Lista 2 de Computação Concorrente

João Pedro Silveira Gonçalves

March 2022

1 Questão 1

1.1 Letra a

O semáforo x é utilizado na implementação da exclusão mútua.
O semáforo h e s são utilizados na implementação da sincronização condicional. O h faz com que as threads que deram wait sejam retidas até que recebam um sinal de notify() ou notifyAll() e que estes desbloqueiem as threads que deram wait. Já o s faz com que só se termine a execução de notify() e notifyAll() quando todas as respectivas(uma única no caso de notify) threads que deram wait() tenham sido desbloqueadas.

1.2 Letra b

Sim, pois wait espera uma notificação de notify() e notifyALL() se bloqueia enquanto não recebe um sinal;e enquanto espera pelo sinal, desbloqueia o lock; mas depois de receber o sinal, readquire o lock, assim como aprendemos em sala. Ademais, notify() e notifyAll() executam sua tarefa de verificar se há alguma thread bloqueada e caso haja, a desbloqueiam, como deveriam.

1.3 Letra c

Não, o uso de 2 variáveis condicionais(além da de exclusão mútua) praticamente extingue essa possibilidade. Há com isso uma grande sincronização de uma thread ao avanço da outra, ao passo que uma não pode acumular sinais, pois há outras de outro tipo que dependem desses sinais para liberar sinais que a thread original precisa para seguir.

1.4 Letra d

Sim, há condições para ocorrência de deadlock. Sim, pois a thread que deu wait() precisa de h, que é incrementado pela thread que está dando o sinal, porém essa thread que está dando o sinal só pode prosseguir(pode estar bloqueada) com base em s, que, por sua vez, somente é incrementado pela thread que bloqueada em espera de h após esta receber h.

2 Questão 2

2.1 Letra a

Pode ter ocorrido a seguinte sequências de acontecimentos organizados pelo scheduler do sistema operacional:
Observação: Começaremos nossa análise após a.start() e b.start() e com balance¡-500.

1. Iniciou-se em runThread, executando sua linha 1. Teremos o seguinte estado:
   amount = 100
   balance = 500

2. Em runThread: Executou-se a linha 2, e como balance(500) ¿= amount(100), passou-se no teste do if.

3. Em modifierThread: Executou-se a linha 1 e teremos o estado
   balance= 10

4. Em runThread: Executou-se a linha 3:
   balance = balance - amount, ou seja, balance = 10 - 100 = -90

5. Na thread principal: imprimiu-se saldo = -90 (negativo)

2.2 Letra b

Se observamos a sequência exposta acima que ilustra uma situação errônea, veremos que o problema que resultou em um saldo negativo foi gerado pela execução concorrente e paralela de duas threads que contém partes que modificam o valor de balance, que é uma variável compartilhada pelas threads(área crítica sem exclusão mútua).

Para evitar que isso se repita e cause resultados errôneos e imprevisíveis seria importante montar um sistema de exclusão mútua de forma a evitar que funções em threads diferentes que modificam o valor de variáveis compartilhadas como balance sejam executadas paralelamente/concorrentemente, ou seja, criar mecanismos que impeçam o acesso simultâneo de threads distintas à regiões de dados compartilhadas nas chamadas áreas críticas do código.

A melhor forma de fazê-lo seria remodelando o código para que apresente o padrão leitores x escritores visto em sala utilizando variáveis de condição. Assim, poderia haver :

- Threads leitoras: apenas são responsáveis pela chamada de funções que consultam as informações da conta.
  Exemplo: consulta de saldo por meio de getBalance().
  Essas threads leitoras poderiam executar em paralelo, pois não modificariam nenhum valor de estado da conta, apenas os consultaria. É esperado que se não tenha havido nenhuma operação, os valores de estado

da conta devam permanecer os mesmos em todas as consultas. Ademais, uma thread leitora(consulta de variáveis de estado da conta) só poderia executar depois ou antes da execução de uma thread escritora(alteração das variáveis de estado compartilhadas entre threads), mas nunca simultaneamente, a fim de evitar inconsistências no acesso às informações da conta.

- Threads escritoras: apenas são responsáveis pela chamada de funções que alteram os valores de estado da conta.
  Exemplo: atribuir novo saldo a uma conta por meio de setBalance()
  Diferentemente das leitoras, essas threads não poderiam executar em paralelo, justamente por modificar as variáveis de estado compartilhadas de um objeto, no caso, conta bancária. Assim, por mecanismos de exclusão mútua e sincronização condicional, seria fundamental implementar que apenas uma thread escritora possa executar por vez uma área crítica onde modifica o as variáveis compartilhadas entre as threads. É fundamental que a modificação de variáveis compartilhadas ocorra de forma sequencial e não concorrente como no caso do exemplo do item acima onde foram ilustrados os problemas desse caminho(modificação concorrente das variáveis de estado da conta). Por fim, uma thread escritora não pode modificar as variáveis compartilhadas de estado da conta enquanto uma thread leitora estiver os acessando. Isso significa que não é possível observar o saldo e realizar um saque simultaneamente. Na verdade, esse fato é uma consequência direta do fato de que se um escritor está escrevendo, nenhum leitor pode ler/acessar a mesma região crítica de dados compartilhada sendo modificada.

