Objektorienteeritud programmeerimine Javas
IDK0051

7. loeng

Paralleeltöö, lõimed

Rain Öpik
rain.opik@ttu.ee
TTÜ Informaatikainstituut

Loengu teemad

- Protsess, lõim
- Jadatöötlus ja paralleeltöötlus
- Protsessoriaja jagamine
- Operatsioonid lõimedega
- Klassid: Thread, Runnable
- Jagatud ressursi kasutus
  - Piiratud ligipääs
  - Lõimede töö sünkroniseerimine
  - Lõimede koostöö
- Lõime olekumudel
- Koodinäited SVNis:

  https://rain.ld.ttu.ee/svn/idk0051/2013-loengud

  pakett: **com.rainopik.joop.loeng07**

Mõisteid

- Tööjärg (*control flow*)
  - Käsud täidetakse üksteise järel
  - Järgmine käsk käivitub, kui eelmine on lõppenud
  - 1 protsessor saab korraga käivitada ühte käsku
- Protsess (*process*)
  - Iseseisev programm - 1 tööjärg
  - Omab mälus isiklikku aadressiruumi
  - Op.süsteem võib käivitada mitu protsessi korraga
    - Protsessoriaja jagamine (*scheduling*)
- Lõim *(thread)*
  - Käivitusüksus e. tööjärg protsessi sees
  - Protsessi siseselt on mälu ja ressurssid lõimede vahel jagatud
  - Igas Java protsessis vähemalt 1 lõim (*main*-thread)
- Mitmelõimeline programm (*multithreaded program*)
  - Protsess, mille sees on mitu tööjärge (e. lõime)

Jada- vs. paralleeltöö

Frog.java

```
run() {
    croak();
    jump();
    swim();
}
```

ConcertSequential.java

```
main() {
    Frog f0 = new Frog();
    Frog f1 = new Frog();
    f0.run();
    f1.run();
}
```

ConcertParallel.java

```
main() {
    Frog f0 = new Frog();
    Frog f1 = new Frog();
    f0.start();
    f1.start();
}
```

Aeg	Tööjärg
0	f0.run();
1	f0.croak();
2	f0.jump();
3	f0.swim();
4	return f0.run()
5	f1.run();
6	f1.croak();
7	f1.jump();
8	f1.swim();
9	return f1.run();
10	return main();

Aeg	Tööjärg		
0	f0.start();	Käivita f0.run();	
1	f1.start();	f0.croak();	Käivita f1.run();
2		f0.jump();	f1.croak();
3		f0.swim();	f1.jump();
4		return f0.run()	f1.swim();
5			return f1.run();
6	return main();		

Paralleeltöö eelised

- Reaktiivne kasutajaliides
  - Nupuvajutus käivitab ajamahuka tegevuse
  - Kasutajaliides reageerib sündmustele ka tegevuse jooksul
- Kõrgem läbilase *(throughput)* e. jõudlus
  - Jagame ülesande sõltumatuteks osadeks
    - Arvutusülesanded, (aeglane) I/O vm.
  - Arvutame iga osa vahetulemuse (saab teha paralleelselt)
  - Liidame osatulemused ja arvutame vastuse

- Ühelõimeline (lõimedeta) programm
  - Ajamahukas tegevuses tuleb perioodiliselt kontrollida kasutajalt saabunud sündmusi
    - Kas vajutas "Close"-nupule? Kas allalaadimine on lõppenud?
- Mitmelõimeline programm
  - Lihtsam loogika reaalaja-simulatsioonides (nt. arvutimängud)
  - Skaleeritavus: rohkem protsessoreid = rohkem tööd korraga tehtud

Lõim
thread

- Realiseerib `Runnable` liidest
  - Runnable = paralleelselt käivitatav tööjärg

```
class Rocket implements Runnable {}
```

- Tööjärg kirjeldatakse `run()`-meetodis
  - Kui `run()` lõpetab töö (*return*), lõpetab ka lõim töö
  - Nimetatakse ka lõime töötsükliks
- Käivitus
  - Pakime oma lõime `Thread`-objekti sisse

```
Rocket r = new Rocket();
Thread t = new Thread(r);
```

