Computação Concorrente (MAB-117)
Monitores

Prof. Silvana Rossetto

[1]Departamento de Ciência da Computação (DCC)
Instituto de Matemática (IM)
Universidade Federal do Rio de Janeiro (UFRJ)
Maio de 2012

1. Conceitos básicos sobre monitores

O uso das primitivas de semáforos para tratar as questões de sincronização em ambientes de memória compartilhada tem duas dificuldades associadas [1]:

1. **erros de programação**: as operações sobre semáforos são chamadas a subrotinas, uma inversão na ordem das chamadas pode facilmente levar o programa a situações de bloqueio indefinido;
2. **dificuldade de manutenção do código**: o uso dos semáforos aparece de forma explícita ao longo do programa, tornando difícil a tarefa de manutenção do código.

O conceito de **monitores** foi sugerido para lidar com essas dificuldades. Neste texto descrevemos o conceito de monitor, as diferentes formas de implementação e outras questões relacionadas.

1.1. Definição de monitor

Um **monitor** é uma construção de Linguagem de Programação que provê **funcionalidade equivalente a semáforos** e é "mais fácil" de controlar. O conceito de monitores foi proposto por Hoare, em 1974, e pode ser definido como um **módulo de software que consiste de um ou mais procedimentos, uma sequência de inicialização, variáveis de condição e estado interno** [2]. Exemplos de linguagens que implementam o conceito de monitores incluem: Modula-2, Modula-3, Concurrent Pascal e Java.

Uma das características que distingue monitores de outras estruturas de programação é que **apenas uma operação interna pode estar ativa a cada instante de tempo**. Se uma thread chama uma operação do monitor e ele encontra-se ocupado, a thread é automaticamente bloqueada até o monitor ficar livre [1]. Essa característica permite implementar a **sincronização por exclusão mútua**.

Outra característica particular de monitores é que **qualquer operação pode suspender a si mesma colocando-se em espera por uma variável de condição** [1]. Essa característica permite implementar a **sincronização por condição**.

Uma operação pode também **sinalizar** uma variável de condição fazendo com que uma thread bloqueada nessa variável seja "acordada" (desbloqueada). As variáveis de condição **não** possuem memória (diferente de semáforos): **se um sinal é emitido e não existe nenhuma thread bloqueada nessa variável o sinal é perdido** [1].

1.2. Semântica original de monitores

Na definição original de monitores, proposta por Hoare, as características principais de monitores são [2]:

1. As variáveis locais (definidas dentro do monitor) são acessíveis apenas pelos procedimentos oferecidos pelo monitor;
2. Uma thread entra no monitor invocando um dos seus procedimentos;
3. Apenas uma thread pode estar executando no monitor a cada instante, qualquer outra thread que invocar o monitor será bloqueada até que o monitor fique livre.

As variáveis de condição são tipos de dados especiais dentro do monitor e acessíveis apenas dentro dele. Essas variáveis são operadas por duas funções apenas [2]:

- **cwait(c)**: suspende a execução da thread na variável de condição **c**, o monitor fica disponível para uso por outras threads;
- **csignal(c)**: retoma a execução de alguma thread bloqueada depois de um *cwait* sobre a mesma variável de condição, se há várias threads bloqueadas nessa condição, escolhe uma delas, caso contrário o sinal é "perdido".

Hoare define várias filas de threads para entrada no monitor [1]:

- uma fila associada a cada variável de condição;
- uma fila de entrada no monitor (para as threads que aguardam entrada quando o monitor está ocupado);
- uma fila de urgência (para as threads que executam uma sinalização de dentro do monitor e existe uma thread esperando nessa condição).

Quando uma thread sai do monitor — completando a operação requisitada ou esperando em uma variável de condição — ela desbloqueia a primeira thread na **fila de urgência**, ou, caso essa fila esteja vazia, na **fila de entrada**.

