Programação concorrente com objectos

Paulo Sérgio Almeida

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho

2007/2008

Programação concorrente com Objectos

Tópicos:

- Suporte nas linguagens para concorrência e distribuição.
- Objectos e actividades: objectos activos e passivos.
- Controlo de concorrência intra-objecto: objectos atómicos, quase-concorrentes e concorrentes.
- Controlo de concorrência inter-objecto.
- Imutabilidade.

Suporte nas linguagens para concorrência

O suporte à concorrência pode variar:

- Bibliotecas para exprimir concorrência e comunicação, em cima de uma linguagem OO (como C++).
- Linguagens OO com algum suporte nativo de concorrência, juntamente com bibliotecas (e.g. JavaMI).
- Linguagens com suporte directo para concorrência e distribuição
  - linguagens experimentais, como Emerald ou Acute,
  - linguagens em uso “sério” como Erlang.

Primitivas básicas de controlo de concorrencia

- Os mecanismos básicos de controlo de concorrência mais populares baseam-se nos conceitos clássicos de:
  - Semáforos (Dijkstra) e
  - Monitores (Brinch-Hansen / Hoare).
- O desenho de API's como POSIX Threads ou de linguagens como Java ou C# tem privilegiado variantes de monitores.
- (Outros conceitos vão sendo testados experimentalmente, como as *Chords* de C_w baseadas no *join calculus*.)
- Um monitor associa a possibilidade de controlar o acesso a um objecto em termos de:
  - exclusão mútua entre métodos e
  - sincronização via variáveis de condição.

Objectos e actividades

Objectos passivos

objectos e threads são considerados conceitos independentes

- ambos são manipulados explicitamente pelo programador.
- o mais frequente nas linguagens comuns, como C++ ou Java.

Objectos activos

existe uma unificação entre objecto e thread

- um objecto activo pode ter uma thread associada;
- a execução de uma operação pode ter uma thread dedicada;
- a concorrência é criada implicitamente:
  - pela instanciação assíncrona;
  - pela invocação assíncrona.

Objectos activos

Instanciação assíncrona

O objecto criado fica a executar um *body* (que pode ser implícito ou explícito) que fica em ciclo à espera de pedidos.

Invocação assíncrona

o cliente prossegue concorrentemente e o resultado é obtido mais tarde, de diferentes modos:

one way invocations não devolvem resultados; se necessário o servidor envia o resultado através de outra invocação.

objectos futuros podem ser devolvidos por invocações assíncronas. O futuro é usado pelo cliente para obter o resultado. Os futuros podem ser implícitos ou explícitos.

Objectos activos como design pattern

- Normalmente é oferecido o conceito de objecto passivo.
- Tal não impede que se possa utilizar o conceito de objecto activo, quando apropriado, para estruturar o software.
- Objectos activos, quando não suportados directamente pela linguagem, podem ser construidos como um *design pattern*.
- Ver [POSA2] Pattern-Oriented Software Architecture, Vol 2 (Patterns for Concurrent and Networked Objects), Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann, Wiley, 2000.

Controlo de concorrência intra-objecto

- Um objecto que permita várias actividades suporta concorrência intra-objecto.
- Controlo de concorrência é necessário para manter a validade do estado do objecto, protegendo contra corridas e bloqueando invocações até estarem atingidas certas condições.
- O controlo de concorrência pode ser classificado como:
  - completamente **explícito** por parte do programador, ou apenas parcialmente especificado (**implícito**);
  - pode ser efectuado pelo cliente (**externo**) ou pela implementação do objecto (**interno**);
  - pode ser escrito em cada método do objecto (**dependente**) ou apenas em certas partes da implementação do objecto dedicadas ao controlo de concorrência (**independente**).

Classificação da concorrência intra-objecto

Relativamente à concorrência intra-objecto, podemos classificar um objecto de:

Sequencial ou atómico quando não suporta concorrência intra-objecto; processa uma mensagem de cada vez.

Quase-concorrente quando várias invocações podem coexistir mas no máximo uma não está suspensa; semelhante ao conceito de monitor.

Concorrente suporta verdadeira concorrência entre invocações, exigindo controlo a ser especificado pelo programador.

Objectos sequênciais ou atómicos

- A forma mais simples de objecto para uso concorrente processa uma invocacão de cada vez: as invocações são serializadas.
- Todos os seus métodos adquirem um *lock* no inicio da execução, libertando-o quando terminam. (Métodos `synchronized` em Java.)
- Todos os métodos acabam em tempo finito, não ficando bloqueados, e garantidamente libertam os *locks*.
- O estado do objecto obedece aos invariantes no ínicio e no fim de cada método.
- Facilita construção do software com garantias formais de correcção.