Somente assim se evitaria as inconsistências e erros consequentes de condições de corrida como visto no item anterior e evitaria-se a chegada em um estado de deadlock.

3 Questão 3

4 Questões 4

Para criar uma solução, podemos pensar da seguinte forma:

Inicialmente, de acordo com o enunciado, podemos observar que há 2 processos básicos: o de atendimento à uma requisição de impressão e o de geração de uma requisição. Devido a essa natureza distinta e oposta, podemos atribuir suas realizações a fluxos de execução também distintos. Vamos analisar agora como se dará de fato essa divisão do trabalho em fluxos distintos.

Do enunciado, temos que o atendimento à uma requisição de impressão será realizado por uma thread de gerenciamento da impressora, atividade alvo do código. Essa thread realizará um loop de atividades que consiste em esperar por novos materiais a serem impressos(i.e. requisições de impressão) e imprimi-los na ordem em que foram recebidos. Assim, se há uma fila de impressão que

contém esses materiais a serem impressos(uma fila de strings) e organizado na ordem em que foram recebidos, podemos imaginar:

- **uma requisição:** como um material do tipo string a ser impresso
- **geração de uma nova requisição de impressão:** como a inserção de uma string em alguma posição dessa fila de impressão para que seu conteúdo seja impress
- **o atendimento a uma requisição:** a impressão do conteúdo de uma string da fila de impressão e posteriormente sua retirada dessa fila

Veja como há a noção de inserção e retirada de elementos de uma região de memória compartilhada entre threads executando tarefas distintas. Isso nos sugere a aplicação do padrão de concorrência visto em aula de produtores x consumidores, em que se divide o trabalho entre: threads que apenas inserem objetos em uma região compartilhada, as produtoras; e threads que apenas retiram esses objetos dessa região compartilhada que foram inseridos nelas pelas consumidoras. Portanto, nesse caso, teremos as seguintes threads e suas funcionalidades:

- Thread Produtura: responsável pela geração de requisições de impressão. Essa thread fará em loop:
  - Se a fila de impressão não estiver cheia:
    Depositará uma requisição de impressão na fila de impressão na próxima posição vazia da fila.
    Observação: a fila de impressão não está cheia quando o número de posições vazias é não nulo(i.e. o número de posições ocupadas não é igual ao tamanho da fila); e uma posição vazia da fila de impressão é apenas um ponteiro para NULL.
  - Se a fila já estiver cheia: Se bloqueará até que haja uma posição vazia para se depositar a requisição e só então seguirá com sua execução normal(inserir string na fila).
    Observação: a fila de impressão está cheia quando o número de posições vazias é nulo(i.e. o número de posições ocupadas é igual ao tamanho da fila).
- Thread Consumidora: responsável pelo atendimento à requisições de impressão:
  Essa thread fará em loop:
  - Se a fila de impressão não estiver vazia:
    - Imprimirá o conteúdo da primeira posição não vazia da fila, ou seja, imprimirá a requisição de impressão.
    - Retirará da fila de impressão essa requisição que já foi impressa.

– Se a fila de impressão estiver vazia:
  Como não há nada a ser impresso, se bloqueará até que haja a inserção de uma requisição de impressão na fila de impressão por uma thread produtora.

Seguindo, deve-se prosseguir a detalhes da implementação. Para isso, utilizaremos a linguagem C e o mecanismo de semáforos para garantir a sincronização condicional e exclusão mútua, necessário para casos em que há mais de 1 threads produtoras e/ou consumidoras; tudo isso visto em sala, na aula 2 da semana 1 do módulo 3 do curso. Nessa aula vimos a facilidade do uso de semáforos para implementação desse padrão consumidor x produtor. É em razão disso que escolheremos semáforos para sincronizar o trabalho entre threads distintas de um mesmo tipo e/ou de tipos distintos, em detrimento do uso de variáveis condicionais compartilhadas.

A facilidade do uso de semáforos para essa aplicação(padrão consumidor x produtor) se dá justamente pela possibilidade do acúmulo de sinais neles, que facilita a sincronização condicional entre threads produtoras e consumidoras. Se tivermos um semáforo cujo número de sinais acumulados corresponde ao número de número de slots vazios na fila de impressão e outro cujo número de sinais acumulados corresponde ao número de número de slots cheios/ocupados/preenchidos, teremos a qualquer momento informações sobre o estado dessa fila, isto é, o número de requisições que estão aguardando a serem atendidas, que corresponde ao número de slots cheios(mais especificamente, sinais acumulados no semáforo slotsCheios) em determinado momento. Vamos imprimir essa informação sempre após a retirada de uma requisição de impressão fa fila de impressão. É dessa forma que responderemos à questão 5.

