Concorrência com Múltiplos Objectos

Carlos Baquero (Slides: Paulo Sérgio Almeida)

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho

Objectos e threads

- Objectos e threads são conceitos independentes;
- Ambos são manipulados explicitamente pelo programador;
- Objectos são passívos, acedidos por threads;
- Encapsulamento de estado não chega para evitar corridas;
- Controlo de concorrência necessário:
  - dar visibilidade de escritas a outras threads;
  - proteger contra corridas;
  - manter invariantes de estado dos objectos;

Controlo de concorrência interno vs externo

- O controlo de concorrência pode ser:
  - *externo*: efectuado pelo cliente do objecto;
  - *interno*: efectuado pela implementação do objecto;
- Interno é mais seguro e preferível por omissão:
  - a implementação encapsula CC;
  - o cliente não tem que se preocupar ao aceder ao objecto;
- Externo é mais flexível mas perigoso:
  - cliente tem que escrever código de CC ao aceder ao objecto;
  - permite padrões de CC mais genéricos, e.g., abrangindo várias invocações de métodos sobre vários objectos;

Controlo de concorrência intra- vs inter-objecto

- Intra-objecto:
  - diz respeito ao CC numa invocação sobre um único objecto;
  - de preferência interna: "o objecto protege-se";
- Inter-objecto:
  - envolvendo interacção entre invocações sobre objectos distintos;
  - e.g., o conjunto de operações deve ser visto como atómico;
  - normalmente externa, pelos clientes dos objectos;
  - não é suficiente cada objecto proteger-se individualmente;

Classificação da concorrência intra-objecto

Relativamente ao CC intra-objecto, um objecto pode ser:

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

Quase-concorrente suporta várias invocações em curso, mas no máximo uma não está suspensa; semelhante ao conceito de monitor.

Concorrente suporta verdadeira concorrência entre invocações, com CC mais fino 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 métodos adquirem um *lock* no inicio da execução, libertando-o no fim. (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 de objecto atómico: conta bancária em Java

classe com todos os métodos synchronized:

```
class Conta {
  private 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

- Suportam várias invocações em curso, ainda que só uma no máximo esteja não bloqueada.
- As invocações não são serializadas: podem já ter executado parcialmente, estando à espera de um evento externo.
- Permitem colaboração entre clientes sem esperas activas.
- Exemplo: *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

- Permitem verdadeira concorrência entre invocações em curso.
- Um serviço poderá disponibilizar operações:
  - potencialmente demoradas;
  - que não dependam umas das outras;
  - e.g., envolvendo sub-objectos distintos;
- Nestes casos é útil ser um objecto concorrente.
- Implementação tem CC de granularidade mais fina, envolvendo *locks* sobre 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.
- 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
    c1.lock();
    c2.lock();
    c1.levantamento(3000);
    c2.deposito(3000);
    c1.unlock();
    c2.unlock();

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

Ordem de aquisição de locks

- Adquirir locks por ordem arbitrária pode causar deadlock.

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

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

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

Ordem de aquisição de locks

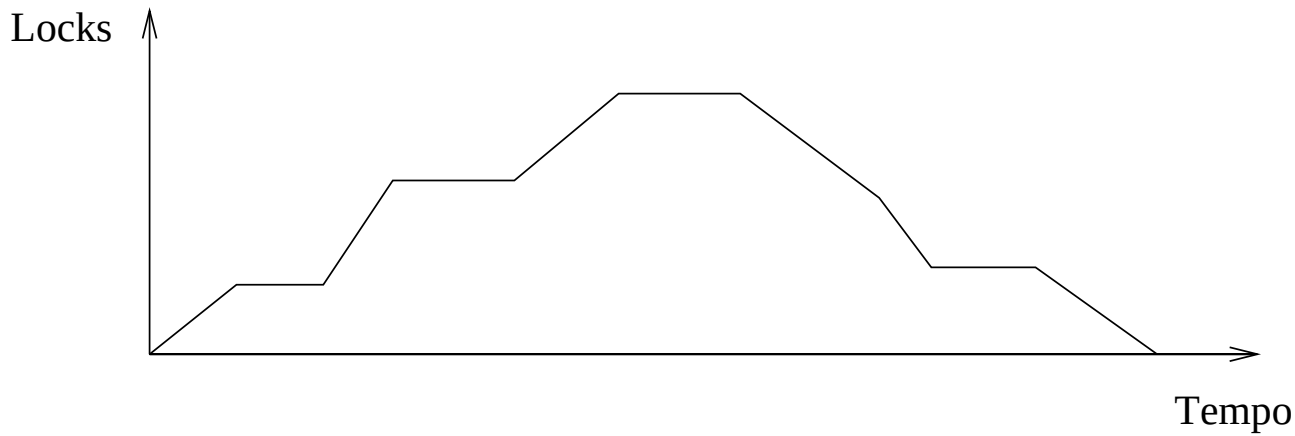
- Dependências cíclicas de aquisição podem causar deadlock.
- Uma solução para evitar deadlocks é:
  - impor uma ordem total sobre os locks envolvidos;
  - adquirir os lock necessários por ordem, do menor para o maior;
  - (ao libertar a ordem não é importante.)

```
// cliente 1; realiza uma transferencia
    c1.lock();
    c2.lock();
    ...

