UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Egon Kocjan

Vzporedno podatkovno pretokovno programiranje v jeziku C++

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: doc. dr. Boštjan Slivnik

Ljubljana, 2009

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01582/2009
Datum: 01.09.2009

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **EGON KOCJAN**

Naslov: **VZPOREDNO PODATKOVNO PRETOKOVNO PROGRAMIRANJE V JEZIKU C++**

PARALLEL DATAFLOW PROGRAMMING IN C++

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Zasnujte in realizirajte knjižnico za realizacijo vzporednih podatkovno pretokovnih algoritmov v programskem jeziku C++. Natančno opišite model, na katerem temelji knjižnica, in na kratko podajte pregled najpomembnejših obstoječih sorodnih rešitev. Uporabo izdelane knjižnice prikažite na za realiziran model ustreznem primeru.

Mentor:
B. Slivnik
doc. dr. Boštjan Slivnik

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:
Franc Solina
prof. dr. Franc Solina

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01582/2009
Datum: 01.09.2009

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **EGON KOCJAN**

Naslov: **VZPOREDNO PODATKOVNO PRETOKOVNO PROGRAMIRANJE V JEZIKU C++**

PARALLEL DATAFLOW PROGRAMMING IN C++

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Zasnujte in realizirajte knjižnico za realizacijo vzporednih podatkovno pretokovnih algoritmov v programskem jeziku C++. Natančno opišite model, na katerem temelji knjižnica, in na kratko podajte pregled najpomembnejših obstoječih sorodnih rešitev. Uporabo izdelane knjižnice prikažite na za realiziran model ustreznem primeru.

Mentor:
doc. dr. Boštjan Slivnik

Fakulteta za računalništvo in informatiko
2
UNIVERZA V LJUBLJANI

Dekan:
prof. dr. Franc Solina

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani Egon Kocjan,

z vpisno številko 63000180,

sem avtor diplomskega dela z naslovom:

Vzporedno podatkovno pretokovno programiranje v jeziku C++

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom
  doc. dr. Boštjana Slivnika
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 30.11.2009 Podpis avtorja:

Zahvala

Najprej bi se zahvalil staršema, ki sta mi ves čas stala ob strani in mi omogočila študij. Zahvaljujem se mentorju doc. dr. Boštjanu Slivniku za pomoč pri izdelavi diplomskega dela. Hvaležen sem tudi vsem sodelavcem v družbi XLAB d. o. o. Skupaj smo rešili veliko tehničnih in poslovnih problemov, pri čemer sem pridobil izjemne praktične izkušnje. Posebna zahvala pa gre ženi Jani za podporo in lektoriranje.

Kazalo

Seznam uporabljenih kratic in simbolov

CCR	(Microsoft) concurrency and coordination runtime
CPE	centralna procesna enota
FIFO	first in, first out
IP	internet protocol
RSA	kriptografski algoritem Rivest, Shamir in Adleman
SSL	secure sockets layer
TBB	(Intel) threading building blocks
TCP	transmission control protocol
V/I	vhod/izhod ali vhodno-izhodno

Povzetek

Diplomsko delo vsebuje opis knjižnice LibY v programskem jeziku C++, ki je namenjena vzporednemu podatkovno pretokovnemu programiranju. Knjižnica rešuje težave pri razvoju programske opreme, ki mora učinkovito izkoriščati moderne večjedrne procesorje, pošiljati večjo količino podatkov prek nezanesljivih internetnih povezav in komunicirati z velikim številom odjemalcev. Pristop k reševanju problemov vključuje večletne izkušnje s področja razvoja komunikacijskih uporabniških aplikacij in velikih strežniških sistemov z 10 000 odjemalci na posamezen strežniku. Navedene so primerjave knjižnice LibY z nekaterimi obstoječimi rešitvami s tega področja. Opisane so osnovne prvine knjižnice in prednosti, ki neposredno izhajajo iz uporabe prvin. Med poglavitne prednosti lahko štejemo preprečevanje smrtnih objemov, optimalno število sistemskih izvajalnih niti, enoten model izvajanja za procesorsko zahtevna opravila in vhodno-izhodna opravila ter učinkovito preslikavo prvin LibY na prvine operacijskega sistema. Prikazana sta tudi dva pristopa k zaklepanju skupnih virov, ki ohranjata osnovno prednost preprečevanja smrtnih objemov, tj. aktivni objekti kot preprosta in učinkovita metoda ter splošno zaklepanje z globalno urejenostjo zaklepanja. V zadnjem delu je predstavljen primer simulacije večpredstavnega strežnika za videokonference. Z analizo strežnika so ugotovljene številne prednosti uporabe knjižnice LibY in tudi nekatere slabosti. Prav tako je opisan razmislek o prihodnjem razvoju knjižnice, ki se opira predvsem na naprednejše izvajalce in razporejevalce opravil ter učinkovitejšo in varnejšo implementacijo prvin LibY.

Ključne besede:

vzporedno programiranje, podatkovno pretokovno programiranje, C++, preprečevanje smrtnih objemov, enoten model opravil, prihodnosti

Abstract

This dissertation describes the parallel dataflow programming library LibY written in the C++ programming language. The library solves software development problems most often encountered in relation to efficient multi-core processor support, extensive data transmission over unreliable Internet connections and communication with a large number of clients. The solutions to the presented problems were strongly influenced by many years of experience in building communication applications for end users and large server systems with 10 000 clients per each server. A comparison is made between LibY and the existing solutions to problems in the mentioned field. Basic library primitives are described along with the direct benefits of their use. The most important improvements in LibY are deadlock prevention, optimal count of system execution threads, unified model of processor intensive and input/output task execution, and efficient mapping of the LibY basic primitives into the operating system primitives. Moreover, two methods of shared resource locking are described. Both methods retain the basic beneficial property of LibY, i.e. deadlock prevention. Active objects are a simple and efficient method of locking, whereas general locking is implemented with a global lock ordering method. The last part covers the simulation of a multimedia videoconferencing server. Numerous benefits along with some disadvantages of LibY usage have been recognised by analysing the server. More sophisticated executors, task schedulers and safer implementation of the basic primitives are identified as the basis for further development of LibY.

Keywords:

parallel programming, dataflow programming, C++, deadlock prevention, unified task model, futures

Poglavje 1

Uvod

Današnji večjedrni procesorji za uporabo v osebnih računalnikih so nastali zaradi težav pri izdelavi čipov. V letih od 2003 do 2005 se je neprestano višanje notranjih frekvenc procesorjev, ki jih proizvaja Intel, skoraj ustavilo, kar se od samih začetkov proizvodnje mikroprocesorjev v sedemdesetih letih prejšnjega stoletja do sedaj še ni zgodilo. Višanje števila tranzistorjev na procesor pa se ni ustavilo in še sledi Moorovem „zakonu“, ki pravi, da se število tranzistorjev podvoji vsaki dve leti. Dodatni tranzistorji se zdaj uporabljajo za večanje števila jeder in vgrajenih predpomnilnikov.

Klasičen pristop k programiranju večjedrnih procesorjev je uporaba prvin kot so muteksi, semaforji, niti itd. Vse te prvine so zelo nizkonivojske, neposredna uporaba pa povzroča vrsto težav:

- smrtni objemi,
- nemodularnost zaklepanja skupnih virov,
- privzeto nedeterministično izvajanje,
- ločenost prvin od samih podatkov in dejanskega namena programa.

Zaradi naštetih težav naj bi se klasične prvine uporabljalo predvsem v jedrih operacijskih sistemov, kjer sta izjemno pomembni majhna poraba sistemskih virov in hitrost. Za trenutno razširjene programske jezike, namenjene uporabniškim aplikacijam, pa so na voljo številne knjižnice, ki bistveno olajšajo večnitno programiranje. V raziskovalni skupnosti je bila razvita nepregledna množica programskih jezikov, ki so namenjeni vzporednemu programiranju in imajo veliko koristnih lastnosti, vendar izboljšave v bolj razširjene jezike žal pritekajo zelo počasi.

Predmet tega diplomskega dela je knjižnica LibY za vzporedno podatkovno pretokovno programiranje v programskem jeziku C++. Prvi cilj je odprava klasičnih težav večnitnega programiranja z naslednjimi ukrepi:

- odprava smrtnih objemov v največji možni meri,
- modularnost zaklepanja skupnih virov (če program A in program B nimata smrtnih objemov, potem tudi program $A + B$ ne bo imel smrtnih objemov),
- uvedba determinizma v izvajanju programa na mesta, ki bi lahko povzročila dvoumnosti,
- podatki in s tem namen programa so postavljeni v ospredje pri podatkovno pretokovnem programiranju.

Zaradi porasta interneta in spletnih aplikacij v zadnjem desetletju se pri razvoju programske opreme le redko izognemo mrežni komunikaciji. Internet pogosto prinaša preveč prednosti, da bi ga lahko ignorirali. Tudi takemu klasičnemu programskemu paketu, kot je Autodesk Autocad, so v zadnjem času dodali podporo za skupinsko delo prek interneta. Razvoj aplikacij, ki temeljijo na internetni komunikaciji, ponuja pisan nabor izzivov:

- napake pri prenosih so razmeroma pogoste, zlasti v zadnjem času zaradi porasta brezžičnih in mobilnih tehnologij prenosa podatkov;
- programske napake v notranjih vozliščih interneta niso redke, določen podatkovni vzorec lahko sesuje usmerjevalnik ali požarni zid;
- večje družbe, finančne ali državne institucije imajo običajno zelo natančno določeno varnostno politiko, prenos podatkov se mora odvijati z veliko omejitvami, ki pa žal niso standardizirane;
- število odjemalcev pri strežniških sistemih lahko zelo niha, v primeru uspešne tržne promocije lahko število odjemalcev izjemno naraste in povsem zasiči sistem;
- pri večjih porazdeljenih sistemih pogosto opazimo posledice tehničnih težav na hrbteničnih povezavah v internetu, včasih jih sprožijo tudi poslovni spori med ponudniki telekomunikacijskih storitev.

V luči izzivov programiranja internetnih aplikacij lahko opazimo drugi cilj knjižnice LibY, tj. dati razvijalcu močna orodja za programiranje vhodno-izhodnih opravil:

- izenačitev procesorsko zahtevnih opravil z V/I opravili (s stališča uporabnika knjižnice LibY so procesor, trdi disk in mrežna povezava naprave, ki sprejmejo podatek, ga obdelajo in vrnejo ali posredujejo naprej),
- razvijalcu preprosta in učinkovita preslikava V/I opravil na prvine operacijskega sistema z najmanjšo možno porabo virov,
- podatkovni pretoki v knjižnici LibY so konceptualno blizu dejanski izvedbi podatkovnih prenosov v strojni opremi in na internetu, kar omogoča učinkovito implementacijo same knjižnice.

V naslednjem poglavju bodo opisane sorodne programske rešitve in podane primerjave s knjižnico LibY. Nadaljuje se z opisom osnovnih prvin knjižnice LibY, tj. prihodnosti, izvajalci in opravila, vzporedni klici in lovljenje izjem vzporednih klicev. Te prvine sestavljajo jedro knjižnice in pokazano bo, da večina prednosti uporabe LibY izhaja iz teh prvin. Opisana bodo tudi nekatera dodatna uporabna orodja in razširitev osnovnega modela izvajanja za modularno zaklepanje skupnih virov. Zaključek vsebuje večji praktičen primer večpredstavnega strežnika, ki simulira pretvorbo video podatkovnih tokov v videokonferenčnih sejah.

Poglavje 2

Sorodne rešitve

2.1 Intel Threading Building Blocks

Intel TBB [10] je knjižnica za programski jezik C++ in je namenjena izkoriščanju večjedrnih procesorjev. Vsebuje višjenivojske konstrukte, ki odpravijo težavno neposredno uporabo niti, in sicer:

- algoritme:
  - algoritme za deljenje podatkov z namenom vzporedne obdelave,
  - vzporedne različice najbolj pogostih algoritmov (`parallel_for`, `parallel_reduce`, `parallel_sort` itd.),
  - cevovod z uporabo filtrov;
- različice pogostih podatkovnih struktur, ki so namenjene hkratni uporabi iz več niti (`concurrent_hash_map`, `concurrent_queue`, `concurrent_vector` itd.),
- dodeljevalce pomnilnika, ki so zasnovani za učinkovito uporabo iz več niti in zagotavljajo pravilno poravnavo podatkovnih struktur v predpomnilnikih,
- nadzor hkratnega dostopa do virov:
  - različne izvedbe muteksov,
  - najpogostejše funkcije za uporabo atomičnih spremenljivk;
- globalen (velja za vsa procesorska jedra) in natančen časovni števec,

- skupine opravil, na katere je možno čakati ali jih prekiniti,
- razporejevalnik opravil, ki uporablja algoritem „kraja dela“ (work-stealing)
- prenos izjem med nitmi,
- prenosljiv programski vmesnik niti, ki je implementiran kot ovojnica za sistemski programski vmesnik niti.

Intel TBB je značilen predstavnik knjižnic in razširitev programskih jezikov, ki so namenjene vzporednemu izvajanju računsko zahtevnih opravil na večjedrnih procesorjih. Vredno je omeniti še OpenMP [15] in osnutek naslednjega standarda C++ [13]. Skupna težava teh rešitev je neobravnavanje vhodno-izhodnih opravil, ki so zelo pogosta v današnjih aplikacijah, namenjenih za uporabo v internetu.

2.2 Paket java.util.concurrent v platformi Java 5

Paket [12] vsebuje nekaj višjenivojskih razredov, ki olajšajo večnitno programiranje v okolju Java, in sicer:

- nekatere izvedbe izvajalcev opravil (`ThreadPoolExecutor`, `ScheduledThreadPoolExecutor`),
- različice pogostih podatkovnih struktur, ki so namenjene hkratni uporabi iz več niti (`ConcurrentHashMap<K,V>`, `ConcurrentLinkedQueue<E>` itd.),
- podatkovne strukture, ki so namenjene usklajevanju izvajanja niti (`DelayQueue<E extends Delayed>`, `Exchanger<V>` itd.),
- predstavitev opravil, ki so ločena od same izvajalne niti (`Callable<V>`),
- predstavitev rezultata izračuna, ki se izvaja v drugi niti (`Future<V>`).

Programski vmesniki v paketu java.util.concurrent omogočajo mešanje računsko zahtevnih in vhodno-izhodnih opravil, saj lahko uporabimo razreda `Callable<V>` in `Future<V>` za obe vrsti opravil. Žal pa je razred `Future<V>` zasnovan z uporabo metode `get`, ki ustavi izvajanje trenutne niti, kar povzroča več težav:

- nevarnost smrtnih objemov,
- večjo porabo pomnilnika,
- oteženo učinkovito porazdeljevanje programskih niti na strojne niti.