As dimensões da fila de impressão serão dadas no argumento da chamada da função main do arquivo que contém o código. Essas dimensões são: o número de frases ou palavras que cabem nessa fila e quantos caractereres uma frase/palavra pode ter no máximo. O uso do código será dado por:

```
./questao4 <tamanho da fila de impressão>
<limite de caracteres em uma frase>
<número de threads consumidoras>
<número de threads produtoras>
```

Seguindo, um array de strings representará a fila de impressão. Os fatos da inserção/produção de uma requisição ser feita sempre na posição vazia mais próxima(de menor índice em relação a 0); e dá remoção/consumo de uma requisição ser feita na posição já preenchida mais próxima garantem que a ordem de recebimento seja guardada, o que possibilida que as requisições sejam atendidas na ordem em que foram recebidas, atendendo aos requisitos descritos no enunciado.

Código para questão está em:

https://github.com/jps2002/computacaoConcorrente/tree/main/lista2

Observação: na versão que está o código cria threads produtoras, que realizam em loop a atividade de inserção da string "hey" na fila de impressão;

e threads consumidoras que, realizam em loop a atividade de remoção de requisições de impressão da fila de impressão. Há a impressão de todos os logs de execução. A interrupção da execução para análise deve ser feita por meio do comando CRTL + C no terminal de execução.

5 Questão 5

Para melhor atender à questão 5, foram pensados os seguintes requisitos: Como se quer consultar o estado da fila de impressão, é importante ter uma função que leia e imprima esse estado, realizando essa consulta. Entretanto, esta consulta não pode ser concomitante à execução de uma atividade de remoção ou inserção de requisições na fila, a fim de evitar inconsistências e condições de corrida. Assim, para atender a esses requisitos, modificou-se o código, que estava em um padrão produtor x consumidor, para incorporar elementos do padrão escritor x leitor visto em sala. Nesse caso, há 3 funções básicas:

- uma de inserção de requisições de impressão na fila de impressão, realizada por threads produtoras(tipo de escritora). Trata-se de uma atividade de escrita.
- uma de gerenciamento da fila impressão, responsável pela execução de requisições e a posterior retirada destas da fila de impressão, realizada por threads consumidoras(tipo de escritora). Trata-se de uma atividade de escrita.
- uma de leitura da fila de impressão, que imprime seu estado, mostrando quantas requisições estão aguardando para serem atendidas. Trata-se de uma atividade de leitura.
  É com essa função que se espera poder atender ao requisito da questão 5.

Essa função de consulta à fila está presente no código de duas formas(Leitura() e LeituraModificada()) de acordo com seus dois possíveis usos:

- Seria interessante que após qualquer inserção ou remoção de requisições de impressão na fila, o estado dessa fila fosse impresso para o usuário.
  Assim, criou-se a função LeituraModificada() que deve imprimir o estado da fila de impressão após cada execução de uma atividade de escrita por quaisquer threads produtoras ou consumidoras.
- Seria que houvesse uma forma simples de consulta ao estado da fila.
  Assim, criou-se a função Leitura() cuja função é simplesmente imprimir o estado da fila quando possível e chamada. Isso significa que em uma thread(seja ela produtora ou consumidora), quando essa função é chamada chamada, espera-se por um momento que não haja alguma função de escrita executando e ai sim imprime o estado da fila de impressão. Observe que nessa versão não há a preocupação com a exatidão do momento de impressão do estado da fila de impressão, o que faz com que essa consulta seja feita e seus resultados impressos muito após a chamada dessa função.

Observe que as modificações foram feitas de forma a manter o padrão consumidor x produtor responsável pelo atendimento às condições da questão 4, mas incorporar também elementos do padrão leitor(consulta à quantidade de elementos na fila) x escritor(inserção ou remoção de elementos na fila).

Por fim: Na versão que está, o código possui cria threads produtoras e consumidoras. No código da tarefa das funções que essas threads realizaram há um caso de teste:

- No caso das threads produtoras: executar-se-á, por thread, a chamada 3 vezes da função Insere(), que deve resultar na inserção, por thread de uma string qualquer de teste "hey" na fila de impressão.
- No caso das threads consumidoras: executar-se-á, por thread, a chamada 3 vezes da função Retira(), que deve resultar na remoção, por thread, na remoção de 3 requisições de impressão.

Há também em ambas, de forma comentada um outro caso de teste:

- No caso das threads produtoras: execução em loop da função Insere() para inserir a requisição de impressão da string de teste "hey" repetidas vezes na fila de impressão
- No caso das threads consumidoras: execução em loop da função Retira() para remover requisições de impressão da fila de impressão, na ordem em que foram recebidas.

Código para questão está em:
https://github.com/jps2002/computacaoConcorrente/tree/main/lista2