Relatório de Programação Concorrente

Sérgio Araújo | up201608365 | 21/0/2020

Índice

Introdução

Este trabalho tem como objetivo programar, validar a correção, e avaliar o desempenho de filas concorrentes com elementos guardados num *array*, com diversos tipos de aproximações - baseadas em *locks*, primitivas atómicas ou STM - e modalidades de implementação - capacidade fixa ou ilimitada.

Programar um *crawler* concorrente de páginas Web baseado no uso de uma *fork-join pool*. Analisar o seu desempenho com diversas *threads* em comparação com a capacidade de computação do servidor

Desenvolvimento

Filas Concorrentes

Fila Baseada em Monitores

Monitor é um mecanismo que controla o acesso concorrente a um objeto, em java o objeto x, é monitorizado pela indicação no bloco *syncronized*(x). Qualquer operação sobre x adquire um *lock*, que é libertado no final da execução do *syncronized*, deste modo, apenas um agente pode executar o código, tornando atómico.

Em conjunção com os métodos *wait(), notify()* e *notifyAll(),* é possível construir lógicas de espera por um evento, que ocorra numa *thread* em paralelo. Será essencial para a implementação de uma fila bloqueante.

MbQueue Implementação inicial e bugs

A implementação inicial o código apresenta erros nos métodos *add* e *remove* da fila, foi dado como pista que os dois blocos *syncronized* e as chamadas ao método *notify*, são os conteúdos errados.

Corri o *cooperari* com o objetivo de obter uma análise mais precisa dos erros presentes. Como resultado surgiu dois erros, *null pointer* sugerindo um problema entre a sincronização da lógica de adição/remoção e o atual tamanho da fila, e *deadlock* que poderia ser problema de sincronização entre os dois métodos.

Focando na resolução dos dois blocos de sincronização e tentado remover o *null pointer*, irei analisar este erro primeiro erro e de seguida o *deadlock*.

Null pointer

Numa primeira analise reparei que os blocos de sincronização separam a lógica de adição, com a lógica de aumento ou redução da variável *size*. Esta é responsável por guardar o número de elementos atualmente presentes na lista, a falta de coerência desta com o estado atual da lista leva a diversas falhas. Por exemplo, uma vez que o lock é libertado antes do *size* ser atualizado, outra *thread* pode escrever em cima de um elemento antigo da lista não acionando nenhuma exceção; ou até remover um elemento já removido, causando o erro que estou a tentar resolver.

A solução passa por integrar este código num só bloco *syncronized*, movi a lógica do segundo para o final do primeiro, as chamadas *notify* também foram movidas para o final de toda a lógica, para não me deparar com futuras inconsistências de estados.

DeadLock

Com a solução anterior implementada, o problema do *deadlock* não ficou resolvido, seguindo a dica dada inicialmente, conclui que seria por causa da presença do *notify* em vez de *notifytall*.

O método *notify* acorda uma única *thread* de modo não determinístico, ou seja, uma *thread* aleatória. Não faz sentido, porque precisamos de notificar todas, as que estão a espera em *add* e as que se encontram em remove, acordar apenas num destes métodos várias vezes sucessivas origina um bloqueio, porque o outro lado precisa de uma *ação* para continuar.

Fila ilimitada

A transformação numa fila ilimitada, requer apenas adaptações no método *add*. Este é um bloco atómico, o que significa que apenas um agente pode o executar num dado espaço tempo. Se verificação de espaço no *array* for igual a zero, criamos um *array* novo com o dobro do espaço e copiamos os valores do antigo para este, executando por vim a adição no novo *array* e libertando o *lock*.

Fila baseada em STM

STM é um mecanismo de controlo concorrente, é uma alternativa ao mecanismo de *lock*. Baseasse em software em vez de um componente de hardware. Uma transação de contexto ocorre quando uma lógica de código executa uma serie de leituras e escritas para uma memoria partilhada, estas ações ocorrem num instante temporal único e estados intermédios não são visíveis para outras transações com sucesso.

STMQueue Implementação inicial e bugs

Os erros desta implementação assemelham-se aos erros presentes na fila baseada em monitores, o bloco atómico não abrange toda a logica dos métodos, criando incoerência entre o tamanho real da fila e o representado pela variável. Possível rescrita em valores já presentes e tentativa de leitura em valores fora do domínio do *array,* são erros provenientes deste estado.

Para solucionar o problema, a lógica foi movida para um único bloco STM atómico, simulando a zona monitorizado do *syncronized*.

Fila ilimitada

A região *stm.atomic* garante atomicidade de execução dentro deste bloco, tal como na implementação com os monitores, a alteração de *array* está podemos aumentar o tamanho do *array* em segurança do mesmo modo que fizemos para a implementação anterior.

Fila baseada em primitivas atómicas

LFBQueue Implementação inicial e bugs

A implementação inicial da fila já encontra variáveis atómicas que permitem a sincronização entre várias *threads*, através da execução de testes é possível verificar que no caso de correr apenas um método da fila, não ocorre problemas de sincronização.

