Computação Concorrente (DCC/UFRJ)

Aula 8: Sincronização com monitores

Prof. Silvana Rossetto

8 de maio de 2012

..recaptulando semáforos

```
const int tam_buffer = N;
T_item buffer[N];
semaphore EM=1;
semaphore CHEIO=0, VAZIO=N;

void Produtor() {
  while(true) {
    produz_item();
    semWait(VAZIO);
    semWait(EM);
    insere_item(buffer);
    semSignal(EM);
    semSignal(CHEIO);
  }
}
```

```
void Consumidor() {
      semWait(CHEIO);
      semWait(EM);
      retira_item(buffer);
      semSignal(EM);
      semSignal(VAZIO);
      consome_item();
  }

  void main() {
    //inicia threads prod/consum
  }
```

Dificuldades com semáforos

1. **erros de programação**: uma inversão na ordem das chamadas das operações *wait/post* pode levar o programa a situações de bloqueio indesejado
2. **dificuldade de manutenção do código**: as operações sobre semáforos aparecem de forma explícita ao longo do programa, complicando a manutenção do código

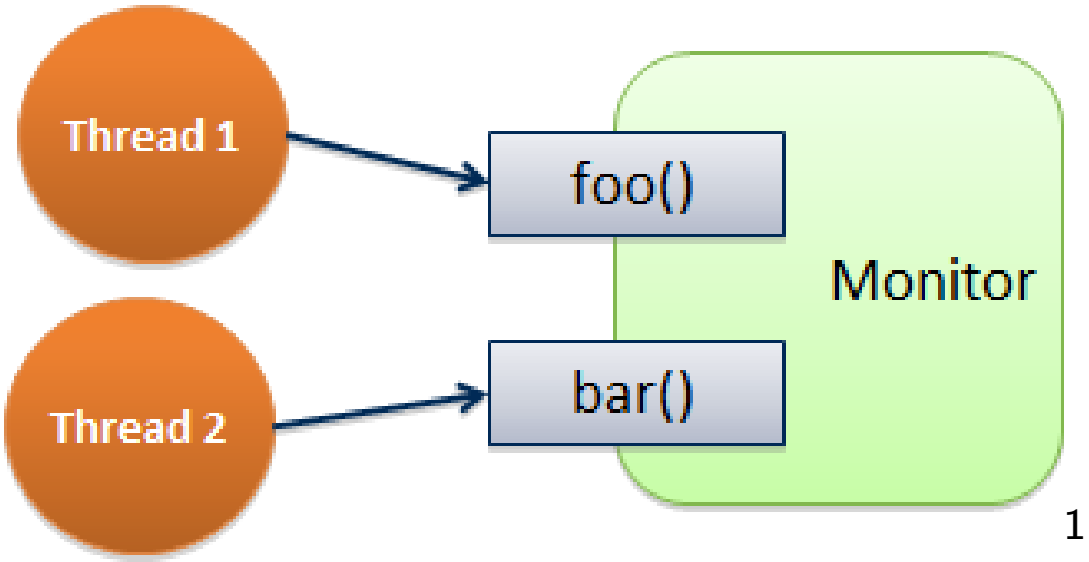
Definição de monitor

Módulo de software que consiste de:

1. **um ou mais procedimentos**
2. uma **sequência de inicialização**
3. **variáveis de condição**
4. **estado interno**

- Um **monitor** é uma construção de Linguagem de Programação que provê **funcionalidade equivalente a semáforos** e é "mais fácil" de controlar
- Proposto por Hoare, em 1974

Visão geral de monitor



1

[1] *Fonte: http://lycog.com*

Propriedades de monitores

1. **Apenas uma operação interna pode estar ativa a cada instante de tempo**: se uma thread chama uma operação do monitor e ele está ocupado, a thread é bloqueada
   - permite implementar a sincronização por exclusão mútua
2. **Qualquer operação pode suspender a si mesma em uma variável de condição**
   - permite implementar a sincronização por condição

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**

Variáveis de condição

Variáveis de condição são tipos de dados especiais acessíveis apenas dentro do monitor

Funções sobre variáveis de condição:

- **WAIT(c)**: suspende a execução da thread na variável de condição **c**
- **SIGNAL(c)**: retoma a execução de alguma thread bloqueada em um *wait* sobre a mesma variável de condição

Outra vantagem de monitores sobre semáforos

Todas as funções de sincronização são confinadas no monitor

- ...mais fácil verificar a corretude da implementação
- ...uma vez que o monitor foi corretamente programado, o acesso protegido ao recurso compartilhado está garantido para todas as threads