  - Käivitamiseks kutsume välja `start()`-meetodi:

```
t.start();
```

  - `start()` paneb lõime "taustal tööle" ja annab tööjärje koheselt järgmisele käsule

Koodinäide: **parallel/Cosmodrome.java**

Tööaja jagamine
scheduling

- Tööaja jagamise strateegiad
  - kooperatiivne koostöö (*cooperative*)
    - lõimed peavad ise CPU tööaja jagamises kokku leppima
    - iga lõim vastutab, et teised ka tööd teha saaks
  - ennetav koostöö (*preemptive*)
    - op.süsteem jagab protsessori tööaega igale lõimele kordamööda
    - garanteerib igale lõimele võimaluse tööd teha
    - lõimel ei ole võimalik protsessorit 100% enda käes hoida
    - kõrgema prioriteediga lõim saab madalamalt tööaja ära võtta
    - alamliik – *time-sharing*
      - iga lõim saab CPU tööajast mingi ajakvandi *(time slice)*
      - ajakvandi lõppedes antakse protsessori juhtimine üle järgmise lõime tööjärjele
- Javas jagab protsessoriaega lõimede planeerija (*thread scheduler*)
  - Lõimede töö on mittedeterministlik
    - igal käivituskorral erinev järjekord
    - sõltub operatsioonisüsteemist, JVM versioonist
  - Java kasutab ennetavat jagatud-ajaga mudelit
- Tööaja üleandmise meetodid:
  - Loobumine `yield()`-meetodiga
    - Vihje planeerijale: osa tööst on tehtud, las teised sama prioriteediga lõimed saavad ka tööd teha

Lõimede loomise ja käivitamise viise

- Thread
  - Klass laiendab Thread-klassi
  - Tööjärje loomiseks alistame run()-meetodi
  - Kõige lihtsam, vahetu ligipääs lõime meetoditele
    - Vahetu käivitus: simple.start();
  - Ei saa kasutada oma pärimishierarhiat (Animal→ Dog → ???)
  - Koodinäide: **classes/SimpleThread.java**
- Runnable
  - Võimaldab mitmest pärimist: säilitame hierarhia ja lisame paralleelkäivituse
  - Realiseerime liideses run()-meetodi
  - Jooksvale lõimele ligipääs läbi Thread.getCurrentThread() meetodi
  - Sageli lisatakse konstruktorisse "käivitusabi" - Thread-objekti loomine
  - Käivitamiseks pakime objekti Thread-objekti sisse
  - Koodinäide: **classes/WithInterface.java**
- Executor
  - Lõime objekti loomine on ressursimahukas, teeme "käivitusabi" taaskasutatavaks
  - Thread-objektide haldamiseks võib kasutada `java.util.concurrent.Executors` klassi
  - `Executors.newCachedThreadPool()` loob dünaamilise suurusega lõimede kogumi
  - `Executors.newFixedThreadPool()` võimaldab lõimede arvu määrata
  - Töö lõpetab `shutdown()` meetod
  - Koodinäide: **classes/LaunchPad.java**

Operatsioonid lõimedega

- Käivituse peatamine - `sleep()`
  - Blokeerib lõime töö *n* millisekundiks
  - Katkestamine võib heita `InterruptedException`-i
  - `java.util.concurrent` paketi klass `TimeUnit` teeb erinevad ajaühikud mugavalt kättesaadavaks:

```
import java.util.concurrent.*;
...
TimeUnit.SECONDS.sleep(5);
```

  - Koodinäide: **parallel/Tickers.java**
- Magamise katkestamine - `interrupt()`
  - Katkestamisel pannakse lõimes püsti vastav lipuke - `isInterrupted()`
  - catch-blokis võetakse lipp uuesti maha
- Teise lõime järel ootamine - `join()`
  - Meetod blokeerib jooksva lõime, kuni teine lõim on töö lõpetanud - tagastus `run()` meetodist)

```
sleeper.join();     // peatab jooksva lõime, kuniks sleeper lõpetab
```