As variáveis *head* e *tail* são atómicas e pontos de sincronização, responsáveis por apontar para onde os elementos devem ser adicionados ou removidos. Quando apenas um método é chamado por várias *threads*, estes pontos de sincronização garantem uma ordem de execução, impossibilitando incoerências. Por outro lado, a ordem correta dentro de um método pode ser sobreposta por ações de outro, ocorrendo então erros de incoerência no estado de memória.

É necessário garantir que apenas um método possa ser executado ao mesmo tempo, para manter então a ordem correta de execução.

Rooms

A classe *Rooms* disponibilizada pelo professor, é uma abstração de quartos em que inicialmente nenhum quarto está ocupado, até que uma *thread* entre num quarto. Estando o quarto ocupado, não há limite para o número de *threads* que podem entrar. *Threads* a tentar entrar noutro quarto serão bloqueadas enquanto o primeiro estiver ocupado.

Se associarmos cada método à sua sala, simula o comportamento de correr apenas um método de cada vez, tornando o código estável num ambiente *multithread*. Para os métodos *add* e *remove*, a entrada na sala é realizada a cada iteração do while, e a saída no fim do bloco while, mesmo que as adições/remoções não sejam bem-sucedidas.

Porque sair da sala se não adicionamos ou removemos nenhum elemento?

Sair da sala mesmo que a iteração não tenha sucesso, permite que outra sala seja aberta, e desbloqueei a fila.

No método *size*, é necessária uma sala, porque engloba a execução de dua operações, acesso a cabeça e cauda da fila, que podem alterar com o acesso as outras salas. Assim a sala SIZE torna a logica de acesso atómica. A entrada e saída são executadas a entrada e saída do método respetivamente.

Fila ilimitada

Todos os métodos exceto o *add*, não necessitam nenhuma alteração. As presenças de salas garantem toda a sincronização necessária, impedindo que ao acesso a outras salas quando a adição esta a decorrer.

Na adição, temos de impedir o acesso concorrente ao *array*, este pode vir a precisar de ser aumentado no caso de já se encontrar cheio. Foi adicionada um booleano atómico de modo a funcionar como uma "*flag*" de exclusão mútua, sinalizando quando uma *thread* esta de momento a trabalhar sobre o *array,* e colocando em espera ativa as restantes que tentem aceder, entretanto.

Back-off

As entradas nas salas são feitas dentro de um ciclo de espera ativa, enquanto que o seu objetivo não seja concluído múltiplas chamadas serão feitas a entrada na sala, acionando o mecanismo *back-off* implementado na abstração *Rooms*.

Análise de execução linearizável

Apenas existe precedência nas ações de adição ou remoção, caso a fila esteja cheia ou vazia, respetivamente. Todas as operações fora deste contexto podem acontecer em paralelo. Após a execução de vários testes, parece que todos os casos possíveis foram cobertos.

Avaliação de desempenho

A avaliação de desempenho das respetivas filas foi realizada através de um processo automático disponibilizado pelo docente da cadeira. Este processo devolve o número de operações que a fila consegue desempenhar durante um segundo de execução.

O processo foi executado cinco vezes de modo a eliminar dispersão de resultados, e transferido para um gráfico para melhor análise (Fig1).