2.3 Microsoft Concurrency and Coordination Runtime

Microsoft CCR [4] je knjižnica, ki temelji na programskem okolju .NET. Namenjena je programiranju prenosa podatkov med različnimi podsistemi v računalniških programih, kot so grafični vmesnik, datotečni sistem in omrežna komunikacija. Hkrati podpira tudi vzporedno izvajanje računsko zahtevnih opravil ter enotno obravnavanje napak vseh vrst opravil. Microsoft CCR vsebuje:

- asinhrono čakalno vrsto, ki ne temelji na čakajočih nitih (`Port<T>`, `PortSet<T1 [, T2, ...]>`),
- nabor razredov, ki predajo sporočila iz čakalnih vrst v nadaljnjo obdelavo (`Task`, `Choice`, `Receiver`, `Interleave` itd.),
- izvajalne čakalne vrste, ki določijo način predaje opravil v izvajanje (`Dispatcher`, `DispatcherQueue` itd.),
- izvajalce (`TaskExecutionWorker` itd.),
- obravnavanje napak z uporabo čakalnih vrst, kamor se sporočajo izjeme, ali neposredno prenašanje napak skupaj z rezultati.

Zasnova knjižnice Microsoft CCR je razmeroma podobna knjižnici LibY, vendar se razlikuje v bistveni podrobnosti – CCR temelji na uporabi čakalnih vrst, ki je konstrukt, s katerim se preprosto povzroči smrtni objem. Klasičen primer smrtnih objemov pri uporabi čakalnih vrst je programiranje koprocesov z uporabo sistemskih cevi na operacijskem sistemu UNIX [24].

2.4 Ostale rešitve

V raziskovalni dejavnosti obstaja nepregledna množica različnih programskih knjižnic, razširitev za obstoječe jezike in jezike, namenjene vzporednemu programiranju. V tem odstavku bodo predstavljene nekatere rešitve, ki so vplivale na zasnovo knjižnice LibY.

2.4.1 Io

Io [11] je objektni programski jezik z dinamičnim sistemom tipov, ki za asinhrono izvajanje uporablja pristop z aktivnimi objekti (asinhrona čakalna vrsta za sprejemanje klicev metod, metode pa se izvajajo v korutini z omejitvijo izvajanja ene metode na objekt hkrati). Rezultat asinhronega klica metode aktivnega objekta je prihodnost. Uporaba prihodnosti povzroči ustavitev izvajanja, dokler se vrednost prihodnosti ne izračuna. Za asinhron klic je treba dodati le simbol `@` pred ime metode. Podobno zasnovo uporablja tudi knjižnica LibY, kjer je treba oviti običajni klic funkcije s funkcijo `pcall`. Rezultat funkcije `pcall` je tudi prihodnost.

2.4.2 Erlang

Erlang [6] je funkcijski programski jezik z dinamičnim sistemom tipov, brez stranskih učinkov in sintakso, ki je podobna Prologu. Za vzporedno izvajanje se uporabljajo procesi, ki so aktivni objekti (asinhrona čakalna vrsta za sprejemanje sporočil, sposobnost oddajanja sporočil drugim objektom, obdelava največ enega sporočila hkrati v vsakem procesu). Komercialno uporabo je našel v telekomunikacijski opremi, uporablja pa se tudi v nekaterih odprtokodnih strežniških projektih.

Zanimiv je predvsem način obravnave napak. Procese je mogoče med sabo povezati tako, da dobimo več različnih podsistemov, vsakega s svojo podmnožico procesov. Napaka v enem procesu bo samodejno zaprla vse ostale procese v podsistemu. Prehajanje napak iz podsistema v podsistem in posledično zapiranje procesov se prepreči z mejami, ki ujamejo in obravnavajo napake. Knjižnica LibY uporablja podoben pristop, ki ob napaki preneha z izvajanjem vzporednih klicev. Mejo prehajanja napak se v LibY postavi s funkcijo `pcatch`.

2.4.3 Haskell

Haskell [9] je funkcijski programski jezik s statičnim sistemom tipov, brez stranskih učinkov in z lenim načinom izvajanja. Jezik med drugim omogoča opredelitev neskončno velikih podatkovnih struktur, saj se elementi strukture izračunajo šele takrat, ko so nujno potrebni. Običajno izvajanje programa, ki obdeluje take podatkovne strukture, je naslednje:

1. če je izpolnjen ustavitveni pogoj, se pregledovanje tega dela podatkovne strukture konča,

2. iz podatkovne strukture se vzame naslednji element,

3. uporabi se element,

4. izvajanje se nadaljuje s korakom 1.

Izračun elementa je potreben šele pri točki 2, torej ni treba izračunati vseh elementov vnaprej kot pri običajnih programskih jezikih. Na takem pristopu temelji vzporedno programiranje v LibY, pri čemer se uporablja prihodnosti, ki v času opredelitve prav tako nimajo nujno izračunane vrednosti. Primer zanke, ki je zelo podobna zgoraj opisanemu primeru za jezik Haskell, je podan v odstavku o čakalnih vrstah FIFO v knjižnici LibY.

Poglavje 3

Knjižnica LibY

LibY je programska knjižnica namenjena vzporednemu in porazdeljenemu programiranju. V okviru tega dela pomeni vzporedno programiranje izvajanje programa v več programskih nitih, ki znotraj enega samega procesa tečejo na več strojnih nitih. Porazdeljeno programiranje pomeni izvajanje programa v več ločenih procesih, ki so med sabo običajno povezani s protokoli iz družine TCP/IP in tečejo na enem ali več računalnikih. Knjižnica je bila narejena v programskem jeziku C++; tudi vsi primeri v tem delu bodo v tem jeziku. Sama zasnova knjižnice pa je preprosto prenosljiva med programskimi jeziki, kot sta na primer Common Lisp in Python. Z nekaj več dela bi bilo mogoče LibY prenesti tudi na programski jezik Java, težavo predstavlja zlasti razlika med generiki v Javi in predlogami v C++. Izvedba LibY temelji na dobro znani knjižnici Boost [3], ki vsebuje razrede za prenosljivo programiranje niti, kazalce s štetjem referenc, razširitev koncepta *union* iz programskega jezika C itd.

Jeziku C++ manjka sintaksa za varne funkcijske klice z več parametri, zato bo v primerih uporabljena naslednja psevdokoda za število parametrov 1 in več:

```
// psevdokoda
template <typename X1 ... Xn>
void fn(X1 x1 ... Xn xn)
{
   do(x1);
   ...
   do(xn);
}
```

```
// pomen psevdokode:
template <typename X1>
void fn(X1 x1)
{
   do(x1);
}
template <typename X1, typename X2>
void fn(X0 x1, X1 x2)
{
   do(x1);
   do(x2);
}
template <typename X1, typename X2, typename X3>
void fn(X1 x1, X2 x2, X3 x3)
{
   do(x1);
   do(x2);
   do(x3);
}
// itn.
```

V primerih bo uporabljena naslednja psevdokoda za število parametrov 0 in več:

```
// psevdokoda
template <typename [X1 ... Xn]>
void fn([X1 x1 ... Xn xn])
{
   do(x1);
   ...
   do(xn);
   do2();
}

// pomen psevdokode:
void fn()
{
   do2();
}
template <typename X1>
```

```
void fn(X1 x1)
{
   do(x1);
   do2();
}
template <typename X1, typename X2>
void fn(X0 x1, X1 x2)
{
   do(x1);
   do(x2);
   do2();
}
// itn.
```

Podporo za varne funkcijske klice z več parametri je mogoče izvesti s pomočjo makrojev v preprocesorju ali z generiranjem kode.

3.1 Prvine vzporednega programiranja v LibY

3.1.1 Prihodnosti

Prihodnosti so ovojnice za vrednosti, ki še niso nujno izračunane v času uporabe same ovojnice. Ta prvina omogoča preprosto izražanje vzporednih algoritmov brez uporabe nizkonivojskih prvin večnitnega programiranja, kot je na primer semafor. Implementacija prihodnosti v LibY je razred `fut<T>`, kjer tip `T` določa tip ovite vrednosti. Objekti tipa `fut<T>` imajo tri možna notranja stanja:

stanje *0*: vrednost še ni izračunana,

stanje *1*: vrednost je izračunana,

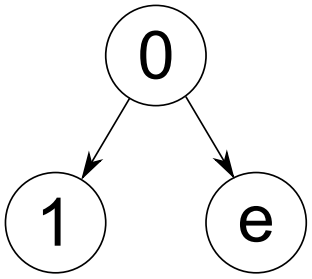
stanje *e*: pri izračunu vrednosti je prišlo do napake (dejanska vrednost objekta `fut<T>` je izjema s tipom `std::exception`).

Začetno stanje je *0*, možna prehoda pa sta le dva:

***0* → *1*:** vrednost se je izračunala,

***0* → *e*:** pri izračunu vrednosti je prišlo do napake.

Na sliki 3.1 so prikazani prehodi med stanji prihodnosti `fut<T>`.



Slika 3.1: Prehodi med stanji prihodnosti `fut<T>`.

Programski vmesnik razreda `fut<T>`

Opis razreda `fut<T>` vsebuje samo najnujnejše metode:

```
template <typename T>
class fut {
public:
   fut();
   fut(const T &v);

   const T &get() const;

   void set(const T &);
   void set_exception(const std::exception &);

   class notify {
   public:
      virtual void fut_value(
         fut<T> *, const T &) = 0;

      virtual void fut_exception(
         fut<T> *, const std::exception &) = 0;
   };
   void register_notify(notify *);
};
```

Opis programskega vmesnika:

`fut<T>::fut()`
: Konstruktor razreda, ki ustvari prihodnost s stanjem *0*.

`fut<T>::fut(const T &v)`
: Konstruktor razreda, ki ustvari prihodnost s stanjem *1* in vrednostjo `v`.

`const T &fut<T>::get() const`
: Vrne referenco na vrednost, če je prihodnost v stanju *1*. Sproži izjemo tipa `std::exception`, če je prihodnost v stanju *e*. Klic metode `get` se obravnava kot napaka v programu in sproži izjemo `std::logic_error`, če je prihodnost v stanju *0*, saj se prehod v stanji *1* in *e* obvesti s povratnima klicema `fut_value` in `fut_exception` v razredu `fut<T>::notify`.

`void fut<T>::set(const T &v)`
: Prihodnosti se nastavi novo stanje *1* in vrednost `v`, če je stanje prihodnosti *0*. Sproži izjemo tipa `std::runtime_error`, če je prihodnost v stanju *1* ali *e*.

`void fut<T>::set_exception(const std::exception &e)`
: Prihodnosti se nastavi novo stanje *e* in izjema `e`, če je stanje prihodnosti *0*. Sproži izjemo tipa `std::runtime_error`, če je prihodnost v stanju *1* ali *e*.

`void fut<T>::register_notify(notify *n)`
: Shrani kazalec na objekt `*n` v prihodnost. Ko preide prihodnost v stanje *1* ali *e*, se pokliče `n->fut_value( this, v )` ali `n->fut_exception( this, e )`. Če je prihodnost že v stanju *1* ali *e*, se metodi pokličeta takoj.

Razred `fut<T>` uporablja štetje referenc kot notranjo metodo za upravljanje s pomnilnikom, kar ponazori naslednji primer:

```
{
   fut<int> a;
   fut<int> b = a;
   a.set(5);
   // a in b kažeta na isto vrednost tipa int:
   assert(b.get() == 5);
}
// uporabljen pomnilnik objektov a in b je sproščen
// po koncu bloka { }
```

Objekti `fut<T>` niso varni za hkratno uporabo iz več niti zaradi hitrostnih optimizacij v prevajalnikih (preurejanje zaporedja izvajanja) in lastnosti

predpomnilnikov v računalniških arhitekturah [19], kar je podobno običajnim kazalcem v jezikih C in C++. Hkratni klici metod na različnih prihodnostih, ki kažejo na isto vrednost v pomnilniku, pa so seveda varni.

Primer 1:

```
fut<int> globalna_prihodnost;

void nit_a(fut<int> a) {
   // NEPRAVILNO! (lahko povzroči sesutje programa)
   globalna_prihodnost = a;
}

void nit_b(fut<int> b) {
   // NEPRAVILNO! (lahko povzroči sesutje programa)
   globalna_prihodnost = b;
}
```

Primer 2:

```
fut<int> globalna_prihodnost;

void nit_a(fut<int> a) {
   // v redu (vrednost globalne prihodnosti
   // je sicer nejasna po koncu izvajanja: 1 ali 2)
   a->set(1);
}

void nit_b(fut<int> b) {
   // v redu (vrednost globalne prihodnosti
   // je sicer nejasna po koncu izvajanja: 1 ali 2)
   b->set(2);
}

void main() {
   ustvari_nit(nit_a, globalna_prihodnost);
   ustvari_nit(nit_b, globalna_prihodnost);
}
```

3.1.2 Izvajalci in opravila

Izvajalec je zelo razširjen konstrukt [7, 27], ki poskrbi za razporejanje opravil na programske niti in posledično na strojne niti. Najbolj pogosta izvedba izvajalca je bazen niti, pri čemer je število programskih niti običajno od n do $2n$, kjer je n število strojnih niti v računalniku. Število strojnih niti vrne npr. funkcija *hardware_ concurrency()* v Intel TBB [17].

Opredelitev najnujnejših metod izvajalca in opravila v LibY:

```
class task {
public:
   class status {
   public:
      virtual void task_finished() {}
   };

   virtual void run(status*) = 0;
};

class executor_i {
public:
   virtual void execute(task*) = 0;
};

// uporabimo štetje referenc za izvajalce
typedef shared_ptr<executor_i> executor;
```

Opis programskega vmesnika:

`void task::run(status *s)`
: Metoda, ki jo implementira vsako opravilo, se pokliče iz izvajalne niti. Metoda `task::status` je običajno implementirana v izvajalni niti. Klic metode `task::status::task_finished` obvesti izvajalno nit `s`, da se je glavni del opravila zaključil, kar je mogoče koristno uporabiti za določene pomembne hitrostne optimizacije. To bo podrobneje opisano pozneje.

`void executor_i::execute(task *t)`
: Metoda, ki jo implementira vsak izvajalec, doda opravilo `t` v izvajalno vrsto.