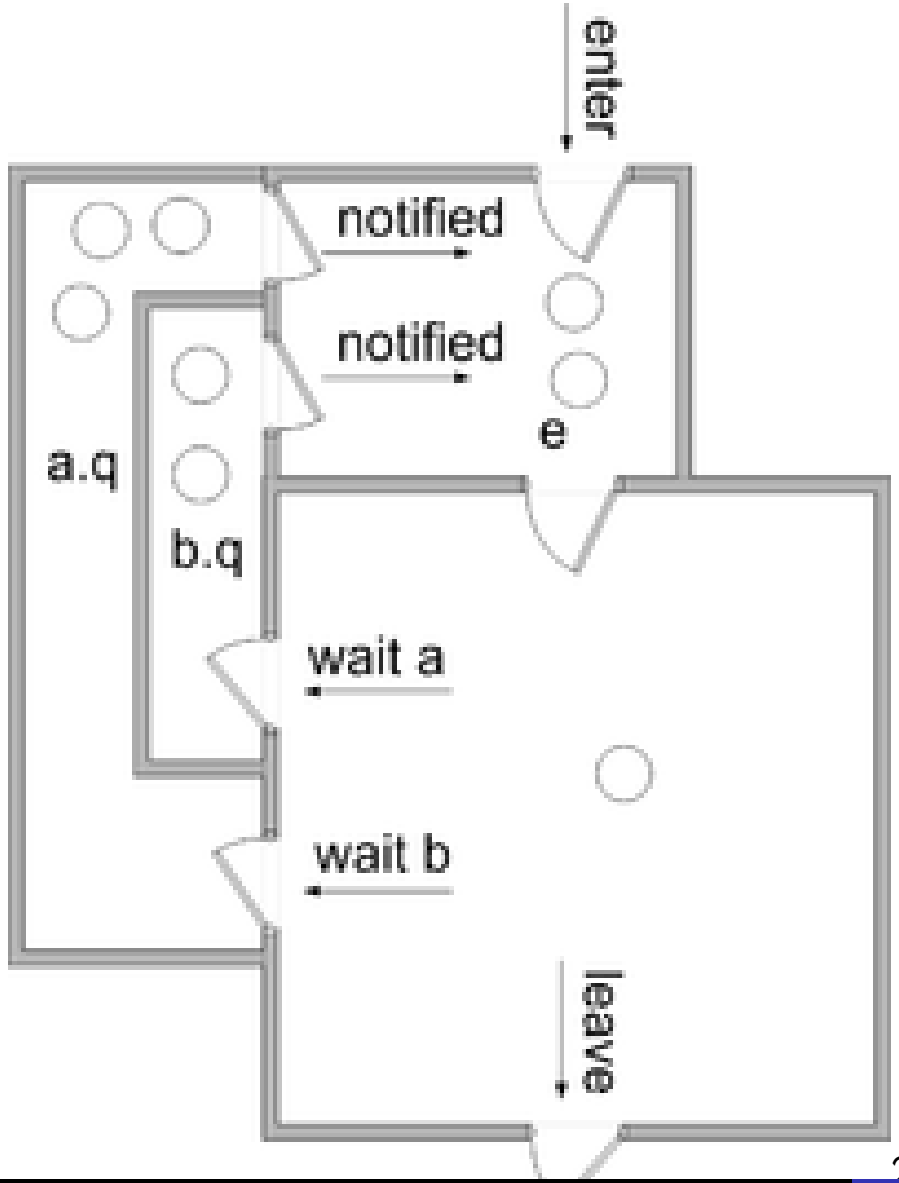
Com semáforos, ao contrário, o acesso ao recurso compartilhado só está corretamente implementado quando todos os trechos de código (em todas as threads) que fazem acesso ao recurso estão corretamente implementados

Semântica dos monitores de Lampson e Redell

Provê a 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 continua executando
- uma thread da fila de condição é desbloqueada
- como não há garantia de que a condição **c** será preservada até que essa thread volte ao monitor, a thread deverá reavaliar a condição

Semântica dos monitores de Lampson e Redell



Produtor/consumidor com monitores e *notify*

```
T_item buffer[N];
int in, out, count;
cond naoCheio, naoVazio;

in = 0; out = 0; count = 0;

void INSERE (T_item item) {
  while (count == N) cwait (naoCheio);
  buffer[in] = item;
  in = (in + 1) % N;
  count++;
  cnotify (naoVazio);  }

T_item RETIRA () {
  while (count == 0)  cwait (naoVazio);
  item = buffer[out];
  out = (out + 1) % N;
  count--;
  cnotify (naoCheio);  }
```

Primitiva *cbroadcast*

A primitiva **cbroadcast** estende a primitiva básica *cnotify*:

- faz **todas as threads na fila da condição serem notificadas**
- desejável em situações onde não é possível saber quantas threads deveriam ser reativadas

um exemplo é o caso do problema do P/C quando o produtor inclui vários itens de uma só vez

Monitores em Java

Uso da sentença **synchronized**:

- todas as execuções de sentenças *synchronized* que se referem ao mesmo objeto compartilhado excluem a execução simultânea de outras execuções
- dentro de uma **sentença** ou **método** *synchronized*, uma thread pode suspender ela mesma chamando o método **wait()** sem argumentos

```
void foo(){
   synchronized(this) {
      x++;
      y = x;
   }
}
```

```
synchonized void bar(){
      y++;
      x += 3;
}
```