  - Koodinäide: **parallel/Dwarves.java**

Deemonid
daemon threads

- Deemon on lõim, mis pakub taustateenust
  - Teenus, mis ei ole programmi jaoks kriitilise tähtsusega
  - Programmi pealõim (*main-thread*) on mitte-deemon
- Töö lõpetamine
  - Programm lõpetab töö, kui kõik mitte-deemonid on lõpetanud
  - Kõik deemonid lõpetatakse "jõuga" - deemon peab olema valmis ootamatuks e. mitte-viisakaks sulgemiseks
  - Deemon-lõime töötsüklis (`run`) ei toimi `finally`-klausel
- Deemonite loomine
  - Lõim tuleb enne käivitamist "demoniseerida"

```
cacodaemon.setDaemon(true);
cacodaemon.start();
```

  - Iga deemoni sees loodud lõim on samuti deemon

Koodinäide: **parallel/Daemons.java**

Lõimede rakendusi

- Kasutajaliides
  - Vähemalt kaks lõime:
    - Kasutajasündmustele reageerimine
      - Swingis - event dispatch thread
    - Andmetöötluse / ajamahukas töö eraldi lõimes
  - Koodinäide: **applications/UI.java**
- Simulatsioon e. paljude objektidega virtuaalmaailm
  - Iga aktiivse objekti jaoks eraldi lõim
  - Ka elutu objekt võib omada käitumist
    - uksed, vilkuvad lambid jm.
- Tänapäevane klient-server rakendus
  - Iga kliendipäringu töötluseks loob server eraldi lõime
  - Kliendipäring ei pea ootama “järjekorras”
  - Asünkroonne päringu täitmine: klient saadab serverile päringu, millele ta ei jää vastust ootama
  - Täiendavad lõimed:
    - Sõnumite-käskude marsruutimiseks
    - Sõnumite puhverdamiseks, puhvrite tühjendamiseks

Ressursi jagamine

- Ühelõimelises programmis on elu lihtne
  - Üks tegevus e. käsk korraga
- Mitmelõimeline programm = palju samaaegseid tegevusi
  - Kaks inimest üritavad rääkida samal ajal
  - Kaks kandidaati ühele parkimiskohale
  - Kaks inimest astuvad korraga läbi sama ukseaugu
- Mitmelõimelises programmis
  - Üks lõim üritab muutujat suurendada (`i++`), teine vähendada (`i--`)
  - Kaks lõime tahavad korraga samasse printerisse trükkida
  - Üks üritab faili kirjutada, teine seda kustutada
- Mis on ressurss?
  - Mäluaadress (muutuja)
  - Sisend-väljundseade (klaviatuur, konsooliväljund, printer, ekraan)
  - Fail, andmebaasikirje

Ressursi jagamine
väärkasutus

- Kahel lõimel ühine andmeobjekt: `AlwaysEven common`
  - Üks lõim suurendab arvu
  - Teine kontrollib, kas vastus sai ikka paarisarv
  - NB! Lõimede töö paralleelselt

AlwaysEven.java
```
int value = 0;
increment() {
  value++;
  value++;
}
getValue() {
  return value;
}
```

AdderThread.java
```
run() {
  common.increment();
}
```

WatcherThread.java
```
run() {
  if (common.getValue()%2 != 0) {
    // paanika paanika
  }
}
```

- Paarisarvu suurendamine koosneb kahest tehtest
- Probleem: vaatlejal õnnestub kätte saada (vigane) vahetulemus

Koodinäide: **shared/Shared.java**

Ressursi jagamine
mis läks valesti?

Naiivne eeldus - loodame, et jagatud ressurssi kasutatakse korda-mööda

Aeg	AdderThread adder	WatcherThread watcher	AlwaysEven common	value
0	common.increment();	*ootab tööjärge*	void increment() {	0
1			value++	1
2			value++	2
3			return increment();}	
4		value = common.getValue();	int getValue() {	
5	*ootab tööjärge*	// sai väärtuseks 2	return value; }	2
6	common.increment();	*ootab tööjärge*	void increment() {	2
7			value++	3
8			value++	4
9			return increment();}	