Knjižnica LibY postavlja zahtevo glede implementacije izvajalca: ko se preda opravilo izvajalcu z metodo `executor_i::execute`, mora izvajalec poskrbeti, da se opravilo začne izvajati v končnem času, torej ne sme ostati v izvajalni vrsti neskončno časa. Pri zasnovi in uporabi izvajalca je treba biti pazljiv na opravila, ki se nikoli ne izvedejo do konca (metoda `task::run` ne vrne izvajanja klicatelju). Primer takega opravila je komunikacija programa z virom operacijskega sistema, kot je sprejemanje odjemalcev TCP na strežniku. Izvajalec mora biti obveščen o takih opravilih, da ostala opravila ne zastajajo v čakalni vrsti. V splošnem je pri zasnovi opravil dobro upoštevati dve pravili:

- Opravila naj se izvedejo v čim krajšem času, da ne pride do pretirane neodzivnosti sistema.
- Opravila naj ne bodo prekratka, da ne bo porabljen čas zaradi preklopov presegel razumne meje.

3.1.3 Vzporedni klici

Vzporedni klici so zadnja in najpomembnejša prvina vzporednega programiranja v LibY. Vzporedni klici dajejo programom, ki uporabljajo knjižnico LibY, značilno zasnovo podatkovno pretokovnega programiranja. Sam vzporedni klic je ovojnica za običajne funkcije, katerih izvajanje prepusti izvajalcu, začetek izvajanja pa določajo prehodi notranjih stanj prihodnosti. Vzporedni klici izjemno poenostavijo izvedbo programov, ki tečejo na velikem številu strojnih niti in imajo veliko število odvisnosti med posameznimi opravili ali pa kombinirajo več modelov vzporednega programiranja, kot so delovne niti in cevovodi. Za začetek sledi preprost primer:

```
int f(int, int);
int g(int, int);
void print(int);

void main()
{
   int a = f(5, 6);
   int b = g(a, 3);
   print(b);
}
```

Ta program lahko enostavno paraleliziramo s pomočjo vzporednih klicev `pcall`, prihodnosti `fut<T>` in izvajalca `get_standard_cpu_executor()`, ki poskrbi za izvajanje na več strojnih nitih:

```
int f(int, int);
int g(int, int);
void print(int);

void main()
{
   executor exe = get_standard_cpu_executor();
   fut<int> a = pcall(exe, f, 5, 6);
   fut<int> b = pcall(exe, g, a, 3);
   pcall(exe, print, b);
}
```

Potek funkcije `main`:

1. `executor exe = get_standard_cpu_executor();`
   Običajen bazen niti se določi kot izvajalec.

2. `fut<int> a = pcall(exe, f, 5, 6);`
   Ustvari se opravilo, ki požene `f(5, 6)` v bazenu niti.

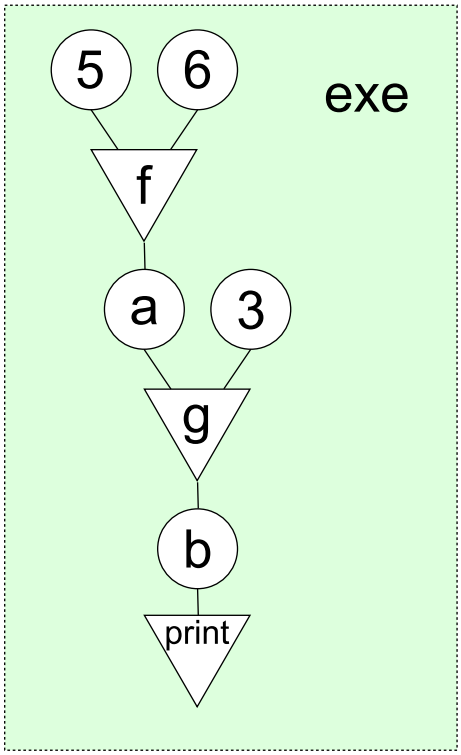
3. `fut<int> b = pcall(exe, g, a, 3);`
   Ustvari se opravilo, ki požene `g(a, 3)` v bazenu niti. Opravilo `g` se ne bo izvedlo, dokler opravilo `f` ne konča izvajanja (počakamo, da gre prihodnost `a` v stanje *1*).

4. `pcall(exe, print, b);`
   Ko se izračuna `b`, se izpiše vrednost s funkcijo `print`.

Na sliki 3.2 je prikazano izvajanje funkcije `main`.

Programski vmesnik vzporednih klicev

```
template <typename Return,
   typename [Y1 ... Yn],
   typename [X1 ... Xn]>
fut<Return> pcall(executor,
   Return (*f)([Y1 ... Yn]),
   [const X1 &x1 ... const Xn &xn]);
```



Slika 3.2: Izvajanje funkcije `main`.

Opis parametrov funkcije `pcall`:

`executor`
: Izvajalec, kjer se bo izvedel klic funkcije `f`.

`Return (*f)([Y1 ... Yn])`
: Kazalec na funkcijo, ki bo poklicana vzporedno. Izhodni tip `Return` se uporabi v prihodnosti `fut<Return>`, ki ga vrne funkcija `pcall`. Vhodni parametri funkcije `f` so tipov `Y1...Yn`.

`[const X1 &x1 ... const Xn &xn]`
: Vhodni parametri za vzporedni klic funkcije `f` so tipov `const X1 &x1 ...const Xn &xn`. Tipi `X1...Xn` so lahko različni od `Y1...Yn`, saj tako podpremo samodejne pretvorbe tipov v jeziku C++ za ceno slabše razumljivejših sporočil o napakah tipov. V primeru, da je vhodni parameter prihodnost tipa `fut<T>`, se bo z uporabo metode `fut<T>::get` neposredno pred izvajanjem funkcije `f` pretvoril v vrednost s tipom `T`.

Izvajanje vzporednih klicev

Klic funkcije `pcall` v ozadju ustvari opravilo, ki bo izvedlo želen vzporeden klic na funkciji `f` s pomočjo podanega izvajalca. Funkcija `pcall` vrne prihodnost `fut<Return>`, kjer je `Return` tip vrednosti, ki jo vrne funkcija `f`. Prihodnost `fut<Return>` predstavlja izhodno vrednost funkcije `f`, ko se bo izvršila. Opravilo vzporednega klica je lahko v treh stanjih, ki so podobna stanjem prihodnosti `fut<T>`:

stanje *0*: Od vhodnih parametrov obstaja vsaj ena prihodnost, ki je v stanju *0*, nobena prihodnost pa ni v stanju *e*.

stanje *1*: Vsi vhodni parametri so izračunani (vse prihodnosti so v stanju *1*), opravilo je bilo predano izvajalcu.

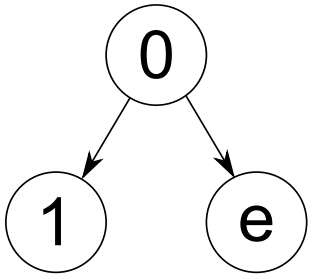
stanje *e*: Vsaj en vhodni parameter je prihodnost v stanju *e*, opravilo je bilo predano izvajalcu, da prenese sporočilo o napaki (izjemo) na izhodno prihodnost `fut<Return>`.

Začetno stanje je *0*. Možna prehoda stanj sta samo dva, podobno kot pri stanjih prihodnosti `fut<T>`:

***0* → *1*:** vsi vhodni parametri, ki so prihodnosti, so prešli v stanje *1*,

***0* → *e*:** vsaj eden od vhodnih parametrov, ki je prihodnost, je prešel v stanje *e*.

Na sliki 3.3 so prikazani prehodi med stanji vzporednega klica.



Slika 3.3: Prehodi med stanji vzporednega klica.

Iz naštetih stanj in prehodov med njimi lahko ugotovimo naslednje pomembne lastnosti vzporednih klicev:

- če so vsi vhodni parametri običajne vrednosti (niso prihodnosti), se klic takoj preda v izvajanje,
- dokler vse vhodne prihodnosti niso izračunane, je treba čakati na razplet,
- če se uspešno izračunajo vse vhodne prihodnosti, se klic preda v izvajanje,
- pri prvi prihodnosti, katere izračun ni uspel, opravilo vzporednega klica takoj prenese izjemo na izhodno prihodnost `fut<Return>`, vsi ostali vhodni parametri se zavržejo.

Prestrezanje izjem

V prejšnjem odstavku je bil prikazan osnovni model izvajanja vzporednih klicev. Do celotnega modela izvajanja pa manjka še metoda za prestrezanje izjem, brez katere bi se program preprosto nehal izvajati ob prvi napaki – vsi vzporedni klici bi prešli v stanje *e*. Uporabimo funkcijo `pcatch`, ki pretvori običajno prihodnost `fut<T>` v prihodnost z možnostjo napake `fut<variant<T, std::exception> >`:

```
template <typename T>
fut<variant<T, std::exception> > pcatch(executor exe, fut<T> f)
```

Vhodni parameter `exe` določi, v katerem izvajalcu se bo izvršila pretvorba prihodnosti `f` v prihodnost s tipom `fut<variant<T, std::exception> >`. Običajno je najbolje izbrati izvajalca, ki bo izvajal telo reševanja napake.

Primer uporabe funkcije `pcatch`:

```
int fn1()
{
   if(rand() & 3)
      throw std::runtime_error();

   return 1;
}

void fn2(variant<int, std::exception> value)
{
   if(int *v = value::get<int>())
```

```
      ; // v 75 % primerov
   else if(std::exception *e = value::get<std::exception>())
      ; // v 25 % primerov
}

void main()
{
   executor exe = get_standard_cpu_executor();
   pcall(exe, fn2,
      pcatch(exe,
         pcall(exe, fn1));
}
```

3.2 Prednosti osnovnega modela izvajanja LibY

Do zdaj so bile prikazane osnovne prvine vzporednega programiranja v LibY. V tem poglavju bodo prikazane nekatere pozitivne lastnosti uporabe LibY v programih, ki neposredno izhajajo iz zasnove knjižnice.

3.2.1 Preprečevanje smrtnega objema

Prvine vzporednega programiranja LibY so zasnovane tako, da zelo poenostavijo pisanje vzporednih programov, ki ne dopuščajo nobene možnosti za smrtni objem. Pravzaprav obstaja samo en način, da program, ki za vzporedno programiranje uporablja samo knjižnico LibY, preide v stanje smrtnega objema. Ta način je uporaba neinicializiranih prihodnosti `fut<T>`. Sledi preprost primer, ki ponazarja smrtni objem v LibY:

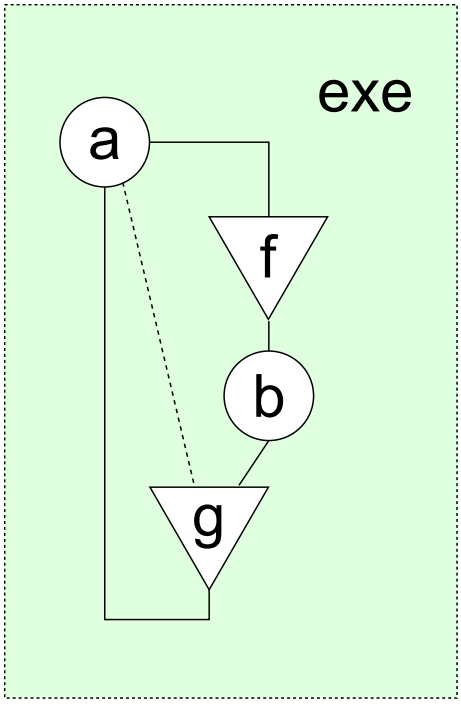
```
int f(int x)
{
   return x * 5;
}

void g(fut<int> x, int y)
{
   x.set(y);
}

void main()
```

```
{
   fut<int> a;
   fut<int> b = pcall(exe, f, a);
   pcall(exe, g, fut<fut<int> >(a), b);
}
```

Postopek:

1. naredi se vzporedni klic funkcije `f`, ki se bo izračunala, ko bo izračunana prihodnost a,
2. naredi se vzporedni klic funkcije `g`, ki bo nastavila vrednost prihodnosti a, vendar je za to potreben rezultat iz funkcije `f`,
3. oba vzporedna klica funkcij `f` in `g` sta v stanju *0*, program se ustavi.

Na sliki 3.4 je prikazan smrtni objem.



Slika 3.4: Smrtni objem.

Podan primer se lahko posploši na **opredelitev smrtnega objema pri uporabi knjižnice LibY: do smrtnega objema v programu pride, ko so izpolnjeni naslednji pogoji:**

1. obstaja podmnožica vzporednih klicev v stanju *0*, ki nimajo izpolnjenih predpogojev za prehod v stanje *1* ali *e*,
2. ne obstaja niti en vzporedni klic, ki ima izpolnjene predpogoje za prehod v stanje *1* ali *e* in ima izhodno prihodnost kot vhodni parameter vsaj enemu vzporednemu klicu iz točke 1,
3. ne obstaja nobena druga sistemska nit, ki bi lahko nastavila vrednost vsaj enemu vhodnemu parametru vzporednih klicev iz točke 1.

Opredelitev smrtnega objema pri uporabi knjižnice LibY se razlikuje od klasične opredelitve smrtnega objema [21]. Vendar sta si opredelitvi precej podobni, tako da je uporaba izraza smrtni objem upravičena:

- vzporedni klici so podobni procesom,
- vzporedni klic v stanju *0* je podoben procesu, ki čaka na sprostitev vira,
- vzporedni klic v stanju *1* ali *e* je podoben procesu, ki je uspešno zasegel vir,
- prihodnosti so podobne virom,
- prihodnost v stanju *0* je podobna zaseženemu viru,
- prihodnost v stanju *1* ali *e* je podobna sproščenemu viru,
- cikel podatkovnih odvisnosti je podoben ciklu v grafu zaseganja virov.

Sledi dokaz trditve, da **ob prepovedi neinicializiranih prihodnosti program nikoli ne preide v stanje smrtnega objema**. Dokaz mora pokazati, da vsak vzporedni klic zapusti stanje *0*, torej se izvede. Obstajata dva načina, da ustvarimo novo prihodnost. Prvi način ustvari prihodnosti v stanju *0*: klic funkcij `pcall` in `pcatch`. Drugi način ustvari prihodnosti v stanju *1*: klic konstruktorja prihodnosti `fut<T>::fut(const T &v)`. Pri dokazovanju se omejimo samo na prihodnosti v stanju *0*, ker so edini predpogoj, ki drži vzporedni klic v stanju *0*. Prihodnosti, ki jih ustvari funkcija `pcatch`, ne potrebujejo posebne obravnave, saj stanje *0* povsem sovpada s stanjem *0* izvorne prihodnosti. Vsaka prihodnost v stanju *0* je torej posledica dveh možnih vzporednih klicev:

1. vzporedni klic v stanju *1* ali *e*, ki se je ravnokar predal v izvajanje, izhodna prihodnost se bo v naslednjem koraku izračunala in predpogoj za smrtni objem ni izpolnjen,

2. vzporedni klic v stanju *0*.

