INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Sincronização com Monitores na CLI e na Infra-estrutura Java

Carlos Martins

Lisboa
Novembro de 2009
3ª Edição

Índice

1 Introdução

O objectivo da sincronização é coordenar a execução de *threads* que partilham recursos. No essencial, a sincronização atrasa a execução das *threads* quando não estão reunidas as condições necessárias para realizar determinada operação, até que essas condições se verifiquem. Mesmo no caso mais simples de sincronização – a exclusão mútua – o objectivo da sincronização é atrasar a execução de cada *thread* que pretenda executar a secção crítica de código enquanto esta está a ser executada por outra *thread*. A sincronização no acesso a recursos mais complexos é mais elaborada, mas, no essencial, baseia-se nas mesmas operações básicas: bloquear a execução da *thread* corrente quando esta detecta que não estão reunidas as condições para prosseguir a execução; depois, quando outra *thread* criar as condições adequadas, acorda a *thread* bloqueada para que esta prossiga a execução.

Em programação orientada por objectos, a gestão de um recurso partilhado por múltiplas *threads* é normalmente associado ao tipo de dados definido para o efeito. Os campos do tipo armazenam a informação necessária para representar o estado do recurso e são partilhados por múltiplas *threads*; os corpos dos métodos são os algoritmos de sincronização que gerem o acesso ao recurso. O acesso aos campos do tipo que armazenam estado partilhado é, tipicamente, realizado em exclusão mútua.

Todos os algoritmos de sincronização envolvem os seguintes passos de processamento:

- Quando a *thread* tenta adquirir o recurso, começa por adquirir o acesso exclusivo aos dados partilhados e, depois, avalia o predicado que indica se o recurso está disponível; se o predicado for verdadeiro, a *thread* assume a posse do recurso, sai da secção crítica e prossegue a execução; caso contrário, insere-se eventualmente numa fila de espera, liberta a secção crítica e bloqueia-se.
- Quando a *thread* disponibiliza o recurso, começa por adquirir acesso exclusivo aos dados partilhados e, depois, realiza uma das seguintes sequências:
    - Actualiza os dados partilhados de forma a indicar que o recurso está disponível; depois, determina se existem *threads* bloqueadas a aguardar a aquisição do recurso; em caso afirmativo, acorda uma ou mais dessas *threads*; finalmente, liberta a o acesso à secção crítica e continua a execução;

- Selecciona uma das *threads* bloqueadas para lhe atribuir o recurso libertado se existir; actualiza os dados partilhados para indicar que o recurso está atribuído àquela *thread*; a seguir, acorda a *thread* seleccionada, liberta o acesso à secção crítica e continua a execução.

– Quando a *thread* acorda, após bloqueio, começa por adquirir a posse da secção crítica e executa uma das seguintes sequências:

  - No caso do reatamento da execução indicar que o recurso está disponível, avalia o predicado que indica a disponibilidade do recurso e se este for verdadeiro, actualiza os dados partilhados indicando que assumiu a posse do recurso, liberta o acesso à secção crítica e continua a execução; caso contrário, liberta o acesso à secção crítica e volta a bloquear-se;

  - No caso de o recurso lhe ter sido atribuído, assume que os dados partilhados estão actualizados, pelo que liberta o acesso à secção crítica e continua a execução;

  - Quando uma *thread* é acordada devido à ocorrência de condições excepcionais – interrupção ou *timeout* –, actualiza os dados partilhados (e.g., remove-se da fila de espera onde estava inserida), realiza o processamento associado à desistência, se existir, liberta o acesso à secção crítica e continua a execução.

Os algoritmos de sincronização que gerem múltiplos recursos têm, por vezes, operações compostas pela sequência do processamento de aquisição de um recurso seguido do processamento da libertação de outro recurso. Por exemplo, no problema do *bounded buffer*, a operação `Put` começa por adquirir o recurso "posição livre" e depois de consumar a transferência da informação para o *buffer* disponibiliza o recurso "posição com informação" que é requerido pela operação `Take`.

Existem outros algoritmos de sincronização que gerem recursos dependentes (e.g., o problema dos leitores escritores), onde é necessário implementar critérios de *scheduling* mais elaborados, que têm em consideração não só o estado dos recursos como também os pedidos em fila de espera. Nestes algoritmos, o facto de uma *thread* em espera desistir da aquisição de um determinado tipo de recurso pode criar condições de progressão a outras *threads* bloqueadas. Por exemplo, no problema dos leitores

escritores, quando o único escritor em espera desiste de obter acesso para escrita e o recurso está a ser lido, este tem que acordar todos leitores que foram bloqueados pela presença do escritor na fila de espera.

Uma infra-estrutura para implementar algoritmos de sincronização deve permitir: manipular dados partilhados entre *threads* em exclusão mútua; bloquear a *thread* corrente, e; reatar a execução de uma, ou mais, *threads* bloqueadas.

Quando se implementam algoritmos de sincronização em código de sistema operativo, para além de estarem disponíveis vários mecanismos para implementar exclusão mútua, é possível agir directamente sobre cada *thread* em particular, porque as estruturas de dados associadas às *threads* e ao *scheduling*, como é o caso do bloco de controlo da *thread* e da fila das *threads* no estado *ready* estão acessíveis. Assim, para bloquear a *thread* em execução basta removê-la da fila *ready* – se ela aí estiver inserida –, inseri-la na fila de espera associada ao recurso pretendido, copiar para o bloco de controlo da *thread* a informação necessária para que outra *thread* possa determinar as condições em que deve ser retomada a execução e, finalmente, invocar o *scheduler* para atribuir o processador a outra *thread ready*. Mais tarde, quando outra *thread* determina que a *thread* bloqueada tem condições para prosseguir a execução, remove-a da fila de espera, passa-lhe através do bloco de controlo a informação necessária, insere-a na fila das *threads ready* e acciona o *scheduler* para atribuir o processador à *thread* adequada. Quando uma *thread* retoma a execução após bloqueio, consulta a informação passada pela outra *thread* e o estado do recurso subjacente e determina se deve continuar a execução ou voltar a bloquear-se.

Quando se implementam algoritmos de sincronização em código de modo utilizador, como é o caso da CLI [4] e Java [6], não é possível aceder directamente às estruturas de dados do sistema operativo como sejam os blocos de controlo das *threads* e as estruturas de dados associadas ao *scheduling*. Contudo, para que os algoritmos de sincronização possam ser implementados de forma tão directa é necessário ter as mesmas possibilidades de aceder a dados partilhados em exclusão mútua, provocar directamente transições no estado das *threads* (bloquear/acordar) e passar informação entre *threads* nessas transições de estado.

Dos mecanismos de sincronização mais conhecidos, considera-se que o conceito de monitor proposto por Brinch Hansen [7] e Hoare [10] é adequado para implementar

algoritmos de sincronização em código em modo utilizador, porque unifica: os dados partilhados, o código que os manipula, a exclusão mútua no acesso aos dados partilhados e a possibilidade de bloquear e acordar *threads* de forma coordenada com a exclusão mútua. Como todos estes aspectos estão envolvidos na implementação de algoritmos de sincronização, não é de estranhar que o conceito de monitor tenha sido adoptado como infra-estrutura para implementação de algoritmos de sincronização nas linguagens que suportam directamente programação concorrente, como é o caso do Java e das linguagens suportadas pela CLI.

Neste texto, sistematizam-se soluções para resolver problemas de sincronização. No que se segue, o texto está organizado do seguinte modo.

Na Secção 2, analisam-se as semânticas propostas para o conceito de monitor desde a sua introdução até aos monitores do CLI e Java, discutindo algumas das razões que motivaram essas semânticas. São também discutidas as implicações das várias semânticas do conceito de monitor na concepção de algoritmos de sincronização baseados em padrões de código genéricos.

Na Secção 3, são apresentados padrões de código genéricos para resolver problemas de sincronização, usando os monitores do CLI e Java e avalia-se o desempenho dos algoritmos baseados nesses padrões.

Na Secção 4, discute-se a notificação específica de *threads* e a sua aplicação aos padrões de código apresentados na secção anterior e analisam-se os ganhos no desempenho obtidos com esta técnica.

Nos exemplos apresentados neste texto é sempre incluído o processamento das condições de excepção: a interrupção das *threads* é sempre considerada; o suporte para *timeout* é considerado quando pertinente. Ainda que o código resulte mais extenso, considera-se que o impacto da interrupção de *threads* na correcção dos algoritmos de sincronização é tão significativa que não faz sentido ignorar a possibilidade de interrupção das *threads*, como acontece na maioria dos exemplos que encontramos na literatura sobre monitores.

2 Monitores

2.1 Semântica de Brinch Hansen e Hoare

O conceito de monitor apresentado por Brinch Hansen [7] e Hoare [10], como infra-estrutura para sincronizar adequadamente o acesso a dados partilhados por parte de múltiplas *threads*, é um conceito fundamental porque unifica todos os aspectos envolvidos na sincronização: os dados partilhados, o código que acede a esses dados, o acesso aos dados partilhados em exclusão mútua e a capacidade de bloquear e desbloquear *threads* em coordenação com a exclusão mútua. Os dados partilhados são protegidos por um monitor e apenas podem ser acedidos no corpo de um procedimento do monitor. Os procedimentos são de dois tipos: os procedimentos de entrada, que podem ser invocados de fora do monitor, e os procedimentos internos que apenas podem ser invocados pelos procedimentos de entrada. As *threads* só realizam operações sobre os dados do monitor invocando os procedimentos de entrada. O monitor garante que, num determinado momento, quanto muito uma *thread* executa um procedimento de entrada: diz-se que esta *thread* está *dentro* do monitor. Quando uma *thread* está dentro do monitor, é atrasada a execução de qualquer outra *thread* que invoque um dos seus procedimentos de entrada.

Para bloquear uma *thread* dentro de um monitor, a aguardar condições para poder prosseguir, Brinch Hansen e Hoare propuseram o conceito de variável condição. As *threads* podem bloquear-se nas variáveis condição e serem sinalizadas por outras *threads*. A definição de Hoare [10] requer que uma *thread* que esteja bloqueada numa variável condição execute imediatamente assim que outra *thread sinaliza* essa variável condição, e que a *thread* sinalizadora execute assim que a *thread* sinalizada abandone o monitor. Esta definição garante à *thread* desbloqueada que o estado dos dados partilhados é o que foi definido antes da sinalização, mas implica pelo menos duas comutações de contexto quando uma *thread* continua a execução após bloqueio.

Em [10], Hoare propõe uma extensão à semântica da primitiva *wait*, que designa por *scheduled waits*. Esta proposta contempla a passagem de um argumento inteiro que define a prioridade da *thread* em espera, isto é, a sua posição relativa face às outras *threads* bloqueadas na mesma variável condição.

Relativamente às capacidades desejáveis para um mecanismo de suporte à sincronização que foram definidas na introdução, a semântica dos monitores de Brinch Hansen e Hoare é flexível. É possível associar às *threads* em espera uma classe (definida pela variável condição onde se bloqueiam), mas é impossível distinguir as várias *threads* bloqueadas na mesma variável condição. A extensão *schedule waits* permite distinguir as *threads* bloqueadas na mesma condição, ainda que de forma limitada. Ao ordenar a fila de espera da condição, determina-se qual, de entre as *threads* bloqueadas, é alvo da próxima sinalização – a *thread* que passou o menor valor à primitiva *wait*. No que se refere à passagem de informação no momento da sinalização, a semântica de Brinch Hansen e Hoare não poderia ser mais generosa: o estado do monitor – o estado dos dados partilhados – é passado à *thread* sinalizada. Portanto, qualquer informação que se pretenda associar à sinalização pode ser passada através dos dados partilhados do monitor.

As limitações do monitor de Brinch Hansen e Hoare como infra-estrutura para suportar directamente algoritmos de sincronização são perceptíveis analisando os exemplos apresentados por Hoare [10]: *single resource*, *bounded buffer*, *alarmclock*, *buffer allocator* (segunda versão), *diskhead* e *readers and writers*. Destes cinco exemplos, apenas o *bounded buffer* tem uma implementação directa, razão pela qual é o exemplo de eleição apresentado na literatura sobre monitores. Três exemplos, *alarmclock*, *buffer allocator* e *diskhead*, baseiam a implementação em *scheduled waits*. No caso de *alarmclock*, as *threads* são ordenadas na fila de espera de uma única condição por ordem crescente do instante em que devem prosseguir. No *buffer allocator*, onde o critério de *scheduling* definido força equidade do número de *buffers* na posse de cada *stream*, as *threads* são ordenadas, na fila de espera da única variável condição, para que as *threads* associadas aos *streams* com menos *buffers* fiquem à frente na fila. Em *diskhead*, são usadas duas variáveis condição – `upsweep` e `downsweep` – e a ordenação das *threads* nas respectivas filas de espera reflecte a ordem pela qual os pedidos devem ser servidos, respectivamente, no movimento ascendente e descendente das cabeças. A solução do último exemplo, *readers and writers*, tira partido da passagem do estado do monitor na sinalização. Só é possível libertar todos os leitores bloqueados quando termina uma escrita fazendo com que cada leitor sinalize a variável condição associada aos leitores, porque se gera uma cadeia de notificação durante a qual não é permitida a entrada de novas *threads* no monitor. Na análise do código apresentado por Hoare,

considera-se que não é evidente que um leitor sinalize outro leitor, quando os leitores não se excluem entre si. A solução apresentada por Hoare para resolver este problema é engenhosa, contudo não nos parece que tenha generalidade.

Nos vários exemplos apresentados em [10], apenas no *bounded buffer* o conceito de monitor é usado com naturalidade: as *threads* em espera são classificadas em produtoras e consumidoras e o conceito de variável condição distingue esta classificação. A implementação trata todas as *threads* consumidoras e todas as produtoras da mesma forma, pelo que não é necessário fazer distinção entre as *threads* produtoras nem entre as *threads* consumidoras.

Com os exemplos que escolheu em [10], Hoare parece pretender conferir generalidade à extensão *scheduled wait*, pois utiliza-a em três dos cinco exemplos. Em última análise, esta extensão associa informação a cada uma das *threads* bloqueadas na mesma variável condição. No entanto, essa informação apenas permite posicionar, uma *thread* relativamente às em espera da variável condição, pelo que o caracter genérico da utilização desta informação é discutível. Nos exemplos que apresentamos na Secção 3, mostramos que são necessárias estruturas de dados adicionais para implementar semânticas de *scheduling* reais. Nestes cenários, o conceito de monitor é, por si só, insuficiente para suportar a implementação dos algoritmos de sincronização, ainda que assuma um papel importante como infra-estrutura de suporte.

