O Modelo de memória de Java

Paulo Sérgio Almeida

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho

2007/2008

Comportamentos inesperados de programas sem sincronização

- Consideremos inicialmente $a = b = 0$ e duas threads a ler e escrever sem sincronização:

```
a = 1;          b = 1;
x = b;          y = a;
```

- Considerando apenas interleavings, quando as threads terminam poderiamos ter apenas os resultados:
  - $x = 0, y = 1$;
  - $x = 1, y = 0$;
  - $x = 1, y = 1$.
- Será possível obter o resultado:
  - $x = 0, y = 0$?

Coerência sequencial e coerências mais fracas

- Coerência sequencial é quando o comportamento observado equivale a:
  - existir uma ordem total resultante de um interleaving das operações das várias threads;
  - uma escrita é vista por qualquer leitura posterior (na ordem total) mesmo de outra thread.
- No exemplo anterior, com coerência sequencial o caso $x = y = 0$ não seria possível.
- Para obter uma execução eficiente em arquitecturas modernas, são normalmente oferecidas coerências mais fracas, motivadas nomeadamente por:
  - caches dos processadores;
  - optimizações no código gerado; e.g. uso de registos e reordenamento de instruções.
- Em arquitecturas e linguagens modernas, como Java, é normal poder acontecer o caso $x = y = 0$.

O modelo de coerência de memória de Java

- O JMM (Java Memory Model) define as garantias de visibilidade entre escritas e leituras feitas por diferentes threads.
- As garantias são mais fracas do que coerência sequencial e podem levar a resultados não intuitivos.
- Definição independente da arquitectura de hardware, permite portabilidade de código.
- Diferenças de modelos de memória das arquitecturas de hardware são escondidas pelas implementações da JVM.

Programas com sincronização insuficiente

- Motivado por optimização do desempenho de programas.
- Alguns exemplos:
  - iniciação lazy de objectos;
  - testar terminação;
- Vários idiomas em uso generalizado estão errados.
- É possível obter idiomas correctos com pouco uso de locking.
- Java oferece suporte, nomeadamente:
  - volatile
  - final

Níveis de abstracção

Do mais seguro para o mais perigoso:

- Abstracções de alto nível na biblioteca de concorrência
- Uso de primitivas de sincronização nativas:
  - blocos synchronized;
  - wait(), notify(), ...
- Primitivas de baixo nível:
  - variáveis volatile
  - classes atómicas
- Ausência deliberada de sincronização:
  - de evitar; extremamente difícil.
  - intuição engana; grande probabilidade de erros .

Ordem parcial happens-before

- O JMM define uma ordem parcial, happens-before, sobre as acções de um programa.
- As acções envolvem:
  - leituras e escritas em variáveis,
  - locks e unlocks de monitores,
  - start e join de threads.
- É garantido que uma thread a executar uma acção B vê o resultado de uma acção A (possivelmente de outra thread) apenas se A e B estiverem ordenados por happens-before.

Regras para happens-before

A ordem parcial happens-before é obtida por regras que incluem:

ordem no programa duas acções da mesma thread são relacionadas se aparecem pela ordem do programa;

lock de monitor um unlock de um monitor é relacionado com os locks subsequentes no mesmo monitor

variável volatile uma escrita numa variável volatile é relacionada com todas as leituras subsequentes da mesma variável;

thread start uma invocação de Thread.start() é relacionada com todas as acções da thread que começou a executar;

thread join todas as acções da thread que terminou são relacionadas com Thread.join() sobre a thread;

Ordem total sobre acções de sincronização

- Apesar de em geral existir apenas uma ordem parcial, acções de sincronização são totalmente ordenadas:
  - locks e unlocks de monitores;
  - leituras e escritas de variáveis volatile;
  - start e join de threads.
- Isto permite descrever happens-before em termos de acções subsequentes (locks de monitores e leituras de variáveis volatile).

Happens-before e visibilidade

```
T1:

r1.x = 1;
lock(M);
g = r1;
unlock(M);
```

```
T2:

lock(M);
r2 = g;
unlock(M);
y = r2.x;
```

- Se T1 adquirir o lock primeiro, todas as acções até ao unlock de T1 são observadas por todas as acções de T2 depois do lock.
- Nomeadamente, a escrita de 1 no campo x é vista por T2, mesmo tendo sido feita sem nenhum lock adquirido.

