MAC 438 - Programação Concorrente - Primeiro Semestre de 2006

Quarto Exercício-Programa: `ServContas`

Data de Entrega: 27 de junho de 2006

Este exercício deve ser desenvolvido em equipes de duas pessoas, a fim de suscitar discussão. Dúvidas sobre o enunciado devem ser enviadas para a lista de discussão de MAC-438.

1 O problema

Você deve implementar o programa `ServContas`, que gerencia N contas bancárias. As contas são numeradas de 1 a N. Para cada conta são guardadas somente duas informações: o saldo atual e o limite mínimo do saldo.

O programa `ServContas` é um servidor que aceita conexões TCP num port especificado na linha de comando. Conectando-se ao servidor, um cliente pode efetuar as operações saldo, depósito, saque, transferência, info e limite, conforme o exemplo abaixo (esse exemplo supõe que o servidor está aguardando conexões no port 6789 da máquina `algum.host`):

```
$ telnet algum.host 6789
Trying xxx.xxx.xxx.xxx...
Connected to algum.host.
Escape character is '^]'.
Bem vindo!
*saldo 1 4 6
conta 1: 100
conta 4: 1000
conta 6: -2000
*deposito 200 1
conta 1: 100 -> 300
*transf 50 4 1
conta 4: 1000 -> 950
conta 1: 300 -> 350
*saque 100 1
conta 1: 350 -> 250
*saque 1000 6
saque ultrapassaria limite
*info 6
conta 6: saldo -2000, limite -2500
*limite -5000 6
limite da conta 6: -2500 -> -5000
*saque 2000 6
conta 6: -2000 -> -3000
*quit
Volte sempre!
$
```

As três linhas abaixo da chamada `telnet` foram geradas pelo próprio `telnet`. O "`*`" é o prompt do servidor de contas. As linhas iniciadas com "`*`" foram digitadas pelo usuário do `telnet` (exceto pelo primeiro caracter da linha, que foi mandado pelo servidor). Todas as outras linhas foram enviadas pelo servidor. Note que o usuário do telnet pode manipular qualquer conta (pense nele como um funcionário do banco especialmente autorizado para isso).

2 Requisitos da solução

1. O servidor de contas deve ser multithreaded. Para cada conexão TCP recebida, o programa `ServContas` deve criar ou alocar uma thread que atende o cliente naquela conexão. "Criar ou alocar uma thread" significa que o servidor deverá ser capaz de trabalhar tanto no modo thread-per-session como com um thread pool (mais detalhes na próxima seção).

   No arquivo `ServidorPrimos.java` você encontrará um exemplo de programa que cria uma thread por cliente. O programa `ServidorPrimos` é um servidor que aceita uma seqüência de números (através de uma conexão TCP) e, para cada número, devolve ao cliente uma string que informa se o número é primo ou não. Para cada conexão recebida ele cria uma thread para lidar com a seqüência de números oriunda desta conexão. O recebimento de uma linha vazia (uma `String` vazia) indica que o cliente não tem mais números para fornecer ao servidor.

2. As operações saldo, depósito, saque, transferência, info e limite devem ter a propriedade de *isolação*, ou seja: mesmo que várias operações estejam sendo executadas concorrentemente, o resultado de uma dada operação *op* é o que seria obtido caso ela fosse a única em execução no servidor e as operações concorrentes com *op* tivessem rodado antes de *op* (algumas dessas operações) ou depois de *op* (as demais operações). Em outras palavras, a propriedade de isolação exige que o resultado da execução de operações concorrentes seja igual ao resultado de alguma execução serial (numa ordem qualquer) dessas mesmas operações.

   Para deixar isso mais concreto, pense num cliente transferindo X da conta A para a conta B. O servidor pode passar por um estado intermediário, em que X já foi subtraído do saldo de A mas ainda não foi adicionado ao saldo de B, mas esse estado não deve ser visível para nenhum outro cliente! Se, concorrentemente com a transferência, algum cliente pedir (numa só operação) os saldos das duas contas, ele deve receber os dois saldos antes da transferência ou os dois saldos depois da transferência, mas nunca um estado intermediário.

   O programa `ServContas` deve garantir a isolação das operações usando read/write locks (um lock por conta) e o protocolo two-phase locking (2PL) discutido em classe. Deve ser utilizada a implementação de read/write lock presente no pacote `java.util.concurrent.locks`.

