Verificarea programelor concurente

28 noiembrie 2013

Probleme asociate programelor concurente

Blocaj (deadlock)

Livelock (ciclare care nu produce progres util)

Starvation: inechitate în accesul la resurse
(fire de execuție care nu primesc acces; fără blocaj pe ansamblu)

Condiții de cursă (race conditions)
în particular: privind valorile datelor (data races)

Atomicitate nerespectată
instrucțiuni sursă simple (++) pot fi neatomice în cod mașină
sau: atribuiri pe dimensiuni de mai multe cuvinte de memorie

Condiții de cursă pentru date (data races)

Se întâmplă când două fire de execuție accesează o variabilă și
- cel puțin un acces e de scriere
- cele două fire nu folosesc vreun mecanism explicit de sincronizare

Analiza condițiilor de cursă e complicată de *reordonări într-un fir de execuție* (prin optimizări de compilator)

```
init: x = 0; y = 0;
t1: r1 = x;     t2: r2 = y;
    y = 2;          x = 1;
```

Final (r1, r2) posibil: (0, 0), (1, 0), (0, 2)

Dar prin reordonare în t1 sau t2 putem obține și r1 = 1, r2 = 2 !

Acest rezultat nu corespunde modelului de *consistență secvențială* (cu care suntem obișnuiți intuitiv):
- toate accesele la memorie corespund unei *ordini totale*, și
- ordinea acceselor din orice fir e *ordinea din program*.

Modelul de memorie în Java

Un limbaj concurent trebuie să aibă un model de memorie *intuitiv*, și care să *nu limiteze performanța*, restricționând optimizările.

Soluția [JSR 133; Manson, Pugh, Adve, PLDI'05]:
definirea unei clase de programe care sunt bine sincronizate (*data race free*), pentru care se asigură *consistența secvențială*
+ garanții minimale pentru restul programelor (chiar dacă sunt incorect sincronizate)

Principiul: definirea unei relații de ordine *happens-before* [Lamport] între acțiunile programului, provenind tranzitiv din:
ordonarea acțiunilor de sincronizare (între unlock și orice lock pe același monitor și intre o scriere de variabilă volatilă și citirea ei)
și *ordinea din program* (în cadrul firelor de execuție)

Volatile și sincronizarea

Citire variabilă *volatile*:

ultima valoare scrisă în ordinea de sincronizare.

Citirea unei variabile non-volatile:

orice valoare care *nu e scrisă ulterior* în *happens-before* și nu e perimată de altă scriere

Volatile și sincronizarea

Citire variabilă *volatile*:
ultima valoare scrisă în ordinea de sincronizare.
Citirea unei variabile non-volatile:
orice valoare care *nu e scrisă ulterior* în *happens-before*
și nu e perimată de altă scriere

Atenție: *volatile* NU înseamnă *atomic* !

Condiție de cursă =
accese conflictuale (r-w, w-w) neordonate prin *happens-before*.

Program bine sincronizat = nu are condiții de cursă.

De ce sunt greu de verificat programele concurente?

Înțelegerea problemelor de concurență e adesea dificilă

E dificil de exercitat o anume secvență de execuție
necesită controlul/modificarea planificatorului/condițiilor externe

Secvențele de eroare pot fi foarte rare (în situații complexe anume)

Condițiile de eroare sunt dificil de reprodus ("Heisenbugs")

Explorarea exhaustivă a secvențelor de execuție e nefezabilă
(exponențial în numărul firelor de execuție / dimensiunea lor)

Tipare de erori în programe concurente

[după Farchi, Nir, Ur – IBM Haifa]

Ignorarea non-atomicității

`x = 0` || `x = 0x101` $\Rightarrow$ posibil `x == 1` (!!)

dacă cei doi octeți sunt scriși separat (hi din 0, low din 0x101)

Acces în două etape

chiar dacă sunt protejate, obiectul se poate schimba între...

```
lock(); idx = table.find(key); unlock();
if (...)  { lock(); table[idx] = newval; unlock(); }
```

Lacăt omis / greșit (un nou programator a uitat...)

```
t1: synchronized(o1) { n++; }     t2: n++;
```

sau

```
t1: synchronized(o1) { n++; }     t1: synchronized(o2)
{ n++; }
```

Tipare de erori în programe concurente (cont.)