1.3. Implementação do problema do produtor/consumidor usando monitores

O pseudo-código abaixo ilustra como usar monitores para implementar o problema do produtor/consumidor [2].

```
monitor Prod-Cons {
   const int tam_buffer = N;
   T_item buffer[N];
   int in, out;              //ponteiros para entrada e saída de itens
   int count;                //número de itens no buffer
   cond naoCheio, naoVazio;  //variáveis de condição

   in = 0; out = 0; count = 0; //código de inicialização

   void INSERE (T_item item) {
      if (count == N)
         cwait(naoCheio); //buffer cheio, bloqueia o produtor
      buffer[in] = item;
      in = (in + 1) % N;
      count++;
      csignal(naoVazio);  //desbloqueia qualquer consumidor
   }

   T_item RETIRA () {
      if (count == 0)
         cwait(naoVazio); //buffer vazio, bloqueia o consumidor
      item = buffer[out];
      out = (out + 1) % N;
```

```
      count--;
      csignal(naoCheio);  //desbloqueia qualquer produtor
   }
}

void Produtor() {
   while(true) {
      produz_item(x);  //fora da secao critica
      INSERE(x);
   }
}

void Consumidor() {
   while(true) {
      y = RETIRA();
      consome_item(y); //fora da secao critica
   }
}

void main() {
   //inicia threads produtoras e consumidoras
}
```

Outra **vantagem importante de monitores sobre semáforos é que todas as funções de sincronização são confinadas no monitor** [2]. Dessa forma é mais fácil verificar se a sincronização foi implementada corretamente e corrigir os possíveis erros. Além disso, uma vez que o monitor foi corretamente programado, o acesso protegido ao recurso compartilhado está garantido para todas as threads. Com semáforos, ao contrário, o acesso ao recurso compartilhado só está corretamente implementado quando todos os trechos de código (em todas as threads) que fazem acesso ao recurso estão corretamente implementados.

1.4. Modelo alternativo de monitores com *notify* e *broadcast*

A definição de monitores de Hoare exige que, caso exista ao menos uma thread na fila de uma condição quando essa condição é sinalizada, o controle do monitor deve ser passado imediatamente para essa thread. Para isso, a thread que chamou *csignal* deverá **sair imediatamente do monitor** ou **ser bloqueada na fila de entrada do monitor**.

Há dois problemas com essa abordagem [2]:

1. Se a thread que emitiu o *csignal* não terminou ainda com o monitor, então duas novas trocas de contexto serão necessárias: uma para bloquear essa thread e outra para retomá-la quando o monitor ficar livre.
2. O escalonamento da thread desbloqueada com o *csignal* deverá ser perfeitamente confiável. Quando o *csignal* é emitido, uma thread na fila da condição sinalizada deverá ser ativada imediatamente e o escalonador deverá garantir que nenhuma outra thread entre no monitor antes da ativação, caso contrário, a condição sobre a qual a thread foi desbloqueada poderá mudar.

Para lidar com esses problemas, Lampson e Redell propuseram uma definição alternativa de monitores (usada nas linguagens Mesa e Modula-3) [2]. A primitiva *csignal* é substituída pela primitiva **cnotify** com a seguinte interpretação: quando uma thread em um monitor executa *cnotify(c)*, a fila da condição **c** é "notificada" e a thread corrente

continua executando (não é retirada do monitor). O resultado da notificação é que uma thread da fila de condição é desbloqueada e poderá voltar ao monitor quando ele ficar livre. Como não há garantia de que a condição **c** será preservada até que essa thread volte ao monitor, a thread deverá reavaliar a condição.

O pseudo-código abaixo ilustra a alteração na implementação do monitor para o problema do produdor/consumidor usando *cnotify*. A sentença **if** é substituída pelo loop **while** [2]:

```
void INSERE (T_item item) {
   while (count == N)
      cwait(naoCheio); //buffer cheio, bloqueia produtor
   buffer[in] = item;
   in = (in + 1) % N;
   count++;
   cnotify(naoVazio);  //desbloqueia qualquer consumidor
}

T_item RETIRA () {
   while (count == 0)
      cwait(naoVazio); //buffer vazio, bloqueia consumidor
   item = buffer[out];
   out = (out + 1) % N;
   count--;
   cnotify(naoCheio);  //desbloqueia qualquer produtor
}
```

Com essa alteração, há pelo menos uma avaliação extra da condição, por outro lado não há a necessidade de trocas de contexto adicionais e nenhuma restrição sobre quando a thread deverá voltar ao monitor depois de receber um *cnotify*.