2.2 Semântica de Lampson e Redell

Em [12], Lampson e Redell identificam os problemas do monitor de Brinch Hansen e Hoare na implementação de sistemas reais e apresentam outra semântica – implementada na linguagem Mesa – para a espera sobre as variáveis condição: quando uma *thread* estabelece uma condição que é esperada por outra *thread*, eventualmente bloqueada, é notificada a respectiva variável condição. Assim, a operação de `notify` é entendida como aviso ou conselho à *thread* bloqueada, e tem como consequência que esta retome a execução algures no futuro. O monitor é readquirido quando uma *thread* bloqueada por `wait` retoma a execução. Não existe garantia de que qualquer outra *thread* não entre no monitor antes da *thread* notificada – em [1], esta possibilidade é designada por *barging*. Ainda, após o retorno de `wait`, nada é garantido para além da invariante do monitor, pelo que é obrigatório que as *threads* notificadas reavaliem a situação, voltando a testar o predicado que conduziu ao bloqueio. Em contrapartida, não

existem comutações adicionais de *threads* nem restrições relativamente a quando a *thread* notificada deve prosseguir a execução após `wait`. Com esta semântica, é possível acrescentar mais três formas de acordar as *threads* bloqueadas nas variáveis condição: por ter sido excedido um limite especificado para o tempo de espera (*timeout*), interrupção ou aborto da *thread* bloqueada e fazer *broadcast* da condição, isto é, notificar todas as *threads* nela bloqueadas.

A semântica de Lampson e Redell tem, relativamente à semântica de Hoare, as seguintes vantagens: diminui, de duas para zero, o número de comutações de contexto obrigatórias no processo de notificação, permite que a espera possa ser interrompida por exceder um limite de tempo ou por interrupção ou aborto das *threads* bloqueadas e permite o *barging* [1] no acesso ao monitor que o pode ser vantajoso para aumentar as possibilidades de concorrência em sistemas multiprocessador.

Um inconveniente desta semântica de monitor é que conduziu a uma infra-estrutura de suporte à sincronização menos flexível: deixou de ser possível passar informação entre *threads* no processo de notificação, como acontecia na sinalização de Hoare. Por exemplo, a solução para os leitores escritores apresentada por Hoare em [10] não se pode implementar com semântica de Lampson e Redell. Com esta semântica não é possível distinguir os leitores que se encontram bloqueados no momento em que a escrita termina dos outros que possam posteriormente aceder ao monitor. Outro inconveniente desta semântica é tornar possível a perda de notificações. Notificações em excesso não são problema, pois esta semântica determina que a *thread* notificada tem que reavaliar o predicado após adquirir a posse do monitor. A perda de notificações pode criar situações em que existem *threads* bloqueadas que, face ao estado corrente do monitor, lá não deveriam estar. A razão porque se podem perder notificações é que a execução do `wait` pode envolver duas situações de bloqueio: uma a aguardar a notificação na variável condição e outra a aguardar a reaquisição do *lock* do monitor. Se a implementação permitir a interrupção de qualquer destas esperas, uma *thread* pode ser notificada na variável condição e ser interrompida quando se encontra bloqueada para readquirir o monitor, retornando com excepção do método `wait`; se esta excepção não for capturada e regenerada a notificação, esta será perdida. É necessário ter em atenção esta particularidade, pois a mesma pode causar erros subtis nos algoritmos de sincronização.

Sendo o cancelamento da espera em condições excepcionais indispensável em sistemas reais e a necessidade de optimizar o número de comutações na notificação, considera-se que a contribuição de Lampson e Redell tornou o conceito de monitor mais adequado para implementar sistemas reais.

2.3 Monitores Intrínsecos em CLI e Java

A linguagem de programação Java e as linguagens suportadas pela CLI são linguagens com ampla divulgação que incluem suporte para programação concorrente. Estas linguagens permitem criar novas *threads* e controlar a sua execução e usam o conceito de monitor com semântica de Lampson e Redell como suporte à implementação de exclusão mútua e de sincronização. Os monitores intrínsecos em CLI e Java apenas suportam uma variável condição anónima, pelas razões que são explicadas a seguir.

Antes de analisarmos as consequências desta restrição na concepção de algoritmos de sincronização, é importante identificar a sua origem. Entre outras, identificam-se as seguintes razões.

Em CLI e Java, qualquer objecto tem implicitamente associado um monitor. Contudo, a associação real do monitor ao objecto é feita no último instante possível (*lazy*), isto é, quando a funcionalidade do monitor é solicitada pela primeira vez. Os monitores físicos são geridos em *pool* pela máquina virtual. Mais, quando um objecto tem associado um monitor este não está a ser necessário (não está na posse de nenhuma *thread* nem tem *threads* bloqueadas na condição), a máquina virtual pode cancelar a associação do monitor físico ao objecto e devolvê-lo ao controlo do *pool* para posterior reutilização pelo mesmo ou por outro objecto. Uma das razões para os monitores serem geridos desta forma é o facto de nas aplicações típicas serem poucos os objectos que usam o monitor implícito. Por isso, a mobilização do monitor apenas quando este é necessário, resulta numa economia de memória significativa (e.g., cada monitor *heavyweight* no JDK 1.5 ocupa 32 *byte*).

Para permitir esta gestão optimizada, os monitores tem que ser todos iguais do ponto de vista da implementação. O número de variáveis condição de um monitor determina o número de filas de espera que têm que ser implementadas e a memória que é necessária para armazenar o estado do monitor. Se o número de condições pudesse ser definido com base no objecto, seria mais complexa a gestão dos *pools* de monitores pela máquina virtual e seria necessário reservar memória em todos os objectos para

armazenar o número de condições, sempre que o objecto não tivesse um monitor físico associado. Suportar monitores com um número arbitrário de variáveis condição coloca também o problema de como e quando se define o número de variáveis condição do monitor implícito de um objecto particular. Resumindo, existiram fortes razões para que os projectistas do Java e da CLI tivessem optado por associar a cada monitor implícito apenas uma variável condição anónima.

No Java 5 foram introduzidos monitores explícitos que suportam um número arbitrário de variáveis condição, criadas dinamicamente. (Como estes monitores são implementados por objectos Java, não estão sujeitos às restrições dos monitores implícitos, atrás referidas.)

O *lock* usado nos monitores implícitos do Java e CLI admite entradas recursivas o que não acontecia nos monitores de Hoare e de Lampson e Redell. Uma *thread* pode entrar um número arbitrário de vezes no mesmo monitor. Para abandonar o monitor, é necessário sair do monitor o mesmo número de vezes que entrou.

Em Java os procedimentos de entrada do monitor são os métodos com o qualificativo `synchronized`. Podem também ser definidos procedimentos de entrada usando blocos de código com atributo `synchronized`. A funcionalidade de bloqueio e notificação é acedida através de métodos de instância da classe `java.lang.Object`. O método `Object.wait` bloqueia a *thread* chamadora na variável condição do monitor. Existe uma versão deste método que aceita como argumento o número máximo de milésimos de segundo que dura o bloqueio. O método `Object.wait` pode retornar: por nenhuma razão especial (segundo a especificação da JVM, algumas máquinas virtuais podem exibir este comportamento), porque a *thread* foi notificada ou por ter decorrido o tempo máximo de espera. Este método lança a `InterruptedException` se a *thread* for interrompida durante a espera. Quando `Object.wait` é invocado sem que a *thread* tenha entrado no monitor o método lança `IllegalMonitorStateException`. O método `Object.wait` não devolve informação, pelo que não existe forma de distinguir o retorno espúrio do retorno por notificação ou por limite de tempo. O método `Object.notify` notifica uma das *threads* bloqueadas na condição do monitor e o método `Object.notifyAll` notifica as *threads* todas (funcionalidade *broadcast* proposta por Lampson e Redell).

Na CLI, os monitores dos objectos são manipulados através de métodos estáticos da classe `System.Threading.Monitor`, que recebem como argumento a referência para objecto que define o monitor. Os métodos `Monitor.Enter` e `Monitor.Exit` permitem, respectivamente, entrar e sair do monitor. O método `Monitor.TryEnter` permite a entrada condicional num monitor: se o monitor ficar livre dentro do tempo especificado para *timeout*, a *thread* invocante assume a sua posse; no caso contrário, a *thread* retorna sem adquirir a posse do monitor. O método `Monitor.Wait` bloqueia a *thread* invocante na variável condição do monitor. Existem versões deste método que permitem especificar, de várias formas, um limite para o tempo de bloqueio. O método `Monitor.Wait` retorna quando a *thread* é notificada, por ter expirado o limite de tempo especificado. O método `Monitor.Wait` lança as seguintes excepções: `ThreadInterruptedException` se a *thread* for interrompida durante a espera ou quando tentava reentrar no monitor; `SynchronizationLockException` se a *thread* não entrou no monitor; e, `ArgumentNullException` se for passada uma referência nula. O método `Monitor.Pulse` notifica uma *thread* bloqueada na condição do monitor e o método `Monitor.PulseAll` notifica as *threads* todas.

Não existem diferenças semânticas significativas entre os monitores do Java e da CLI, para além do facto da entrada e saída dos monitores em Java ser obrigatoriamente aninhada, pois a entrada e saída do monitor está subjacente às chamadas e retorno dos métodos e à entrada e saída de blocos de código sincronizados. Em CLI, a entrada e saída do monitor pode ser feita em qualquer ponto do programa, basta invocar o respectivo método. Pode ser aplicado aos métodos o atributo `MethodImplAttribute` com o argumento `MethodImplOption.Synchronized` para obter o mesmo efeito que o qualificativo `synchronized` nos métodos Java. Algumas linguagens suportadas pelo CLI oferecem uma construção que permite definir blocos de código sincronizados. Em C# essa construção é o `lock` [5], que tem o mesmo aspecto e semântica dos blocos sincronizados Java.

A diferença significativa entre os monitores nas duas infra-estruturas é a natureza aninhada da entrada/saída dos monitores no Java e a ausência de aninhamento no CLI. Ainda, que a opção do Java evite erros do programador, a flexibilidade do CLI permite simplificar algoritmos que envolvem mais do que um monitor, como se mostra na Secção 4.

Nos exemplos de código a seguir apresentados é usada a linguagem C# sempre que for usada a semântica comum a CLI e Java e versões em C# e Java quando isso não acontecer.

O monitor do Java e da CLI não permite distinguir as *threads* bloqueadas na única variável condição nem permitem passar informação entre *threads* na notificação (consequência da semântica de Lampson e Redell). Acrescente-se que a especificações CLI e Java admitem que podem ser perdidas notificações quando as mesmas ocorrem em simultâneo com a interrupção ou aborto das *threads*. (O JSR 133 [13] faz uma precisão à especificação do Java que elimina a hipótese de perda de notificações, mas a análise do código das classes de biblioteca no Java 5 e 6 mostra que o código continua prevê essa possibilidade.)

Resumindo, com os monitores Java e CLI:

a) Não é possível distinguir as *threads* bloqueadas no monitor, nem mesmo saber se existem *threads* bloqueadas;

b) Toda a troca de informação entre *threads* tem que ser realizada através de variáveis de estado partilhadas e protegidas pelo monitor;

c) Podem ser perdidas notificações, quando estas ocorrerem em simultâneo com a interrupção das *threads*.

d) Não se pode assumir ordenação das *threads* presentes nas filas de espera do monitor (fila de entrada e fila da variável condição).

e) Pode ocorrer o retorno do *wait* sem haver notificação nem ocorrer *timeout* ou interrupção (especificação da JVM).

Estas características têm as seguintes implicações na concepção dos algoritmos de sincronização:

– É necessário usar *broadcast* sempre que podem estar no monitor *threads* bloqueadas por mais do que uma razão. Por exemplo, na implementação do *bounded buffer* podem estar bloqueadas em simultâneo *threads* produtoras e *threads* consumidoras. Por isso, quando o consumidor recolhe informação do *buffer*, ou quando o produtor coloca informação no *buffer*, é necessário notificar todas as *threads* para garantir a notificação da *thread* interessada na alteração de estado produzida.

- Nas circunstâncias em que é possível usar *notify* – todas as *threads* estão bloqueadas pela mesma razão (avaliam o mesmo predicado) e apenas uma delas pode prosseguir a execução – é sempre necessário capturar a excepção de interrupção e invocar *notify* para garantir a regeneração de uma notificação que possa ser perdida devido a interrupção da *thread* em espera na variável condição.
- Quando se pretendem implementar critérios de ordenação nas filas de espera das *threads* bloqueadas no monitor – por exemplo, FIFO – é necessário implementar explicitamente as respectivas filas de espera.

A maior parte dos erros em algoritmos de sincronização em código CLI e Java resultam de não serem tidas em consideração todas as implicações da semântica dos monitores do CLI e Java. Na próxima secção são apresentados padrões de código genéricos para implementar algoritmos de sincronização em CLI e Java.

3 Algoritmos de Sincronização em CLI e Java

Apesar de se encontrar na literatura referências às dificuldades que a semântica dos monitores CLI e Java colocam aos projectistas, considera-se os monitores são uma infra-estrutura versátil, ainda que de baixo nível, para suporte à implementação de algoritmos de sincronização.

Considera-se que qualquer programador familiarizado com a programação concorrente e conhecedor da semântica do monitor CLI e Java não terá dificuldade em implementar correctamente algoritmos de sincronização nestas plataformas. O facto de ser necessário usar sistematicamente *broadcast* na notificação, pelas razões discutidas anteriormente, não aumenta a dificuldade na concepção dos algoritmos, ainda que possa degradar o seu desempenho, devido a ocorrência de comutações de contexto adicionais, sempre que se notificam *threads* que não podem prosseguir a execução. Na Secção 4 apresenta-se uma solução para minimizar esta ineficiência.

O facto de não se separar as preocupações relativas à correcção dos algoritmos de sincronização do respectivo desempenho é a principal razão dos erros mais comuns. Um dos erros mais frequentes decorre da substituição da notificação por *broadcast* pela notificação por *notify*, o que compromete a correcção sempre que a interrupção de uma *thread* resulta na omissão de uma notificação relevante.

A seguir, são apresentados os padrões de código para algoritmos de sincronização, usando os problemas clássicos apresentados na literatura. Para todos os exemplos, são discutidas os requisitos do problema e as respectivas soluções.

Os algoritmos apresentados processam a excepção de interrupção e alguns deles ilustram as consequências da especificação de limite ao tempo de espera. Ainda que a limitação do tempo de espera seja optativa, o processamento das interrupções nos algoritmos de sincronização é obrigatório. Em CLI e Java, a interrupção de *threads* é uma facilidade de utilização geral que pode ser usada por qualquer componente *software*.

3.1 Exemplo 1

Nas situações em que existe apenas uma classe de *threads* bloqueadas no monitor (i.e., todas as *threads* avaliam o mesmo predicado) e por cada notificação apenas uma *thread*

pode prosseguir a execução, é possível usar *notify* (em CLI, método `Monitor.Pulse`). Neste caso, o tratamento da `ThreadInterruptedException` é obrigatório, pois a omissão de uma notificação compromete a correcção do algoritmo.