// cliente 2; consulta as duas contas
    c1.lock();
    c2.lock();
    ...
```

Two-phase locking

- Técnica de controlo de concorrência usada em bases de dados, para obter isolamento entre transacções.
- Garante mesmo resultado que exclusão mútua com mutex global, com mais concorrência.
- 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
    c1.lock();
    c1.levantamento(3000);
    c2.lock();
    c1.unlock();
    c2.deposito(3000);
    c2.unlock();

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

Hand-over-hand locking

- Certas estruturas exibem ordem de acesso:
  - exemplos: listas, árvores
- Não é posível atingir um determinado objecto percorrendo caminhos diferentes.
- Motiva técnica de *hand-over-hand locking* ou *lock chaining*:
  - fazer lock da raiz;
  - em cada iteração na travessia da estrutura:
    - adquirir lock do próximo objecto;
    - libertar lock do corrente;
- Permite que diferentes threads estejam a trabalhar concorrentemente em partes diferentes.
- Usar apenas se concorrência esperada e operações justificarem.
- Se abusada pode provocar grande ineficiência.

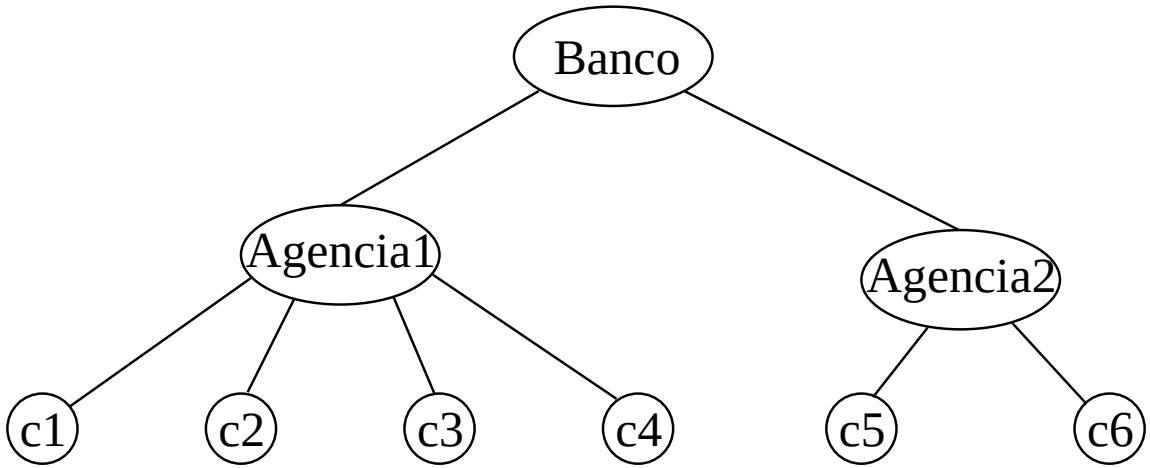
Modos de locks

- Locks de exclusão mútua podem ser demasiado restrictivos.
- É útil distinguir diferentes tipos de acesso e oferecer *locks* de leitura e de escrita.
- Várias leituras podem prosseguir concorrentemente.
- Tabela de compatibilidade de *locks*:

	read	write
read	+	-
write	-	-

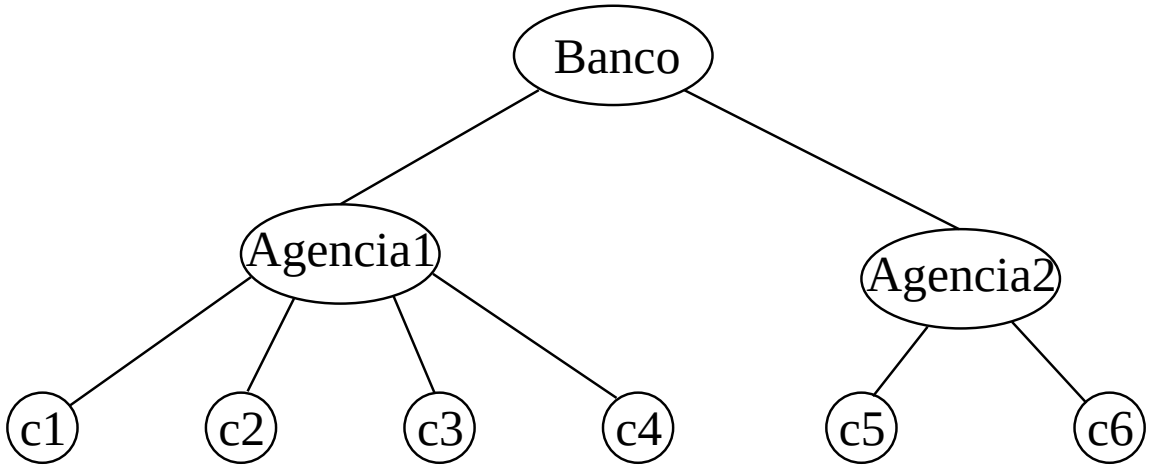
Locking hierárquico

- Em sistemas de objectos hierárquicos, objectos podem ser *containers* de um conjunto de objectos *componentes*.
- *Locking* hierárquico permite fazer *lock* de toda uma sub-árvore de um *container* sem o fazer individualmente para cada objecto.
- 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 é feito um `intention-lock` desde a raíz até ao *container* de X, seguida de um `lock` de X.
- Locks são libertados das folhas para a raíz, ou por qualquer ordem no fim da transação.




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

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

Compatibilidade de locks hierárquicos

- É útil distinguir leituras e escritas;
- Assim há 4 combinações possíveis:

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

Compatibilidade de locks hierárquicos

- Um quinto modo RIW combina *read* e *intention write*.
- Útil num caso comum em que uma transacção quer:
  - ler uma sub-árvore inteira e,
  - fazer update em alguns objectos dessa sub-árvore.
- Sem este modo:
  - ou se fazia lock de escrita na raiz; não permite concorrência;
  - ou se teria que fazer lock explícito a potencialmente muitos objectos da sub-árvore; ineficiente.

	IR	R	IW	RIW	W
IR	+	+	+	+	-
R	+	+	-	-	-
IW	+	-	+	-	-
RIW	+	-	-	-	-
W	-	-	-	-	-