Kurb tegelikkus - kaks lõime võivad ühise objekti kallal <u>samaaegselt</u> toimetada

sama objekt *common*

Aeg	AdderThread	AlwaysEven	WatcherThread	AlwaysEven	value
0	common.increment();	void increment() {	value = common.getValue();	int getValue() {	0
1		value++	// sai väärtuseks 1	return value; }	1
2		value++			2
3		return increment();}	*ootab tööjärge*		

Ressursi jagamine
jadastatud pöördumine

- Meeldetuletus: lõimede töö ei ole deterministlik
  - Vaikimisi puudub garanteeritud pöördumiste järjekord
- Lahendus: jagatud ressursi poole tuleb pöörduda korda-mööda
  - Pöördumised jadastatakse *(serialized access)*
- Vastastikkuse välistamise printsiip (*mutual exclusion* e. *mutex*)
  - Kui üks lõim soovib objektiga töötada, siis ta lukustab objekti
  - Objekti saab kasutada ainult see, kes ta lukustas
  - Teised lõimed on ootel e. blokeeritud, kuniks esimene lõpetab
  - Kui esimene lõim lõpetab, siis ta vabastab objekti lukust
  - Järgmine lõim võib objekti enda kätte võtta (lukustada)
- *Mutex* on nagu vannituba hommikusel ajal – ainult üks inimene korraga

Koodinäide: **shared/OnePersonOnly.java**

Ressursi jagamine
lukustamine

- Java mehhanism objekti lukustamiseks - sünkroniseerimine
  - Koodiblokk kaitstakse `synchronized` võtmesõnaga
  - Iga objekt omab monitori e. lukku
  - Kui lõim siseneb sünkroniseeritud koodiblokki
    - Kontollitakse, kas objekt on saadaval / lukus
    - Objekt lukustatakse
  - Kui lõim väljub sünkroniseeritud blokist, vabastatakse lukk
- Sünkroniseeritud blokis saab korraga viibida ainult üks lõim
- Hästi disainitud klassis:
  - Kõik andmeväljad on private-ligipääsuga
  - Ligipääs ainult meetodite vahendusel
  - Ühiskasutuse jaoks sünkroniseeritakse kõik ligipääsud (e. meetodid)
- Sünkroniseerida saab:
  - Meetodit – lukustab objekti enda (e. `this`-i)

```
public synchronized void increment()
```

  - Mingit teist etteantud objekti (sh. lõime ennast):

```
synchronized (common) {
        // kasuta jagatud objekti südamerahus
```

Koodinäide: **shared/Synch.java, shared/SynchBlock.java**

Sünkroniseerimine
üks lõim korraga

Aeg	SynchAdderThread adder	SynchWatcherThread watcher	AlwaysEvenSynch common	common value	common monitor
0	common.increment();	*ootab monitori vabanemist*	`synchronized void increment() {` *hõiva objekti monitor*	0	*(vaba)* -> *adder*
1			`value++`	1	*adder*
2			`value++`	2	*adder*
3			`return increment();` *vabasta objekti monitor* `}`	2	*adder* -> *(vaba)*
4	*ootab monitori vabanemist*	`value = common.getValue();`	`synchronized int getValue() {` *hõiva objekti monitor*	2	*(vaba)* -> *watcher*
5		`// sai väärtuseks 2`	`return value;` *vabasta objekti monitor* `}`	2	*watcher* -> *(vaba)*
6	common.increment();	*ootab monitori vabanemist*	`synchronized void increment() {` *hõiva objekti monitor*	2	*(vaba)* -> *adder*
7			`value++`	3	*adder*
8			`value++`	4	*adder*
9			`return increment();` *vabasta objekti monitor* `}`	4	*adder* -> *(vaba)*