A limitação do tempo de espera (*timeout*) levanta dois problemas. O primeiro é a correcta contagem do tempo. Na implementação de algoritmos de sincronização usando monitores com semântica de Lampson e Redell, é necessário considerar que as *threads* podem ser acordadas várias vezes, antes do predicado que condiciona o seu progresso ser verdadeiro. Isto obriga a que o valor do *timeout* tenha que ser ajustado após cada invocação do método `Wait`. Para implementar este algoritmo, é necessário adquirir uma referência temporal antes de invocar `Wait` e ajustar o *timeout* do próximo `Wait` em função do instante em que se faz a invocação. No código apresentado neste texto, é usado método estático `SyncUtils.AdjustTimeout` (cujo código é mostrado a seguir) que recebe como argumentos o instante a que se refere o valor do *timeout* (`lastTime`) e a referência para a variável com o valor do *timeout* em milésimos de segundo. A função devolve por valor o *timeout* remanescente e afecta com esse valor a variável cuja referência é passada no segundo argumento e actualize o valor de `lastTime` com o tempo presente.

```
public static class SyncUtils {
   ...
   public static int AdjustTimeout(ref int lastTime, ref int timeout) {
      if (timeout != Timeout.Infinite) {
         int now = Environment.TickCount;
         int elapsed = (now == lastTime) ? 1 : now - lastTime;
         if (elapsed >= timeout) {
            timeout = 0;
         } else {
            timeout -= elapsed;
            lastTime = now;
         }
      }
      return timeout;
   }
   ...
}
```

Esta função trata como caso particular o valor de *timeout* infinito (`Timeout.Infinite`) e ajusta o *timeout* de exactamente de um milésimo de segundo quando o tempo decorrido entre ajustes for inferior.

O segundo problema colocado pelo processamento do *timeout* é decidir o que fazer quando a situação de *timeout* ocorre em "simultâneo" com uma notificação. A solução mais simples é considerar que houve notificação e ignorar o *timeout*. Assim, após o

retorno do método `Monitor.Wait`, é dada prioridade à avaliação do predicado e, só depois, considerada a condição de *timeout*.

Neste exemplo, é implementado o semáforo, com operações `Acquire(timeout)` e `Release()`. A operação `Acquire(timeout)` aguarda a obtenção de uma autorização no semáforo no máximo `timeout` milésimos de segundo; se esse tempo expirar sem que seja possível obter a unidade, o método retorna com indicação de *timeout*. A operação `Release()` devolve uma autorização ao controlo do semáforo.

Este é um tipo de problema em que pode ser usada notificação simples, porque a invocação de `Release()` apenas cria condições para a continuação de uma *thread* bloqueada e pode ser seleccionada qualquer uma das *threads* bloqueadas (não se pretende implementar um critério de ordenação específico da fila de espera).

O código que implementa o semáforo contador é o seguinte:

```
public sealed class Semaphore {
  private int permits;

  // Constructor
  public Semaphore(int initial) {
    if (initial > 0)
      permits = initial;
  }

  // Acquire one permit from the semaphore
  public bool Acquire(int timeout) {
    lock(this) {
      if (permits > 0) {
        permits--;
        return true;
      }
      // if a timeout was specified, get a time reference
      int lastTime = (timeout != Timeout.Infinite) ?
                           Environment.TickCount : 0;
      // loop until one permit is available, the specified timeout expires or
      // the thread is interrupted.
      do {
        try {
          Monitor.Wait(this, timeout);
        } catch (ThreadInterruptedException) {
          // if we were interrupted and there are permits available, we can have been
          // notified and interrupted.
          // so, we leave this method throwing ThreadInterruptException, but before
          // we regenerate the notification, if there are available permits
          if (permits > 0) {
            Monitor.Pulse(this);
          }
          throw;          // re-throw exception
        }
        if (permits > 0) {     // permits available, decrement and return
          permits--;
```

```
            return true;
          }
          if (SyncUtils.AdjustTimeout(ref lastTime, ref timeout) == 0){
            // the specified timeout elapsed, so return failure
            return false;
          }
        } while (true);
      }
    }

    // Release one permit
    public void Release(){
      lock(this) {
        permits++;
        Monitor.Pulse(this);  // only one thread can proceed execution
      }
    }
}
```

Na definição desta classe, constata-se que o processamento da excepção de interrupção e da limitação ao tempo de espera é responsável por um número significativo de linhas de código. No entanto, a inclusão deste código não levanta qualquer problema, pois o seu padrão é independente do algoritmo de sincronização. A única parte que depende do algoritmo é a resposta concreta à desistência por interrupção ou *timeout*. Neste caso, regenera-se a notificação quando a *thread* é interrompida durante a espera e existem autorizações disponíveis no semáforo; quando ocorre *timeout* não é necessário fazer nada.

Nos algoritmos de sincronização, deve ser optimizada a sequência de instruções que corresponde ao percurso que não bloqueia a *thread*, por estando tipicamente envolvidas poucas instruções máquina, mesmo optimizações simples podem ser significativas. Por outro lado, omitir algumas instruções máquina na sequência que inclui o bloqueio da *thread* não tem qualquer impacto no desempenho, porque esse caminho envolve, no mínimo, duas comutações de contexto. Neste e nos outros exemplos de código que são apresentados a seguir, segue-se este princípio.

Em CLI, o método `Monitor.Wait` retorna um booleano que indica se ocorreu, ou não, *timeout*. Em Java, o método `Object.wait` não devolve essa indicação. Optou-se por não usar o valor de retorno do `Monitor.Wait` para que os algoritmos possam ser usados nas duas plataformas.

3.2 Exemplo 2

Este exemplo, ilustra a situação mais geral, onde é necessário notificar com *broadcast* para ter a garantia de que são notificadas todas as *threads* interessadas na alteração do estado partilhado no monitor. A alternativa à utilização do *broadcast* é gerar um encadeamento de notificações equivalente, o que é mais difícil de programar correctamente. Quando se notifica com *broadcast*, pode não ser necessário processar a excepção de interrupção, pois nunca é necessário regenerar notificações omitidas. Como são sempre notificadas todas as *threads* bloqueadas no monitor, uma eventual *thread* interrompida não tem responsabilidade especial, como acontece no exemplo anterior onde a *thread* notificada é a destinatária da autorização entregue ao semáforo pela operação `Release()`. Nessa situação, se a *thread* notificada desistir de receber a autorização, tem a responsabilidade de notificar outra *thread*. Quando se usa o padrão de código com *broadcast*, só é necessário processar a excepção de interrupção quando a desistência da *thread*, por interrupção, implica a alteração aos dados partilhados protegidos pelo monitor.

O seguinte código implementa um semáforo, com operações `Acquire(n, timeout)` e `Release(n)`:

```
public sealed class Semaphore_2 {
  private int permits = 0;

  // Constructor
  public Semaphore_2 (int initial) {
    if (initial > 0)
      permits = initial;
  }

  // Acquire n permits
  public bool Acquire(int n, int timeout) {
    lock(this) {
      // if there are sufficient permits, update semaphore value and return success
      if (permits >= n) {
        permits -= n;
        return true;
      }
      // get a time reference if a timeout was specified
      int lastTime = (timeout != Timeout.Infinite) ?
                        Environment.TickCount : 0;
      // loop until acquire n permits, the specified timeout expires or thread
      // is  interrupted
      do {
        Monitor.Wait(this, timeout);
        if (permits >= n) {   // permits available
          permits -= n;
          return true;
        }
        if (SyncUtils.AdjustTimeout(ref lastTime, ref timeout) == 0){
          // timeout expired
```

```
            return false;
          }
        } while (true);
      }
    }

    // Release n permits
    public void Release(int n){
      lock(this) {
        permits += n;
        Monitor.PulseAll(this);
      }
    }
}
```

Um semáforo com esta implementação é injusto para as *threads* que solicitam múltiplas autorizações. As *threads* que solicitam poucas autorizações podem nunca deixar que o número de autorizações disponíveis no semáforo seja suficiente para satisfazer os pedidos das *threads* que solicitam mais autorizações. Para resolver este problema é necessário dotar o semáforo de uma fila de espera com disciplina FIFO, como se ilustra no próximo exemplo.

3.3 Exemplo 3

Quando se pretende impor uma disciplina particular (e.g., FIFO) na aquisição de um recurso subjacente ao monitor, é necessário implementar explicitamente a fila de espera, porque os monitores CLI e Java não garantem nenhuma ordenação nas respectivas filas de espera. Assim, considera-se que a fila onde se encontram as *threads* bloqueadas no monitor é mais um componente do estado partilhado, que é protegido pelo monitor.

O código que implementa o semáforo com as operações `Acquire(n)` e `Release(n)` e fila de espera FIFO é o seguinte:

```
public sealed class Semaphore_FIFO {
  private int permits = 0;
  // Queue of waiting threads.
  // For each waiting thread we hold an integer with the number of
  // permits requested. This allow us optimizing the notifications as we
  // can see in the method notifyWaiters ()
  private readonly LinkedList<int> queue = new LinkedList<int>();

  // The constructor.
  public Semaphore_FIFO(int initial) {
    if (initial > 0)
      permits = initial;
  }

  // Notify the waiting threads, if the number of permits is greater or equal than
  // the request of the waiting thread that is at front of queue.
  private void notifyWaiters() {
    if (queue.Count > 0 && permits >= queue.First.Value) {
```

```
      Monitor.PulseAll(this);
    }
  }

  // Acquire n permits within the specified timeout
  public bool Acquire(int n, int timeout){
    lock(this) {
      // if the queue is empty and there are sufficient permits, acquire immediately
      if (queue.Count == 0 && permits >= n) { // entry predicate
        permits -= n;
        return true;
      }
      // enqueue our request
      LinkedListNode<int> rn = queue.AddLast(n);
      // if a timeout was specified, get a time reference
      int lastTime = (timeout != Timeout.Infinite) ?
                          Environment.TickCount : 0;
      do {
        try {
          Monitor.Wait(this, timeout);
        } catch (ThreadInterruptedException) {
          // interrupted exception: give up processing
          queue.Remove(rn);   // remove request node from the queue
          notifyWaiters();    // notify waiters, because queue was modified
          throw;              // re-throw interrupted exception
        }
        // predicate after wakeup: if we are at front of queue, ckeck if there
        // sufficient permits available
        if (rn == queue.First && permits >= n) {
          queue.Remove(rn);   // dequeue request
          permits-= n;        // get permits
          notifyWaiters();    // notify waiters, because the shared state was modified
          return true;
        }
        // adjust the timeout and check if it expired
        if (SyncUtils.AdjustTimeout(ref lastTime, ref timeout) == 0){
          // timeout: give up processing
          queue.Remove(rn);   // remove request item from queue
          notifyWaiters();    // notify waiters, because queue was modified
          return false;
        }
      } while (true);
    }
  }

  // Acquire n permits, without timeout
  public void Acquire(int n){
    Acquire(n, Timeout.Infinite);
  }

  // Release n permits
  public void Release(int n){
    lock(this) {
      //update the available permits and notify waiters because the shared
      // state was mofified
      permits += n;
      notifyWaiters();
    }
  }
```

```
}
```

Neste algoritmo e nos seguintes, usamos a classe genérica `LinkedList<T>`, definida no espaço de nomes `System.Collections.Generic`, para implementar as filas de espera de *threads*. Esta classe implementa as operações usadas no algoritmo – inserção no fim da lista e remoção de elementos a partir da sua referência – com custo constante (O(1)). Esta classe implementa uma lista duplamente ligada cujos nós são instâncias do tipo `LinkedListNode<T>`. Usando este tipo e um tipo valor para armazenar os dados associados a cada *thread* em espera, apenas é necessário criar um objecto (i.e., uma instância de `LinkedListNode<int>`) pode cada *thread* que se bloqueia no semáforo.

Neste sincronizador, o estado partilhado é composto pelo contador com o número de autorizações disponíveis e pela fila de espera de *threads*. A existência da fila de espera de *threads* obriga à definição de dois predicados: o predicado de entrada e o predicado após bloqueio. Quando uma *thread* avalia o predicado de entrada, não se encontra inserida na fila de espera, o que não acontece quando a mesma *thread* avalia o predicado após bloqueio. Neste exemplo, para que a *thread* obtenha as autorizações que pretende do semáforo na entrada do método, é necessário que a fila de espera esteja vazia e que o número de autorizações disponíveis seja suficiente. Por outro lado, após bloqueio, a *thread* tem que se encontrar à cabeça da fila de espera e as autorizações disponíveis serem suficientes. É ineficiente inserir sempre um item na fila de espera para diminuir o número de linhas de código, pois isso envolve a criação e a destruição de uma instância do tipo `LinkedListNode<int>`. Existe ainda outra razão para distinguir a situação de entrada no método da situação após bloqueio, que reside no facto da *thread* ter normalmente, nas duas situações, responsabilidades diferentes relativamente à notificação de outras *threads*. Neste exemplo, a aquisição de autorizações no semáforo na entrada do método só é possível quando não existem *threads* bloqueadas, pelo que a *thread* não tem a responsabilidade de notificar outras *threads*, o que não acontece quando as autorizações pretendidas são adquiridas após bloqueio.

Como os monitores CLI e Java não permitem distinguir as *threads* bloqueadas, todas as *threads* bloqueadas tem que ser notificadas sempre que é alterado o estado partilhado: número de autorizações disponíveis e o estado da fila de espera. Isso acontece quando a *thread* é notificada e obtém as autorizações (altera a fila de espera e o número de

autorizações do semáforo); quando a *thread* é interrompida (altera a fila de espera), e; quando ocorre *timeout* (altera a fila de espera).

Neste exemplo, existe no monitor informação sobre as *threads* bloqueadas – a instância de `LinkedListNode<int>` que está na fila de espera. Se o monitor permitisse notificar individualmente cada uma das *threads* bloqueadas, o algoritmo produzia o número de comutações de *thread* estritamente necessário. (A notificação específica de *threads* é discutida na Secção 4.)

A gestão da fila de espera de *threads* simplifica-se quando a operação de inserção e remoção na fila é feita pela própria *thread*, porque, quando a *thread* acorda, não tem que determinar se está, ou não, inserida na fila. Não é possível usar este critério quando se usa delegação da execução, como veremos no exemplo seguinte.

Uma consequência benéfica da existência da fila é que permite saber se existem *threads* bloqueadas no semáforo e qual o número de unidades que cada *thread* pretende adquirir. Dispondo desta informação, é possível omitir invocações desnecessárias ao método `Monitor.PulseAll`, sempre que não existem condições para que nenhuma *thread* em espera possa prosseguir a execução (no exemplo anterior, isso não era possível porque não se dispunha de informação sobre os pedidos das *threads* bloqueadas).