Monitores em Java

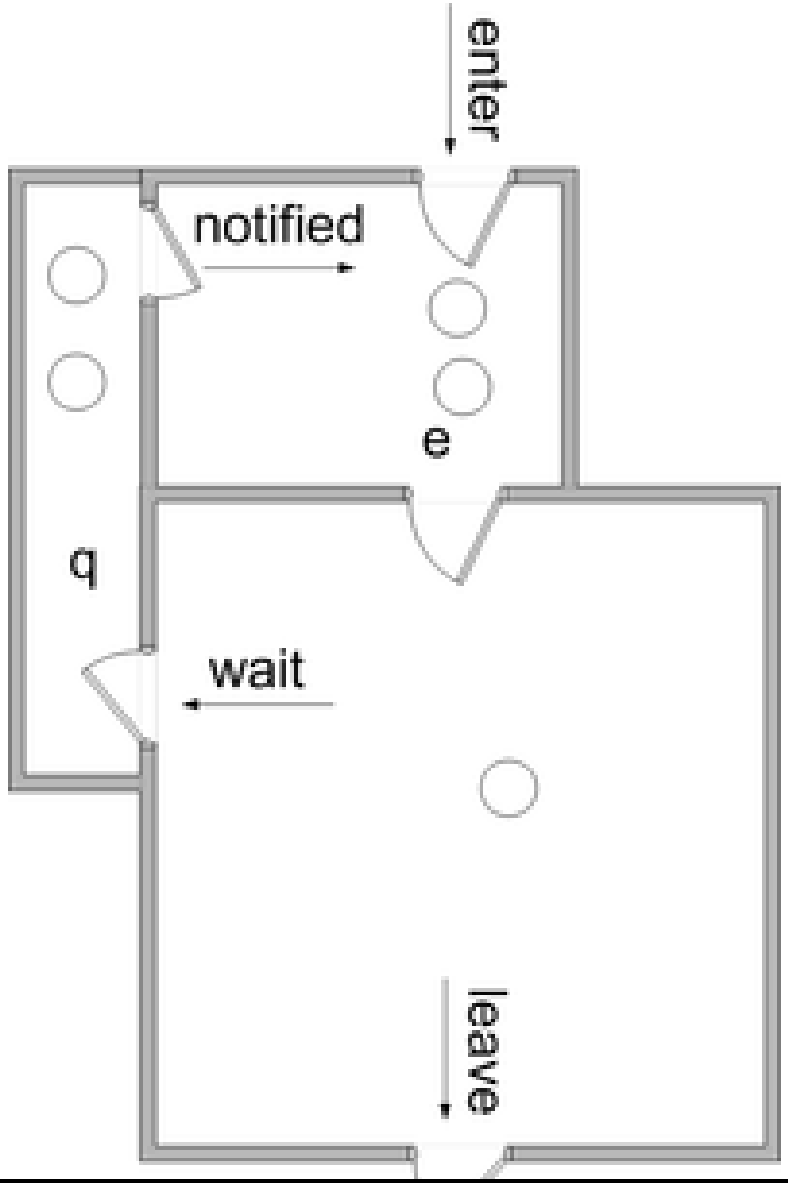
Notificação para uma thread

- 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 *synchronized* que se refere ao mesmo objeto
- O método *notify* não recebe argumentos, uma thread qualquer suspensa na fila do objeto é desbloqueada

Notificação para todas as threads

O método **notifyAll()** desbloqueia todas as threads suspensas no objeto

Semântica dos monitores em Java



Monitores em Java

As operações **wait**, **notify** e **notifyAll**, combinadas com métodos **synchronized** permitem construir objetos Java com características de **monitores**

Suporte parcial do conceito geral de monitores

1. Em Java **não existe suporte de compilação para checar e previnir condições de corrida no programa** (se algum método de acesso a variáveis compartilhadas não é precedido de `synchronized`, pode ocorrer condições de corrida)
2. Em Java **não há variáveis de condição explícitas** (quando uma thread executa uma operação **wait**, ela fica bloqueada na fila de uma **variável de condição implícita**, associada com o objeto do bloco synchronized)

Filas de condição

- Todo objeto Java pode agir como um **lock** e 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

Object.notify versus Object.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

Impacto de desempenho com notifyAll

- Uma chamada `notifyAll` acorda (desbloqueia/sinaliza) **todas as threads** esperando naquele objeto, mesmo que estejam em subgrupos de espera distintos
- 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)

Possibilidade de erro com notify

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)

Quando usar notify() ao invés de notifyAll()

O uso de **notify** (ao invés de **notifyAll**) deveria ocorrer apenas quando os seguintes requisitos são atendidos:

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

Simulando várias variáveis de condição

- É possível usar objetos Java para obter o efeito que é similar ao uso de várias variáveis de condição
- 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

Referências bibliográficas

1. *Programming Language Pragmatics*, Scott, Morgan-Kaufmann, ed. 2, 2006
2. *Operating Systems – Internals and Design Principles*, Stallings, Pearson, ed. 6, 2009