Za vzporedni klic v stanju *0* pa velja, da so bili vsi vhodni parametri ustvarjeni pred svojo izhodno prihodnostjo. Tako se lahko rekurzivno vzpenjamo do vedno starejših vzporednih klicev, dokler ne pridemo popolnoma na vrh, kjer pa obstajajo samo vhodni parametri dveh tipov:

1. vrednosti, ki niso prihodnosti,

2. prihodnosti, ki so bile inicializirane v stanje *1* z neko vrednostjo.

Zdaj lahko zaključimo, da ima vsak vzporedni klic svoje starejše prednike, ki se bodo gotovo izvedli.

Na tem mestu se poraja vprašanje: zakaj neinicializirane prihodnosti `fut<T>` preprosto ne prepovemo? Težava nastopi pri uporabi virov operacijskega sistema ali pa pri uporabi zunanjih knjižnic, ki ne podpirajo knjižnice LibY. Neinicializirane prihodnosti so uporabne za premostitev razlik med modelom LibY in zunanjo kodo. Primer, ki ponazarja uporabo neinicializiranih prihodnosti:

```
void ask_operating_system_to_start_some_task(
   void (*finished_callback)(void*), void*);

void task_finished(void *arg)
{
   fut<int> *f = (fut<int>*)arg;
   try { f->set(1); } catch(...) {}
   delete f;
}

fut<int> start_task()
{
   fut<int> ret;
   ask_operating_system_to_start_some_task(task_finished,
      new fut<int>(ret));

   // task_finished se pokliče, ko se opravilo,
   // ki jo izvaja operacijski sistem, zaključi

   // iz funkcije vrnemo prihodnost, ki bo
   // v stanju 1, ko se opravilo zaključi
```

```
    return ret;
}
```

V splošnem se večinoma izkaže, da ni pretirano težko omejiti neinicializirane prihodnosti samo na nujna mesta v programu – stičišča z operacijskim sistemom in ostalimi zunanjimi knjižnicami. Običajno je lahko preveriti pravilnost programa, kar pomeni da ni možnosti za smrtni objem.

3.2.2 Velika prilagodljivost stopnje vzporednega izvajanja in nizka poraba pomnilnika

V splošnem je mogoče predstaviti izvajanje programa, ki uporablja knjižnico LibY, kot acikličen usmerjen graf. Prihodnosti in vzporedne klice predstavimo kot vozlišča, premike podatkov med prvinami LibY pa predstavimo kot usmerjene povezave. Vozlišča, ki nimajo očetov, so v stanju *1* ali *e*, kar pomeni, da je izvajanje mogoče predati izvajalcu. V vsakem trenutku lahko torej razdelimo celoten graf na dva dela: vozlišča, ki jih je možno predati v izvajanje, in vozlišča, ki morajo še počakati na izračun vhodnih parametrov. Iz napisanega sledi, da imamo v vsakem trenutku možnost stopnje vzporednega izvajanja v intervalu $[k, n + k]$, kjer k predstavlja trenutno število niti, ki izvajajo opravila, in n število vzporednih klicev v stanjih *1* in *e*. **Stopnjo vzporednega izvajanja lahko torej z uporabo LibY vedno določimo na trenutno najboljšo možno vrednost računalniškega sistema.** Ugotovimo lahko dve pomembni posledici:

- število izvajalnih niti lahko določimo na najboljšo možno vrednost,
- vse obstoječe niti so v stanju izvajanja programa, razen niti, ki so v bazenu zaradi nižanja amortiziranega časa ustvarjanja nove niti.

Ker je število niti določeno tako, da ni nobena nit neizkoriščena, je poraba navideznega pomnilnika najmanjša možna pri dani stopnji vzporednosti. Neizkoriščene niti so v računalniških sistemih lahko velik problem. Pri računalniških arhitekturah z 32-bitnim naslavljanjem navideznega pomnilnika in velikostjo sklada npr. 1 MB običajno zmanjka navideznega pomnilnika že pri nekaj tisoč nitih [23]. Sklad je običajno implementiran z uporabo neveljavnih strani. Če program z rastjo sklada preseže trenutno dodeljen nabor strani, se skladu s pomočjo napak strani dodeli nove strani. Žal pa je navidezni pomnilnik za celoten sklad rezerviran vnaprej. Težava ni samo poraba navideznega pomnilnika, saj

so jedra operacijskih sistemov optimizirana za razmeroma nizko število niti. V splošnem velja pravilo, da naj število niti ne odstopa bistveno od števila procesorjev oziroma strojnih niti v računalniku, pri čemer LibY nudi veliko pomoči.

3.2.3 Enak model izvajanja za procesorska in vhodno-izhodna opravila

Knjižnica LibY popolnoma enako obravnava zaključek uporabniških opravil, ki jih je izvedel procesor, in opravil, ki jih izvede operacijski sistem pri vhodni-izhodni komunikaciji. Razlog za to bistveno poenostavitev leži v uporabi prihodnosti. Programski vmesniki, ki so prirejeni za uporabo knjižnice LibY, povsem zakrijejo različne modele izvajanja v implementaciji:

- procesorsko zahtevna opravila – izvajanje v ospredju (odjemalec počaka na konec izvajanja opravila),
- procesorsko zahtevna opravila – izvajanje v niti v ozadju (odjemalec lahko nadaljuje z izvajanjem),
- vhodno-izhodno opravilo – odjemalčeva nit se ustavi, da počaka na konec izvajanja,
- vhodno-izhodno opravilo – konec izvajanja se obvesti s povratnim klicem iz sistemske niti,
- vhodno-izhodno opravilo – konec izvajanja se obvesti s sporočilom iz sistemske čakalne vrste ali sprožitvijo semaforja.

Običajni programski vmesniki ne zakrijejo različnih modelov in zato pogosto prisilijo uporabnika vmesnika, da izbere enega samega. Če se pozneje izkaže, da bi bil primernejši drug model izvajanja, je že prepozno, saj bi bilo treba preurediti celoten program [16, 25]. **Prihodnosti v knjižnici LibY lahko obravnavamo kot dodaten mehanizem za abstrakcijo, ki je ortogonalen na mehanizme, kot so funkcije, objekti, predikati, makroji, funkcijske ovojnice itd.**

3.2.4 Učinkovita izvedba vzporednih klicev

Običajna izvedba opravila vzporednega klica vsebuje:

- kazalec na funkcijo `Return (*f)([Y1...Yn])`,

- vhodne parametre za funkcijo `f` tipov `X1...Xn`,
- prihodnost `fut<Return>`,
- števec še neizračunanih (v stanju *0*) prihodnosti iz vhodnih parametrov,
- zastavico, če je katera od vhodnih prihodnostih sporočila napako (je v stanju *e*).

Vhodni parametri se obravnavajo glede na tip. Če gre za prihodnost `fut<T>`, je postopek sledeč:

1. števec neizračunanih vrednosti se poveča za 1;
2. opravilo vzporednega klica se prijavi na sporočilo o spremembi stanja z metodo `fut<T>::register_notify`;
3. obvestilo `fut<T>::notify::fut_value` zmanjša števec neizračunanih prihodnosti za 1. Obvestilo `fut<T>::notify::fut_exception` zmanjša števec neizračunanih prihodnosti za $n+1$, kjer n predstavlja število vhodnih parametrov, in nastavi zastavico za napako.

Vhodni parametri, ki so običajne vrednosti, pa nimajo posebnega začetnega postopka, zato lahko spustimo korake od 1 do 3. Takšna opredelitev izvedbe opravila pa ponuja možnost, da implementiramo vzporedni klic povsem brez uporabe muteksov ali semaforjev. Treba je uporabiti le atomične spremenljivke (*volatile* v jezikih C in C++) za števec neizračunanih prihodnosti ter zastavico, ki označuje napako. Pogoj za predajo v normalno izvajanje je prehod števca v vrednost *0*, pogoj za predajo v izvajanje z obveščanjem o napaki pa prehod zastavice za napako v vrednost *1*. Zaznavanje prehodov se učinkovito izvede z ukazi za zaklenjen dostop do pomnilnika, ki so prisotni na vseh modernih 32-bitnih procesorjih [22].

3.2.5 Prednosti metode `fut<T>::register_notify`

Poglavitna prednost razreda `fut<T>` izhaja iz metode `fut<T>::register_notify`. Običajen pristop pri večini obstoječih knjižnic ali jezikov, ki implementirajo prihodnosti, je uporaba metode `wait` [8, 27]. Ta metoda ustavi izvajanje trenutne niti in nadaljuje izvajanje šele, ko je vrednost izračunana. Metodo `wait` lahko zelo preprosto implementiramo v razredu `fut<T>` z uporabo metode `fut<T>::register_notify` in standardne prvine večnitnega programiranja semafor:

```
const T &fut<T>::wait() const
{
   semaphore sem;
   register_notify(&sem);
   sem.wait();
   return get();
}

// razred semaphore vsebuje implementacijo metod
// fut_value in fut_exception, ki pokličejo post()
```

Žal uporaba metode `wait` povzroča dve resni težavi:

- nevarnost smrtnih objemov, kar nas vrne nazaj v svet klasičnih težav večnitnega programiranja,
- ustavljene niti nepotrebno večajo porabo pomnilnika in znižujejo učinkovitost izkoriščanja strojne opreme.

Dodatna prednost uporabe metode `fut<T>::register_notify` pa je preprosta in učinkovita preslikava na uporabo virov operacijskega sistema. Velik del zasnove modernih operacijskih sistemov temelji na dogodkovnem programiranju [1], ki izhaja iz prekinitev in drugih načinov komunikacije v strojni opremi. Ta zasnova se v uporabniških programih odraža npr. v *I/O completion port* v Windows NT in *epoll* v jedru Linux. Obe omenjeni prvini pošiljata informacije o stanju vhodno-izhodnih in drugih tipov komunikacije programa z operacijskim sistemom. V splošnem delujeta kot čakalna vrsta statusnih sporočil, ki se jih običajno tako uporabi v programu, ki uporablja LibY:

```
for(message *m; m = get_message(port);) {
   if(event_type(m) == executable_task) {
      m->run(this);
   } else {
      io_channel *chan = get_channel(m);
      switch(event_type(m)) {
         case read_event:
            bytebuffer buf = read_data(chan);
            fut<bytebuffer> f = find_fut_for_channel(chan);
            f->set(buf);
            break;
         case write_event:
```

```
            chan->flush_unsent_buffers();
            break;
      }
   }
}
```

Opisana vhodno-izhodna zanka poleg izmenjave podatkov z operacijskim sistemom podpira tudi izvajanje poljubnih drugih opravil kar z vmesnikom običajnega izvajalca. S takim pristopom znižamo število sistemskih niti in tako tudi znižamo število preklopov niti na enem procesorju.

3.2.6 Prednosti pri uporabi razreda `task::status`

Še enkrat povzemimo opis razredov `task` in `task::status`:

```
class task {
public:
   class status {
   public:
      virtual void task_finished() {}
   };

   virtual void run(status*) = 0;
};
```

Metoda `task::status::task_finished` je namig izvajalcu, da je opravilo zaključilo glavni del izvajanja. Dobro implementiran izvajalec lahko ta namig izkoristi tako, da označi nit, ki izvaja opravilo, kot prosto za prevzem novih opravil. Dodatno lahko izvajalec poviša še prioriteto niti v vrsti prostih niti tako, da bo naslednje opravilo najverjetneje šlo na to nit. Težavo, ki jo rešujemo s signalom `task::status::task_finished`, najbolje opišemo kar na naivni implementaciji vzporednega klica funkcije `f`, ki ne obvesti izvajalca o zaključku izvajanja `f`:

```
int f();

class pcall_task: public task {
   fut<int> return_value;

public:
```

```
   void run(status*)
   {
      int tmp = f();
      return_value.set(tmp);
   }
};
```

Možen potek izvajanja:

1. pokliče se `f()`,
2. prihodnosti `return_value` se priredi vrednost,
3. prihodnost obvesti vsa opravila, ki so zahtevala obvestilo o spremembi stanja, nadaljujemo samo z opisom ene poti,
4. v naslednjem opravilu smo bili obveščeni o uspešno izračunani vrednosti `f()` – ker ni več nobenih prihodnosti v stanju $\emptyset$, predamo opravilo izvajalcu,
5. izvajalec pogleda trenuten nabor niti in ugotovi, da obstaja samo ena zasedena nit, ki trenutno izvaja obveščanje o izračunani vrednosti `f()`,
6. izvajalec ustvari novo nit in ji dodeli opravilo,
7. nova nit se požene na drugem procesorju,
8. rezultat izračuna `f()` ni v predpomnilniku drugega procesorja, zato je treba najprej obvestiti prvi procesor, naj zapiše predpomnilniški blok v glavni pomnilnik, nato pa je treba ta blok prenesti še v predpomnilnik drugega procesorja,
9. novo opravilo se lahko požene in uporabi rezultat izračuna `f()`.

Iz napisanega lahko sklepamo, da lahko povsem zaporedni programi tečejo veliko počasneje z uporabo LibY, če bi uporabili tako naivno implementacijo. Razlog za počasno delovanje sledi iz točk do 5 do 8. Popravimo razred `pcall_task`, da sporoči konec izvajanja glavnega dela opravila:

```
int f();

class pcall_task: public task {
   fut<int> return_value;
```

```
public:
   void run(status *s)
   {
      int tmp = f();
      s->task_finished();
      return_value.set(tmp);
   }
};
```

Najverjetnejši potek dogajanja pri uporabi metode `task_finished`:

1. pokliče se `f()`,
2. trenutno nit se obvesti, da je konec izvajanja glavnega dela opravila,
3. prihodnosti `return_value` se priredi vrednost,
4. prihodnost obvesti vsa opravila, ki so zahtevala obvestilo o spremembi stanja, nadaljujemo samo z opisom ene poti,
5. v naslednjem opravilu smo bili obveščeni o uspešno izračunani vrednosti `f()` – ker ni več nobenih prihodnosti v stanju *0*, predamo opravilo izvajalcu,
6. izvajalec pregleda trenuten nabor niti in ugotovi, da obstaja primerna prosta nit, ter ji dodeli novo opravilo,
7. prihodnost preneha z obveščanjem,
8. trenutna nit se sprosti in takoj nadaljuje z izvajanjem novega opravila, ki uporabi rezultat izračuna `f()`.

Vidimo, da se z dodatnim obveščanjem o stanju izvajanja opravila izognemo časovno zelo potratnim opravkom, kot je ustvarjanje nove niti in premikanje predpomnilniških blokov med procesorji. V redkem primeru, da pri točki 6 ne obstaja primerna prosta nit, se postopek nadaljuje s točko 6 prejšnjega primera poteka dogajanja.