Quando ocorre a excepção de interrupção, é necessário remover a *thread* da fila de espera e, eventualmente, notificar as outras *threads* bloqueadas. A *thread* interrompida podia estar no início da fila de espera e o número de autorizações disponíveis no semáforo ser maior do que zero, mas insuficiente para satisfazer o seu pedido. Contudo, é possível que as *threads* que se encontram a seguir na fila de espera tenham condições para satisfazer os respectivos pedidos. Ocorre uma situação idêntica, quando uma *thread* desiste por exceder o limite de tempo de espera. Em todos os algoritmos em que se implementam explicitamente filas de espera de *threads* é necessário considerar as consequências do cancelamento de pedido nas outras *threads* bloqueadas no monitor.

Os padrões de código até agora apresentados têm uma característica comum: as *threads* bloqueadas no monitor são desbloqueadas para avaliarem os respectivos predicados e realizam a acção subjacente ao pedido (neste caso, obter autorizações do semáforo) se o predicado for verdadeiro. Como os monitores CLI e Java admitem *barging*, qualquer *thread* pode entrar no monitor na janela temporal definida pelo instante da notificação e o instante que a *thread* notificada reentra no monitor. O *barging* não coloca problema,

porque o predicado na entrada do método `Wait` considera o estado da fila de espera. Neste exemplo, como as *threads* bloqueadas estão inseridas na fila de espera e só são removidas quando satisfazem o respectivo pedido, o estado da fila de espera permite que qualquer *thread* que entre pela primeira vez no monitor determine se tem acesso às unidades do semáforo ou se tem que se colocar na fila de espera. Apesar dos monitores CLI e Java permitirem *barging* sobre o monitor, este algoritmo impede o *barging* sobre o semáforo. No entanto, existem problemas em que as decisões de *scheduling* têm em consideração transições do estado do monitor e que não são passíveis de resolver usando o padrão de código apresentado neste algoritmo. O exemplo seguinte ilustra estas situações.

3.4 Exemplo 4

Neste exemplo, resolve-se o problema dos leitores escritores com o critério de *scheduling* proposto por Hoare [10]. Para evitar que o acesso ao recurso controlado pelo monitor possa ser negado a leitores e/ou escritores, Hoare propõe o seguinte critério de *scheduling*: a solicitação de um acesso para escrita interrompe a sequência de acessos para leitura; concluída a escrita, todos os leitores que se encontram em fila de espera são desbloqueados. Com este critério de *scheduling*, sequências infinitas de leituras não impedem a escrita, nem sequências infinitas de escritas impedem a leitura.

A maior dificuldade neste problema é permitir que, após a escrita, sejam autorizadas todas as leituras bloqueadas, e apenas essas. Seguindo estritamente este critério, se no decurso da escrita se bloquearem L leitores e um escritor, após a conclusão da escrita devem ser permitidas exactamente L leituras. Qualquer pedido de leitura que chegue depois terá de ser atrasado até que o escritor bloqueado conclua a escrita. Portanto, no momento em que termina a escrita têm que ser autorizados os pedidos pendentes de acesso para leitura; isto é, facultar o acesso aos L leitores bloqueados, mas não aos outros leitores que possam entrar no monitor antes dos leitores notificados o conseguirem (devido ao *barging*).

A solução genérica para este tipo de problemas, usando monitores com a semântica de Lampson e Redell, é usar uma técnica que designaremos por delegação da execução: as operações sobre o monitor que tenham que ser atrasadas são realizadas, não pela *thread* que invoca a operação, mas pelas *threads* que criam, posteriormente, as condições para

que a operação se realize. O seguinte código demonstra a utilização da técnica da delegação da execução.

```
public sealed class ReadersWriters {
   private int  readers = 0;       // current number of readers
   private bool writing = false;   // true when writing

   // We use a queue for waiting readers and a queue for waiting writers.
   // For each queue node holds a Boolean that says if the requested
   // access was already granted or not.
   private readonly LinkedList<bool> rdq = new LinkedList<bool>();
   private readonly LinkedList<bool> wrq = new LinkedList<bool>();

   // Constructor.
   public ReadersWriters(){}

   // Acquire read (shared) access
   public void StartRead() {
      lock(this) {
         // if there isn't blocked writers and the resource isn't being written, grant
         // read access immediately
         if (wrq.Count == 0 && !writing) {
            readers++;
            return;
         }
         // enqueue a read access request
         LinkedListNode<bool> rd = rdq.AddLast(false);
         do {
            try {
               Monitor.Wait(this);
            } catch (ThreadInterruptedException) {
               // if the requested shared access was granted, we must re-assert exception,
               // and return normally.
               // otherwise, we remove the request from the queue and re-throw exception
               if (rd.Value) {
                  Thread.CurrentThread.Interrupt();
                  return;
               }
               rdq.Remove(rd);
               throw;
            }
            // if shared access was granted then return; otherwise, re-wait
            if (rd.Value)
               return;
         } while (true);
      }
   }

   // Acquire write (exclusive) access
   public void StartWrite() {
      lock(this) {
         // if the resource isn't being read nor writing and the writers' wait queue is
         // empty, grant the access immediately
         if (readers == 0 && !writing && wrq.Count == 0) {
            writing = true;
            return;
         }
         // enqueue a exclusive access request
         LinkedListNode<bool> wr = wrq.AddLast(false);
```

```
      do {
         try {
            Monitor.Wait(this);
         } catch (ThreadInterruptedException) {
            // if exclusive access was granted, then we re-assert exception,
            // and return normally
            if (wr.Value) {
               Thread.CurrentThread.Interrupt();
               return;
            }
            // when a waiting writer gives up, we must grant shared access to all
            //  waiting readers that has been blocked by this waiting writer
            wrq.Remove(wr);
            if (!writing && wrq.Count == 0 && rdq.Count > 0) {
               do {
                  rdq.First.Value = true;  // signal reader that access was granted
                  rdq.RemoveFirst();
                  readers++;       // account shared access
               } while (rdq.Count > 0);
               Monitor.PulseAll(this);    // wakeup readers
            }
            throw;
         }
         // if the write access request was granted, return; else, re-wait
         if (wr.Value) {
            return;
         }
      } while (true);
   }
}

// Release read (shared) access
public void EndRead() {
   lock(this) {
      readers--;     // decrement the number of readers
      // if this is the last reader, and there is at least a blocked writer, grant access
      // to the writer that is at front of queue
      if (readers == 0 && wrq.Count > 0) {
         wrq.First.Value = true;    // mark the exclusive access request as granted
         wrq.RemoveFirst();
         writing = true;            // accomplish write
         Monitor.PulseAll(this);    // notify writer
      }
   }
}

// Release write (exclusive) access
public void EndWrite() {
   lock(this) {
      //  release the exclusive access
      writing = false;
      //if there are blocked readers, grant shared access to all of them;
      // otherwise, if there is at least a waiting writer, grant the exclusive access to it
      if (rdq.Count > 0) {
         do {
            rdq.First.Value = true; // signal reader that shared access was granted
            rdq.RemoveFirst();
            readers++;       // accomplish read
         } while (rdq.Count > 0);
```

```
            Monitor.PulseAll(this);    // notify readers
          } else if (wrq.Count > 0) {
            // grant exclusive access to the next writer
            wrq.First.Value = true;    // mark the shared access as granted
            wrq.RemoveFirst();
            writing = true;     // accomplish write
            Monitor.PulseAll(this); // notify the writer
          }
        }
      }
    }
```

Este algoritmo usa a técnica da delegação da execução e funciona do seguinte modo: ao entrar no monitor para realizar uma operação bloqueante, cada *thread* avalia um predicado para determinar se pode realizar a operação; em caso, afirmativo, modifica os dados partilhados e, se necessário, notifica as outras *threads* dessa modificação; no caso contrário, cria um *request item*, para descrever a operação e os seus argumentos de entrada, saída e entrada/saída, e insere-o na respectiva fila de espera; quando cada *thread* é acordada, esta testa o respectivo *request item* para determinar se a operação já foi realizada; em caso afirmativo, abandona o monitor, no caso contrário, volta a bloquear-se.

Neste exemplo, como cada operação usa a sua fila de espera e as operações (acesso para leitura e acesso para escrita) não têm argumentos, o *request item* (`LinkedListNode<bool>`) contém apenas um campo do tipo booleano para indicar se a operação já foi concluída.

A utilização da delegação de execução resolve o problema criado pela possibilidade de *barging* nos monitores de Lampson e Redell. Como não é possível garantir que outras *threads* entrem no monitor entre o momento em que uma *thread* bloqueada é notificada e o momento em que esta reentra no monitor, usamos um *request item* com toda a informação que é necessária para que a operação possa ser realizada antes da notificação e comunica-se à *thread* notificada apenas a conclusão da operação e os seus resultados, se existirem. Isto permite realizar atomicamente a alteração ao estado partilhado e consumar as respectivas consequências sobre as *threads* em espera. Neste exemplo, quando é sinalizado o fim de uma escrita e existem L *threads* leitoras bloqueadas, o próximo estado partilhado indica que estão L *threads* em leitura; além disso cada uma das *threads* leitoras a quem foi garantido o acesso recebe essa informação através do respectivo *request item*. Neste padrão de código, o predicado

avaliado por cada *thread* após bloqueio envolve apenas informação armazenada no respectivo *request item*.

Quando as operações sobre o monitor têm afinidade à *thread* que as invoca, não pode ser usada a técnica da delegação da execução. Contudo, este tipo de solução tem generalidade, porque a maior parte dos sincronizadores implementam o controlo do acesso aos recursos, não os próprios recursos. Saliente-se que, tipicamente, a afinidade à *thread* é característica da operação sobre o recurso.

A utilização da delegação da execução neste problema permite consolidar atomicamente decisões de *scheduling* que estão associadas a transições do estado partilhado. Neste exemplo, é efectivamente iniciada a leitura por parte de todos os leitores bloqueados na sequência da terminação da escrita. Se as *threads* leitoras fossem libertadas sem incrementar o contador `readers` (consumado o início da leitura), era possível que outra *thread* entrasse no monitor e iniciasse a escrita (porque `writing` é `false` e `readers` é `0`), o que contrariava o critério de *scheduling*.

Como acontece no Exemplo 3, têm que ser analisadas as consequências do cancelamento de um pedido em fila por interrupção da *thread* ou por *timeout*. Como a presença de escritores em fila de espera determina o bloqueio de leitores, quando o único escritor em fila de espera cancela o pedido, é necessário garantir o acesso a todos os leitores bloqueados, se o recurso não estiver a ser escrito. O cancelamento dos pedidos presentes na fila de espera de leitores não tem consequências, porque o estado desta fila de espera não é considerado nos predicados das operações bloqueantes (`StartRead` e `StartWrite`).

No que se refere ao cancelamento dos pedidos, a delegação da execução apresenta a seguinte particularidade: é possível ocorrer interrupção ou *timeout* em "simultâneo" com a realização da operação por parte de outra *thread*. Assim, no caso da excepção de interrupção, é necessário retornar normalmente do método para que o chamador considere que a operação foi realizada – o que é verdade. Para que o sinal de interrupção da *thread* não se perca, o código restaura o estado de interrupção – chamando o método `Thread.Interrupt` sobre a *thread* corrente –, de modo a que a interrupção possa ser detectada pelo código dos níveis superiores. O processamento do *timeout* é mais simples, pois basta "considerar" que o limite de tempo de espera expirou depois de a operação ter sido realizada, e retornar normalmente.

A técnica da delegação de execução suporta a implementação directa de sincronizadores cuja semântica associa comportamento a transições do estado partilhado, mas obriga à utilização de filas de espera para manter os pedidos pendentes. Em alguns cenários de utilização da delegação de execução, como é o caso do problema dos leitores/escritores, a implementação das filas de espera pode ser optimizada. Neste exemplo, a operação `StartRead` não tem argumentos e todas as *threads* leitoras bloqueadas tem o mesmo tratamento. Por isso, não é necessário manter um *request item* por cada *thread* leitora bloqueada no monitor; basta apenas saber quantas *threads* leitoras estão bloqueadas e ser capaz de passar a todas elas a indicação de que o respectivo pedido de acesso foi garantido, quando termina uma escrita ou quando um escritor bloqueado desiste por interrupção. Neste tipo de situações, podemos considerar que as operações de `StartRead` pendentes são processadas por lotes. Quando forem reunidas as condições necessárias, são processados todos os pedidos pendentes (um lote). Apenas é necessário garantir que as *threads* envolvidas tomam conhecimento de que podem prosseguir a execução e assegurar que qualquer *thread* leitora que chegue depois ao monitor, e tenha que se bloquear, pertence a outro lote. A fila de espera cujo código se apresenta a seguir responde às necessidades de implementação da delegação de execução para o caso das *threads* leitoras no problema dos leitores/escritores.

```
public sealed class SimpleBatchReqQueue {
   private int current = 0;   // identifies the current batch
   private int count = 0 ;    // number of request items in the current batch

   //Add a request to the queue and return its identifier
   public int Add() {
      count++;
      return current;
   }

   // Remove a request from the queue
   public void Remove(int r) {
      if (count == 0 || r != current)
         // the request belongs to a previous generation or there is no requests in the queue
         throw new InvalidOperationException();
      count--;
   }

   // Clear all request items of the current batch
   public void Clear() { count = 0; }

   // Start a new batch of request items
   public void NewBatch() {
      current++;
      count = 0;
   }
```

```
   // Returns the current number of requests
   public int Count { get { return count; } }

   // Returns true if a request item was processed; that is, its batch identifier is different
   // from the current batch identifier
   public bool IsCompleted(int r) { return r != current; }
}
```