Exemplo: conta bancária em Java

Uma classe com todos os métodos `synchronized`, sendo os objectos atómicos:

```
class Conta {
  int saldo;

  public synchronized int consulta() {
    return saldo;
  }

  public synchronized void deposito(int valor) {
    saldo = saldo + valor;
  }

  public synchronized void levantamento(int valor) {
    saldo = saldo - valor;
  }
}
```

Monitores / objectos quase-concorrentes

- O conceito de monitor permite a obtenção de objectos quase-concorrentes: que suportam várias invocações em curso, ainda que só uma no máximo esteja não bloqueada.
- Ao contrário dos objectos atómicos, as invocações não são serializadas: ainda que bloqueadas, podem já ter executado parcialmente, estando à espera de um evento externo a elas.
- Permite controlar a colaboração entre clientes que fazem uso de um serviço.
- Cada método faz uso de um *mutex* e de variáveis de condição. (Java usa apenas uma variável de condição, implícita.)
- Exemplo: produtor-consumidor em Java. Um *bounded buffer* é um monitor para uso por threads produtoras e consumidoras.

  O buffer poderá estar cheio ou vazio, podendo ser necessário bloquear pedidos.

Objectos concorrentes

- Um serviço poderá disponibilizar operações que poderão ser demoradas a executar (por exemplo input/output), ainda que não dependam umas das outras.
- Nestes casos deverá ser implementado como um servidor concorrente, ou seja como um objecto que permita verdadeira concorrência entre invocações a ser processadas.
- Tal leva a uma implementação com controlo de concorrência de granularidade mais fina.
- Em vez de *locking* a nível do objecto, são por exemplo usados *locks* relativamente a sub-objectos.
- A implementação é mais complexa, necessitando mais cuidado.

Exemplo: operações sobre objectos em repositórios

```
Interface Operacao { void aplica(Object o); }

class Repositorio {
  public synchronized void insere(String nome, Object o) {
    // insere o objecto no repositorio
  }

  public void aplica(String nome, Operacao op) {
    Object obj;
    synchronized (this) {
      obj = ...  // procura o objecto pelo nome
    }
    synchronized (obj) {
      op.aplica(obj);       // operacao potencialmente demorada
    }
  }
}
```

Controlo de concorrência inter-objecto

- Concorrência inter-objecto: na invocação de operações em objectos diferentes.
- Controlo de concorrência pode ser necessário para garantir coerência no estado global do sistema (e não de objectos individuais).
- Tal pode acontecer quando existem dependências entre operações a ser efectuadas em objectos diferentes.

Exemplo: operações sobre duas contas bancárias

- Para realizar uma transferência é realizada uma operação de levantamento na primeira conta e outra de depósito na segunda.

```
c1.levantamento(3000);
c2.deposito(3000);
```

- Suponhamos que é consultado concorrentemente o saldo de cada conta (para por exemplo obter a soma dos saldos).

```
i = c1.saldo();
j = c2.saldo();
```

- Se tal for efectuado depois do levantamento mas antes do depósito, o resultado é inválido.
- Nestes casos é necessário prevenir interferência entre cada conjunto de operações: obter isolamento.
- Uma hipótese é forçar serialização das operações.

Exemplo: operações sobre duas contas bancárias

- Uma solução para o problema pode passar por utilizar um objecto `mutex`.

```
Conta c1, c2; Mutex m;

// cliente 1; realiza uma transferencia
  m.lock();
  c1.levantamento(3000);
  c2.deposito(3000);
  m.unlock()

// cliente 2; consulta as duas contas
  m.lock();
  i = c1.saldo();
  j = c2.saldo();
  m.unlock()
```

Exemplo: operações sobre duas contas bancárias

- No caso geral vários clientes podem manipular várias contas.
- Solução: cada cliente adquire os *locks* dos objectos a manipular, efectua as operações em questão, e finalmente liberta os *locks*.

```
// cliente 1; realiza uma transferencia
    lc1.lock();
    lc2.lock();
    c1.levantamento(3000);
    c2.deposito(3000);
    lc1.unlock();
    lc2.unlock();

// cliente 2; consulta as duas contas
    lc1.lock();
    lc2.lock();
    i = c1.saldo();
    j = c2.saldo();
    lc1.unlock();
    lc2.unlock();
```

Ordem de aquisição de locks

- Adquirir locks por ordem arbitrária pode causar deadlock.

```
// cliente 1; realiza uma transferencia
    lc1.lock();
    lc2.lock();
    c1.levantamento(3000);
    c2.deposito(3000);
    lc1.unlock();
    lc2.unlock();

// cliente 2; consulta as duas contas
    lc2.lock();
    lc1.lock();
    i = c1.saldo();
    j = c2.saldo();
    lc2.unlock();
    lc1.unlock();
```

- Podemos ter cada thread com um lock e à espera do outro.

