cs^4_{wk} nedeljainformatike^4

Uvod u paralelno procesiranje

Miloš Stanojević

Matematička gimnazija
NEDELJA$^4_{\text{INFORMATIKE}}$

26. mart 2018.

Uvod

- Dostignut teoretski maksimum single-thread performansi 2002.
- Rešenje je pronađeno u paralelizmu – uvođenjem više jezgara u jedan procesor
- U obe prethodne arhitekture je postojala (i postoji) mogućnost multi-threadinga
- Od sada pa na dalje posmatraćemo sisteme koji podržavaju multi-threading

Thread i konflikti

- **Thread** predstavlja *kontekst izvršavanja* nekog programa
- **Konflikt** među threadovima ne postoji ukoliko rade nad različitim podacima ili nad disjunktnim celinama istih podataka/resursa
- Konflikt nastaje kada više threadova želi da vrši operacije nad *istim* podskupom resursa

Bagovi

- Čak i kada konflikt postoji *on se ne mora uvek ispoljiti*
- Zbog toga bagove nije lako reprodukovati
- Postoje problemi kao deadlock, livelock i contention
- Debugging multi-thread programa je generalno veoma težak

Thread safety

- U nastavku ćemo smatrati da su programi koje koristimo tačni
- To ne znači da oni neće izazvati konflikte kada se izvršavaju u više threadova
- Zato treba da obezbedimo da oni budu **thread-safe**

Primer arhitekture

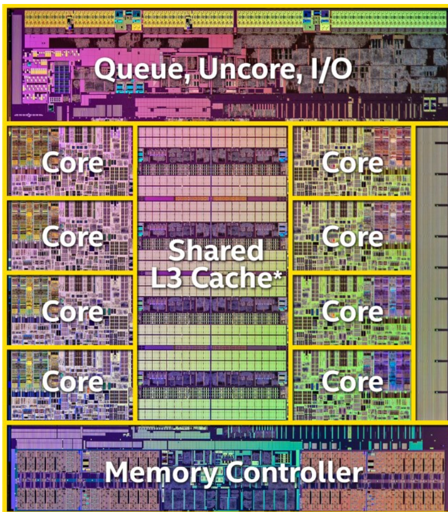
CPU

CORE 1	CORE 2	CORE 8
L1 CACHE	L1 CACHE	L1 CACHE
L2 CACHE	L2 CACHE	L2 CACHE

L3 CACHE

MEMORY

Intel Haswell-E



Primer situacije

- Multi-thread program koji vrši neku komplikovanu simulaciju
- Za potrebe kasnije analize svaka iteracija simulacije se beleži u jedinstven log fajl koji je zajednički za sve threadove
- Samo beleženje se vrši pozivom konkretne funkcije `writeLog(text)`
- Ceo program se izvršava na jednom CPU koji ima efektivno 8 jezgara
- Koji su sve problemi koje moramo da rešimo?
  - Konkurentno pisanje u fajl?

writeLog funkcija

- Najjednostavnija implementacija bila bi:

```
void writeLog(char *text) {
    fprintf(out, "%s", text);
}
```

- Šta je problem ovde?

`writeLog` funkcija – cont'd

- `fprintf` u ovom slučaju predstavlja *critical section* – kod koji samo jedan thread sme da izvršava u nekom trenutku
- Stoga u kodu želimo da onemogućimo da više od jednog threada istovremeno piše u fajl
- Jedno potencijalno rešenje:

```c
void writeLog(char *text) {
    acquireLock(lock);
    fprintf(out, "%s", text);
    releaseLock(lock);
}
```

- Ali kako implementirati ove funkcije?

Prvi pokušaj

```
void acquireLock(bool *lock) {
    while(true) {
        if(*lock == false) {
            *lock = true;
            break;
        }
    }
}

void releaseLock(bool *lock) {
    *lock = false;
}
```

Atomske operacije

- Kod sa prethodnog slajda nije tačan
- Potrebna nam je posebna **atomska** (nedeljiva) mašinska instrukcija!
- Za tu svrhu postoji `compare_and_swap(Destination, Comparand, Exchange)` atomska operacija koja:
  - postavi Exchange na Destination ukoliko je prvobitna vrednost Destination bila jednaka sa Comparand
  - vrati trenutnu vrednost Destination u suprotnom
- Ova instrukcija zavisi od arhitekture:
  - `InterlockedCompareExchange` – MSDN operacija na Windowsu
  - `__sync_val_compare_and_swap` – u slučaju GCC kompajlera na Linux sistemima