Esta fila de espera pode ser utilizada no algoritmo apresentado anteriormente sem alterar nenhum aspecto relevante do raciocínio subjacente, como se mostra a seguir. As alterações são destacadas a negrito.

```
public sealed class ReadersWriters_2{
   private int  readers = 0;       // number of readers
   private bool writing = false;  // true when writing

   // For waiting readers we use a simple batch request queue.
   private readonly SimpleBatchReqQueue rdq =
             new SimpleBatchReqQueue();

   // We use a generic queue for waiting writers.
   // Each request item in the queue holds a Boolean that says if the requested
   // access was already granted or not.
   private readonly LinkedList<bool> wrq = new LinkedList<bool>();

   // Constructor.
   public ReadersWriters_2(){}

   // Acquire read (shared) access
   public void StartRead() {
      lock(this) {
         // if the resource is not being written and there is no waiting writers,
         // we can start reading
         if (!writing && wrq.Count == 0) {
            readers++;
            return;
         }
         // enqueue a item in the readers queue to request shared access
         int rd = rdq.Add();
         //loop until the shared access was granted or someone interrupts the reader thread
         do {
            try {
               Monitor.Wait(this);
            } catch (ThreadInterruptedException) {
               // if shared access was granted, we must re-assert exception, and return normally;
               // otherwise, we remove the request from the queue and re-throw exception
               if (rdq.IsCompleted(rd)) {
                  Thread.CurrentThread.Interrupt();
                  return;
               }
               rdq.Remove(rd);
               throw;
            }
            // if shared access was granted then return; otherwise, re-wait
            if (rdq.IsCompleted(rd)) {
               return;
            }
         } while (true);
```

```
    }
  }

  // Acquire write (exclusive) access
  public void StartWrite() {
    lock(this) {
      // if the resource is not being written nor read, and there is no waiting writers,
      // we can start reading
      if (readers == 0 && !writing && wrq.Count == 0) {
        writing = true;
        return;
      }
      // enqueue a request item in the writers queue to request exclusive access
      LinkedListNode<bool> wr = wrq.AddLast(false);
      // loop until the exclusive access was granted or someone interrupts the writer thread
      do {
        try {
          Monitor.Wait(this);
        } catch (ThreadInterruptedException) {
          // if exclusive access was granted, then we re-assert exception,
          // and return normally
          if (wr.Value) {
            Thread.CurrentThread.Interrupt();
            return;
          }
          // when a waiting writer gives up, we must grant shared access to all
          // waiting readers if the resource is not being written, and the writers
          // queue becomes empty
          wrq.Remove(wr);
          if (!writing && wrq.Count == 0 && rdq.Count > 0) {
            readers += rdq.Count; // all waiting readers start reading
            rdq.NewBatch();      // start a new batch of start read requests
            Monitor.PulseAll(this);   // wakeup the readers
          }
          // throw ThreadInterruptedException
          throw;
        }
        // if the exclusive access was granted, return; else, re-wait
        if (wr.Value) {
          return;
        }
      } while (true);
    }
  }

  //  Release a read (shared) access
  public void EndRead() {
    lock(this) {
      readers--;      // decrement the current number of readers
      // if this is the last reader, and there is at least a blocked writer, grant exclusive access
      // to the writer that is at front of the writers queue
      if (readers == 0 && wrq.Count > 0) {
        //  signal the writer that exclusive access was granted
        wrq.First.Value = true;
        wrq.RemoveFirst();       // remove the writer from the queue
        writing = true;          // mark the resource as being written
        Monitor.PulseAll(this); // notify writer
      }
    }
```

```
   }

   // Release the write (exclusive) access
   public void EndWrite() {
      lock(this) {
         writing = false;       // release exclusive access
         // if there are any blocked readers, grant shared access to all of them;
         // otherwise, grant exclusive access to one of the waiting writers, if exists
         if (rdq.Count > 0) {
            readers += rdq.Count;       // all waiting readers start reading
            rdq.NewBatch();             // start a new batch of start read requests
            Monitor.PulseAll(this);     // notify readers
         } else if (wrq.Count > 0) {
            // grant exclusive access to the first waiting
            wrq.First.Value = true;
            wrq.RemoveFirst();          // remove the writer from queue
            writing = true;             // mark the resource as being written
            Monitor.PulseAll(this);     // notify the writer
         }
      }
   }
}
```

A optimização decorre do facto da implementação da fila de espera usar apenas dois inteiros e de não ser necessário criar um objecto (para armazenar o *request item*) de cada vez que é necessário bloquear uma *thread* leitora. Como se pode constatar pela análise do código, nada mudou na sua estrutura e as alterações estão apenas relacionadas com a definição e manipulação da fila de espera das *threads* leitoras.

Existem também sincronizadores onde os pedidos pendentes são atendidos todos ao mesmo tempo, mas onde as operações têm argumentos e/ou resultados. Nestas situações, não é necessário criar um objecto diferente para suportar o *request item* de cada *thread* a bloquear, pois o mesmo *request item* pode ser partilhado por todas as *threads* bloqueadas. A classe, cuja definição se apresenta a seguir, implementa uma fila de espera de pedidos optimizada ser utilizada nestas circunstâncias.

```
public sealed class BatchReqQueue<T> {
   // inner class to hold the request item as a reference type
   public sealed class Request {
      public T Value;
      public Request(T v) { Value = v; }
   }

   private Request current;   // current shared request item
   private int count ;        // number of requests in the current batch

   // Queue constructor: the argument passed to the constructor defines the initial state
   // of the request item in the first batch
   public BatchReqQueue(T r) {
      current = new Request(r);
      count = 0;
   }
```

```
   // Add a request to the queue, and return the reference to the request item
   public Request Add() {
      count++;
      return current;
   }
   // Remove all items from queue
   public void Clear() {
      current = null;
      count = 0;
   }

   // Remove a request item from the queue
   public void Remove(Request r) {
      if (count == 0 || r != current)
         // the request item belongs to another batch
         throw new InvalidOperationException();
      count--;
   }

   // Start a new batch of requests
   // the argument passed to this method defines the initial state of the request
   // item in the next batch
   public void NewBatch(T r) {
      current = new Request(r);
      count = 0;
   }

   // Return the number of requests in the current batch
   public int Count { get { return count; } }

   // Return the reference for the request item of the current batch
   public Request Current { get { return current; } }
}
```

Para exemplificar a utilização deste tipo de fila de espera, apresenta-se a seguir a implementação de uma barreira cíclica, com semântica idêntica à da classe `CyclicBarrier` do `package java.util.concurrent` introduzida no Java 5. Este sincronizador permite a um conjunto de *threads* parceiras esperarem umas pelas outras, até que todas elas atinjam um ponto de barreira comum. As barreiras cíclicas são úteis em programas que envolvem um conjunto fixo de *threads* parceiras que, ocasionalmente, necessitem esperar umas pelas outras. A designação barreira cíclica indica que a barreira fica pronta para ser reutilizada após uma ronda de sincronização de todas as *threads*. A barreira suporta, optativamente, a especificação de uma acção que é executada uma vez por cada ponto de barreira, depois de chegar a última *thread* parceira, mas antes das outras *threads* prosseguirem a execução. A acção associada à barreira é útil para actualizar estado partilhado entre as *threads* parceiras, antes de qualquer delas continuar a execução após sincronização no ponto de barreira.

O método `Wait` da barreira aceita a especificação de *timeout* e também é permitida a interrupção das *threads* bloqueadas na barreira. Para integrar esta funcionalidade, considera-se que a barreira é quebrada quando uma das *threads* parceiras desiste da sincronização, por *timeout* ou interrupção. Quando a barreira é quebrada por uma *thread*, as restantes parceiras têm que ser informadas desse facto, para que possam promover as necessárias acções de *cleanup* e/ou sincronização. Por isso, o método `Wait` pode lançar `ThreadInterruptedException`, `TimeoutException`, `BrokenBarrierException` ou a excepção lançada pelo método que executa a acção associada à barreira. Na *thread* responsável pela quebra da barreira, o método `Wait` lança uma excepção específica para informar qual a razão para a quebra da barreira (`ThreadInterruptedException`, `TimeoutException` ou a excepção lançada durante a execução da acção da barreira); nas restantes *threads* parceiras, o método `Wait` lança `BrokenBarrierException`. Após quebrada, a barreira permanece nesse estado até seja restabelecida com a invocação do método `Reset`.

A semântica da barreira cíclica enquadra-a nos sincronizadores cuja implementação é adequada à utilização da técnica de delegação de execução. A semântica especifica que a barreira é levantada exactamente no momento em que chega a última *thread* parceira, ou é quebrada para todas as *threads* no instante em que uma das *threads* parceiras desiste da sincronização, por *timeout* ou interrupção. Além disso, quando a barreira é levantada para um grupo de *threads*, é necessário voltar a fechá-la para preparar nova sincronização. A implementação simplifica-se se for a *thread* que abre ou quebra a barreira a única responsável por actualizar o estado partilhado em proveito de todas as *threads* e, depois, proceder à notificação das *threads* em espera, passando-lhe o código resultado da sincronização: abertura ou quebra. Após acordarem, as *threads* em espera retornam do método `Wait` de acordo com o código resultado que lhes foi passado, pela *thread* que abriu ou quebrou barreira, isto é, retornando normalmente ou lançando `BrokenBarrierException`.

Como todas as *threads* em espera na barreira são acordadas ao mesmo tempo e com o mesmo código resultado, a fila de espera dos pedidos pendentes pode ser implementada com uma instância da classe `BatchReqQueue<T>`. Um único objecto *request item* – partilhado por todas as *threads* bloqueadas – irá armazenar o código resultado da sincronização.

```
// The BrokenBarrierException definition
```

```
class BrokenBarrierException : Exception {}
// The delegate type used to define the barrier's action
public delegate void BarrierAction();

// The Cyclic Barrier
public sealed class CyclicBarrier {
   private readonly int parties;  // number of parties
   private int count;             // number of parties missing
   private bool broken;           // true when barrier was broken
   private readonly BarrierAction action;   // the barrier action

   // Defines the result of the barrier's synchronization
   enum Result {
      Closed,   // barrier is still closed
      Opened,   // barrier was already opened
      Broken    // barrier was broken
   }

   // Queue of pending requests
   private readonly BatchReqQueue<Result> queue =
               new BatchReqQueue<Result>(Result.Closed);

   // constructor
   public CyclicBarrier(int parties, BarrierAction action) {
      this.parties = this.count = parties;
      this.action = action;
      this.broken = false;
   }

   // Prepare the barrier for the next synchronization
   private void nextSynch() {
      queue.NewBatch(Result.Closed);    // start a new batch of requests
      count = parties;
      broken = false;
   }

   // Open the barrier, and prepare it for the next synchronization
   private void openBarrier() {
      if (queue.Count > 0) {
         // wake all waiting parties with Result.Opened
         queue.Current.Value = Result.Opened;
         Monitor.PulseAll(this);
      }
      // prepare the barrier for the next synchronization
      nextSynch();
   }

   // Break the barrier
   private void breakBarrier() {
      if (queue.Count > 0) {
         // wake all waiting parties with Result.Broken
         queue.Current.Value = Result.Broken;
         queue.Clear();    // remove all request from queue
         Monitor.PulseAll(this);
      }
      broken = true;
   }

   // Wait until all parties has arrived
```

```
public int Wait(int timeout) {
   lock(this) {
      // if the barrier was already broken, throw exception
      if (broken)
         throw new BrokenBarrierException();
      // get our  arrival order, and decrement count of parties
      int index = --count;
      if (index == 0) {
         // all parties arrived; execute barrier action, if exists
         bool ranAction = false;
         try {
            if (action != null) {
               action();
            }
            ranAction = true;
            openBarrier();
            return 0;
         } finally {
            // if the barrier action and thrown an exception break the barrier
            if (!ranAction)
               breakBarrier();
         }
      }
      // we must wait, so enqueue a request item
      BatchReqQueue<Result>.Request req = queue.Add();
      // get a time reference to adjust the timeout value
      int lastTime = (timeout == Timeout.Infinite) ?
                        Environment.TickCount : 0;
      // loop until barrier is opened, broken, the specified timeout expires
      // or the thread is interrupted.
      do {
         try {
            Monitor.Wait(this, timeout);
         } catch (ThreadInterruptedException) {
            // if the barrier was already opened, re-assert the exception and return normally
            if (req.Value == Result.Opened) {
               Thread.CurrentThread.Interrupt();
               return index;
            }
            // remove the request item from queue
            queue.Remove(req);
            // break the barrier, and throw interrupt exception
            breakBarrier();
            throw;
         }
         // if the barrier was opened, return normally
         if (req.Value == Result.Opened)
            return index;
         // if the barrier was broken, return with barrier broken exception
         if (req.Value == Result.Broken)
            throw new BrokenBarrierException();
         // adjust timeout, and check if timeout expired
         if (SyncUtils.AdjustTimeout(ref lastTime, ref timeout) == 0){
            // timeout expired
            // remove the request item from queue, break the barrier and throw the
            // timeout exception
            queue.Remove(req);
            breakBarrier();
            throw new TimeoutException();
```

```
            }
         } while (true);
      }
   }

   // Queries if this barrier is in a broken state
   public bool IsBroken {
      get {
         lock(this) {
            return broken;
         }
      }
   }

   // Reset the barrier
   public void Reset() {
      lock(this) {
         // break the current synchronization, and start a new one
         breakBarrier();
         nextSynch();
      }
   }
}
```

Para além da utilização da classe `BatchReqQueue<T>`, este exemplo não acrescenta nada de relevante àquilo que foi discutido nos exemplos anteriores que utilizam a técnica da delegação de execução.

3.5 Conclusão

Nesta secção, foram apresentados padrões de código genéricos para implementar correctamente em CLI e Java soluções para problemas clássicos de sincronização. Se analisarmos esses padrões de código, verificamos que a sua adaptação a qualquer outro problema, com os mesmos requisitos, difere apenas nos seguintes aspectos: estruturas de dados de suporte (filas ou outras), definição do predicado de entrada, definição do predicado a avaliar após notificação e processamento dos cancelamentos de pedidos.

À primeira vista, o código apresentado parece extenso, quando comparado com os exemplos que encontramos na literatura sobre monitores. Existem duas razões para isso: primeiro, não foi feito qualquer esforço para escrever o código de forma a ocupar menos linhas e, segundo, foi sempre incluído o processamento da interrupção das *threads* bloqueadas no monitor e, em alguns casos, a desistência por *timeout*. Na implementação de sincronizadores destinados a integrar em *software* de produção, considera-se indispensável suportar a especificação de *timeout* nas operações bloqueantes e suportar adequadamente a interrupção das *threads* bloqueadas.

Os exemplos apresentados, com a excepção do primeiro, têm problemas de desempenho devido à utilização sistemática de notificação com *broadcast*. Isto é uma consequência dos monitores em CLI e Java não permitirem distinguir as *threads* bloqueadas. Nas situações em que existem muitas *threads* bloqueadas no monitor, a utilização de *broadcast* provoca um número significativo de comutações de contexto desnecessárias. Na maioria dos casos, apenas uma das N *threads* notificadas pode prosseguir a execução. Na plataforma de referência, uma comutação de *thread* medida em programas CLI ou Java consume cerca de 0.7 µs de tempo de processador. Como a "falsa" notificação cada *thread* provoca duas comutações de contexto, o tempo de processador desperdiçado por cada *thread* notificada em "falso" é 1.4 µs. É, por isso, importante eliminar as comutações de contexto sem consequências, o que será abordado na próxima secção.

Resultados experimentais, com programas CLI e Java, mostram que a utilização de filas de espera e de estruturas de dados para suporte ao *scheduling* nos algoritmos de sincronização não têm impacto significativo no desempenho. Foi comparado o desempenho da implementação do semáforo contador do Exemplo 2 com o do Exemplo 3, para N *threads* de alta prioridade a executar `Acquire(k)` e com uma *thread*, de prioridade inferior, a executar `Release(k)`. O impacto da fila de espera no desempenho só é perceptível para valores de N inferiores a 4, sendo inferior a 3%. Destes resultados, conclui-se que a utilização de estruturas de dados auxiliares para suportar critérios de *scheduling* mais elaborados não compromete o desempenho.