Sünkroniseerimine
ei taga determineeritud käitumist

Aeg	SynchAdderThread adder	SynchWatcherThread watcher	AlwaysEvenSynch common	common value	common monitor
0	*ootab monitori vabanemist*	`value = common.getValue();`	`synchronized int getValue() {` *hõiva objekti monitor*	0	*(vaba)* -> *watcher*
1		`// sai väärtuseks 0`	`return value;` *vabasta objekti monitor* `}`	0	*watcher* -> *(vaba)*
2	*ootab monitori vabanemist*	`value = common.getValue();`	`synchronized int getValue() {` *hõiva objekti monitor*	0	*(vaba)* -> *watcher*
3		`// sai väärtuseks 0`	`return value;` *vabasta objekti monitor* `}`	0	*watcher* -> *(vaba)*
4	`common.increment();`	*ootab monitori vabanemist*	`synchronized void increment() {` *hõiva objekti monitor*	0	*(vaba)* -> *adder*
5			`value++`	1	*adder*
6			`value++`	2	*adder*
7			`return increment();` *vabasta objekti monitor* `}`	2	*adder* -> *(vaba)*
8	`common.increment();`	*ootab monitori vabanemist*	`synchronized void increment() {` *hõiva objekti monitor*	2	*(vaba)* -> *adder*
9			`value++`	3	*adder*
10			`value++`	4	*adder*
11			`return increment();` *vabasta objekti monitor* `}`	4	*adder* -> *(vaba)*

Ressursi jagamine
lukustus

- Luku operatsioonid:
  - saadavuse kontroll (*isAvailable*)
  - lukustamine (*acquire*)
  - *vabastamine* (*release*)
- Erinevad lukustamisviisid
  - Semafor (*semaphore*)
    - Võimaldab *n* lõime limiteeritud samaaegselt ligipääsu
    - Omab sisemist loendurit
    - Iga lukustamine vähendab loendurit
    - Iga vabastamine suurendab
    - Ligipääs lubatakse siis, kui loendur > 0
    - Realiseeritud `java.util.concurrent.Semaphore` klassis
  - Vajadusel saab lukke ka käsitsi luua
    - Klass `java.util.concurrent.Lock`
    - lukustamine: `myLock.lock();`
    - vabastamine: `myLock.unlock();`

Ressursi jagamine
vahekokkuvõte

- Tehing e. transaktsioon *(transaction)*
  - **A**tomicity e. jagamatus
    - tehingus sooritatakse kas kõik operatsioonid või mitte ühtegi
    - tehingu väljakutsuja ei pea muretsema, et see katkeb poole peal
  - **C**onsistency e. kooskõlalisus
    - pärast iga tehingut on süsteem kooskõlalises seisundis
  - **I**solation e. isolatsioon
    - teised tehingud ei näe pooleli oleva tehingu vahetulemusi
    - vajadusel peab tehing ootama, et teine tehing oma töö lõpetaks
  - **D**urability e. püsivus
    - tehingu tulemus on püsiv - kui süsteemiga midagi juhtub, siis selle seisund on taastatav
- Programmi terviklik seisund e. olek koosneb:
  - programmi muutmälu ← JOOP
    - muutujad e. objektid ja väljad
    - kasutajaliides ja ekraanikuva
  - programmi püsimälu
    - failid
    - andmebaasis talletatud kirjed ← Andmebaasid II
  - väliste programmide seisund (süsteem) ← Rakendusarhitektuurid

Kokkuvõte

- Paralleeltöö on keeruline
  - Programmeerimine → OOP → Paralleeltöö programmeerimine
  - Lihtsal testimisel ei pruugi vead avalduda
  - Intuitsiooni paralleeltöö puhul usaldada ei saa
- Iga jagatud ressurssi poole pöördumine tuleb sünkroniseerida
- Lukustuse toimimiseks
  - Kõik ligipääsud sünkroniseerida
  - Loome isetehtud lukumehhanismi (wait/notify)
- Lõimedest ei ole pääsu
  - GUI, veebirakendused (nt. *servlet*), serveriprogrammid ja enamus mobiilirakendusi on olemuselt mitmelõimelised
    - "Tahan, et programm toimetaks taustal..." → taustatöö lõim

Kodune töö

- Läbi lugeda *Thinking in Javas*t:
  - 13. peatükk - Concurrency

Viited

Kasutatud materjalid

[1] Bruce Eckel - Thinking in Java 3rd ed.

http://rain.ld.ttu.ee/idk0051/tij

Lisalugemist

Sun Thread tutorial

http://docs.oracle.com/javase/tutorial/essential/concurrency

Exploring Java, P. Niemeyer, J.Peck. Chapter 6 – Threads

http://oreilly.com/catalog/expjava/excerpt/index.html