados lidos) não importa a ordem de execução.

Em geral, nas seguintes situações podemos utilizar soluções concorrentes:

- **Leitura de arquivo**: se quisermos ler todo o conteúdo do arquivo, normalmente a ordem de leitura não é importante para o processamento final dos dados
- **Multiplicação de matrizes**: o cálculo do valor de cada uma das células da matriz final é independente da ordem de execução
- **Resolução de sistemas lineares**: em alguns casos, cada uma das equações do sistema pode ser resolvida de forma simultânea
- **Cálculo de somas (integrais, séries)**: o cálculo de somas de valores em geral não depende da ordem em que os elementos são somados
- **Cálculo de multiplicações**: o cálculo de multiplicação de valores em geral não depende da ordem em que os elementos são multiplicados

c Consideremos que o tempo total de execução sequencial da aplicação (5 tarefas) é **T**, então o tempo de cada tarefa é de $\frac{T}{5}$. Sabemos que a aceleração é dada por $\frac{T_{sequencial}}{T_{concorrente}}$. A aceleração vai variar de acordo com a quantidade de núcleos de processamento disponíveis, porque isso vai alterar o tempo concorrente pelo fato de termos ou não threads aguardando sua vez dependendo do caso.

Considerando o caso em que temos 3 ou mais núcleos disponíveis, vamos ter as 3 tarefas (threads) executadas simultaneamente. Dessa forma, teremos que o tempo total da aplicação concorrente

será dado por $\frac{2T}{5} + \frac{T}{5}$, onde o primeiro termo é dado pelas duas tarefas sequenciais e o segundo pelas outras três que são executadas concorrentemente. Então a aceleração será $\frac{T}{\frac{3T}{5}} = \frac{5}{3} \approx 1,67$.

Considerando o caso em que temos 2 núcleos disponíveis, vamos ter duas threads executando simultaneamente e assim que uma delas acabar a execução, a terceira será executada. Dessa forma, teremos que o tempo total da aplicação concorrente será aproximadamente dado por $\frac{2T}{5} + \frac{2T}{5}$, onde o primeiro termo é dado pelas duas tarefas sequenciais e o segundo pelas outras três que serão executadas aproximadamente no tempo de duas tarefas, pela limitação do número de núcleos. Então a aceleração será $\frac{T}{\frac{4T}{5}} = \frac{5}{4} \approx 1,25$.

Com apenas um núcleo, não haverá aceleração.

d No contexto de um programa concorrente, dizemos que uma seção crítica é uma área de código do algoritmo que acessa uma variável compartilhada que não pode ser acessada concorrentemente por mais de uma thread, seja para leitura ou escrita. De forma mais tangível, podemos citar o exemplo dado em aula de uma variável global de soma que é acessada por todas as threads, de forma a computar o valor total da soma. A seção crítica do código seria justamente o trecho de acesso a essa variável, uma vez que, se não houver o devido tratamento e uma thread interromper a execução de outra, ocorrem inconsistências no resultado pelo fato de uma única linha de código ser convertida em várias instruções de máquina (ler o valor atual, guardar em um registrador, realizar a soma e guardar o valor novamente).

e A sincronização por exclusão mútua é um mecanismo que permite o controle da ordem na qual as operações ocorrem em diferentes threads. Ela garante que as seções críticas do código não sejam executadas por mais de uma thread simultaneamente. Como o próprio nome desse mecanismo de sincronização diz, há uma exclusão mútua, de forma que quando uma thread inicia a execução da seção crítica, ocorre o "travamento"da possibilidade de outras threads executarem esse trecho do código até que a thread que está executando termine a execução. Fazendo uma analogia com um fato do cotidiano, duas threads podem ser consideradas duas pessoas tentando ligar simultaneamente para um mesmo número de telefone, que seria considerado a seção crítica: quando uma pessoa é atendida, a outra recebe sinal de ocupado e só consegue contato com esse número de telefone quando a linha estiver desocupada, de modo que duas pessoas nunca conseguem falar com o mesmo número ao mesmo tempo.