Double-checked locking: "optimizarea" inițializării la cerere

```
class Foo {
  private Helper helper = null;
  public Helper getHelper() {  // evita unele sincronizari
    if (helper == null)        // deja alocat ? se returneaza
      synchronized(this) { // sincronizat cand e alocat
        if (helper == null) // a doua verificare e protejata
          helper = new Helper();
      }
    return helper; // alt fir va vedea obiect creat incomplet
  }
}
```

Problema: compilatorul e liber să facă reordonări pentru optimizare

Tipare de erori în programe concurente (cont.)

Situații presupuse imposibile (dar care se întâmplă):

`sleep()` folosit greșit pentru a *garanta* o întârziere

Lost Notify: se pierde când e executat *înainte* de `wait`:

t1: `synchronized(o) { o.wait(); }` || t2: `synchronized(o) { o.notifyAll(); }`

Wait neverificat: la revenire trebuie verificată condiția așteptată (revenirea se putea întâmpla și din alte cauze)

Situații de blocare

Cod scris presupunând că secțiunea critică nu se blochează

fals, dacă codul e furnizat (eronat) de altcineva...

Fire de execuție "orfane"

dacă firul care le-a creat se termină eronat $\Rightarrow$ poate duce la blocare

Soluții pentru testarea de modul (unit testing)

Implicit, `JUnit` observă firul care a lansat execuția testului
$\Rightarrow$ nu detectează excepții în fire de execuție create ulterior

Soluție: ConcJUnit [Rice University]
creează/observă un *grup* de fire de execuție
avertizează dacă fire create mai rulează după încheierea firului principal (ar fi trebuit așteptate cu un `join` ...)
poate insera întârzieri arbitrare $\Rightarrow$ generează alte secvențieri

Soluții: testare la nivel de sistem

Ideea: crearea de variație în planificarea firelor de execuție

ConTest [IBM Haifa]
- instrumentează programul (`sleep()`, `yield()`, etc.)
- sau simulează întârzieri/pierderi de mesaje
  - $\Rightarrow$ variație aleatoare în planificare
- măsoară acoperirea raportată la toate planificările posibile

CHESS [Microsoft Research]
- captează funcții de sincronizare
- generează *sistematic* execuții cu planificări noi
  - în ordine crescătoare a nr. de întreruperi (preemptions)
- poate *reproduce* execuțiile generate

Eraser: detectarea condițiilor de cursă

Combină analiza dinamică și statică
- prin analiza unei execuții determină erori potențiale în altele
- reține lacătele deținute de fiecare fir de execuție
- încearcă să deducă ce lacăt protejează fiecare obiect partajat

init: $C(v) = all\_locks$; // pentru fiecare variabilă v
access: $C(v) = C(v) \cap locks\_held(t)$; // la acces în firul t
if $C(v) = \emptyset$ warning(); // acces neprotejat!

Extins, poate distinge între lacăte pentru citire și scriere, urmărind starea fiecărei variabile (virgin, exclusive, shared, shared-modified)

Algoritm conservator, poate da alarme false pentru programe corecte (care nu asociază o variabilă cu un lacăt unic pe întreaga execuție)

High-level data races

[Artho, Havelund, Biere 2003]
Erori: când *granularitatea* variabilelor protejate variază:

```
void swap() {
  int lx, ly;
  synchronized(this) {
    lx = this.x;
    ly = this.y;
  }
  synchronized(this) {
    this.x = ly;
    this.y = lx;
  }
}
```

```
void reset() {
  synchronized(this) {
    this.x = 0;
  }
  synchronized(this) {
    this.y = 0;
  }
}
```

Accesul la membri e sincronizat, dar `swap` și `reset` pot să interfereze!
$\Rightarrow$ Analiză nu doar dpdv al variabilelor (ce lacăte le protejează) dar și dpdv al lacătelor (ce mulțimi de variabile acoperă fiecare)

Java PathFinder [NASA]: verificare prin model checking

Un sistem de verificare prin explorare completă a programului
simulează nedeterminismul printr-o mașină virtuală proprie
care permite alegerea alternativelor de planificare la fiecare pas
și revenirea la cele neexplorate încă (vezi: backtracking)

Permite verificarea (pornind de la bytecode)
situatiilor de blocaj
condițiilor de excepție
aserțiunilor din cod

Limitat la programe mici (10 kloc): "explozia spațiului stărilor"
dimensiunea stărilor memorate (nr. variabile în program)
numărul de execuț¡ii posibile (exponențial în nr. threads)