Nesta secção tratou-se a concepção de algoritmos de sincronização com base nos monitores CLI e Java, sem optimizar o número de comutações. Na próxima secção mostra-se como a notificação específica de *threads* minimiza o número de comutações, no caso dos algoritmos apresentados nos Exemplos 3 e 4.

4 Notificação Específica de *Threads*

Alguns dos algoritmos apresentados anteriormente ainda que funcionalmente correctos, não estão optimizados porque podem colocam em execução mais *threads* do que aquelas que têm as condições necessárias para prosseguir a execução, após uma alteração particular do estado partilhado do monitor.

Na maioria desses algoritmos, são sempre colocadas em execução todas as *threads* bloqueadas no monitor, para que estas reavaliem os respectivos predicados sobre o estado partilhado. Por cada *thread* acordada que não possa prosseguir a execução – tendo que voltar a bloquear-se –, são realizadas duas comutações de contexto inúteis.

Utilizando múltiplas variáveis condição no mesmo monitor, é possível ornar selectiva a notificação das *threads* bloqueadas num monitor. Apesar dos monitores implícitos na CLI e Java, suportarem apenas uma variável condição anónima, esse não é o único recurso disponível nestas plataformas. A partir do Java 5 (*package* `java.util.concurrent.locks`), está disponível a implementação de monitores explícitos que suportam um número arbitrário de variáveis condição, criadas dinamicamente. Na CLI, é possível implementar monitores com múltiplas condições, usando os monitores implícitos de vários objectos.

É exactamente a selectividade na notificação, proporcionada pelas variáveis condição, que vamos utilizar para optimizar o número de comutações de contexto nos padrões de código apresentados anteriormente.

O padrão do Exemplo 1 – semáforo – faz uma utilização natural dos conceitos de monitor e de variável condição e não apresenta, normalmente, comutações redundantes, pois apenas é notificada uma *thread* de cada vez que se incrementa o contador `permits`, o que garante que a *thread* notificada encontra condições para prosseguir a execução. No entanto, este código pode provocar comutações de contexto redundantes devido à possibilidade de *barging* nos monitores de Lampson e Redell. É possível que uma *thread* invoque o método `Acquire()` e adquira o *lock* do monitor antes de uma outra *thread* que tenha sido notificada pelo método `Release()` depois de afectar `permits` com 1. Nesta situação, a *thread* que invoca `Acquire()` encontra o campo `permits` com 1, decrementa-o e retorna de imediato; quando a *thread* notificada adquire o *lock* do monitor, encontra o campo `permits` com 0 e tem que voltar a

bloquear-se. Estas comutações redundantes podem ser eliminadas, mas considera-se que a pouca probabilidade delas ocorrerem não justifica que se complique o código neste padrão de solução.

O padrão do Exemplo 2 – semáforo – faz a utilização natural da notificação múltipla proposta por Lampson e Redell. Com o padrão de código sugerido não é possível optimizar o número de comutações. Como cada *thread* boqueada no semáforo pretende um número arbitrário de autorizações e não se conhece o número de autorizações pretendida por cada *thread*, não existe a possibilidade de tornar a notificação selectiva.

No Exemplo 3, implementa-se um semáforo com operações `Acquire(n)` e `Release(n)` e fila de espera com disciplina FIFO. Para servir os pedidos de autorizações ao semáforo por ordem de chegada, é necessário implementar explicitamente uma fila de espera onde constam os pedidos de autorizações pendentes, por ordem de chegada. Neste exemplo, cada *thread*, após bloqueio, avalia um predicado diferente, isto é, testa se o "seu" *request item* está à cabeça da fila de espera e só se isso se verificar e que testa se existem autorizações disponíveis suficientes. Nesta implementação é possível tornar a notificação selectiva, bloqueando cada *thread* numa variável condição diferente e, por cada alteração do número de autorizações disponíveis, verificar se são suficientes para satisfazer o pedido da *thread* que se encontra à cabeça de fila de espera e, em caso afirmativo, notificar apenas essa *thread*. Depois, cada uma das *threads* notificadas, será responsável por notificar a próxima *thread* da fila se estiverem reunidas as condições adequadas. Esta cadeia de notificações individuais extingue-se quando a fila ficar vazia ou quando não existirem autorizações suficientes para satisfazer o pedido da *thread* que se encontra à cabeça da fila. O número de variáveis condição utilizadas em cada momento é igual ao número de *threads* bloqueadas no semáforo. Salienta-se que esta implementação do semáforo impede o *barging* sobre o monitor, pois o predicado de entrada do método `Acquire()` apenas tem em consideração o número de autorizações disponíveis quando a fila de espera está vazia.

No Exemplo 4 – problema leitores e escritores – foi necessário usar duas filas de espera (de leitores e de escritores) para implementar a semântica especificada. A fila de espera com os pedidos de leitura permite saber exactamente o número de *threads* bloqueadas para leitura, às quais tem que ser garantido o acesso solicitado no instante em que termina a escrita. Estas *threads* são todas libertadas ao mesmo tempo, pelo que podem

ser bloqueadas na mesma variável condição e notificadas com *broadcast*. A fila de espera com os pedidos de escrita pendentes é usada para determinar se há *threads* escritoras bloqueadas e para garantir que os pedidos de acesso para escrita são atendidos pela ordem de chegada. As *threads* escritoras são notificadas, uma de cada vez, pela ordem com que se encontram na fila de espera. Neste exemplo, a optimização das comutações de contexto, usando notificação específica, leva a que se bloqueiem todas as *threads* leitoras na mesma variável condição e cada uma das *threads* escritoras numa variável condição diferente. As *threads* leitoras são notificadas com *broadcast* sobre a respectiva variável condição, e cada *thread* escritora é notificada com *notify* sobre a respectiva variável condição. Também neste exemplo, o número de variáveis condição necessário, em cada momento, para tornar a notificação selectiva depende do número e da natureza das *threads* bloqueadas.

Salienta-se que a utilização das variáveis condição, aqui proposta, é diferente da sugerida originalmente com o conceito de monitor, onde o número de variáveis condição a utilizar era determinada pela classificação das *threads* em função da sua relação com o sincronizador. Assim, emergiam naturais duas condições no problema dos leitores/escritores, uma para bloquear as *threads* leitoras (`okToRead`) e outra para bloquear as *threads* escritoras (`okToWrite`), ou duas condições no problema do *bounded-buffer*, para bloquear separadamente as *threads* produtoras e *threads* consumidoras. Na proposta original, a definição do número de condições a utilizar num sincronizador particular era ponto de partida para definir o respectivo algoritmo.

A abordagem que se propõe neste texto é diferente. Desenha-se cada sincronizador sem pensar nas variáveis condição, usando apenas a funcionalidade de bloqueio e notificação de *threads*. É mesmo possível testar os algoritmos usando os monitores implícitos, bloqueando todas as *threads* na variável condição anónima e fazendo a notificação com *broadcast* sobre essa variável condição. Depois, analisando as condições em que as *threads* são notificadas, determina-se quantas variáveis condição são necessárias para notificar apenas as *threads* que, garantidamente, têm condições para prosseguir a execução. É esta diferença na utilização das variáveis condição que faz com que designemos este tópico por notificação específica de *threads*.

Salienta-se que pelo facto de se notificar individualmente as *threads* bloqueadas num monitor, depois de se avaliar o respectivo predicado, isso não significa que se deva omitir a avaliação do predicado quando cada *thread* notificada reentra no monitor. A

semântica do monitor proposta por Lampson e Redell não garante que o retorno da operação *wait* seja consequência de uma notificação. Além disso, a *Java Language Specification* afirma que uma implementação compatível pode produzir "*spurious wake-ups*" nos monitores implícitos. Além disso, o retorno da operação *wait* pode se consequência de ter ocorrido *timeout*.

4.1 Notificação Específica de Threads na CLI

A implementação da notificação específica de *threads* – ou seja, o suporte de múltiplas variáveis condição usando os monitores implícitos na CLI e Java – tem sido proposto por vários autores, nomeadamente Cargill em [2]. A estratégia proposta por vários autores, e a que vamos adoptar, é suportar a implementação de monitores com múltiplas condições em monitores implícitos de vários objectos.

Segundo esta estratégia, o monitor implícito de um dos objectos é usado como monitor propriamente dito, sendo os restantes usados apenas para aproveitar a funcionalidade da variável condição, mas mantendo a necessária ligação com o monitor implícito do objecto que funciona como monitor.

O padrão de código apresentado em [2] coloca ao nível superior dos algoritmos dos sincronizadores a problemática da utilização de mais do que um monitor implícito. Se adoptássemos esta estratégia, seria necessário alterar completamente a estrutura dos algoritmos que apresentámos na secção anterior. Neste texto, propõe-se uma solução diferente: esconder a utilização de vários monitores implícitos, na implementação de métodos que implementem a funcionalidade *wait*, *notify* e *broadcast* de monitores com múltiplas variáveis condição. Com esta abordagem, é possível continuar a raciocinar como se existisse apenas um monitor e optimizar o número de comutações de contexto de qualquer dos algoritmos apresentados anteriormente, sem introduzir novos problemas de concepção.

A seguir apresenta-se a implementação da funcionalidade *wait*, *notify* e *broadcast* de monitores com um número arbitrário de variáveis condição, acrescentando métodos estáticos à classe `SyncUtils`, já referida anteriormente.

```
public static class SyncUtils {
  public static int AdjustTimeout(ref int lastTime,
                                   ref int timeout) { ... }

  // Auxiliary method to acquire the object's lock filtering the
  // ThreadInterruptedException. Through its out parameter this method informs
```

```
// if the current thread is interrupted while tries to acquire the lock.
private static void EnterUninterruptibly(object mlock,
                                         out bool interrupted) {
   interrupted = false;
   do {
      try {
         Monitor.Enter(mlock);
         break;
      } catch (ThreadInterruptedException) {
         interrupted = true;
      }
   } while (true);
}

// This method waits on a specific condition of a monitor.
//
// This method is called with mlock locked and the condition's lock unlocked.
// On return, the same conditions are meet: mlock locked, condition's lock unlocked.
public static void Wait(object mlock, object condition, int tmout) {
   // if the mlock and condition are the same object, we just call Monitor.Wait on mlock
   if (mlock == condition) {
      Monitor.Wait(mlock, tmout);
      return;
   }
   // if the mlock and condition are different objects, we need to release the mlock's lock to
   // wait on condition's monitor.
   //
   // first, we acquire lock on condition object before release the lock on mlock, to prevent
   // the loss of notifications.
   // if a ThreadInterruptException is thrown, we return the exception with the mlock locked.
   // we considerer this case as the exception was thrown by the Monitor.Wait(condition).
   Monitor.Enter(condition);
   // release the mlock's lock and wait on condition's monitor condition
   Monitor.Exit(mlock);
   try {
      // wait on the condition monitor
      Monitor.Wait(condition, tmout);
   } finally {
      // release the condition's lock
      Monitor.Exit(condition);
      // re-acquire the mlock's lock uninterruptibly
      bool interrupted;
      EnterUninterruptibly(mlock, out interrupted);
      // if the thread was interrupted while trying to acquire the mlock, we consider that it was
      // interrupted when in the wait state, so, we throw the ThreadInterruptedException.
      if (interrupted)
         throw new ThreadInterruptedException();

   }
}

// This method wait on a specific condition of a monitor.
//
// This method is called with mlock locked and the condition's lock unlocked.
// On return, the same conditions are meet: mlock locked, condition's lock unlocked.
public static void Wait(object mlock, object condition) {
   Wait(mlock, condition, Timeout.Infinite);
}
```

```
   // This method notifies one thread that called Wait using the same mlock and condition objects.
   //
   // This method is called with the mlock's lock held, and returns under the same conditions
   public static void Notify(object mlock, object condition) {
      // if mlock and condition refers to the same object, we just call Monitor.Pulse on mlock.
      if (mlock == condition) {
        Monitor.Pulse(mlock);
        return;
      }
      // If mlock and condition refer to different objects, in order to call Monitor.Pulse on
      // condition we need to acquire condition's lock. We must acquire this lock ignoring the
      // ThreadInterruptedExceptions, because this method in not used for wait purposes,
      // so it must not throw ThreadInterruptedException.
      bool interrupted;
      EnterUninterruptibly(condition, out interrupted);
      // Notify the condition object and leave the corresponding monitor.
      Monitor.Pulse(condition);
      Monitor.Exit(condition);
      // if the current thread was interrupted, we re-assert the interrupt, so the exception
      // will be raised on the next call to a wait method.
      if (interrupted)
        Thread.CurrentThread.Interrupt();
   }

   // This method notifies all threads that called Wait using the same mlock and condition objects.
   //
   // This method is called with the mlock's lock held, and returns under the same conditions
   public static void Broadcast(object mlock, object condition) {
      // if mlock and condition refer to the same object, we just call Monitor.PulseAll on mlock.
      if (mlock == condition) {
        Monitor.PulseAll(mlock);
        return;
      }
      // If mlock and condition refer to different objects, in order to call Monitor.PulseAll
      // on condition, we need to hold the condition's lock.
      // We must acquire the condition's lock ignoring the ThreadInterruptedExceptions, because
      // this method is not used for wait purposes, so it must not throw
      // ThreadInterruptedException.
      bool interrupted;
      EnterUninterruptibly(condition, out interrupted);
      // notify all threads waiting on the condition and leave the condition object's monitor
      Monitor.PulseAll(condition);
      Monitor.Exit(condition);
      // In case of interrupt, we re-assert the interrupt, so the exception can be raised on
      // the next wait.
      if (interrupted)
        Thread.CurrentThread.Interrupt();
   }
}
```

Os métodos `Wait`, `Notify` e `Broadcast` têm dois parâmetros comuns: a referência para o objecto usado como monitor e a referência para o objecto usado como variável condição. Assume-se que estes métodos são invocados com o *lock* do monitor implícito do objecto usado como monitor na posse da *thread* corrente e que o retorno é feito nas mesmas condições.

Esta implementação suporta a funcionalidade *wait*, *notify* e *broadcast*, suportando-se em dois monitores implícitos. No texto que segue, vamos usar os termos objecto monitor para referir o monitor implícito do objecto usado como monitor e objecto variável condição para referir o monitor implícito do objecto usado como variável condição.