Ordem de aquisição de locks

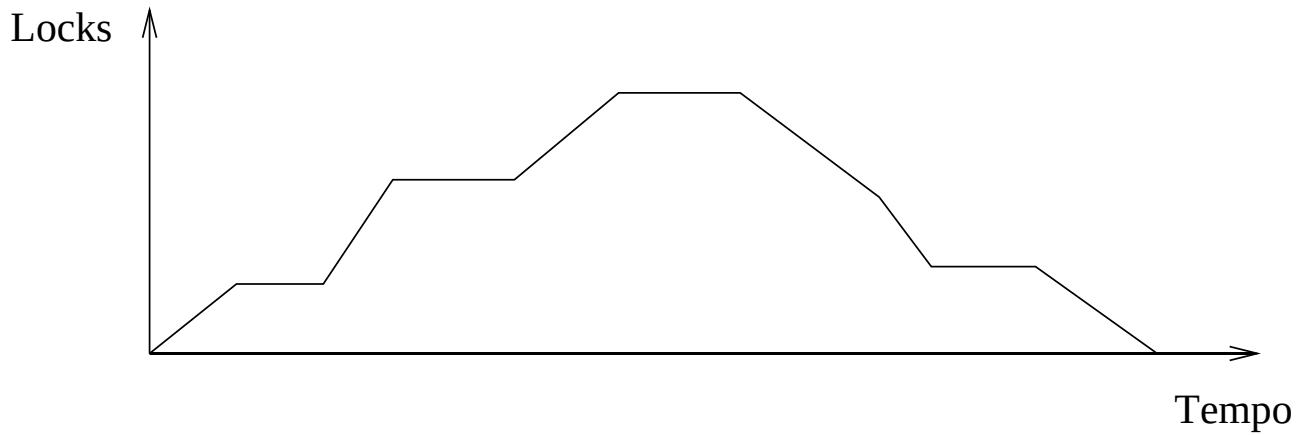
- Dependências cíclicas de aquisição de locks podem causar deadlock.
- Uma solução para evitar deadlocks é:
  - impor uma ordem total sobre os locks envolvidos;
  - adquirir os lock necessários por ordem, do menor para o maior;
  - (ao libertar a ordem não é importante.)

```
// cliente 1; realiza uma transferencia
    lc1.lock();
    lc2.lock();
    ...

// cliente 2; consulta as duas contas
    lc1.lock();
    lc2.lock();
    ...
```

Two-phase locking

- Técnica de controlo de concorrência usada em bases de dados e sistemas de objectos distribuídos, para obter isolamento entre transacções ao garantir equivalência a serialização.
- Cada transacção envolvida passa por duas fases: aquisição de *locks*; libertação de *locks*.
- Depois de algum *lock* ser libertado, mais nenhum é adquirido.



- Um *lock* de um objecto só é libertado quando a transacção já possui todos os *locks* de que necessita.

Operações sobre duas contas bancárias com 2PL

Nova versão com a estratégia two-phase locking:

- um lock é adquirido o mais tarde possível, na primeira fase;
- um lock é libertado o mais cedo possível, na segunda fase;
- operações sobre os objectos em ambas as fases.

```
// cliente 1; realiza uma transferencia
    lc1.lock();
    c1.levantamento(3000);
    lc2.lock();
    lc1.unlock();
    c2.deposito(3000);
    lc2.unlock();

// cliente 2; consulta as duas contas
    lc1.lock();
    i = c1.saldo();
    lc2.lock();
    lc1.unlock();
    j = c2.saldo();
    lc2.unlock();
```