Poglavje 4

Višjenivojski konstrukti

V tem poglavju bo opisano nekaj uporabnih konstruktov, ki so implementirani kot dodatek LibY z uporabo osnovnih prvin ali kot razširitev zasnove LibY z ohranjanjem osnovnih prednosti.

4.1 Operatorji

Operatorji so preproste funkcije, s pomočjo katerih vodimo tok izvajanja, brez da bi bili prisiljeni v zaporedje izvajanja, ki povsem sledi toku podatkov. Mnogi vzporedni algoritmi pa sploh nimajo očitnega toka podatkov, ki bi ga lahko izračunavali vzporedno.

4.1.1 Operator `punwrap`

Pri zasnovi programskih vmesnikov hitro naletimo na funkcije, ki vračajo prihodnost. Taki vmesniki so zelo pogosti v vhodno-izhodnem programiranju, saj konec nekega vhodno-izhodnega zahtevka običajno predstavimo s prihodnostjo. Nastopi težava gnezdenja prihodnosti. Primer:

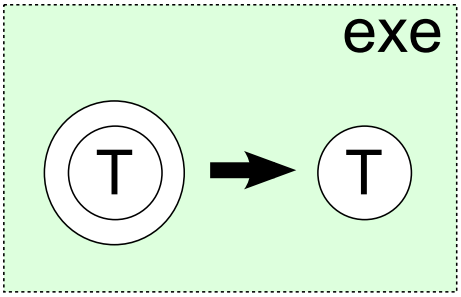
```
fut<int> turn_on_printer(int printer_id);
fut<int> fetch_printer_id();

void main()
{
   fut<int> id = fetch_printer_id();

   fut<fut<int> > status = pcall(exe, turn_on_printer, id);
```

```
   // notranji fut<int> označuje konec opravila turn_on_printer,
   // fut<fut<int> > označuje začetek opravila turn_on_printer,
   // kar je ekvivalentno koncu opravila fetch_printer_id
   // in zato ne nosi nobene dodatne informacije
}
```

Uporabimo funkcijo `punwrap`. Funkcija ustvari novo prihodnost, ki ima odstranjen zunanji ovitek `fut< >` in je povsem ekvivalentna notranji prihodnosti. Programski vmesnik:

```
template <typename T>
fut<T> punwrap(executor, fut<fut<T> >);
```

Na sliki 4.1 je prikazana izvedba operatorja `punwrap`.



Slika 4.1: Izvedba operatorja `punwrap`.

Definirajmo še funkcijo `pcallu`, ki združi operator `punwrap` in funkcijo `pcall` zaradi jedrnatosti:

```
template <typename Return,
   typename [Y1 ... Yn],
   typename [X1 ... Xn]>
fut<Return> pcallu(executor exe,
   fut<Return> (*f)([Y1 ... Yn]),
   [const X1 &x1 ... const Xn &xn])
{
   return punwrap(exe, pcall(exe, f, [x1 ... xn]));
}
```

4.1.2 Operator »

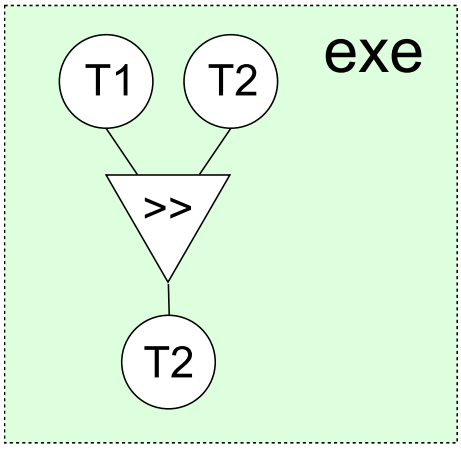
Z operatorjem » umetno vstavimo podatkovne odvisnosti v prihodnosti. Operator je uporaben za usmerjanje vzporednih algoritmov, ki nimajo očitnega toka podatkov in temeljijo na stranskih učinkih pri izvajanju. Programski vmesnik:

```
template <typename T1, typename T2>
fut<T2> operator >> (executor, fut<T1>, fut<T2>);
```

Operator ustvari novo prihodnost, ki je povsem ekvivalentna `fut<T2>`, razen naslednjih dveh razlik:

- nova prihodnost preide v stanje *e*, če preide v stanje *e* prihodnost `fut<T1>` ali `fut<T2>`,
- nova prihodnost preide v stanje *1* z vrednostjo prihodnosti `fut<T2>`, če v stanje *1* preideta obe prihodnosti `fut<T1>` in `fut<T2>`.

Na sliki 4.2 je prikazana izvedba operatorja ».



Slika 4.2: Izvedba operatorja ».

4.1.3 Vzporedni quicksort z uporabo `punwrap` in »

Operatorja `punwrap` in » sta dovolj za elegantno implementacijo vzporednega quicksorta. Implementacija funkcije `partition` je izpuščena, saj ni vzporedna. Povejmo le, da funkcija `partition` vrne iterator, ki kaže na mejni element.

Elementi levo od meje so manjši od mejnega elementa, elementi desno od meje so večji ali enaki mejnemu elementu. Predstavljena implementacija quicksorta podpira poljubno območje elementov z uporabo iteratorjev za naključni dostop in poljubno funkcijo urejanja po zgledu standardne knjižnice C++ 1998.

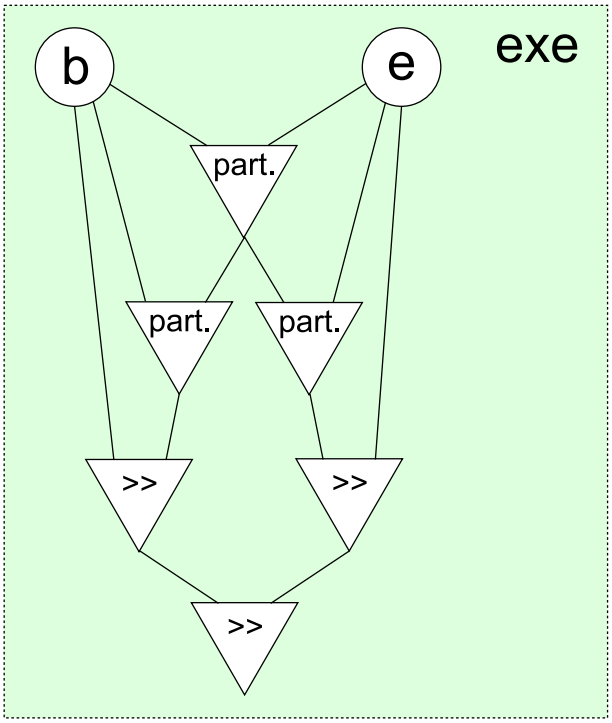
```
template <typename Iterator, typename Less>
fut<int> pqsort(Iterator begin, Iterator end, Less less)
{
   if(end - begin < 2)
      return 1;

   fut<Iterator> boundary = pcall(partition, begin, end, less);
   fut<int> left = pcallu(exe, pqsort, begin, boundary, less));
   fut<int> right = pcallu(exe, pqsort, boundary+1, end, less));
   return left >> right;
}
```

Na sliki 4.3 je prikazano poenostavljeno izvajanje funkcije `pqsort`. Izpuščene so vmesne prihodnosti in klici operatorja `punwrap`.

4.2 Čakalna vrsta FIFO

Čakalna vrsta FIFO je preprost konstrukt, ki olajša programiranje podatkovnih tokov, kot so povezave TCP, ali drugih podatkovnih izvirov, ki temeljijo na dogodkih. V izvedbi čakalne vrste FIFO potrebujemo neinicializirane prihodnosti, kar pomeni nevarnost smrtnega objema. Čakalno vrsto zato uporabljamo samo v notranji izvedbi stika programa in operacijskega sistema ali zunanjih knjižnic, ki ne temeljijo na LibY. Za odsotnost smrtnega objema moramo poskrbeti sami. Programski vmesnik:

```
template <typename T>
class queue_c {
public:
   fut<T> pop();
   void push(const T &);
   void set_exception(const std::exception &);
};

// uporabimo štetje referenc
typedef shared_ptr<queue_c> queue;
```



Slika 4.3: Poenostavljen prikaz izvajanja funkcije `pqsort`.

Opis vmesnika:

`fut<T> queue<T>::pop()`
: Metoda vzame najstarejši element iz podatkovne čakalne vrste in ga vrne, če podatkovna čakalna vrsta ni prazna. Če je podatkovna čakalna vrsta prazna, se zahtevek v obliki prihodnosti `fut<T>` doda v čakalno vrsto za zahtevke.

`void queue<T>::push(const T &t)`
: Če je čakalna vrsta za zahtevke prazna, se vrednost `t` doda v podatkovno vrsto. Če čakalna vrsta za zahtevke ni prazna, se najstarejši prihodnosti nastavi vrednost `t` in se jo izbriše iz čakalne vrste zahtevkov.

`void queue<T>::set_exception(const std::exception &e)`
: Čakalni vrsti FIFO nastavi stanje napake `e`. Vsem obstoječim prihodnostim v čakalni vrsti za zahtevke se nastavi stanje e z izjemo `e`. Vse nadaljnje vrnjene prihodnosti z metodo `pop` so v stanju e z izjemo `e`.

Podatke iz čakalne vrste FIFO običajno prejemamo z vzporedno zanko naslednje oblike:

```
void handle(queue<int> q, int data)
{
   // uporabimo data

   // nadaljujemo s sprejemanjem
   pcall(exe, handle, q, q.pop());
}

void main()
{
   queue<int> q;

   // začetek zanke
   pcall(exe, handle, q, q.pop());
}

// zanko handle prekinemo s klicem q.set_exception(...)
```

4.3 Porazdeljeno izvajanje

Knjižnica LibY ne vsebuje programskih vmesnikov za porazdeljeno izvajanje. Obstaja veliko število komunikacijskih knjižnic, ki podpirajo raznovrstne načine serializacije in prenosa podatkov v oddaljene procese. Knjižnice LibY ne bi bilo smiselno vezati na samo en komunikacijski protokol. Zato je naveden samo primer osnovnega programskega vmesnika za oddaljene klice, ki ga programer lahko prenese na poljubno komunikacijsko knjižnico.

4.3.1 Serializacija

Za prenos podatkov prek običajnih bajtnih podatkovnih tokov potrebujemo mehanizem, ki spremeni podatkovne strukture v vektor bajtov in obratno.

```
// uporabimo štetje referenc
typedef shared_ptr<network_packet_c> network_packet;

// funkcija, ki mrežnemu paketu doda vrednost t
```

```
template <typename T>
void serialize(network_packet p, const T &t);

// funkcija, ki iz mrežnega paketa vzame vrednost t
template <typename T>
void deserialize(network_packet p, T &t);
```

4.3.2 Oddaljeni klici

Za pošiljanje zahtevkov za oddaljene klice uporabimo programski vmesnik, ki ga je treba prenesti na izbrano komunikacijsko knjižnico.

```
class remote_process_c {
public:
   // pošlje podatke o klicu in vrne rezultat kot prihodnost
   template <typename T>
   fut<T> make_call(network_packet);

   // pomožni izvajalec za serializacijo podatkov
   executor get_executor();
};

// uporabimo štetje referenc
typedef shared_ptr<remote_process_c> remote_process;
```

4.3.3 Klic oddaljene funkcije prek definicije vmesnika

Na strani odjemalca lahko pokličemo funkcije programskega vmesnika, ki bi jih radi pognali v oddaljenem procesu, s pomočjo funkcije `remote_call` in makroja `REMOTE_METHOD`.

```
class some_interface {
public:
   virtual int some_remote_procedure(int a, int b) = 0;
};

#define REMOTE_METHOD(_i, _p) #_i "::" #_p, &_i::_p

template <typename R, typename [X1 ... Xn]>
fut<R> remote_call_wait_for_futs(
```

```
   remote_process proc,
   const char *name,
   [const X1 &x1 ... const Xn &xn])
{
   network_packet packet;
   serialize(packet, name);
   serialize(packet, x1); ... serialize(packet, xn);
   return proc->make_call<R>(packet);
}

template <typename I,
   typename R,
   typename [X1 ... Xn]>
fut<R> remote_call(
   remote_process proc,
   const char *name,
   R (I::*)([X1...Xn]),
   [X1 x1 ... Xn xn])
{
   return pcallu<R>(proc->get_executor(),
      remote_call_wait_for_futs, proc, name, [x1...xn]);
}

// primer uporabe
void main()
{
   remote_process proc = connect();
   fut<int> x = remote_call(proc,
      REMOTE_METHOD(some_interface, some_remote_procedure),
      5, 10);
   fut<int> y = remote_call(proc,
      REMOTE_METHOD(some_interface, some_remote_procedure),
      x, 20);
}
```

4.3.4 Sprejem klicev

Na strani strežnika implementiramo programski vmesnik in registriramo vse metode za oddaljen klic.

```
class implementation: public some_interface {
public:
   virtual int some_remote_procedure(int a, int b)
   {
      return a + b;
   }
};

#define REMOTE_METHOD(_i, _p) #_i "::" #_p, &_i::_p

template <typename I, typename R, typename [X1 ... Xn]>
void register_remote_call(I *impl,
   const char *name,
   R (I::*method)([X1...Xn]))
{
   // zaradi jedrnatosti izpustimo implementacijo
}

void main()
{
   some_interface *x = new implementation;
   register_remote_call(x,
      REMOTE_METHOD(some_interface, some_remote_procedure));
}
```

Poglavje 5

Nadzor dostopa do virov

Osnovni model izvajanja v knjižnici LibY privzame stopnjo vzporednega izvajanja, ki najbolje izkoristi večprocesorski računalniški sistem. Vendar pa mnoge podatkovne strukture in viri operacijskega sistema ne podpirajo hkratnih vzporednih dostopov iz več niti. V tem primeru moramo omejiti stopnjo vzporednosti na primerno raven. To poglavje opiše dva mehanizma za nadzor dostopa, tj. pristop z uporabo aktivnih objektov in pravo zaklepanje virov.

5.1 Aktivni objekti

Aktivni objektni so dobro poznan koncept vzporednega programiranja [20]. Običajnim objektom priredimo izvajalca, ki poskrbi za izvajanje vseh metod. Pogosto uporabimo izvajalca, ki je omejen na eno vzporedno nit, in tako dobimo preprost ter učinkovit mehanizem za zagotavljanje nadzora dostopa. Pristop z uporabo aktivnih objektov, ki so omejeni na eno nit na objekt, se zanaša na ugotovitev, da je objektov v večini programov veliko več kot procesorjev v računalniškem sistemu. Na ta način zagotovimo dovolj veliko izkoriščenost sistema.