Questão 02

Para calcular o valor de π de forma concorrente, a função abaixo poderia ser executada pelas threads:

```
void* calculatePi(void *arg) {
  int id = (long int) arg; // identificador da thread
  double *localSum; // variavel local de soma de elementos
  int sign; // variavel de controle do sinal de um elemento no somatorio

  localSum = (double *) malloc(sizeof(double));
  if (localSum == NULL) {
    fprintf(stderr, "Erro na alocacao para a soma local");
    exit(1);
  }
  *localSum = 0;

  // cada thread soma elementos de forma intercalada
  for(int i = id; i < n; i+=nthreads) {
    sign = i % 2 == 0 ? 1 : -1;
    *localSum += (sign * ((double) 1 / (2 * i + 1)));
  }

  // retorna o resultado da soma local
  pthread_exit((void *) localSum);
}
```

As variáveis n e nthreads usadas na função são variáveis globais às quais a thread terá acesso ao ser executada.

Além disso, para que a multiplicação por 4 não ocorra em todos os processamentos das threads, na função principal (*main*), a variável pi deve ser usada para somar todos os valores de retorno das threads quando usarmos a *pthread_join* e em seguida multiplicada por 4:

```
pi = 4 * pi;
```

Dessa forma, o valor de pi calculado estará correto ao ser exibido na main.

Questão 03

Valor -1

Sim, o valor -1 pode ser exibido na saída padrão quando a aplicação é executada. Isso ocorre quando as 5 linhas da T1 (T1(1), T1(2), T1(3), T1(4) E T1(5)) são executadas sequencialmente. Depois do resultado esperado, a ordem de execução de T2 e T3 é indiferente.

Valor 0

Sim, o valor 0 pode ser exibido na saída padrão quando a aplicação é executada. Isso ocorre quando T1(1), T1(2), T1(3), T1(4), T2(1) e T1(5) são executadas. Dessa forma, quando T1(4) é executada, o valor de x é -1, mas, como T2(1) será executada antes da impressão do valor de x, o valor impresso em T1(5) será 0. Depois do resultado esperado, a ordem de execução do restante das instruções é indiferente.

Valor 2

Sim, o valor 2 pode ser exibido na saída padrão quando a aplicação é executada. Isso ocorre quando T3(1), T3(2), T2(1) e T3(3) são executadas. Dessa forma, quando T3(2) é executada, o valor de x é 1, mas, como T2(1) será executada antes da impressão do valor de x, o valor impresso em T3(3) será 2. Depois do resultado esperado, a ordem de execução do restante das instruções é indiferente.

Valor -2

Sim, o valor -2 pode ser exibido na saída padrão quando a aplicação é executada. Isso ocorre quando T1(1), T1(2), T1(3) & T2(1), T1(4), T2(2) e T1(5) são executadas. A diferença aqui está no fato de T1(3) e T2(1) estarem sendo simultaneamente executadas, devido ao fato de uma única linha de código ser quebrada em várias instruções de máquina (nesse caso há a leitura do valor de x, o armazenamento desse valor no registrador, a realização da operação e a atribuição do novo valor de x). Logo, nesse caso, ambas leem o valor de $x = 0$ e realizam suas operações, T2(1) calcula o valor 1 para x e o atribui, mas T1(3) calcula o valor -1, e sobrescreve o valor de x, então ao final da execução dessas diversas operações em baixo nível $x = -1$. Com isso, a condição de T1(4) é verdadeira, e ao executarmos T2(2) estaremos subtraindo o valor 1 e obtendo -2 como resultado em x, que será exibido em T1(5).

Valor 3

Sim, o valor 3 pode ser exibido na saída padrão quando a aplicação é executada. Isso ocorre quando T1(1) & T3(1), T3(2), T2(1), T1(2) e T3(3) são executadas. Novamente teremos duas linhas T1(1) e T3(1) de threads distintas executando simultaneamente, devido ao fato da quebra de uma mesma instrução em alto nível em várias de baixo nível. Em ambas há a leitura do valor de $x = 0$ no começo, T1(1) calcula o valor -1 para x e o atribui, mas T3(1) calcula o valor 1 e sobrescreve o valor de x, logo ao final da execução desses passos $x = 1$. Com isso, a condição de T3(2) é verdadeira, e ao executarmos T2(1) e T1(2) estaremos somando o valor 2 e obtendo 3 como resultado em x, que será exibido em T3(3).