O método `Wait` bloqueia a *thread* no monitor implícito do objecto usado como variável condição. Para invocar `Monitor.Wait` sobre este objecto é necessário adquirir o respectivo *lock*. Para não bloquear o acesso ao monitor enquanto a *thread* aguarda notificação, é necessário libertar o *lock* do objecto monitor. Para implementar correctamente a espera nas variáveis condição de um monitor, é necessário realizar atomicamente a libertação do *lock* do monitor e a inserção da *thread* na fila de espera da variável condição. Por outras palavras, quando a *thread* que invoca *wait* liberta o *lock* do monitor já tem que estar inserida na fila de espera da variável condição, para que não possam ser perdidas notificações. Nesta implementação, como estão a ser usados dois monitores implícitos, é necessário impedir que o monitor do objecto usado como variável condição seja notificado (com `Monitor.Pulse` ou `Monitor.PulseAll`) antes de a *thread* ter invocado `Monitor.Wait`. Isso está a ser garantido, porque o *lock* do objecto condição é adquirido antes de libertar o *lock* do objecto monitor. Logo, não é possível a outra *thread* adquirir os dois *locks* para fazer uma notificação, antes da *thread* que se bloqueia libertar o *lock* do objecto condição, o que acontece no método `Monitor.Wait`, depois de a *thread* estar inserida na fila de espera da variável condição.

A implementação dos métodos `Wait`, `Notify` e `Broadcast` tem que respeitar a especificação dos métodos `Monitor.Wait`, `Monitor.Pulse` e `Monitor.PulseAll` no que se refere à excepção `ThreadInterruptedException`, para que possamos usar estes métodos nos algoritmos apresentados anteriormente. Dos três métodos, apenas `Wait` pode retornar com o lançamento de `ThreadInterruptedException`.

Na CLI, a excepção `ThreadInterruptedException` pode ser lançada pelos métodos `Monitor.Enter` e `Monitor.Wait`.

Os métodos `Notify` e `Broadcast` não podem lançar a excepção de interrupção. Assim, o código destes métodos filtra a `ThreadInterruptedException`, quando adquire o *lock* do objecto condição. Ainda, quando se detecta a interrupção da *thread*

corrente, captura-se a respectiva excepção até que seja adquirida a posse do *lock*. Para que a indicação de interrupção da *thread* não seja perdida, antes de retorno, é interrompida a *thread* corrente; assim, a interrupção fica pendente, sendo a respectiva excepção lançada na próxima chamada que a *thread* fizer a um método com semântica *wait*.

Na implementação do método `Wait` existem três circunstâncias onde pode ser lançada `ThreadInterruptedException`: na aquisição do *lock* do objecto condição, no método `Monitor.Wait` sobre o objecto condição e na aquisição do *lock* do objecto monitor. Se for detectada a interrupção da *thread* corrente em qualquer destas três circunstâncias, o método `Wait` lança `ThreadIterruptedException`, para indicar que foi interrompida a espera no objecto condição, mesmo que a excepção não seja detectada na chamada ao método `Monitor.Wait`. O código garante que no retorno deste método a *thread* corrente detém a posse do *lock* do objecto monitor, mesmo no caso em que é lançada `ThreadInterruptedException`.

Para exemplificar a utilização de notificação específica na CLI, apresentamos a seguir a implementação do semáforo com fila de espera FIFO, adaptando o código apresentado no Exemplo 3.

Neste algoritmo, usamos como objecto monitor o próprio objecto semáforo (`this`) e como objectos condição os *request items* colocados na fila de espera por cada *thread* que se bloqueia no semáforo. Podemos constatar que basta alterar duas linhas de código (assinaladas a negrito) para que a implementação do semáforo não provoque comutações de contexto inúteis.

```
public sealed class Semaphore_FIFO_SN {
  private int permits = 0;
  // Queue of waiting threads.
  // For each waiting thread we hold an integer with the number of
  // permits requested. This allow us optimizing the notifications as we
  // can see in the method notifyWaiter()
  private readonly LinkedList<int> queue = new LinkedList<int>();

  // The constructor.
  public Semaphore_FIFO_SN(int initial) {
    if (initial > 0)
      permits = initial;
  }

  // Notify the waiting threads, if the number of permits is greater or equal than the first
  // queued request
  private void notifyWaiter () {
    if (queue.Count > 0 && permits >= queue.First.Value)
```

```
            SyncUtils.Notify(this, queue.First);  // specific notification
        }

        // Acquire n permits, waiting at most timeout milliseconds, or the thread is interrupted
        public bool Acquire(int n, int timeout){
            lock(this) {
                if (queue.Count == 0 && permits >= n) { // entry predicate
                    permits -= n;
                    return true;
                }
                // enqueue the request
                LinkedListNode<int> rn = queue.AddLast(n);
                // if a timeout was specified get a time reference
                int lastTime = (timeout != Timeout.Infinite) ?
                                    Environment.TickCount : 0;
                do {
                    try {
                        SyncUtils.Wait(this, rn, timeout);  // specific wait
                    } catch (ThreadInterruptedException) {
                        // interrupted exception: give up processing
                        queue.Remove(rn);   // remove request node from the queue
                        notifyWaiter();     // notify, because queue was altered
                        throw;              // re-throw interrupted exception
                    }
                    // predicate after wakeup
                    if (rn == queue.First && permits >= n) {
                        queue.Remove(rn);   // dequeue request
                        permits-= n;        // get permits
                        notifyWaiter();     // notify, because permits and queue changed
                        return true;
                    }
                    if (SyncUtils.AdjustTimeout(ref lastTime, ref timeout) == 0){
                        // timeout: give up processing
                        queue.Remove(rn);   // remove request item from queue
                        notifyWaiter();         // notify, because queue was altered
                        return false;
                    }
                } while (true);
            }
        }

        // Acquire n permits, without limit on wait time
        public void Acquire(int n){
            Acquire(n, Timeout.Infinite);
        }

        // Release n permits
        public void Release(int n) {
            lock(this) {
                permits += n;       // update permits
                notifyWaiter();     // notify, because the number of permits changed
            }
        }
    }
```

Qualquer dos algoritmos que usa delegação de execução pode ser alterado, para usar notificação específica de *threads*, com a mesma facilidade. Apresentamos a seguir a

implementação dos leitores e escritores para exemplificar a utilização da notificação específica em Java.

Salienta-se que a utilização da notificação específica na CLI tem desvantagens. A utilização de dois monitores implícitos, no bloqueio e notificação de *threads*, introduz *overhead* e impede a aquisição recursiva do *lock* do objecto usado como monitor. O *overhead* introduzido deve-se a dois factores: custo da aquisição e libertação de um *lock* adicional, sempre que se bloqueia ou notifica uma *thread*, e; custo da associação por parte da máquina virtual do monitor físico aos objectos usados como variáveis condição, na primeira vez que essa funcionalidade é invocada assim como o custo da reciclagem do monitor físico, durante as operações de *garbage collection*. A seguir, discute-se a importância destes dois aspectos, a forma de minimizar o segundo, assim como a razão porque se perde da capacidade de aquisição recursiva do *lock* do objecto monitor.

Se analisarmos o código dos métodos que suportam as múltiplas variáveis e o compararmos com código que use apenas um monitor implícito, verifica-se que a existência de dois *locks* pode aumentar as situações de disputa pelos *locks*, em determinados cenários de interacção entre as *threads*, provocando comutações de contexto adicionais. O número destas comutações depende do cenário particular de interacção entre as *threads* e da implementação dos monitores implícitos na máquina virtual. Se este *overhead* anula, ou não, as vantagens da notificação específica de *threads*, depende do algoritmo do sincronizador e das condições de utilização. Para tirar conclusões definitivas é necessário medir o desempenho das duas implementações. Em cenários onde a disputa pela aquisição seja baixa, este *overhead* é baixo, pois a aquisição sem contenção do *lock* de um monitor implícito pode ser feita com uma instrução atómica.

Pode minimizar-se o custo da associação do monitor implícito aos objectos pela máquina virtual, usando como objecto condição a instância do tipo `Thread` associada à *thread* que se bloqueia. Assim, o número de objectos a que a máquina virtual tem que associar o monitor físico não depende do número de vezes que *threads* se bloqueiam no sincronizador (potencialmente elevado), mas sim do número de que usam o sincronizador (potencialmente baixo). Para implementar esta técnica, basta associar ao *request item* colocado na fila de espera a referência para a *thread* corrente e usar o objecto `Thread` como variável condição, como se mostra no seguinte código. (As

linhas de código alteradas relativamente à implementação anterior estão assinaladas a negrito.)

```
public sealed class Semaphore_FIFO_SN_2 {
   private int permits = 0;
   // Queue of waiting threads.
   // For each waiting thread we hold an integer with the number of
   // permits requested. This allow us optimizing the notifications as we
   // can see in the method notifyWaiter()
   private readonly LinkedList<Req> queue = new LinkedList<Req>();

   // Type to hold the request item
   private struct Req {
      public int n;            // the request number of permits
      public Thread thread;    // the requesting thread
      public Req(int n) {
         this.n = n;
         this.thread = Thread.CurrentThread;
      }
   }

   // Constructor
   public Semaphore_FIFO_SN_2(int initial) {
      if (initial > 0)
         permits = initial;
   }

   // Notify the waiting threads, if the number of permits is greater or equal than the
   // first queued request
   private void notifyWaiter() {
      if (queue.Count > 0 && permits >= queue.First.Value.n)
         SyncUtils.Notify(this, queue.First.Value.thread);
   }

   //  acquire n permits, waiting at most timeout milliseconds, or the thread is interrupted
   public bool Acquire(int n, int timeout){
      lock(this) {
         if (queue.Count == 0 && permits >= n) { // entry predicate
            permits -= n;
            return true;
         }
         // enqueue the request
         LinkedListNode<Req> rn = queue.AddLast(new Req(n));
         // if a timeout was specified take a time reference
         int lastTime = (timeout == Timeout.Infinite) ?
                        Environment.TickCount : 0;
         do {
            try {
               SyncUtils.Wait(this, rn.Value.thread, timeout);
            } catch (ThreadInterruptedException) {
               // interrupted exception: give up processing
               queue.Remove(rn);   // remove request node from the queue
               notifyWaiter();     // notify, because queue was altered
               throw;              // re-throw interrupted exception
            }
            // predicate after wakeup
            if (rn == queue.First && permits >= n) {
               queue.Remove(rn);   // dequeue request
```

```
            permits-= n;        // get permits
            notifyWaiter();     // notify, because permits and queue changed
            return true;
          }
          if (SyncUtils.AdjustTimeout(ref lastTime, ref timeout) == 0){
            // timeout: give up processing
            queue.Remove(rn);  // remove request item from queue
            notifyWaiter();       // notify, because queue was altered
            return false;
          }
        } while (true);
      }
    }

    // Acquire n permits, without limit on wait time
    public void Acquire(int n){ ... }

    // Release n permits
    public void Release(int n){ ... }
}
```

Esta implementação de "monitores com múltiplas condições" inviabiliza a aquisição recursiva do *lock* do objecto usado como monitor, pelos motivos que se explicam a seguir. Quando uma *thread* adquire um *lock* pela primeira vez com `Monitor.Enter`, o contador de aquisições do *lock* é iniciado com 1; este contador é incrementado por cada aquisição posterior do *lock* pela mesma *thread*. Quando uma *thread* invoca `Monitor.Exit`, o contador de aquisições do *lock* é decrementado e, se atingir zero, o *lock* é libertado. Assim, para que uma *thread* liberte um *lock* na sua posse, tem que invocar `Monitor.Exit` o mesmo número de vezes que tinha invocado `Monitor.Enter`. Por outro lado, quando uma *thread* invoca `Monitor.Wait`, para se bloquear na variável condição do monitor, o respectivo *lock* é completamente libertado, guardando a *thread* o número corrente de aquisições do *lock*. Quando, após notificação, a *thread* volta a adquirir o *lock*, restaura o contador das aquisições recursivas com o valor que este tinha antes da libertação do *lock*. Por outras palavras, o método `Monitor.Wait` liberta sempre o *lock* do monitor na posse da *thread* invocante e retorna com o mesmo estado de *locking* que a *thread* tinha antes de chamar o método.

O código do método `SyncUtils.Wait` liberta o *lock* do objecto monitor com `Monitor.Exit` e volta-se, após notificação, a adquiri-lo com `Monitor.Enter`. Se o método `SyncUtils.Wait` for chamado por uma *thread* que adquiriu o *lock* do objecto monitor mais do que uma vez, a chamada a `Monitor.Exit` não é suficiente para libertar o *lock*. Como, a seguir, a *thread* vai bloquear-se no objecto condição com a posse do *lock* do monitor, cria-se uma situação de *deadlock*. Para que a *thread* com a

posse do *lock* o libertasse, seria necessário que outra *thread* entrasse no monitor para a notificar, o que nunca pode acontecer.

A solução aqui apresentada para implementar notificação específica de *threads* na CLI, apesar de introduzir algum *overhead*, tem, tipicamente, vantagens quando é utilizada em sincronizadores onde a notificação acorda mais *threads* do que aquelas que podem, efectivamente, prosseguir a execução.

4.2 Notificação Específica de Threads em Java

No âmbito do *Java Community Process*, JSR 166, 2002, foram propostos um conjunto de mecanismos para suporte à programação concorrente. Estes mecanismos foram integrados no *package* `java.utils.concurrent` no Java 5. Um dos mecanismos propostos foi um *lock* genérico que suporta a funcionalidade completa dos monitores de Lampson e Redell. Este monitor está definido com base nas interfaces `Lock` e `Condition`, cuja definição se mostra a seguir.

```
public interface Lock {
   // Uninterruptible wait for the lock to be acquired
   public void lock();
   // As above but interruptible
   public void lockInterruptibly() throws InterruptedException;
   // Create a new condition variable for use with the Lock
   public Condition newCondition();
   // Returns true is lock is available immediately
   public boolean tryLock();
   // Returns true is lock is available within a timeout
   public boolean tryLock(long time, TimeUnit unit)
                    throws InterruptedException;
   // Releases the lock
   public void unlock();
}

public interface Condition {
   // Atomically releases the associated lock and causes the current thread to wait until:
   // 1. another thread invokes the signal method and the current thread happens to be chosen
   //    as the thread to be awakened; or
   // 2. another thread invokes the signalAll method;
   // 3. another thread interrupt the thread; or
   // 4. a spurious wake-up occurs.
   // When the method returns it is guaranteed to hold the associated lock.
   public void await() throws InterruptedException;

   // As for await() but with a timeout
   public boolean await(long time, TimeUnit unit)
                           throws InterruptedException;
   public long awaitNanos(long nanosTimeout)
                           throws InterruptedException;
   public boolean awaitUntil(java.util.Date deadline)
            throws InterruptedException;
```

```
   // As for await, but not interruptible
   public void awaitUninterruptible();

   // Wake up one waiting thread
   public void signal();

   // Wake up all waiting threads
   public void signalAll();
}
```

O novo tipo de monitor e as variáveis condição associadas podem ser criados por intermédio da classe `ReentrantLock`, cuja definição é a seguinte:

```
public class ReentrantLock implements Lock, java.io.Serializable {
   public ReentrantLock();
   ...
   public void lock();
   public void lockInterruptibly() throws InterruptedException;
   public ConditionObject newCondition();

   // Create a new condition variable and associated it with this lock object
   public boolean tryLock();
   public boolean tryLock(long time, TimeUnit unit)
                                 throws InterruptedException;
   public void unlock();
}
```

A classe `ConditionObject` está definida como classe interna de `ReentrantLock` e implementa a interface `Condition`.

