UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Thierry Pierre Dutoit - 119040432

Trabalho 2 de Computação Concorrente:

Leitores e Escritores/Sensores e Atuadores

Professora: Silvana Rossetto
Data: 05/2021
Local: Rio de Janeiro

Leitores e Escritores com prioridade para Escrita

A aplicação de Leitores e Escritores supõe que alguns requisitos sempre deverão ser mantidos em relação a área compartilhada entre as diferentes threads:

(i) As threads leitoras **apenas lêem** o conteúdo da área de dados
(ii) As threads escritoras **apenas escrevem** conteúdo na área de dados
(iii) As threads leitoras podem ler **simultaneamente** uma região de dados compartilhada
(iv) **Apenas um escritor pode escrever** a cada instante em uma região de dados compartilhada
(v) Se um **escritor está escrevendo**, **nenhum leitor pode ler** a mesma região de dados compartilhada

No entanto, além dos requisitos padrões do problema de leitores/escritores, há a prioridade para escrita (um novo requisito) que precisa ser satisfeita. Assim, quando uma thread escritora quiser escrever, nenhuma thread leitora começará sua execução antes da escritora:

(vi) Se um ou mais escritores estiverem na **fila de espera de execução**, eles sempre serão os **primeiros da fila**.

Principais Decisões do Projeto da Solução

O programa para leitores/escritores em Java é parecido com a implementação em C, porém por ser em Java alguns aspectos mudam devido à natureza da linguagem. A primeira diferença é a criação de uma classe que armazena a área de dados compartilhada entre as threads leitoras e escritoras e a implementação dos métodos que alteram os dados compartilhados.

Teremos mais duas classes, uma para as threads encarregadas de escrever, e outra para as threads encarregadas de lerem. Ambas as classes farão uso do recurso compartilhado.

Nos métodos de início e fim de leitura/escrita precisaremos usar synchronized como exclusão mútua para evitar condições de corrida ruins. O objeto alvo para o uso de synchronized será o recurso compartilhado, que ambas as classes utilizam. Dessa forma, duas ou mais threads não poderão executar simultaneamente em cima do

recurso compartilhado. Na minha implementação, a frase anterior não exclui o caso em que n threads leitoras executam simultaneamente, elas só não podem iniciar e finalizar ao mesmo tempo.

Em relação ao conteúdo novo, é bem simples dar prioridade para a escrita. A solução consiste em alterar tanto os métodos de leitura quanto de escrita para respeitarem a prioridade. No método de Leitura, mudaremos a condição de início do leitor, para que não inicie caso uma thread escritora queira escrever. Mudaremos também o método de início de escrita para que, quando um escritor execute, ele defina a prioridade de execução e a retire quando conseguir começar a escrever. Para implementar essa solução, faremos uso de uma variável que controla a prioridade, como veremos na próxima seção:

Principais Decisões da Implementação da Solução

Analisando a implementação em mais detalhes, a classe de recurso compartilhado atua em três variáveis principais: número de leitores lendo, número de escritores escrevendo e prioridade para escrita. Esta última variável define se existe a prioridade para escrita ou não naquele determinado momento.

Nas classes Escritora e Leitora temos 4 métodos: O construtor, que inicializa o id da thread e o recurso compartilhado que será usado por ela; o método run que consiste na tarefa realizada pela thread; para a classe Escritora os métodos de IniciaEscrita e FimEscrita que controlam o número de escritores; para a classe Leitora os métodos de IniciaLeitura e FimLeitura que controlam o número de leitores. Essas duas classes são extensões da classe Threads, a fim de permitir a programação concorrente.

No método main, recebemos a quantidade de escritores e leitores, inicializamos as threads e estas executam indeterminadamente até que seja usado algum mecanismo de parada. É no método de main que ocorre as inicializações das threads e do recurso compartilhado, que é passado como argumento para as threads.

Estruturas de dados Utilizadas e Funções Implementadas

Como estrutura de dados compartilhada temos a classe chamada Resource que possui 3 variáveis inteiras: leitores, escritores e prioridade_escrita. Todas são inicializadas com valor igual a 0, e a variável leitores corresponde ao número de leitores ativos, a variável escritores ao número de escritores ativos e a variável prioridade_escrita armazena se uma thread escritora deseja escrever.

Ainda na classe de recurso compartilhado temos métodos getters e setters padrões do Java mas também temos os métodos que adicionam leitor/escritor ao somarem em 1 a variável correspondente, e os métodos que removem leitor/escritor ao diminuírem em 1 a variável correspondente.