Test and set (TAS) lock

```
void acquireLock(bool *lock) {
    while(CAS(lock, false, true)) {
        /* Nothing */
    }
}

void releaseLock(bool *lock) {
    *lock = false;
}
```

Test and test and set (TATAS) lock

```
void acquireLock(bool *lock) {
   do {
      while(*lock) {}
   } while(CAS(lock, false, true));
}

void releaseLock(bool *lock) {
   *lock = false;
}
```

Problemi sa TATAS

- Izaziva takmičenje za isti resurs (cache liniju koja sadrži lock) – blago poboljšanje u odnosu na TAS
- Ne postoji kontorla oko toga koji thread dobija lock (ne postoji jasna 'polisa zaključavanja' kao na primer FCFS)
- Postoji problem *stampeda* – svi threadovi istovremeno pokušaju da dobiju lock (kako ovo rešiti?)

Rešenje stampeda: TATAS

- Možemo koristiti nekakav back-off algoritam:
  - Posmatramo lock s iteracija
  - Ako se lock ne oslobodi, čekamo lokalno w iteracija (*bez posmatranja locka!*)
- Generalno, vrednosti s i w biramo u zavisnosti od problema i to najčešće koristeći ograničeni eksponencijalni back-off (back-off se resetuje kada se lock dobije)

Queue-based lockovi

- Svi threadovi koji žele da dobiju lock se postavljaju u red: odmah dobijamo first come first serve (FCFS) ponašanje
- Svaki thread se vrti *lokalno* na flagu u svom queue entry-ju: nemamo pristup dubljim slojevima memorije dok čekamo
- Oslobađanje (release) locka budi sledeći thread direktno: nemamo stampeda

MCS lock

MCS lock acquire

```
void acquireMCS(mcs *lock, QNode *qn) {
   QNode *prev;
   qn->flag = false;
   qn->next = NULL;
   while(true) {
      prev = lock->tail;
      /* Label 1 */
      if(CAS(&lock->tail, prev, qn)) break;
   }
   if(prev != NULL) {
      prev->next = qn; /* Label 2 */
      while(!qn->flag) { } // Spin
} }
```

MCS lock release

```
void releaseMCS(mcs *lock, QNode *qn) {
   if(lock->tail = qn) {
      if(CAS(&lock->tail, qn, NULL)) return;
   }
   while(qn->next == NULL) { }
   qn->next->flag = true;
}
```

Proširimo problem

- Zamislimo da naš log fajl sada ima i programe koji on-line analiziraju (čitaju) log-fajl
- U ovom slučaju nam nije dovoljno da imamo samo mutex (true/false lock) kao do sada (sem u slučaju jednog čitača)
- Stoga uvodimo koncept Reader-writer lockova – lockova kod kojih omogućavamo da više čitača istovremeno pristupa fajlu

Writer acquire i release

```
void acquireWrite(int *lock) {
   do {
      if((*lock == 0) &&
         (CAS(lock, 0, -1))) {
         break;
   } while(true);
}

void releaseWrite(int *lock) {
   *lock = 0;
}
```

Reader acquire i release

```
void acquireRead(int *lock) {
    do {
        int oldVal = *lock;
        if((oldVal >= 0) &&
            (CAS(lock, oldVal, oldVal+1))) {
            break;
    }} while (true);
}

void releaseRead(int *lock) {
    FADD(lock, -1); // Atomic fetch-and-add
}
```

Drugi vidovi konkurentnog procesiranja

- Hijerarhijski lockovi – uspostavljaju prostorno-lokalan redosled na threadove koji zahtevaju lock (npr. po jezgru na kome se izvršavaju)
- Čitanje bez lockova – u slučajevima gde se malo piše a puno čita, koriste se sheme kao verzioni brojevi (version number schemes)

Dodatni materijal

- "The art of multiprocessor programming", Herlihy & Shavit – jako dobar pregled struktura podataka u deljenoj memoriji, sa praktične i teoretske tačke gledišta
- `http://www.cs.rochester.edu/~scott/papers/1991_TOCS_synch.pdf` – orignialni Mellor-Crummey i Scott rad iz 1991. u kojem uvode MCS lockove

Hvala na pažnji!

Pitanja?