Programas correctamente sincronizados

- Dois acessos são conflituosos quando pelo menos um é uma escrita.
- Uma *data race* ocorre quando dois acessos conflituosos não estão ordenados por happens-before.
- Um programa é correctamente sincronizado se todas as execuções possíveis com coerência sequencial não contêm data races.
- Se um programa estiver correctamente sincronizado, as suas execuções serão equivalentes a termos coerência sequencial.

Programas correctamente sincronizados

- Estará o programa abaixo correctamente sincronizado?

```
T1:
x = 1;
lock(M);
y = 1;
unlock(M);
```

```
T2:
lock(M);
r1 = y;
unlock(M);
r2 = x;
```

- Não: pois assumindo uma execução (com coerência sequencial) em que T2 adquire primeiro o lock, temos uma data race, pois os acessos conflituosos a x não são ordenados por happens-before.

Exemplo: iniciação lazy

- Suponhamos que queremos iniciar uma única instância, da primeira vez que é usada, e queremos poupar na sincronização.
- Primeira versão, obviamente errada:

```
class C{
  private static Obj o;
  public static getObj() {
    if (o == null)
      o = new Obj();
    return o;
  }
}
```

- Total ausência de sincronização.
- Várias threads podem ver o=null e iniciar objecto.
- Mais grave, pode ser observado um objecto parcialmente construído.

Exemplo: iniciação lazy

- Segunda versão: double-checked-locking.
- Tenta não fazer locking no caso mais comum.
- A intuição seria que o caso de duas threads verem null seria protegido, testando-se outra vez, agora com sincronização.

```
class C{
  private static Obj o;
  public static getObj() {
    if (o == null) {
      synchronized (C.class) {
        if (o == null)
          o = new Obj();
      }
    }
    return o;
  }
}
```

- Apenas uma thread inicia objecto.
- Mas será que funciona? Ou a intuição engana . . .

Variáveis volatile

- Suponhamos o exemplo:

```
boolean finished;
...
while (!finished)
  doSomething();
```

- O objectivo seria terminar o ciclo quando uma outra thread escrevesse na variável finished.
- Tal não funciona, pois sem sincronização não há garantias que a escrita da outra thread seja vista.
- Este caso pode ser codificado correctamente com:

```
volatile boolean finished;
...
while (!finished)
  doSomething();
```

Variáveis volatile

- volatile é considerado para as garantias do JMM:
  - uma escrita num volatile está relacionada na happens-before com uma leitura subsequente da mesma variável;
  - leituras e escritas num volatile são totalmente ordenadas.
- A vantagem do volatile, face a adquirir locks é o muito baixo custo, pouco mais do que um acesso a uma variável “normal”.
- São usados volatile quando:
  - a escrita não depende de uma leitura, ou apenas uma thread escreve na variável;
  - não estão envolvidos invariantes envolvendo mais do que uma variável;

Variáveis final

- Variáveis final são iniciadas e não podem ser modificadas em seguida.
- O JMM oferece algumas garantias de visibilidade com final, mesmo na presença de races.
- Uma thread que só adquira uma referência para um objecto depois do construtor terminar, vê garantidamente os valores iniciados dos membros final do objecto.
- Vê também iniciados os objectos referenciados pelos membros final.
- A garantia inclui o caso de a referência ser transmitida sem sincronização, numa race.
- Deve ser evitado publicar uma referência antes do construtor terminar; e.g. via variável global.

Exemplo com variáveis final

```
class C {
  final int f;
  int nf;
  static C c;
  public C() {
    f = 1;
    nf = 2;
  }
  static void writer() {
    c = new C();
  }
  static void reader() {
    if (c != null) {
      int i = c.f; // ve sempre 1
      int j = c.nf; // pode ver 2 ou 0
    }
  }
}
```

- Uma thread invoca writer e outra reader, sem sincronização;
- o membro final f é sempre visto como iniciado;
- o membro não final nf pode ser visto não iniciado;

Objectos imutáveis

- Um caso particular importante são os objectos imutáveis.
- Um objecto é imutável se:
  - todos os membros forem final;
  - os objectos referenciados não puderem ser modificados;
  - a sua referência não escapa durante a construção.
- Objectos imutáveis podem ser partilhados sem sincronização.
- O seu estado será visto correctamente, mesmo que publicados sem sincronização.