Já mencionei o número de métodos das Classes Escritora e Leitora, então vamos analisar mais a fundo o que cada função faz:

Método run() - Classe Escritora

Inicia a escrita, ou seja, adiciona um escritor. Printa que o escritor está escrevendo e termina a escrita, removendo o escritor.

método IniciaEscrita() - Classe Escritora

A função é encapsulada por synchronized(instância do recurso compartilhado) e resumidamente define a prioridade de escrita, checa se já existem escritores ou leitores executando e se for o caso ela se bloqueia, adiciona escritor quando conseguir passar do while bloqueante e remove a prioridade.

método FimEscrita() - Classe Escritora

A função é encapsulada por synchronized(instância do recurso compartilhado) e remove o escritor notificando todas as threads bloqueadas que podem se desbloquear (notifyAll()), pois diversas threads tanto escritoras quando leitoras podem ter sido bloqueadas enquanto esta thread escritora escrevia. Além disso, há um sleep de um segundo fora da parte synchronized para que a alternância entre as threads seja mais visível.

Método run() - Classe Leitora

Inicia a leitura, ou seja, adiciona um leitor. Printa que o leitor está lendo e termina a leitura, removendo o leitor.

método IniciaLeitura() - Classe Escritora

A função é encapsulada por synchronized(instância do recurso compartilhado) checa se já existem escritores ou se há prioridade para escrita e se for o caso ela se bloqueia, e adiciona o leitor quando conseguir passar do while bloqueante.

método FimLeitura() - Classe Escritora

A função é encapsulada por synchronized(instância do recurso compartilhado) e remove o leitor, caso seja a única thread leitora executando notifica todas as threads bloqueadas que podem se desbloquear (notifyAll()), pois diversas threads escritoras podem ter sido bloqueadas enquanto ocorria a leitura. Além disso, há um sleep de um segundo fora da parte synchronized para que a alternância entre as threads seja mais visível.

Já foram mencionados o método main e as variáveis presentes em cada classe, então num contexto geral o programa já foi bem descrito. Para analisar o código basta clicar neste link.

Testes Realizados

Os testes serão feitos através de um programa em Python que verifica o log gerado pelo programa em Java. Para isso, criei um programa em Java com a mesma implementação que o programa em Java mencionado mais acima, porém com log adaptado para a análise do programa Python. O programa em Python (fornecido pela professora) que verifica o log precisou ser atualizado para lidar com prioridade de escrita.

O novo programa em Java se encontra aqui, e o programa em Python aqui.

Dessa forma, com estes programas, conseguimos analisar a saída do nosso programa e verificar a corretude dele. No geral, queremos que o seguinte trecho de linha de comando:

```
PS C:\Users\thier\IdeaProjects\Trab2> javac lclog.java
PS C:\Users\thier\IdeaProjects\Trab2> java lclog 4 3 > saida.py
PS C:\Users\thier\IdeaProjects\Trab2> python3 saida.py > erros.txt
SyntaxError: Non-UTF-8 code starting with '\xff' in file C:\Users\thier\IdeaP
-0263/ for details
PS C:\Users\thier\IdeaProjects\Trab2> python3 saida.py > erros.txt
SyntaxError: Non-UTF-8 code starting with '\xff' in file C:\Users\thier\IdeaP
-0263/ for details
PS C:\Users\thier\IdeaProjects\Trab2> python3 saida.py > erros.txt
PS C:\Users\thier\IdeaProjects\Trab2> cat erros.txt
PS C:\Users\thier\IdeaProjects\Trab2> _
```

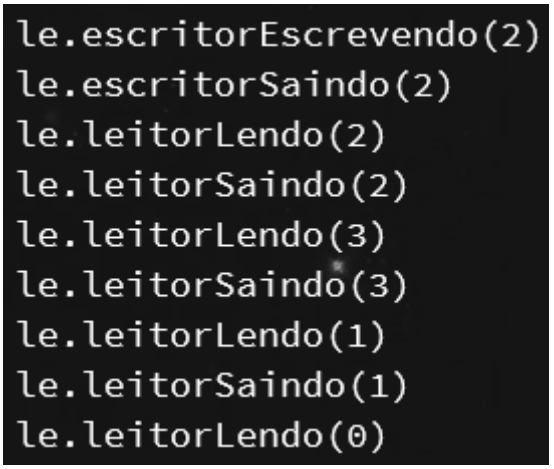
resulte num arquivo erros.txt vazio. Vejamos o porquê.

javac lclog.java -> compila o programa Java
java lclog 4 3 > saida.py -> executa o programa com 4 leitores e 3 escritores e escreve o log num programa em python chamado saida.py
python3 saida.py > erros.txt -> executa o programa em python e escreve qualquer inconsistência do programa Java no arquivo txt erros. Ocorreu um problema de codificação de arquivo, por isso as mensagens de erro. Eu consertei o problema e quando fui analisar o arquivo erros.txt ele estava vazio, significando que nenhum erro foi encontrado.