3. Com o uso de locks, seu servidor pode entrar em deadlock se você não escrevê-lo tomando os devidos cuidados. O `ServContas` deve utilizar uma estratégia de prevenção de deadlocks baseada na ordenação das aquisições de lock: as operações saldo e transferência, que adquirem mais de um lock de conta, devem adquirir os locks sempre numa mesma ordem (em ordem crescente do número da conta, por exemplo).

4. O `ServContas` aceita na linha de comando o argumento opcional "`-cozinh`", que desliga a isolação. Com esse argumento, o servidor ignora o protocolo 2PL e roda em "modo cozinhado", sem adquirir/liberar locks de contas.

5. O `ServContas` aceita na linha de comando o argumento opcional "`-suicida`", que desliga a prevenção de deadlock. Com esse argumento, o servidor adquire os locks de contas na ordem em que as contas aparecem nos operações requisitadas pelos clientes. O comando `saldo 2 3 1` adquiriria o lock da conta 2, depois o da conta 3 e, por último, o da conta 1.

6. Além do programa `ServContas`, você deve implementar um conjunto de clientes que façam seu servidor funcionar mal quando chamado com "`-cozinh`" ou com "`-suicida`".

   - Para o "`-cozinh`", implemente um cliente que fica fazendo sucessivas operações de transferência entre duas contas e outro que repete a operação que dá o saldo dessas duas contas até obter um par de saldos que não deveria ser visto.
   - Para o "`-suicida`", implemente um par de clientes que faz o servidor entrar em deadlock.

7. Como o acesso às contas na memória é muito rápido, pode ser difícil fazer os problemas acima se manifestarem. Para facilitar isso, o `ServContas` pode reconhecer mais um argumento na linha de

comando: "`-atraso n`" (onde `n` é um inteiro positivo) faz com que cada acesso (leitura ou escrita) ao saldo ou ao limite de uma conta leve `n` milisegundos. Se for chamado com esse argumento, o servidor inserirá um `sleep(n)` antes de cada acesso ao saldo ou ao limite de uma conta. Junto com o servidor e com os clientes você deverá incluir um relatório contando se foi ou não foi difícil fazer esses problemas aparecerem, a partir de que valor de `n` eles começam a aparecer, etc.

3 Modelos de programação de servidores multithreaded

O exemplo `ServidorPrimos.java` é um servidor multithreaded que cria uma thread para tratar cada sessão com um cliente. Criar uma thread para cada sessão é um dos possíveis modelos de programação de servidores multithreaded. Os três modelos mais freqüentemente usados são:

Thread-per-request: Este modelo de programação cria uma nova thread para cada requisição de serviço proveniente de algum cliente. A thread efetua o serviço requisitado, manda uma mensagem de resposta para o cliente e encerra sua execução. O modelo thread-per-request é útil para servidores que tratam requisições de longa duração (como consultas a um banco de dados) emitidas por múltiplos clientes. Ele é menos útil no caso de requisições de curta duração, devido ao overhead de criação de uma thread a cada requisição. Pode também consumir muitos recursos do S.O. caso muitos clientes façam requisições simultâneas.

Thread-per-session: Esta variação do modelo thread-per-request amortiza por várias requisições o custo de criação de uma thread. Para cada cliente que se conecta com o servidor é criada uma nova thread, que tem a duração da sessão do cliente. O modelo thread-per-session é útil no caso de múltiplos clientes que travam conversações demoradas com o servidor. Não é útil se os clientes fazem só uma requisição, pois recai no modelo thread-per-request.

Thread pool: Esta variação dos modelos anteriores elimina o custo de criação de threads empregando um pool de worker threads criadas na inicialização do servidor. É útil para servidores que querem colocar um limite na quantidade de recursos do S.O. que eles consomem. A alocação de threads do pool pode ser feita por requisição ou por sessão. No primeiro caso, o tamanho do pool determina o número de requisições que podem ser processadas concorrentemente. As requisições concorrentes que ultrapassarem este número devem ser enfileiradas para posterior tratamento. No segundo caso, o tamanho do pool determina o número de sessões simultâneas com clientes. Os clientes que ultrapassarem este número devem ser enfileirados para posterior atendimento.