Implementacija aktivnih objektov s knjižnico LibY je zelo preprosta, saj je model izvajanja v LibY strogo nadrejen aktivnim objektom. Po analogiji s `pcall` in `pcallu` opredelimo dve funkciji `pcallm` in `pcallmu`, ki sta opisani v prejšnjih poglavjih. Programski vmesnik:

```
template <typename Object,
   typename Class,
   typename Return,
   typename [Y1 ... Yn],
```

```
   typename [X1 ... Xn]>
fut<Return> pcallm(Object o,
   Return (Class::*method)([Y1 ... Yn]),
   [const X1 &x1 ... const Xn &xn]);

template <typename Object,
   typename Class,
   typename Return,
   typename [Y1 ... Yn],
   typename [X1 ... Xn]>
fut<Return> pcallmu(Object o,
   fut<Return> (Class::*method)([Y1 ... Yn]),
   [const X1 &x1 ... const Xn &xn])
{
   return
      punwrap(o->get_executor(),
         pcallm(o, method, [x1 ... xn]));
}

// zaradi jedrnatosti izpustimo celotno opredelitev
// izvedb metod pcallm in pcallmu, ki imata parameter
// objekt kot tip prihodnost fut<Object>

fut<Return> pcallm(fut<Object> o, ...);
fut<Return> pcallmu(fut<Object> o, ...);
```

Opis parametrov funkcije `pcallm` in `pcallmu`:

`Object o`
: Aktivni objekt `o`. Objekt mora podpirati klic `o->get_executor()`, ki vrne izvajalca, in klic `o.get()`, ki vrne kazalec na objekt tipa `Class*`. Tip `Object` je običajno `shared_ptr<Class>`.

`Return (Class::*method)([Y1 ... Yn])`
: Kazalec na metodo, ki jo želimo poklicati na aktivnem objektu.

`[const X1 &x1 ... const Xn &xn]`
: Vhodni parametri za vzporedni klic metode.

5.1.1 Programski vmesnik datoteka

Programski vmesnik za pisanje in branje vrstic v datoteko bo podan kot primer uporabe aktivnih objektov. Ker želimo, da se vsako pisanje in branje zaključi v celoti, uporabimo izvajalca, ki omogoča samo eno hkratno nit.

```
class line_file_c;
// uporabimo štetje referenc
typedef shared_ptr<line_file_c> line_file;

class line_file_c {
   // privatni konstruktor, lahko ustvarimo samo z
   // uporabo funkcije create()
   line_file();

public:
   fut<line_file> create(std::string file_name, bool read_write);

   // vrnemo izvajalca, ki je omejen na eno nit
   executor get_executor();

   void write_line(std::string);
   std::string read_line();
};

void main()
{
   fut<line_file> f = line_file_c::create("abc.txt", true);

   // preberemo vrstico iz datoteke
   fut<std::string> line = pcallm(f, &line_file_c::read_line);

   // in jo še enkrat zapišemo
   pcallm(f, &line_file_c::write_line, line);

   // zapišemo še eno vrstico, izvajalec bo poskrbel,
   // da se write_line ne pokliče dvakrat hkrati
   pcallm(f, &line_file_c::write_line, "123");
}
```

5.2 Zaklepanje virov

V prejšnjem odstavku je podan pristop z omejevanjem na eno samo nit pri izvajanju metod objekta. Pogosto pa je treba zakleniti za dostop dva ali več objektov. V tem primeru je potreben nov konstrukt, ki omogoči zaklepanje objektov pri izvajanju vzporednih klicev. Izpeljan je iz razreda `shared_ptr<T>`, ki je v splošnem namenjen upravljanju pomnilnika s štetjem referenc. Najprej opredelimo programski vmesnik za razred `lockable<T>`, ki ponazarja objekte, ki jih je treba pri izvajanju vzporednih klicev zakleniti pri dostopu:

```
template <typename T>
class lockable: public shared_ptr<T> {
public:
   lockable(T *ptr = 0): shared_ptr(ptr) {}
   // podobno ostale nujne metode, ki prenesejo
   // glavnino štetja referenc na metode iz razreda
   // shared_ptr

   class notify {
      virtual void locked(lockable*) = 0;
   };

   void try_lock(notify*);
   void unlock();
};
```

Opis programskega vmesnika:

`void lockable<T>::try_lock(notify *n)`
: Klic metode sproži poskus zaklepanja vira s tipom `T*`. Če vir ni zaklenjen za dostop, se takoj pokliče `n->locked( this )`. Če je vir zaklenjen za dostop, se kazalec `notify *n` doda v čakalno vrsto FIFO, metoda `notify::locked` pa se pokliče ob sprostitvi vira (ob klicu metode `unlock`).

`void lockable<T>::unlock()`
: Klic metode sprosti vir. Če ima čakalna vrsta zaklepov kak element, se vzame najstarejšega iz vrste in obvesti o zaklepu s klicem `n->locked( this )`.

Primer uporabe programskega vmesnika:

```
void transfer_money(
   int *account_a,
   int account_a_limit,
   int *account_b,
   int amount)
{
   if(*account_a + account_a_limit < amount)
      throw std::runtime_error();

   *account_a -= amount;
   *account_b += amount;
}

void main()
{
   lockable<int> account_a(new int(5000));
   lockable<int> account_b(new int(0));
   lockable<int> account_c(new int(1000));

   // oznaka lockable prepreči tekmovanje za vire
   pcall(exe, transfer_money, account_a, 3000, account_b, 1000);
   pcall(exe, transfer_money, account_a, 3000, account_c, 1000);
   pcall(exe, transfer_money, account_c, 3000, account_b, 1000);
}
```

Programerju prijaznega zaklepanja virov ni mogoče implementirati z osnovnimi prvinami vzporednega programiranja v LibY. Treba je razširiti model izvajanja in pri tem biti previden, da se ohranijo prednosti knjižnice LibY. Pri tem mislimo predvsem na preprečevanje smrtnih objemov. Model bomo razširili tako, da vzporednim klicem dodamo novo stanje L. Nova opredelitev stanj vzporednih klicev:

stanje *0*: od vhodnih parametrov obstaja vsaj ena prihodnost, ki je v stanju *0*, in nobena prihodnost v stanju *e*,

stanje *1*: vsi vhodni parametri so izračunani (vse prihodnosti so v stanju *1*), opravilo lahko začne z zaklepanjem virov,

stanje *L*: vsi vhodni parametri so zaklenjeni (vsi objekti `lockable<T>` so sporočili stanje `locked`), opravilo se lahko preda izvajalcu,

stanje *e*: vsaj en vhodni parameter je prihodnost v stanju *e*, opravilo je bilo predano izvajalcu, da prenese sporočilo o napaki (izjemo) na izhodno prihodnost `fut<Return>`.

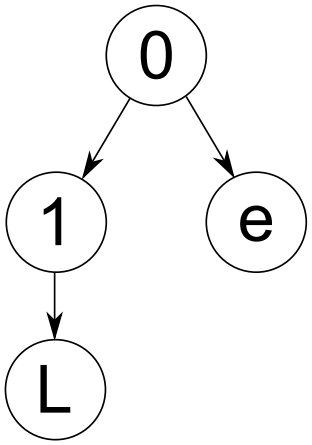
Začetno stanje je *0*. Možni prehodi stanj so trije:

***0* → *1*:** vsi vhodni parametri, ki so prihodnosti, so prešli v stanje *1*,

***1* → *L*:** vsi vhodni parametri, ki so objekti `lockable<T>`, so sporočili stanje `locked`,

***0* → *e*:** vsaj eden od vhodnih parametrov, ki je prihodnost, je prešel v stanje *e*.

Na sliki 5.1 so prikazani prehodi med stanji vzporednega klica z zaklepanjem virov.



Slika 5.1: Prehodi med stanji vzporednega klica z zaklepanjem virov.

Z dodajanjem novega stanja *L* povzročimo nevarnost, da smo ustvarili možnost smrtnih objemov. Vendar pa je razmeroma preprosto dokazati, da se to ne zgodi. Bodimo pozorni na zasnovo novih stanj *1* in *L*. Obe stanji sta povsem ločeni fazi v izvajanju vzporednih klicev:

- počakamo, da se čisto vse prihodnosti izračunajo, potem gremo v fazo zaklepanja,
- počakamo, da se vsi objekti `lockable<T>` zaklenejo, potem predamo vzporedni klic izvajalcu.

Izvorni zadostni pogoj za preprečitev smrtnih objemov je „vsi vzporedni klici se predajo v izvajanje“. Zaradi dodatnega stanja L je treba preoblikovati natančneje izražen izvorni pogoj „vsi vzporedni klici preidejo v stanje *1* ali e“ v „vsi vzporedni klici preidejo v stanje L ali e“. Upoštevamo dejstvo, da se postopek zaklepanja začne samo pri vzporednih klicih v stanju *1*, zato se pri dokazu omejimo samo na vzporedne klice v stanju *1*. Iz tega sledi, da je treba dokazati samo, da postopek zaklepanja ne povzroči smrtnega objema pri vseh podmnožicah vzporednih klicev v stanju *1*. Uporabimo zaklepanje večjega števila muteksov v vrstnem redu, ki upošteva globalno ureditev objektov `lockable<T>`. Najpreprostejša globalna ureditev pa je ureditev po kazalcih na pomnilnik, ki ga zasedajo objekti:

```
lockable<X1> x1; ... lockable<Xn> xn;
void *ordering[] = {(void*)x1.get(), ..., (void*)xn.get()};
std::sort(ordering, ordering + sizeof(ordering)/sizeof(void*));
```

Dokaz pa tukaj lahko zaključimo, saj je bila uporabljena dobro poznano klasično tehniko za preprečitev smrtnih objemov [26]. Slabosti pristopa z globalno ureditvijo vrstnega reda zaklepanja je več:

- Ni mogoče zaklepati objektov `lockable<T>` znotraj funkcij. Zaklep je možen samo prek vzporednega klica.
- Ni mogoče zaklepati objektov `lockable<T>` postopoma v odvisnosti od vhodnih parametrov. Treba je ustvariti nov vzporedni klic in trenutnega zaključiti, kar povzroči vmesni odklep vseh objektov `lockable<T>`.

5.3 Prednosti nadzora dostopa do virov v LibY

5.3.1 Preprečevanje smrtnega objema

Aktivni objekti ohranjajo osnovno lastnost LibY, tj. preprečevanje smrtnega objema. Aktivni objekti so v celoti implementirani z uporabo osnovnih prvin LibY. Zaklepanje virov z uporabo objektov `lockable<T>` pri vzporednih klicih ohranja lastnost preprečevanja smrtnega objema, kar je bilo tudi dokazano.

5.3.2 Prednosti razreda `lockable<T>`

Zaklepanje objektov `lockable<T>` se učinkovito implementira povsem brez uporabe muteksov, saj izvajalni model z dodatnim stanjem L v vzporednih

klicih poskrbi, da se v izvajanje preda največ en vzporedni klic za vsak objekt `lockable<T>`. Iz te zasnove sledi tudi ohranjanje lastnosti najboljše možne stopnje vzporednega izvajanja in najnižje možne porabe navideznega pomnilnika, saj ni niti v stanju čakanja.

Poglavje 6

Mrežni strežniki

V tem poglavju bodo predstavljene nekatere najpogostejše arhitekture mrežnih strežnikov. Na praktičnem primeru bo prikazana uporaba knjižnice LibY za implementacijo mrežnih strežnikov in iz tega izhajajočih prednostih.

6.1 Arhitekture mrežnih strežnikov

6.1.1 Delovne niti ali procesi

Arhitektura mrežnega strežnika z uporabo delovnih niti ali procesov je zelo pogosta v aplikacijskih spletnih strežnikih. Vsak zahtevek oddaljenega odjemalca se dodeli svoji niti ali procesu in se v celoti obdela. Rezultat zahtevka se po koncu obdelave pošlje odjemalcu. Prednosti:

- preprostost programiranja aplikacij. Ta prednost je najverjetneje poglavitni razlog za izjemo rast dinamičnih spletnih aplikacij v zadnjih 15 letih;
- pri uporabi procesov za obdelavo zahtevkov je strežnik zelo robusten, sesutja programa so omejena na samo en zahtevek.

Slabosti:

- najpogosteje se uporabljajo sistemske niti ali sistemski procesi za obdelavo vsakega zahtevka, kar pripelje do velike porabe virov operacijskega sistema;

- arhitektura neposredno ne predvideva medsebojne komunikacije med odjemalci. Običajno se v ta namen uporabi podatkovna baza ali sistem sporočilnih čakalnih vrst, kar pa lahko hitro privede do izgube poglavitne prednosti preprostosti.

6.1.2 Dogodkovni vhod/izhod

Arhitektura dogodkovnega vhoda/izhoda je diametralno nasprotje delovnih niti ali procesov. Najpogosteje se uporablja v strežnikih, ki so namenjeni obdelavi kratkih in preprostih zahtevkov zelo velikega števila hkratnih odjemalcev. Običajna izvedba takih strežnikov vsebuje eno samo nit, ki sprejema dogodke v zvezi z mrežnimi povezavami in ostalimi V/I napravami za vse odjemalce hkrati. Ta arhitektura je pogosta v infrastrukturi interneta (npr. strežniki DNS, NTP, ...) ali pa v spletnih strežnikih, ki so namenjeni posredovanju statičnih vsebin, kot so slike. Prednosti:

- zelo nizka poraba virov operacijskega sistema, kar omogoči veliko število hkratnih odjemalcev;

- preprosta in učinkovita medsebojna komunikacije med odjemalci v primeru, da so sporočila preprosta in ne povzročajo zaklepanja skupnih virov na strežniku.

Slabosti:

- uporaba skupnih virov z dogodkovnim zaklepanjem hitro veča zapletenost sistema;

- redko so vsa opravila dovolj preprosta, da bi jih lahko obdelali neposredno v dogodkovni niti. Uporabiti je treba delovne niti, kar hitro poveča zapletenost sistema.

V primeru, da ne potrebujemo zaklepanja skupnih virov, je preprosto doseči poljubno stopnjo vzporednega izvajanja. Poženemo lahko poljubno število dogodkovnih niti, kjer vsaka skrbi za obdelavo zahtevkov podmnožice odjemalcev. Običajno poskrbimo, da so odjemalci, ki pogosto komunicirajo med sabo, na isti dogodkovni niti. S tem zmanjšamo upočasnitev zaradi prenašanja predpomnilnikov med procesorskimi jedri.