Com a idéia de "notificação" ao invés de "reativação imediata", é possível incorporar outra primitiva ao monitor: **cbroadcast**. Essa primitiva faz **todas as threads na fila da condição serem notificadas**. Isso é conveniente em situações onde não é possível saber quantas threads deveriam ser reativadas (um exemplo é o caso do problema do produtor/consumidor quando o produtor inclui vários itens de uma só vez) [2].

2. Monitores em Java

A linguagem Java implementa uma definição de monitores baseada na proposta de Lampson e Redell. Todo objeto acessível por mais de uma thread pode ter um *lock* de exclusão mútua implícito, adquirido e liberado por meio da sentença de sincronização **synchronized** [1]. Exemplo:

```
synchronized (obj) {
   ... //seção crítica
}
```

Todas as execuções de sentenças *synchronized* que se referem ao mesmo objeto compartilhado excluem a execução simultânea de outras execuções.

Um "açúcar sintático" oferecido por Java é prefixar um método de uma classe com *synchronized*. Nesse caso, o corpo do método é protegido pela sentença *synchronized* (referência ao objeto *this*).

Dentro de uma **sentença** ou **método** *synchronized*, uma thread pode suspender ela mesma chamando o método *wait()*, sem argumentos. Normalmente o uso de *wait* aparece dentro de uma repetição condicional. Exemplo:

```
while (!condicao) {
  wait();
}
```

A thread que chama *wait* de dentro de um objeto libera o lock (bloqueio) desse objeto.

Para retomar a execução de uma thread suspensa em um dado objeto, alguma outra thread deve executar o método **notify()** de **dentro de uma sentença ou método *synchronized* que se refere ao mesmo objeto**. O método *notify* não recebe argumentos, o sistema de execução escolhe uma thread qualquer suspensa na fila do objeto e a desbloqueia. Se não houver threads suspensas para esse objeto, o *notify* é "perdido".

Outro método disponibilizado por Java é **notifyAll()**. Nesse caso **todas as threads suspensas no objeto são desbloqueadas**.

2.1. Restrições da implementação de monitores em Java

Vimos que as operações Java `wait`, `notify` e `notifyAll`, combinadas com métodos `synchronized` e classes definidas pelo usuário permitem construir objetos com características de **monitores**. Há, entretando, algumas diferenças importantes em relação a outras linguagens de programação que oferecem suporte completo a monitores (por exemplo, Concurrent Pascal).

Primeiro, em Java **não existe suporte de compilação para checar e previnir *condições de corrida* no programa**. (*Lembrando, existe condição de corrida em um programa quando o resultado da computação depende da ordem ou do instante de execução dos fluxos de controle concorrentes. Quando o resultado da computação pode ser incorreto, a "condição de corrida é ruim" e deve ser eliminada, em outros casos ela pode ser permitida.*) Adicionar a palavra reservada `synchronized` aos métodos de uma classe Java automaticamente provê *exclusão mútua* para threads que acessam os atributos de uma instância dessa classe via esses métodos. Entretanto, se por descuido algum desses métodos de acesso aos atributos de um objeto não é precedido de `synchronized`, pode ocorrer condições de corrida (a linguagem Java não provê ferramentas de análise do código para apontar possíveis erros de implementação no acesso a variáveis compartilhadas no programa).

Outra particularidade de Java é que **não há variáveis de condição explícitas**. Quando uma thread executa uma operação `wait`, a thread fica bloqueada na fila de uma *variável de condição implícita*, associada com o objeto do bloco synchronized. O fato de ter apenas uma variável de condição por objeto de sincronização torna o uso de monitores mais difícil [3].

2.2. Filas de condição em Java

Assim como todo objeto Java pode agir como um lock, todo objeto Java também pode agir como uma **fila de condição**. Os métodos `wait`, `notify` e `notifyAll` (da classe Object) são a API para acesso às filas de condição intrínsicas de cada objeto [4].