No programa saida.py vemos quantas linhas de log tivemos. O arquivo que eu usei possui mais de 2500 linhas de log, então temos uma boa margem de verificação. Porém, para meios didáticos, analisaremos aqui algumas situações mais relevantes:

```
le.escritorPrioridade(0)
le.escritorEscrevendo(0)
le.escritorPrioridade(2)
le.escritorBloqueado(2)
le.leitorBloqueado(3)
le.escritorPrioridade(1)
le.escritorBloqueado(1)
le.leitorBloqueado(2)
le.escritorSaindo(0)
le.leitorBloqueado(2)
le.escritorEscrevendo(1)
le.escritorSaindo(1)
le.leitorBloqueado(3)
```

Toda vez que um escritor entra no método IniciaEscrita() ele define a prioridade para escrita, assim como mostrado na primeira linha do print. Como neste caso não havia nenhum escritor e leitor executando, ele começou a escrever. Outro escritor entra no método IniciaEscrita(), define a prioridade de escrita e se bloqueia pois há um escritor escrevendo. Obviamente, o leitor que tenta entrar também é bloqueado, assim como o escritor(1) que define mais uma prioridade de escrita e se bloqueia, e o leitor(2) que entra logo após e é bloqueado. Se pararmos para analisar, 2 escritores estão na espera para começar a escrever, então nossa prioridade de escrita é igual a 2. Quando o escritor 0 termina de escrever, o leitor tenta executar mas a prioridade de escrita o barra e ele se bloqueia mais uma vez. Depois, o escritor 1 começa e termina sua execução e o leitor 3 tenta executar, mas como o escritor 2 ainda possui a prioridade de escrita (igual a 1 nesse momento da execução) o leitor se bloqueia mais uma vez. Somente após o escritor 2 terminar sua execução é que será possível para os leitores entrarem em cena (caso nenhum escritor entre na frente), como vemos a seguir:



O escritor 2 termina e os leitores conseguem finalmente ler a área de dados compartilhada. Este exemplo mostra bem como a prioridade de escrita funciona no nosso programa.

Os outros requisitos do padrão foram atendidos e não são muito explicitados aqui pois o código em Python de verificação fornecido pela professora já os satisfaz. Como eu usei esse código em Python e apenas fiz adições por cima, considerei melhor dar ênfase no requisito novo de prioridade de escrita.

Aplicação de Monitoramento de Temperatura (Atuador/Sensor)

A Primeira divergência entre a Aplicação de Monitoramento de Temperatura e o padrão Leitores e Escritores se encontra no uso da Escrita e Leitura. Enquanto que no primeiro padrão, não tínhamos nada sendo realmente lido e escrito, agora estamos usando o padrão para ordenar a escrita e leitura das temperaturas medidas.

Como consequência, no método run tanto de Escritora quanto Leitora, não há mais execução em loop. O loop agora se encontra nas novas classes: Sensor e Atuador. Fora isso, as classes Leitora e Escritora e o recurso compartilhado entre elas se mantêm iguais, e não precisamos testá-los ou entrar em maiores detalhes. O foco daqui para frente será a explicação da aplicação de Monitoramento.

Estrutura de Dados (Atuador/Sensor)

Temos uma nova classe chamada Valor, que armazena três variáveis inteiras: idSensor, idLeitura e Value. Todo objeto dessa Classe Valor corresponderá a uma medição de temperatura. Dessa forma, idSensor armazena o id do Sensor que mediu aquela temperatura. IdLeitura armazena a quantidade de medições feitas por aquele Sensor e Value armazena o valor em si lido. Na classe Valor, temos os métodos getters e setters também, para que os valores possam ser alterados e retornados.

Em seguida, criamos uma nova classe chamada Resource, que armazenará os dados necessários para o padrão Sensor/Atuador. Os valores armazenados são: um vetor de 60 posições do objeto Valor, que guarda no máximo 60 medições feitas; uma variável que guarda a posição atual para escrita e o número de sensores que o programa tem, que também corresponde ao número de atuadores. Além disso, temos dois métodos: addValue que guarda uma medição feita por um Sensor que seja válida a ser armazenada e; getValue que retorna uma medição através de uma posição fornecida.

Valores Passados e Retornados para as Threads (Atuador/Sensor)

Para as Threads Sensor e Atuador precisarão ser passados apenas o Recurso compartilhado entre Leitoras e Escritores e o recurso compartilhado entre as próprias threads Sensor e Atuador. Assim, quando um Sensor quiser escrever, ele poderá modificar o número de escritoras naquele momento, e também colocar no vetor o valor medido. Da mesma maneira, quando um Atuador quiser ler as temperaturas medidas, aumentará o número de leitores e acessará os elementos do vetor de temperaturas.