6.1.3 Sporočilne čakalne vrste

Arhitektura sporočilnih čakalnih vrst je namenjena sistemom, ki omogočajo izmenjavo velike količine sporočil med odjemalci. Osnovna prvina arhitekture je čakalna vrsta, ki sprejema sporočila, jih obdeluje in posreduje v naslednjo čakalno vrsto. Običajno se uporablja v telekomunikacijskih sistemih in pri izmenjavi podatkov med poslovnimi sistemi. Mnogi e-poštni strežniki uporabljajo tako arhitekturo, saj neposredno odraža delovanje celotnega e-poštnega sistema. Prednosti:

- z modelom ena nit na čakalno vrsto lahko zelo preprosto in učinkovito obdelujemo sporočila, saj je vrst v netrivialnih sistemih običajno veliko več kot procesorskih jeder;
- sistem je zelo preprosto razširiti na več ločenih računalnikov, saj obdelava sporočil ne temelji na skupnem deljenem pomnilniku.

Slabosti:

- v primeru zapletenejše komunikacije med odjemalci je treba nositi veliko količino stanja v sporočilih, kar zniža prepustnost sistema in poviša zapletenost implementacije;
- sporočilne čakalne vrste, ki imajo netrivialna notranja stanja in ne obdelujejo sporočila samo v odvisnosti od informacij v samem sporočilu, povzročajo veliko nevarnost smrtnih objemov.

6.2 Simulacija strežnika večpredstavnih vsebin z LibY

Strežnik je simulacija vozlišča za pretvorbo in posredovanje živih video podatkovnih tokov. Vozlišče podpira dva tipa odjemalcev:

- ponudnika izvornega video podatkovnega toka, npr. spletna kamera;
- odjemalca videa, npr. aplikacija za videokonference.

Vsak ponudnik video vsebine ustvari novo sejo na strežniku, v katero se pridružijo ostali odjemalci in spremljajo živo predvajanje. Odjemalci prejemajo video podatkovni tok v različnih kvalitetah, saj se prenos izvaja prek internetnih povezav različnih prepustnosti. Znotraj vsake seje se zato ustvari več video pretvornikov, ki pošiljajo video podatkovni tok določeni podmnožici odjemalcev.

6.2.1 Opis izvedbe strežnika

Zaradi preprostosti je bila namesto zajema pravega videa iz kamer in zapletenih pretvorb video zapisov uporabljena knjižnica zlib [18] za splošno stiskanje poljubnih podatkov, ker je poglavitni namen simulacije prikaz pristopa k programiranju večpredstavnih strežnikov z LibY. Implementacija vsebuje naslednje datoteke:

liby.h: implementacija celotne knjižnice LibY. Manjka implementacija večkratnega zaklepanja `lockable<T>`, saj je v primeru večpredstavnega strežnika dovolj uporaba aktivnih objektov;

liby.cpp: preprosta implementacija splošnega izvajalca opravil;

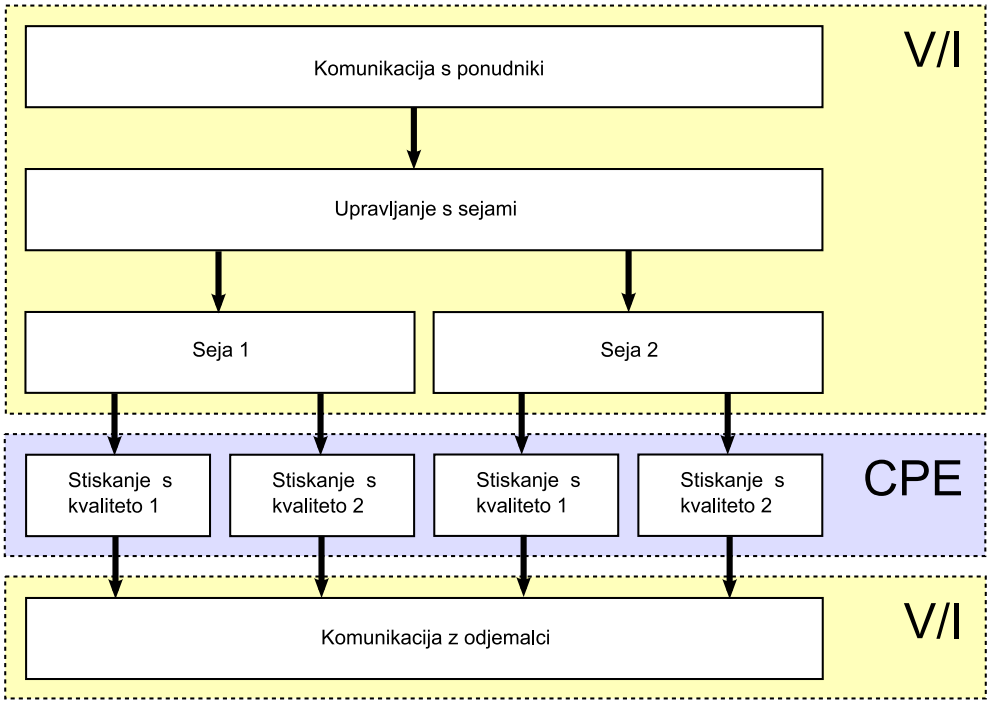
net.h, net.cpp, netp.cpp: implementacija mrežnega vmesnika za povezave TCP s pomočjo knjižnice boost::asio [3];

server.h, server.cpp, comp.h, comp.cpp: implementacija strežniškega dela, ki vsebuje ustvarjanje sej, stiskanje podatkovnega toka in posredovanje toka odjemalcem;

client.h, client.cpp, md5.h, md5.c: implementacija ponudnika in odjemalca podatkovnega toka.

Na sliki 6.1 je prikazan podatkovni pretok med podsistemi v strežniku. S kratico CPE je označen splošno-namenski izvajalec. S kratico V/I je označen izvajalec za vhodno-izhodne naloge.
Poglavitne prednosti, ki so razvidne iz izvedbe strežnika pri uporabi knjižnice LibY:

- Razmeroma preprosto je pokazati, da program ne vsebuje smrtnih objemov. Uporabniški del strežnika (datoteke server.cpp in client.cpp) ne vsebuje nobenih neinicializiranih prihodnosti `fut<T>`, iz česar sledi, da je treba podrobneje analizirati samo mrežni vmesnik v datoteki net.cpp. Mrežni vmesnik uporablja zunanjo knjižnico za komunikacijo z odjemalci in ne podpira knjižnice LibY, zato je žal potrebna uporaba neinicializiranih prihodnosti. V večjih programih bi mrežni vmesnik predstavljal zelo majhen del celotne izvorne kode, kar izjemno olajša analizo možnih smrtnih objemov. Gre za bistveno izboljšavo glede na strežnike z arhitekturo sporočilnih čakalnih vrst.



Slika 6.1: Podatkovni pretok med podsistemi v strežniku.

- Mrežni vmesnik v datoteki net.cpp je implementiran z uporabo več dogodkovnih niti vhod/izhod. S takim pristopom lahko poljubno določimo stopnjo vzporednega izvajanja neodvisno od števila odjemalcev. Podobno je implementirano stiskanje podatkovnega toka v datoteki server.cpp, saj uporablja običajne vzporedne klice `pcall`, ki se jim lahko poljubno določi izvajalca in stopnjo vzporednega izvajanja. Knjižnica LibY torej omogoča preprosto mešanje opravil vhoda in izhoda ter opravil centralne procesne enote brez umetnih omejitev, kar je bistvena izboljšava glede na arhitekturi delovnih niti ali procesov in dogodkovnega V/I.

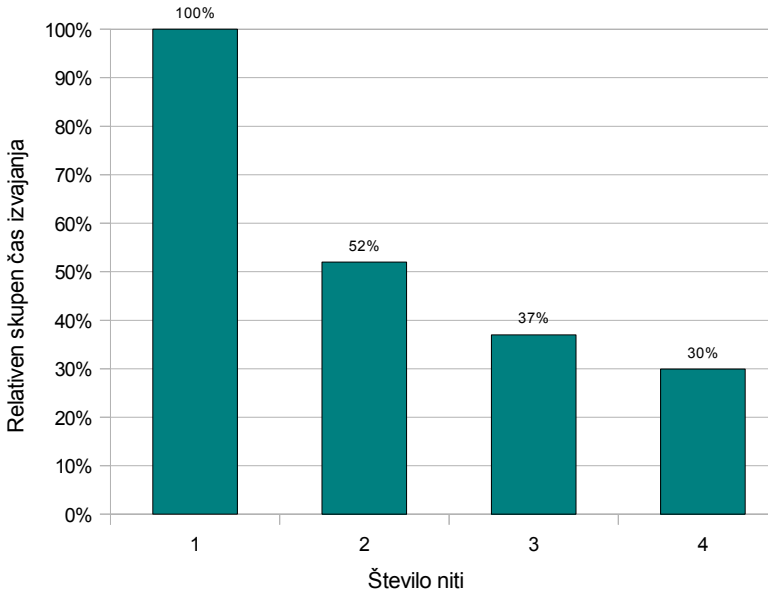
- Implementacija vsebuje malo izvorne kode, ki služi obravnavanju napak. To je posledica modela izvajanja v knjižnici LibY, ki se ga lahko predstaviti z acikličnim usmerjenim grafom. Napake se samodejno prenašajo globlje v graf, kar je bistvena izboljšava glede na dogodkovni V/I in mnoge izvedbe sporočilnih čakalnih vrst.

Glavna slabost pa je razmeroma zapletena implementacija in uporaba knjižnice LibY v programskem jeziku C++. Resno težavo predstavljajo neberljiva

prevajalnikova sporočila o napakah in okorna sintaksa za vzporedne klice metod objektov. Za učinkovitejše programiranje je treba zasnovati nov jezik, ki temelji na podatkovno pretokovnem programiranju, ali pa dodati podatkovno pretokovno razširitev obstoječemu čistejšemu jeziku. Primer slednjega predstavlja Microsoft F# z dodanimi asinhronimi bloki in spremenljivkami [2], ki je dialekt dobro znane družine jezikov ML.

6.2.2 Preizkus delovanja

Preizkus delovanja strežnika je bil izveden na štirijedrnem procesorju. Ustvarjenih je bilo 12 sej, vsaka s štirimi odjemalci z naključno stopnjo stiskanja podatkovnega toka. Meritev vsebuje skupen čas za prenos podatkovnega toka od ponudnikov do odjemalcev, vključno s stiskanjem. Pri večanju števila niti splošnega izvajalca lahko opazimo značilen, skoraj linearen pospešek izvajanja, kar potrdi pravilno zasnovo strežnika. Večanje števila niti v mrežnem vmesniku ni imelo bistvenega vpliva na hitrost izvajanja, saj predstavlja pošiljanje V/I zahtevkov operacijskemu sistemu razmeroma majhen del izvajanja. Na sliki 6.2 je prikazan skupen čas izvajanja v odvisnosti od števila niti v splošnem izvajalcu za procesorsko zahtevna opravila. Z eno nitjo je bil skupen čas izvajanja 127 sekund.



Slika 6.2: Skupen čas izvajanja v odvisnosti od števila niti.

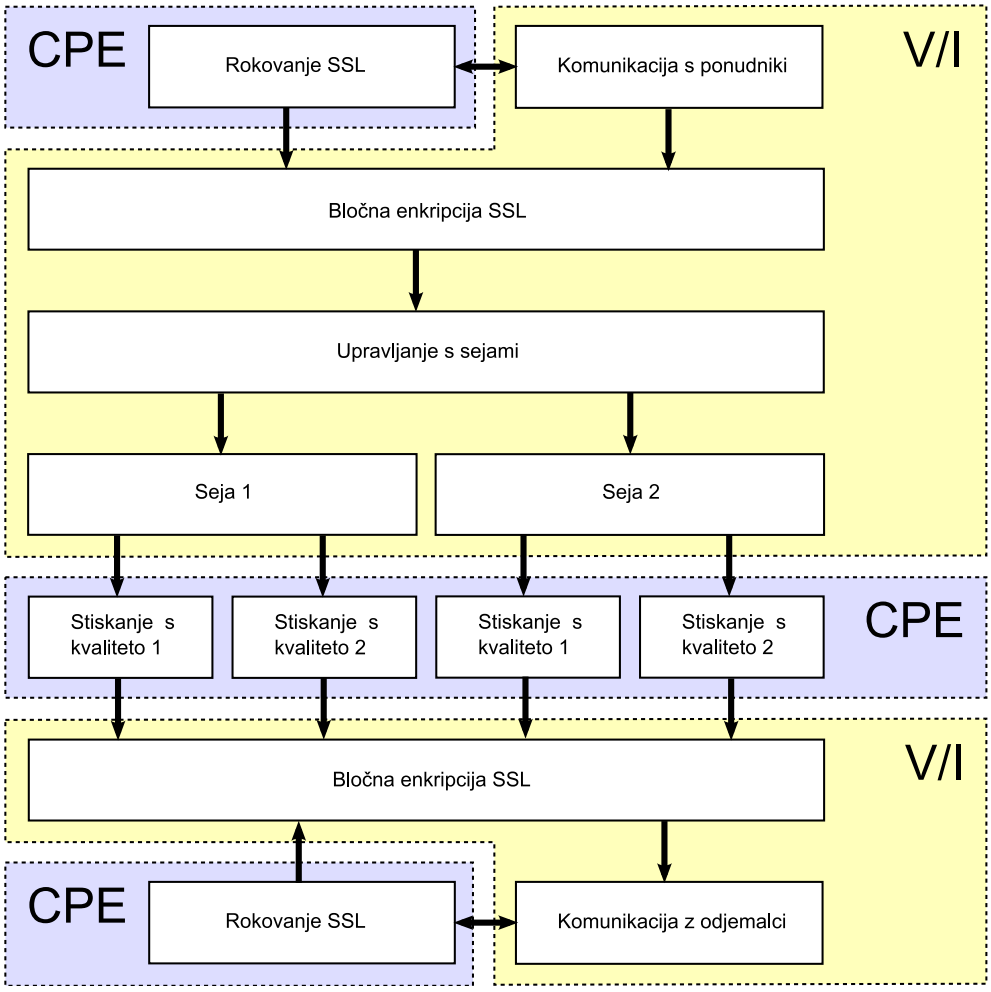
6.2.3 Razširitev strežnika z enkripcijo podatkovnih tokov

V prejšnjem odstavku je podana ugotovitev, da večanje števila V/I niti ni imelo bistvenega vpliva na hitrost izvajanja. Razlog za to so izjemno kratka opravila, ki obvestijo operacijski sistem, naj prenese določen del pomnilnika na mrežno povezavo TCP. Če pa bi želeli zaradi večje varnosti in zasebnosti še kriptirati podatkovne tokove po standardu SSL, lahko izberemo naslednji dvodelni pristop:

- Na začetku povezav SSL je vedno procesorsko zelo potratno rokovanje med odjemalcem in strežnikom z uporabo asimetričnih kriptografskih algoritmov, kot sta RSA in Diffie-Hellman. Rokovanje zato običajno predamo v obdelavo splošnemu izvajalcu, da ne ustvarjamo nepotrebnih časovnih zamikov v V/I nitih.