Um aspecto importante é que o *lock* e a *fila de condição* intrínsicas de um objeto estão sempre relacionados: **para chamar qualquer método da fila de condição sobre um objeto X, a thread deve possuir o lock do objeto**. Essa dependência deve-se ao fato de que a espera por uma determinada condição está sempre associada a uma informação de **estado**, e informações de estado estão necessariamente protegidas por mecanismos que visam preservar a consistência das informações que caracterizam o estado:

- uma thread não pode esperar por uma condição a menos que ela possa examinar as informações de estado; e
- uma thread não pode liberar outra thread que espera pela condição a menos que ela possa modificar as informações de estado.

`Object.wait()` atomicamente libera o *lock* e pede ao Sistema Operacional para suspender a thread corrente, permitindo que outras threads adquiram o *lock* e modifiquem o estado do objeto protegido. Depois de ser desbloqueada, a thread deve readquirir o *lock* antes de retomar a sua execução a partir do ponto em que foi bloqueada.

2.3. notify versus notifyAll

Como há apenas uma variável de condição implícita associada a um objeto de locação, pode ocorrer de duas ou mais threads estarem esperando na mesma variável, mas por condições lógicas distintas. Por isso, o uso das operações `notify` e `notifyAll` deve ser feito com cuidado [3].

Uma chamada `notifyAll` acorda (desbloqueia/sinaliza) **todas as threads** esperando naquele objeto, mesmo que estejam em subgrupos de espera distintos. As threads sinalizadas devem readquirir o *lock* do bloco synchronized e então checar novamente se podem continuar executando. Esse tipo de *semi-espera-ocupada* pode causar impactos no desempenho da aplicação (ex., uma thread é acordado, ganha o controle da CPU e verifica que deve voltar a se bloquear, todo esse processamento poderia ser economizado). Por outro lado, se `notify` for usado ao invés de notifyAll, a única thread acordada pode ser membro de um subgrupo errado (que não tem a condição lógica para prosseguir naquele momento). O uso de `notify` (ao invés de notifyAll) deveria ocorrer apenas quando os seguintes requisitos são atendidos:

- todas as threads esperam pela mesma condição lógica;
- cada notificação deve permitir que apenas uma thread volte a executar.

2.4. Simulando várias variáveis de condição

É possível usar objetos Java para obter o efeito que é similar ao uso de várias variáveis de condição [3]. Usando um bloco synchronized (ao invés de preceder os métodos com synchronized), podemos criar blocos de código sincronizados em locks distintos, e com eles permitir filas de condição distintas.

3. Exercícios

1. O que é um monitor?
2. Como as *variáveis de condição* dos monitores diferem dos semáforos?
3. Enumere as vantagens de monitores sobre semáforos.
4. Quais são os problemas encontrados na utilização da definição original de monitores?

5. Como o modelo alternativo com *notify* e *broadcast* resolve esses problemas?
6. Descreva como o conceito de monitor é implementado em Java.
7. Sumarize como monitores podem ser usados para implementar o controle de acesso a um recurso compartilhado.

4. Exercícios

1. O que é um monitor?
2. Como as *variáveis de condição* dos monitores diferem dos semáforos?
3. Enumere as vantagens de monitores sobre semáforos.
4. Quais são os problemas encontrados na utilização da definição original de monitores?
5. Como o modelo alternativo com *notify* e *broadcast* resolve esses problemas?
6. Descreva como o conceito de monitor é implementado em Java.
7. Sumarize como monitores podem ser usados para implementar o controle de acesso a um recurso compartilhado.

Referências

[1] M. L. Scott. *Programming Language Pragmatics*. Morgan-Kaufmann, 2 edition, 2006.

[2] W. Stallings. *Operating Systems – Internals and Design Principles*. Pearson - Prentice Hall, 6 edition, 2009.

[3] Richard H. Carver and Kuo-Chung Tai. *Modern multithreading*. Wiley, 2006.

[4] Brian Goetz et.al. *Java Concurrency in Practice*. Addison Wesley, 2006.