Ambos os recursos compartilhados são inicializados na função main, que também está responsável pela declaração e inicialização das threads Sensores e Atuadores.

Algoritmo Executado Pelas Threads (Atuador/Sensor)

Cada Sensor armazena seu id e seu idLeitura, para que possa escrever estes valores nas medições. Além disso, como mencionado anteriormente, ele recebe os dois recursos compartilhados.

Ele possui apenas um método, que executa indeterminadamente, que é o método run executado pela thread. Neste método, gera-se um valor aleatório entre 25 e 40 (inclusivo) que corresponde ao valor medido. Se este valor for maior que 30, inicializa-se um objeto da classe Escritora e chamamos o método IniciaEscrita. Adicionamos então o valor medido no vetor de medições compartilhado, adicionamos o número de leituras daquele sensor e 1 e terminamos a escrita chamando o método FimEscrita. Então, a thread Sensor dorme durante 1 segundo e continua sua execução.

Os Atuadores já são mais robustos, e armazenam seu id, uma variável que controla o alerta vermelho, outra que controla o alerta amarelo, a média dos valores lidos, a variável que controla a posição que está sendo lida e os recursos compartilhados.

Temos novamente apenas um método: o método run que também executa indeterminadamente. De início, há a instanciação de um objeto de Leitora e esse objeto chama o método IniciaLeitura. Após, executa-se um laço para encontrar a última posição que o Sensor de mesmo id que o Atuador escreveu. Com a última posição em mãos, inicia-se outro laço, para verificar os alertas, ou seja, se os 5 últimos valores foram maiores que 35, ou se os 15 últimos valores foram acima de 35, ou se nenhum dos casos anteriores são satisfeitos para emissão de um alerta normal. Ao mesmo tempo, para cada valor lido, aumenta-se em 1 a variável posição e soma-se o valor com a média.

Uma vez que o laço termina, já sabemos qual alerta emitir e qual a média dos valores disponíveis daquele sensor. Então, emitimos os logs e zeramos as variáveis que precisam ser resetadas para uma nova execução daquele Atuador. Por fim, a thread Atuador dorme 2 segundos antes de executar novamente.

O programa em Java se encontra neste link.

Interface com o Usuário da Aplicação (Atuador/Sensor)

A interface com o usuário da aplicação é baseada em executar o programa fornecendo o número de Sensores/Atuadores que serão executados. Um ponto a ser notado é que o número de Sensores/Atuadores passado for maior que 4, o programa irá retornar uma mensagem de erro e não continuará, pois não é um número recomendado de Sensores/Atuadores.

Então, após o programa ser executado, o log começará a ser produzido fornecendo os valores medidos que foram maiores que 30 e os sensores que mediram tais valores. Além disso, por parte dos atuadores, teremos um aviso quando algum atuador começar a ler, e cada atuador retornará o alerta correspondente aos valores lidos e também a média dos valores ainda disponíveis a serem lidos daquele sensor. Como já satisfazemos o padrão leitores/escritores, não precisamos produzir o log de quando um leitor começa a ler ou termina, ou quando um escritor começa a ler ou termina, muito menos quando leitores/escritores são bloqueados. Queremos apenas verificar que os valores medidos e lidos correspondem e que os alertas e as médias calculadas estão corretas.

Testes Realizados (Atuador/Sensor)

Como já reforçado, os testes feitos foram apenas nos valores medidos pelos sensores e na leitura feita pelos Atuadores (alertas e médias). Dessa forma, foi possível criar um script em Python que avalia o log produzido e checa se algum erro foi cometido.

Esse Script se assemelha bastante ao código em Java, pois queremos reproduzir o comportamento do programa e analisar se os logs foram emitidos corretamente. Através do programa em Java, criamos um programa em Python chamado saidamonitoramento.py. Nele, temos aproximadamente 1500 linhas com o padrão Sensor/Atuador, e que serão analisadas pelo script em Python. Quando executamos este programa, qualquer erro encontrado nos valores medidos e lidos será notificado e, como intencionado, 0 notificações de erro foram emitidas.

Sendo assim, todos os requisitos da aplicação são garantidos pois, primeiro, o padrão leitores e escritores já é garantido uma vez que foi modularizado e testado. O padrão Sensores e Atuadores também é garantido pois o log gerado foi grande o bastante para cobrir casos críticos como um número de sensores grande disputando a escrita no vetor, o encadeamento do vetor e a posição sempre estando entre 0 e 59, os atuadores lerem as posições certas e emitirem médias/alertas corretos, entre outros. Como obtivemos 0 notificações de erro por meio do script em Python e eu possuo bastante convicção que tudo está correto, acabamos por aqui a fase de testes.