Valor -3

Não, o valor -3 não pode ser exibido na saída padrão quando a aplicação é executada. Esse valor é impossibilitado pelas condicionais para exibição que temos, porque, para T1(4) ser executada, duas subtrações já foram executadas por estarem na mesma thread, então sobra uma subtração que pode ser feita, fazendo com que o menor valor alcançado seja -2. Da mesma forma, para a condicional em T3(2), o valor de x é 1, de forma que para que o valor -3 seja alcançado devem ocorrer 4 subtrações, o que ultrapassa as subtrações existentes na aplicação. Logo é impossível.

Valor 4

Não, o valor 4 não pode ser exibido na saída padrão quando a aplicação é executada. Da mesma forma, esse valor é impossibilitado pelas condicionais para exibição que temos, porque, para T1(4) ser executada e retornar true, o valor de x é -1, necessitando de 5 adições para chegar no valor 4, o que ultrapassa as somas existentes na aplicação. E para a condicional em T3(2), o valor de x já é 1 e uma soma já foi executada, mas necessitaria de mais 3 adições, o que também ultrapassa as duas somas restantes na aplicação. Logo é impossível.

Questão 04

a Considerando que as linhas estarão sendo executadas na ordem em que aparecem, teremos alguns possíveis cenários. Se a thread T0 começar a executar primeiro que a thread T1 e conseguir ser executada sem ser interrompida, teremos que o valor de $queroEntrar_0$ será setado como $true$, indicando que T0 quer executar a seção crítica, o valor de $TURN$ será setado como 1, indicando que T1 tem direito de execução, e o valor de $queroEntrar_1$ será $false$, assumindo que T1 não foi executada e por isso ainda não quer entrar na seção crítica.

Logo, a condição avaliada no while da linha 3 fornecerá $false$ como retorno, já que se trata de um and e o primeiro booleano tem valor $false$ e o segundo $true$, o que indica que apesar de T1 ter direito de execução, ela ainda não manifestou interesse em executar a seção crítica, logo não é necessário que T0 fique em espera.

Então a thread T0 não entrará nesse loop e conseguirá chegar na linha 4 e executar a seção crítica. Após executar esse trecho de código, $queroEntrar_0$ voltará a ser $false$, porque a thread já conseguiu executar a seção crítica.

Em outros casos, em que a thread T0 começa a ser executada mas vai ser interrompida pela thread T1, pode acontecer de T0 ficar presa no loop de espera pelo fato de ser a "vez"de T1 executar **E** T1 ter manifestado interesse em executar a seção crítica, mas depois de sua execução, $queroEntrar_1$ voltará a ser $false$ e T0 sairá do loop, conseguindo executar a seção crítica.

b Novamente, considerando que as linhas estarão sendo executadas na ordem em que aparecem, teremos alguns possíveis cenários. Se a thread T1 começar a executar primeiro que a thread T0 e conseguir ser executada sem ser interrompida, teremos que o valor de $queroEntrar_1$ será setado como $true$, indicando que T1 quer executar a seção crítica, o valor de $TURN$ será setado como 0, indicando que T0 tem direito de execução, e o valor de $queroEntrar_0$ será $false$, assumindo que T0 não foi executada e por isso ainda não quer entrar na seção crítica.

Logo, a condição avaliada no while da linha 3 fornecerá $false$ como retorno, já que se trata de um and e o primeiro booleano tem valor $false$ e o segundo $true$, o que indica que apesar de T0 ter direito de execução, ela ainda não manifestou interesse em executar a seção crítica, logo não é necessário que T1 fique em espera.

Então a thread T1 não entrará nesse loop e conseguirá chegar na linha 4 e executar a seção crítica. Após executar esse trecho de código, $queroEntrar_1$ voltará a ser $false$, porque a thread já conseguiu executar a seção crítica.

Em outros casos, em que a thread T1 começa a ser executada mas vai ser interrompida pela thread T0, pode acontecer de T1 ficar presa no loop de espera pelo fato de ser a "vez"de T0 executar **E** T0

ter manifestado interesse em executar a seção crítica, mas depois de sua execução, $queroEntrar_0$ voltará a ser $false$ e T1 sairá do loop, conseguindo executar a seção crítica.

c Se as duas threads executarem juntas, poderemos ter vários casos, já que não necessariamente cada thread executará "uma linha por vez".