Com a classe `ReentrantLock` podemos fazer notificação específica de *threads* em qualquer dos exemplos apresentados anteriormente. A seguir, apresentamos a implementação do semáforo com fila de espera FIFO.

```
public final class Semaphore_FIFO_Java {
   private int permits = 0;   // permits available
   private final Lock lock;   // the explicit monitor

   // Type to hold the request items
   private static class Req {
      public int n;           // the number of permits requested
      public Condition wcond; // condition where the thread will wait

      //Request item constructor
      public Req(int n, Condition wcond) {
         this.n = n;
         this.wcond = wcond;
      }
   }

   // Queue of waiting threads
   private final LinkedList<Req> queue = new LinkedList<Req>();

   // The constructor.
   public Semaphore_FIFO_Java(int initial) {
      lock = new ReentrantLock();
      if (initial > 0)
```

```
    permits = initial;
}

// Notify the waiting threads, if the number of permits is greater or equal than the request
// of the waiting thread that is at front of queue
private void notifyWaiter() {
   // notify a waiter if there is one, and if we have enough available permits
   if (queue.size() > 0 && permits >= queue.element().n)
      queue.element().wcond.signal();
}

// Acquire n permits
private boolean doAcquire(int n, boolean timed, long timeout)
     throws InterruptedException {
   lock.lock();   // acquire the monitor's lock
   try {
      // if queue is empty are there are enough permits available, acquire permits, and return
      if (queue.size() == 0 && permits >= n) {
         permits -= n;
         return true;
      }
      // create a request item and enqueue it
      // we create a new condition for each request where the thread will wait
      Req r = new Req(n, lock.newCondition());
      queue.addLast(r);
      do {
         try {
            if (!timed)
               r.wcond.await();
            else if (timeout > 0L)
               timeout = r.wcond.awaitNanos(timeout);
         } catch (InterruptedException ie) {
            // interrupted exception: give up processing
            queue.remove(r);    // remove request item from the queue
            notifyWaiter();     // notify, because queue was modified
            throw ie;           // re-throw interrupted exception
         }
         // if the request item is at the front queue and there are enough permits available,
         // get requested permits, and return
         if (r == queue.element() && permits >= n) {
            queue.remove(r); // dequeue request
            permits-= n;     // get permits
            notifyWaiter();     // notify, because permits and queue were modified
            return true;
         }
         if (timed && timeout <= 0L) {
            // timeout: give up processing
            queue.remove(r);    // remove request item from queue
            notifyWaiter();     // notify, because queue was modified
            return false;
         }
      } while (true);
   } finally {
      lock.unlock();   // release the monitor's lock
   }
}

// Acquire n permits, without timeout
public void acquire(int n) throws InterruptedException {
```

```
    doAcquire(n, false, 0L);
  }

  // Acquire n permits, with timeout
  public boolean acquire(int n, long timeout, TimeUnit unit)
              throws InterruptedException {
    return doAcquire(n, true, unit.toNanos(timeout));
  }

  // Release n permits
  public void post(int n){
    lock.lock();        // acquire the monitor's lock
    try {
      permits += n;     // update available permits
      notifyWaiter();   // notify, because the available permits was modified
    } finally {
      lock.unlock();    // release the monitor's lock
    }
  }
}
```

A primeira nota sobre este código é a utilização explícita da instância do tipo `ReentrantLock` como monitor. A linguagem Java não tem suporte embutido para os monitores explícitos – como os métodos e os blocos `synchronized`, para os monitores implícitos –, pelo que é necessário explicitar a aquisição e a libertação do *lock* do monitor. A secção crítica protegida pelo *lock* do monitor deve ser incluída num bloco `try`, ao qual se associa um bloco `finally` para libertar o *lock* do monitor, mesmo que o código da secção crítica lance excepção.

Para usar notificação específica de *threads* neste exemplo, usa-se uma variável condição diferente por cada *request item* (instância do tipo `Req`) em fila de espera, e bloqueia-se a respectiva *thread* nessa variável condição.

Como esta implementação não usa delegação de execução, cada *thread*, após notificação, avalia um predicado para determinar se está à cabeça da fila e existem permissões suficientes disponíveis. À primeira vista, isto parece redundante, porque, no método `notifyWaiter`, condiciona-se a notificação pela avaliação do mesmo predicado. No entanto, como podem ocorrer notificações espúrias ou ocorrer *timeout*, o predicado tem que ser sempre avaliado para determinar o que a *thread* deve fazer: adquirir as autorizações pretendidas, retornar com indicação de *timeout* ou voltar a esperar na variável condição.

Para ilustrar mais utilização da notificação específica de *threads* em Java, apresenta-se a seguir a implementação do problema dos leitores e escritores.

Este algoritmo usa a versão Java da classe `SimpleBatchReqQueue`, cujo código se apresenta a seguir.

```
public final class SimpleBatchReqQueue {
   private int current = Integer.MIN_VALUE; // the number of current batch
   private int count ;      // the number of the items in the current batch

   // add a request to the queue
   public int add() {
      count++;
      return current;
   }

   // remove a request from the queue
   public void remove(int r) {
      if (count == 0 || r != current)
         // the request belongs to another batch
         throw new IllegalStateException();
      count--;
   }

   // remove all pending requests from the queue, and start a new batch
   public void newBatch() {
      current++;
      count = 0;
   }

   // remove all pending requests from the queue
   public void clear() { count = 0; }

   // return the number of requests in the queue
   public int size() { return count; }

   // returns true if a request item was processed; that is, its batch number if different
   // the number of the current batch of requests
   public boolean isCompleted(int r) { return r != current; }
}
```

Na implementação do problema dos leitores e escritores seguindo a técnica da delegação de execução com notificação específica de *threads*, usa-se uma variável condição para bloquear todas as *threads* leitoras e bloqueia-se cada uma das *threads* escritoras numa variável condição própria. Como todas as *threads* leitoras são notificadas ao mesmo tempo nas condições em que podem prosseguir a execução, não existe necessidade de usar uma variável condição para cada *thread*. O mesmo não acontece com as threads escritoras que são notificadas uma de cada vez.

```
public final class ReadersWriters_Java_NS {
   private int  readers = 0;         // current readers
   private boolean writing = false;  // true when writing
   // readers queue
   private final SimpleBatchReqQueue rdq = new SimpleBatchReqQueue();
   // writers queue
   private final LinkedList<WrReq> wrq = new LinkedList<WrReq>();
   // explicit monitor
```

```
private final Lock lock;
// explicition condition to block all readers
private final Condition okToRead;

// type to hold write request items
final static class WrReq {
   public boolean granted;         // starts false
   public final Condition wcond;   // condition where owner thread will wait

   public WrReq(Condition wcond) {
      this.wcond = wcond;
   }
}

// The constructor
public ReadersWriters_Java_NS() {
   lock = new ReentrantLock();
   okToRead = lock.newCondition();
}

// Acquire a read (shared) access
public void startRead() throws InterruptedException {
   lock.lock();        // acquire the monitor's lock
   try {
      if (wrq.size() == 0 && !writing) {
         readers++;
         return;
      }
      // enqueue a read request item in the current batch
      int rd = rdq.add();
      do {
         try {
            okToRead.await();
         } catch (InterruptedException ie) {
            // if operation was granted, re-asset exception and return normally
            if (rdq.isCompleted(rd)) {
               Thread.currentThread().interrupt();
               return;
            }
            // else, remove the request item from queue and rethrow exception
            // one reader gives up its request, there is no consequences!
            rdq.remove(rd);
            throw ie;
         }
         if (rdq.isCompleted(rd))
            return;   // read access granted
      } while (true);
   } finally {
      lock.unlock();   // release the monitor's lock
   }
}

// acquire write (exclusive) access
public void startWrite() throws InterruptedException {
   lock.lock();   // acquire the monitor's lock
   try {
      if (wrq.size() == 0 && readers == 0 && !writing) {
         writing = true;
         return;
```

```
      }
      // create and enqueue a write request item for the current thread
      // create a condition variable where the thread will wait
      WrReq wr = new WrReq(lock.newCondition());
      wrq.add(wr);
      do {
        try {
          wr.wcond.await();
        } catch (InterruptedException ie) {
          // if operation was granted, re-assert exception and return normally
          if (wr.granted) {
            Thread.currentThread().interrupt();
            return;
          }
          // when a waiting writer gives up, we need to wakeup all readers if this is the unique
          // waiting writers and if the resource is not being accessed by another writer.
          wrq.remove(wr);      // remove from queue
          if (!writing && wrq.size() == 0  && rdq.size() > 0) {
            readers += rdq.size();  // accomplish read operation
            rdq.newBatch();
            okToRead.signalAll();    // notify all waiting readers
          }
          throw ie;
        }
        // if request was granted, return
        if (wr.granted)
          return;
      } while (true);
    } finally {
      lock.unlock();    // release the monitor's lock
    }
  }

  //  release the read (shared) access of current thread
  public void endRead() {
    lock.lock();        // acquire the monitor's lock
    try {
      readers--;
      if (readers == 0 && wrq.size() > 0) {
        // wakeup next writer
        WrReq wr = wrq.removeFirst();
        wr.granted = true;
        writing = true;     // accomplish write
        wr.wcond.signal(); // notify the writer
      }
    } finally {
      lock.unlock();    // release the monitor's lock
    }
  }

  //  release the write (exclusive)  access of current thread
  public void endWrite() {
    lock.lock();        // acquire the monitor's lock
    try {
      writing = false;
      if (rdq.size() > 0) {
        // wakeup all blocked readers
        readers += rdq.size();      // accomplish read operation
        rdq.newBatch();             // start a new batch of read requests
```

```
            okToRead.signalAll();       // notify all waiting readers
          } else if (wrq.size() > 0) {
            // wakeup next writer
            WrReq wr = wrq.removeFirst();
            wr.granted = true;
            writing = true;         // accomplish write
            wr.wcond.signal();      // notify the writer
          }
        } finally {
          lock.unlock();    // release the monitor's lock
        }
    }
}
```

Esta implementação segue o padrão que foi explicado no Exemplo 4, escrito em C#. As diferenças são devidas à utilização da linguagem Java, à utilização dos monitores explícitos e à notificação específica das *threads* bloqueadas.

A criação do objecto Java que implementa a variável condição, de cada vez que uma *thread* que se bloqueia no monitor, também introduz *overhead*. Na implementação dos monitores explícitos no JDK 1.6, é mesmo criado segundo objecto Java para inserir a *thread* na fila de espera da variável condição. Nesta implementação, o custo da notificação específica, de modo a distinguir entre si todas as *threads* bloqueadas no monitor, esta relacionado com a criação de dois objectos Java, de cada vez que bloqueia uma *thread*, para além do *request item* que é necessário para associar a *thread* bloqueada à respectiva variável condição. A pressão sobre o *garbage collector* que resulta da criação de objectos adicionais, depende do ritmo de criação dos objectos (i.e., o ritmo com que as *threads* se bloqueiam no sincronizador) e do tempo de vida desses objectos (i.e., tempo que as *threads* permanecem bloqueadas).

Comparando a implementação da notificação específica de *threads* em *Java* e em CLI, conclui-se que a implementação em Java é globalmente mais eficiente.

Uma vantagem da implementação *Java* é que apenas exige a aquisição de um *lock* para bloquear ou notificar *threads*, enquanto na CLI têm que ser adquiridos dois *locks*. Além da CLI obrigar sempre à aquisição da posse de um *lock* adicional – o que tem custo mesmo quando o *lock* está livre –, quando a aquisição é feita com contenção, podem ocorrer comutações de contexto adicionais. Outra vantagem da implementação da notificação específica em Java é que não se perde a capacidade de adquirir recursivamente a posse do *lock* do monitor, como acontece na solução apresentada para a CLI.

A única vantagem da implementação na CLI é que não obriga à criação de um objecto específico para variável condição, pois utiliza-se o objecto *request item* para esse efeito.

4.3 Conclusão

É possível reescrever qualquer algoritmo de sincronização que tenha sido desenhado com base nos monitores CLI e Java para usar notificação específica de *threads*, de modo a eliminar as comutações de contexto inúteis. No pior caso, a reescrita envolve a implementação de uma fila de espera associada às *threads* bloqueadas, quando esta não é necessária por outras razões.

Nos cenários em que se notificam mais *threads* do que aquelas que podem, efectivamente, prosseguir a execução, os ganhos obtidos pela eliminação das comutações de contexto inúteis compensam o aumento do processamento inerente à utilização da fila de espera e, no caso da CLI, os custos adicionais decorrentes da utilização de dois monitores.

Referências

[1] *Buhr, P. A, P. A., H. Cofin, M. H., Monitor classification, ACM Computing Surveys, 1995.*

[2] *Cargill, T., Specific Notification for Java Thread Synchronization, personal communication, http://www.profcon.com/profcon/cargill/jgf/SpecificNotification.html.*

[3] *Dijkstra, E. W., Guarded commands no determinacy and formal derivation of program, Communications of ACM, 1975.*

[4] *ECMA – 335, Common Language Infrastructure (CLI) Specification, 3rd edition, 2005.*

[5] *ECMA – 3334, C# Language Specification, 2nd edition, 2002.*

[6] *Gosling, J., Joy, B., Steele, G., The Java Language Specification, Addison-Wesley, 1996.*

[7] *Hansen, P., Structured multiprogramming, Communications of the ACM, 1972.*

[8] *Hansen, P., Concurrent programming concepts, ACM Computer Survey, volume 5, December, 1973.*

[9] *Hoare, C. A. R., Toward a theory of parallel programming, Operating System Techniques, Academic Press, 1972.*

[10] *Hoare, C. A. R., Monitors: An Operating System Structuring Concept, Communications of ACM, 1974.*

[11] *Hoare, C. A. R., Communicating sequential processes, Communications of ACM, 1978.*

[12] *Lampson, B. W., Redell, D. D., Experience with Processes and Monitors in Mesa, Communications of the ACM, 1980.*

[13] *JSR 133: Java Memory Model and Thread Specification, 2004*

Plataformas de Referência

A plataforma base é uma arquitectura baseada processador *Pentium* M a 1.4 GHz, com 1 GB de memória e com o sistema operativo Windows XP SP2.

A plataforma de referência em Java é a plataforma base com o JKD 1.5.0.

A plataforma de referência em CLI é a plataforma base executando o .NET *Framework* v1.1.4322.