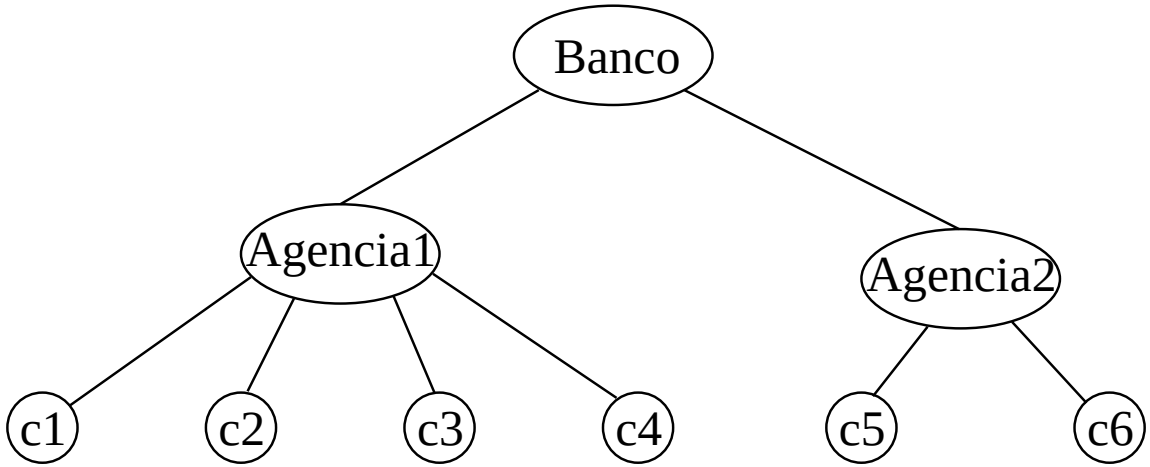
Modos de locks

- Locks de exclusão mútua (binários) podem ser demasiado restrictivos.
- É útil distinguir diferentes tipos de acesso e oferecer *locks* de leitura e de escrita,
- o que permite que várias leituras possam prosseguir concorrentemente.
- Tabela de compatibilidade de *locks*:

	read	write
read	+	-
write	-	-

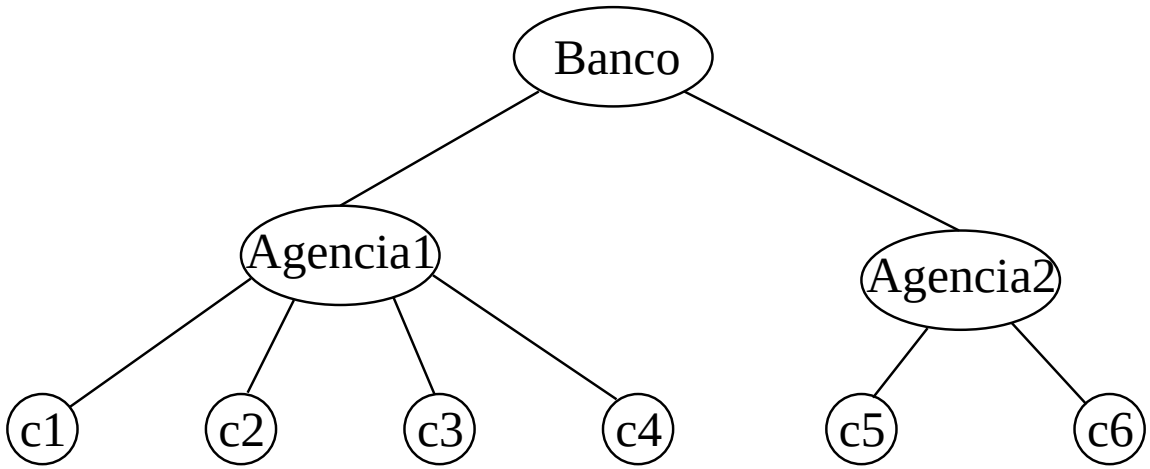
Locking hierárquico

- Em sistemas de objectos hierárquicos objectos podem ser *containers* de um conjunto de objectos *componentes*.
- *Locking* hierárquico permite fazer o *lock* de todos os componentes de um *container* de uma só vez.
- O *locking* hierárquico é vantajoso quando existem muitos objectos e uma hierarquia de composição pouco profunda.



Locking hierárquico

- São oferecidas duas operações: `lock` e `intention-lock`.
- Para fazer *lock* a X (*container* ou componente) é feito um `intention-lock` em todos os *containers* desde a raiz até ao *container* de X, seguida de um `lock` de X.




```
// Processo 1: lock de c1
Banco.lock(intention_write);
Agencia1.lock(intention_write);
c1.lock(write);
```

```
// Processo 2: lock de Agencia2
Banco.lock(intention_write);
Agencia2.lock(write);
```

Compatibilidade de locks hierárquicos

	intention read	read	intention write	write
intention read	+	+	+	-
read	+	+	-	-
intention write	+	-	+	-
write	-	-	-	-

Imutabilidade

- Objectos que não mudam de estado podem ser usados concorrentemente sem restrições.
- Estes podem ser úteis para uso na implementação (como sub-objectos) de objectos concorrentes, diminuindo as necessidades de controlo de concorrência.
- Sempre que possível deve ser usado suporte da linguagem para ter garantias que um dado objecto é imutável; por exemplo a palavra-chave `final` em Java.
- É necessário evitar que se escapem referências para o futuro objecto imutável enquanto este está a ser construído.