Se for o caso de as threads executarem uma linha por vez, teremos simultaneamente $queroEntrar_0$ e $queroEntrar_1$ como $true$, indicando que ambas as threads querem executar a seção crítica. Mas, na linha 2 ambas as threads alteram a variável $TURN$, que é global, assim, se T0 alterar primeiro o valor, T1 irá sobrescrevê-lo, e o valor de $TURN$ será 0, logo a condição do while da linha 3 de T0 será avaliada como $false$ (porque T1 disse que era a "vez"de T0 executar), fazendo com que T0 não fique presa no loop, mas a condição da mesma linha de T1 será avaliada como $true$, fazendo com que T1 fique presa no loop enquanto $queroEntrar_0$ e $TURN == 0$ forem $true$ (é a VEZ de T0 executar a seção crítica e T0 QUER executar).

Da mesma forma, se T1 alterar primeiro o valor, T0 irá sobrescrevê-lo, e o valor de $TURN$ será 1, logo a condição do while da linha 3 de T0 será avaliada como $true$ (porque T0 disse que era a "vez"de T1 executar), fazendo com que T0 fique presa no loop enquanto $queroEntrar_1$ e $TURN == 1$ forem $true$ (é a VEZ de T1 executar a seção crítica e T1 QUER executar), mas a condição da mesma linha de T1 será avaliada como $false$, fazendo com que T1 não fique presa no loop. Com isso, apenas uma das threads executará a seção crítica de cada vez.

No primeiro caso, quando T0 finalizar a execução, $queroEntrar_0$ voltará a ser $false$ (indicando que T0 terminou a execução), liberando T1 do loop e possibilitando que execute a seção crítica. No segundo caso, quando T1 finalizar a execução, $queroEntrar_1$ voltará a ser $false$ (indicando que T1 terminou a execução), liberando T0 do loop e possibilitando que execute a seção crítica.

Mesmo que não haja essa execução "linha a linha"de cada thread (T0 executa a linha 1, T1 executa linha 1, T0 executa linha 2, T1 executa linha 2...), ambas as threads vão conseguir executar a seção crítica devido à condição *and* do while da linha 3, conforme será melhor explicado no item a seguir.

d Desde que a ordem de execução das linhas seja respeitada, essa implementação garante exclusão mútua. Se não houver respeito à ordem de execução, ambas as threads podem ir direto à linha 3 e as condições das duas threads serão $false$ e nenhuma ficará em espera. Com isso, as duas executarão a seção crítica, o que é uma quebra da exclusão mútua.

No caso de respeito à ordem das linhas, a exclusão mútua é garantida através das variáveis $queroEntrar_0$ e $queroEntrar_1$. Se há empate, então é usada a variável $TURN$, que dá a vez para a outra thread.

O algoritmo satisfaz os três critérios essenciais para resolver o problema de condições de corrida ruins por meio da exclusão mútua: T0 e T1 nunca executarão a seção crítica ao mesmo tempo, uma thread não consegue "reentrar"na seção crítica antes de outra que sinalizou o interesse em executar e uma thread nunca espera mais do que uma volta para entrar na seção crítica.

De forma mais aprofundada, podemos dizer que mesmo em diferentes opções de ordem de execução, como a condição do loop da linha 3 depende da variável $TURN$ em ambas as threads e a condição em cada uma das threads avalia se o valor de $TURN$ é igual a dois valores distintos, não haverá o caso em que ambas as condicionais avaliem $true$ ou o caso em que ambas avaliem $false$. Sendo assim, nunca vai acontecer de ambas as threads ficarem presas no loop em espera e nem ambas as threads saírem ao mesmo tempo do loop.

Além disso, a linha 5 de ambas as threads garante que após a thread em questão conseguir executar a seção crítica com sucesso, ela irá liberar a outra thread do loop, porque nos dois casos possíveis (para T0 ou para T1), ela irá setar $queroEntrar_x$ como $false$, fazendo com que a condição do while da linha 3 da outra thread seja avaliada como $false$, pelo fato do primeiro booleano ser $false$.

Esse controle de variáveis garante que mesmo que as threads sejam executadas várias vezes, não haverá casos de espera infinita nem de execução simultânea.