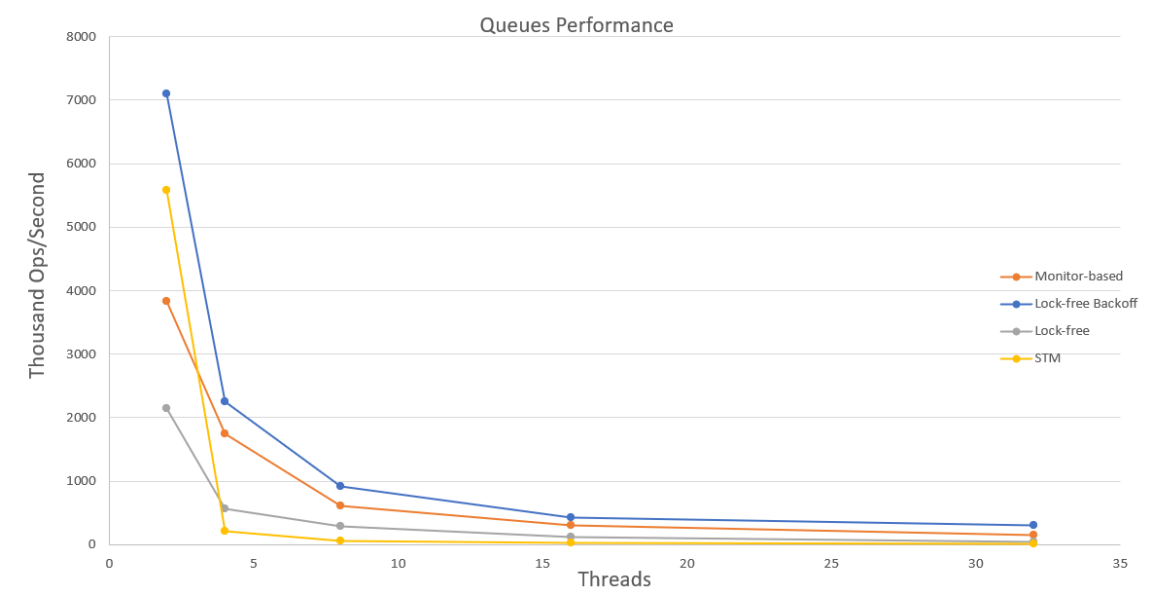

Fig1: Gráfico de desempenho

O aumento de número de *threads* faz com que as implementações reduzem o seu desempenho, numa proporcionalidade inversa. Como o computador não tem núcleos suficientes para paralisar todas as *threads*, mais *threads* apenas aumenta a complexidade e o número de variações de contexto. Por outo lado mais *threads*, significa mais sincronização para as restantes. Estes casos afetam negativamente o desempenho para cada *thread* aumentada, até ao ponto em que o ambiente não concorrente será melhor.

A abstração *backoff* melhora significativamente o desempenho quando comparado com a alternativa, espera ativa.

Crawler

Implementação

O *crawler* concorrente, adaptação do sequencial, não deve fazer o download de uma página mais que uma vez. Cada download tem um identificador único, que começa com o valor inicial de zero até ao número total de downloads.

Uma sincronização de estado geral do trabalho é necessário entre as n *threads* que executam em paralelo, nomeadamente o conteúdo já baixado e o numero da tarefa. Um set concorrente proveniente da biblioteca do java é utilizado para guardar os links já visitados. Um inteiro atómico comum a todas as *threads*, é utilizado para manter o número da tarefa, o método atómico *getAndIncrement* permite manter um estado coerente

A adaptação foi realizada através do esqueleto inicial, que é constituído pela implementação de uma *ForkJoinPool*, a tarefa inicial é submetida na *pool* e todos os seus filhos executam um *fork*.

Teste e avaliação de desempenho

O crawler concorrente tem um desempenho superior do que o sequencial, o último esperara que o download acabe para começar outro. Operações de I/O são dispendiosas, é necessário esperar pelo envio do pedido, tempo de processamento do servidor, e o tempo de reposta. O concorrente consegue nesse tempo enviar n pedidos mitigando assim o tempo de espera porque é disperso por n pedidos.

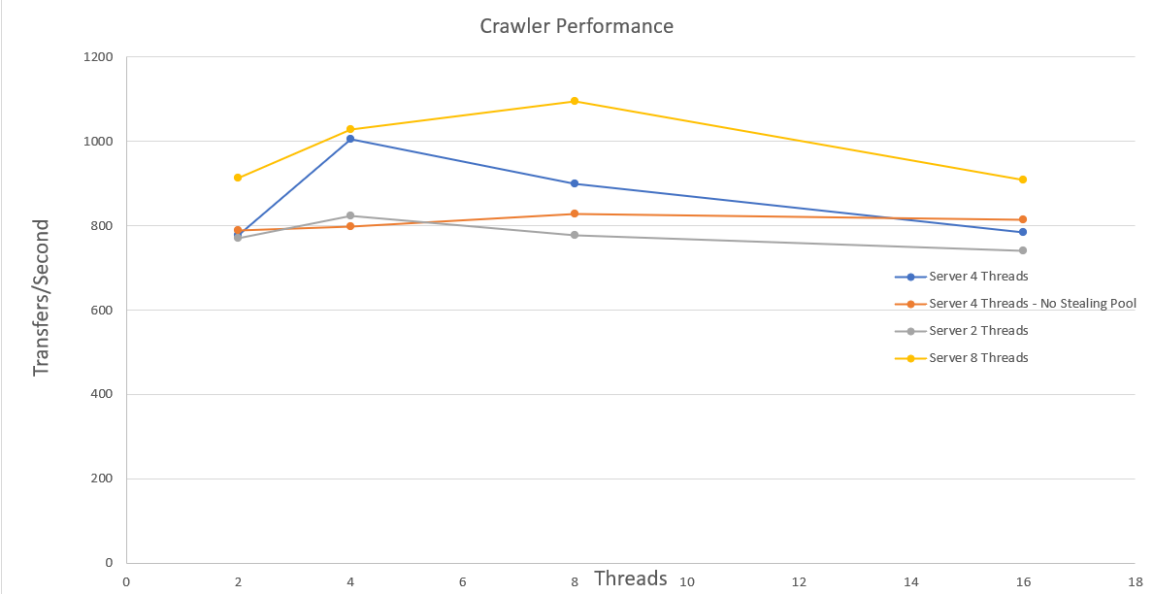

Fig2: Gráfico de desempenho

Na Fig2 é possível avaliar o desempenho do *crawler* com base no número de *threads* do servidor e do mesmo, e a desativação *da stealing pool*.

O aumento do número de *threads* no *crawler* para além das presentes no servidor leva a uma diminuição no performance, o servidor não consegue responder a mesma velocidade de que os pedidos são feitos, e como na conclusão das filas, o maior número de *threads* apenas acarreta complexidade.

Desabilitando a *stealing pool* gera tempo inativo em *threads* que já acabaram o seu ramo, quando não há mais tarefas na pool. Estas ficam a espera que os restantes ramos mais profundos sejam terminados, sem puderem ajudar na sua exploração