4 Requisitos adicionais

O `ServContas` deve ser capaz de operar tanto no modo thread-per-session ou com um thread pool. O default é thread-per-session. Para que o `ServContas` opere com um thread pool é necessário ativá-lo com um argumento adicional na linha de comando. O número máximo de worker threads no pool e o comprimento máximo da fila de sessões aguardando atendimento por alguma worker thread são argumentos especificados na ativação do `ServContas`. Novas conexões de clientes devem ser recusadas caso a fila de sessões tenha comprimento máximo. A fila de sessões deve ser um bounded buffer com múltiplos consumidores (as worker threads) e um produtor (a listener thread, que fica aguardando novas conexões de clientes).

Em vez de implementar do zero o pool de worker threads e a fila de sessões, neste EP você usará as implementações existentes no pacote `java.util.concurrent`. Use um `ThreadPoolExecutor` para o pool de threads e uma `ArrayBlockingQueue` para a fila de sessões.

Além das opções descritas na seção 2, seu programa deverá aceitar as seguintes:

- `-threadpool` n, que especifica operação com um pool de n worker threads;
- `-bound` n, que especifica o comprimento máximo da fila de sessões.

Implemente também um cliente adicional (ou conjunto de clientes) que compare o desempenho do `ServContas` no modo thread-per-session com o desempenho no modo thread pool. Escreva esse(s) cliente(s) com o objetivo de evidenciar a diferença de desempenho entre os modos thread-per-session e thread pool, na situação em que essa diferença deve ser maior: sessões bem curtas, com uma só requisição por sessão. Ou seja, faça o cliente executar um laço que abre uma sessão com o servidor, envia uma requisição, aguarda a resposta e fecha a sessão. No modo thread-per-session o `ServContas` criará uma nova thread a cada volta, mas no modo thread pool ele só alocará uma worker thread do pool.

Use como medidas de desempenho o tempo de resposta para um cliente e a vazão (throughput) com múltiplos clientes.

- O tempo de resposta é o tempo de execução, por um cliente, de uma volta de um laço "abre sessão, manda requisição, aguarda resposta e fecha sessão".
- A vazão é número de sessões que o servidor consegue tratar por unidade de tempo quando múltiplos clientes rodam laços como esse.

5 Seu arsenal

Você deve implementar o `ServContas` em Java, usando o JDK 1.5. Não devem ser usadas versões anteriores do JDK, pois os pacotes `java.util.concurrent` e `java.util.concurrent.locks` foram introduzidos na versão 1.5.

Não chame nenhum método "deprecated".

Os clientes pedidos na seção anterior e no item 6 da seção 2 podem ser implementados em Java ou em outra linguagem qualquer, desde que evidenciem os problemas a que o servidor estará sujeito caso seja desligada a isolação ou a prevenção de deadlock e permitam avaliar a influência do uso de um thread pool sobre o desempenho do servidor. Minha impressão é que o melhor é usar Java também para os clientes, mas fique à vontade para discordar de mim! Note que, embora este enunciado fale em "clientes" (no plural), você pode, se você achar mais conveniente, implementar um único programa cliente com todas as funcionalidades pedidas.

6 Sobre a entrega

Você deverá entregar três coisas:

- um arquivo tar.gz contendo sua solução (arquivos-fonte, makefile, README, . . .);
- um relatório sobre os seguintes pontos:
  - a dificuldade que você teve (ou não teve) para evidenciar a ausência de isolação ou conseguir deadlock;
  - a comparação dos desempenho do `ServContas` nos modos thread-per-session e thread pool.

O desempacotamento do seu arquivo tar.gz deverá produzir um diretório contendo esses ítens. O diretório deve ter nome da forma `ep4-membros-da-equipe` (exemplo: `ep4-joao-maria`).

O arquivo `README` deve incluir uma explicação de como rodar cada cliente. Ele deve, por exemplo, explicar como os clientes devem ser acionados para evidenciar a ausência de isolação num servidor chamado com "`-cozinh`" e para chegar a um deadlock num servidor chamado com "`-suicida`". O que se deseja aqui é uma receita simples para chegar rapidamente a uma situação errônea induzida por clientes escritos com esse propósito. Não é aceitável algo como "rode o cliente C1 e o cliente C2 e espere algumas horas; o deadlock geralmente ocorre depois de quatro horas."

A entrega será via Internet. Detalhes adicionais serão divulgados na lista de discussão da disciplina.

EPs atrasados não serão aceitos!

Bom trabalho!