- Ko se rokovanje preneha, povezava SSL preklopi v način prenosa podatkov z bločnim kriptiranjem. Bločni kriptografski algoritmi s simetričnimi ključi so zasnovani za doseganje velikih hitrosti, zato je smiselno opravljati kriptiranje podatkovnih blokov kar v V/I nitih. Kljub temu pa so današnja procesorska jedra sposobna enkripcije s hitrostjo približno 100 MB/s, kar je bistveno manj od hitrosti prenosov na pomnilniških vodilih. Hkrati moderni strojni mrežni vmesniki dosegajo podobne hitrosti, zato je upravičena uporaba več V/I niti.

Na sliki 6.3 je prikazan podatkovni pretok med podsistemi v strežniku z razširitvijo SSL. S kratico CPE je označen splošno-namenski izvajalec. S kratico V/I je označen izvajalec za vhodno-izhodne naloge.



Slika 6.3: Podatkovni pretok med podsistemi v strežniku z razširitvijo SSL.

Poglavje 7

Sklepne ugotovitve

Pri izdelavi tega diplomskega dela sem motivacijo našel v porastu dostopnosti večjedrnih procesorjev za osebne računalnike in strežnike ter hkrati večji pomembnosti podpore za internetne tehnologije v programski opremi v zadnjem desetletju. Pri pristopu k rešitvi zastavljenih problemov sem uporabil izkušnje, ki sem jih pridobil pri sedem-letnem delu v oddelku ISL Online družbe XLAB d. o. o., v kateri sem bil odgovoren za razvoj komunikacijskih uporabniških programov in internetnih strežnikov. V teh letih dela je bilo treba z dobrim skupinskim delom rešiti veliko tehničnih in netehničnih problemov. Nanizali smo kar nekaj uspehov in priznanj[1], kar nas je pripeljalo do kupcev iz celega sveta, vključno z ZDA, Evropo in Japonsko. Naše omrežje ta trenutek zajema 20 strežnikov, razpršenih po celem svetu, ki so namenjeni zagotavljanju hitrega delovanja storitev 70 000 registriranim uporabnikom. S prodorom do večjih kupcev, kot so trgovske verige, banke in državne institucije z lastnimi strežniki, pa število 10 000 uporabnikov na posamezen strežnik ni več redkost, kar predstavlja svojevrsten izziv.

Na kratko povzamimo prednosti, ki izhajajo iz teoretičnega modela knjižnice LibY:

- v primeru uporabe samih inicializiranih prihodnosti `fut<T>` je zagotovljena odsotnost smrtnega objema;
- stopnjo vzporednega izvajanja v programu lahko z uporabo vzporednih klicev `pcall` določimo na vrednost, ki zagotavlja najhitrejše izvajanje

[1] c't (Computertechnik) magazin, priznana nemška računalniška revija z več kot milijonom bralcev, je preizkusila in ocenila delovanje 22-ih programskih rešitev za računalniško podporo na daljavo. Med vsemi preizkušenimi kandidati je naš izdelek ISL Light prejel najvišjo oceno (7/8 zvezdic). [14]

in najmanjšo možno porabo navideznega pomnilnika, kar je predvsem pomembno v 32-bitnih računalniških arhitekturah;

- poenoten model izvajanja za procesorsko zahtevna opravila in vhodni-izhodna opravila, kar omogoča preprostejši razvoj procesorsko zahtevnih in porazdeljenih internetnih aplikacij;
- vzporedne klice `pcall` je mogoče učinkovito implementirati samo z uporabo atomičnih dostopov do pomnilnika brez muteksov;
- metoda `fut<T>::register_notify` omogoča učinkovito preslikavo na prvine operacijskega sistema;
- razširitev običajne metode `task::run` s povratnim klicem, ki obvesti izvajalca o skorajšnjem koncu izvajanja opravila, omogoča učinkovitejšo razporejanje opravil po strojnih nitih;
- z uporabo aktivnih objektov dosežemo učinkovit in preprost nadzor nad skupnimi viri. Če ta pristop ne zadostuje, lahko uporabimo splošno zaklepanje, ki ravno tako zagotavlja odsotnost smrtnih objemov;
- splošno zaklepanje je mogoče implementirati samo z uporabo atomičnih dostopov do pomnilnika brez muteksov.

Iz praktičnega primera simulacije večpredstavnega strežnika je razvidno, da uporaba knjižnice LibY ponuja večino prednosti različnih arhitektur strežnikov, kot so delovne niti ali procesi, dogodkovni vhod/izhod in sporočilne čakalne vrste. Kot poglavitno slabost knjižnice LibY, implementirane v jeziku C++, pa lahko izpostavimo neberljiva sporočila o napakah od prevajalnika in okorno sintakso klicev metod aktivnih objektov. Navedeni težavi se lahko reši z razvojem lastnega jezika, ki temelji na podatkovno pretokovnem programiranju, ali pa z razširitvijo čistejšega obstoječega programskega jezika, kot je npr. ML.

7.1 Primerjava s sorodnimi knjižnicami

Intel threading building blocks [10] je knjižnica, ki je namenjena predvsem računsko zahtevnim nalogam. Vsebuje veliko nizkonivojskih prvin, ki olajšajo večnitno programiranje, saj zadnji uradni standard za programski jezik C++ iz leta 1998 ne vsebuje vzporednega programiranja. Sledijo prednosti knjižnice LibY pred TBB:

- TBB vsebuje precej uporabnih konstruktov, ki omilijo možnost nastopov smrtnih objemov, vendar smrtni objemi niso povsem odpravljeni kot v knjižnici LibY.
- TBB neposredno ne obravnava vhodno-izhodnih opravil. LibY ponuja enoten model izvajanja mešanih množic računsko zahtevnih in vhodno-izhodnih opravil.
- Knjižnica LibY zmanjša število čakajočih niti na najmanjšo možno vrednost in s tem zniža porabo navideznega pomnilnika. TBB vsebuje veliko abstrakcij, ki temeljijo na čakajočih niti.

Knjižnico TBB je smiselno uporabiti, če je večina opravil računsko zahtevnih. Večina omenjenih prednosti knjižnice LibY se resnično izrazi šele pri snovanju programske opreme z veliko vhodno-izhodnimi opravili. Na svojem področju je knjižnica TBB tudi priročnejša od LibY, saj vsebuje abstrakcije kot je vzporedna zanka „for“. Zaradi večjega števila nizkonivojskih prvin v TBB bi bilo smotrno uporabiti TBB za implementacijo knjižnice LibY. S takim pristopom združimo prednosti obeh knjižnic.

Microsoft concurrency and coordination runtime [4] rešuje probleme iz istega področja kot knjižnica LibY z razmeroma podobnimi ukrepi:

- večje naloge naj bodo razbite na krajša asinhrona opravila,
- izvajanje opravil je ločeno od same opredelitve opravil,
- prenašanje podatkov iz računsko zahtevnih in vhodno-izhodnih opravil je povsem enako,
- obveščanje o napakah naj bo preprosto in robustno.

Microsoft CCR vsebuje bistveno večje število izvajalcev in mehanizmov za predajo podatkov v obdelavo kot knjižnica LibY. Vendar pa knjižnica CCR temelji na uporabi razreda „Port“, ki je podatkovna čakalna vrsta. Dobro znana slaba lastnost čakalnih vrst pa je možnost smrtnih objemov, kar je bilo pogosto omenjeno v literaturi o programiranju koprocesov s cevmi na operacijskem sistemu UNIX [24]. Kot zanimiv primer lahko navedem tudi prvi zadetek poizvedbe „ccr deadlock“ na iskalniku Google, ki je forumski pogovor z naslovom „Deadlock issue in CCR?“ [5]:

> Phillip Palk: I'm sure this is fairly unlikely, but I've come across what *appears* to be a deadlock in the CCR. I have an interleave

> that has 3 receivers on it, 2 concurrent (C1, C2) and 1 exclusive (E). ... Firstly, should this work? Exclusive handler's don't run until all concurrent handlers have completed, but do they prevent new concurrent handlers from starting (this would clearly cause a deadlock in my situation)? Is this a scenario that's been tested? I've not yet tried to produce a basic example that reproduces the deadlock, I figured I'd ask here first. If this is by design (and if so, why!?) or if this is a well tested scenario I'll look elsewhere, but if not I'll try and reproduce it with an example 'plain' CCR program.
>
> Trevor Taylor, MSFT: I believe that what you are seeing is the correct behavior. ... CCR does not eliminate deadlocks. You can still design systems that will deadlock :-(...

Knjižnica LibY z odpravo smrtnih objemov bistveno olajša izvedbo vzporednih aplikacij tako neizkušenim programerjem kot arhitektom večjih sistemov.

Primerjava paketa java.util.concurrent [12] s knjižnico LibY ni potrebna, saj paket vsebuje samo podmnožico abstrakcij iz TBB in CCR.

7.2 Prihodnje delo

7.2.1 Vpliv predpomnilnikov na izvajanje

Treba bi bilo bolje raziskati vplive predpomnilniške arhitekture pri večprocesorskih računalniških sistemih. V tem delu ni bilo veliko napisanega o sami zasnovi izvajalcev. Opisana je bila metoda `task::status::task_finished`, ki obvesti izvajalno nit o koncu izvajanja opravila. To obvestilo je mogoče koristno uporabiti, da zadržimo nadaljnjega opravila na istem procesorju in s tem zmanjšamo vpliv prenašanja podatkov med različnimi procesorji. Tak pristop je preprost za implementacijo, vendar bi bilo treba izmeriti resničen vpliv na uporabo predpomnilnika in podati primerjave z že znanimi izvajalci. V literaturi se pogosto omenja Cilk++, ki je razširitev za jezik C++, namenjena vzporednemu programiranju. Cilk++ implementira algoritem razporejanja opravil na procesorje „kraja dela“ [28]. Podoben algoritem vsebuje tudi knjižnica Intel threading building blocks [17].

7.2.2 Razporejevalnik opravil za porazdeljeno izvajanje

Knjižnica LibY poda zelo preprosto in naivno izvedbo porazdeljenega izvajanja. Treba bi bilo zasnovati naprednejši razporejevalnik opravil na oddaljena

vozlišča. Pri tem je potrebno upoštevati več meril:

- zamik in pasovna širina prenosa podatkovnega kanala,
- procesorska moč oddaljenega vozlišča in trenutna obremenjenost.

Najverjetneje bi bilo treba razširiti razred prihodnosti `fut<R>`, da bi vseboval dodatne informacije o oddaljenosti podatka, ki ga nosi. Vzporedne klice bi bilo treba razširiti tudi z informacijo o predvideni normalizirani zahtevnosti opravila, ki ga je potem mogoče izračunati za konkreten tip procesorja. Ugotavljanje predvidene zahtevnosti opravila je mogoče izvesti s profilom tipičnega izvajanja, kar na primer uporabljajo prevajalniki za optimizacije. S pomočjo teh dodatnih informacij bi bilo mogoče bolje razporejati prenos podatkov med vozlišči. Morebiti se izkaže, da bi bil tak algoritem dovolj dober tudi za razporejanje opravil med procesorje in pomnilniške pretoke na lokalnem računalniku. Dobili bi torej povsem splošen algoritem, ki je znotraj istega ogrodja zmožen obravnavati:

- pasovno širino predpomnilnikov, glavnega pomnilnika in omrežnih povezav,
- zamik pri prenosu podatkov med pomnilniki v lokalnem in oddaljenih računalnikih,
- procesorsko moč in trenutno obremenjenost procesorjev v lokalnem in oddaljenih računalnikih.

Posplošen algoritem bi znal odgovoriti na klasično vprašanje, „ali je bolje počakati na prost procesor na lokalnem računalniku ali je bolje prenesti opravilo na oddaljeno vozlišče kljub večjemu zamiku in manjši pasovni širini“.

7.2.3 Izvedba prvin z atomičnimi spremenljivkami

V delu je predstavljeno nekaj prvin, ki so običajno implementirane z uporabo muteksov. Vendar to ne velja za prvino vzporedni klic, saj je bila opisana možna izvedbo z uporabo atomičnih števcev. Treba bi bilo razmisliti, ali je brez uporabe muteksov mogoče implementirati tudi prihodnosti `fut<T>` in vire `lockable<T>`. Glavno težavo pri prihodnosti povzroča metoda `register_notify`, ki vodi seznam objektov, ki morajo biti obveščeni o spremembi stanja. Podobno velja tudi za metodo `try_lock` pri virih. Znanih je nekaj podatkovnih struktur, ki uporabljajo atomične dostope do pomnilnika, da zagotovijo varno uporabo iz več niti. Če bi uporabili še izvajalca, ki za notranjo sinhronizacijo

ne potrebuje muteksov, bi lahko implementirali celotno jedro knjižnice LibY brez enega samega elementa, ki povzroča čakanje niti.

7.2.4 Splošnejši nadzor dostopa do virov

Predstavljena sta bila dva pristopa k nadzoru virov. Oba pristopa preprečita možnost smrtnega objema, vendar pri tem žrtvujemo splošnost zaklepanja. Treba bi bilo raziskati splošnejše metode zaklepanja, ki preprečujejo smrtne objeme. Alternativna možnost pa so drugi pristopi k reševanju smrtnih objemov, kot je zaznavanje smrtnih objemov, ki mu sledi sporočanje napake.

7.2.5 Odprava neinicializiranih prihodnosti

Kot razlog za težave s smrtnimi objemi je bila podana uporaba neinicializiranih prihodnosti. Treba bi bilo razmisliti, ali obstaja kakšen varnejši način za implementacijo stikov z operacijskim sistemom in drugimi knjižnicami, ki ne temeljijo na LibY. Dodatna prednost bi bila implementacija čakalne vrste FIFO brez uporabe neinicializiranih prihodnosti.

Slike

Literatura

[1] (2009) Asynchronous procedure calls. Dostopno na: http://msdn.microsoft.com/en-us/library/ms681951%28VS.85%29.aspx

[2] (2009) Asynchronous workflows (F#). Dostopno na: http://msdn.microsoft.com/en-us/library/dd233250%28VS.100%29.aspx

[3] (2009) Boost C++ libraries. Dostopno na: http://boost.org/

[4] (2009) CCR introduction. Dostopno na: http://msdn.microsoft.com/en-us/library/bb648752.aspx

[5] (2009, okt.) Deadlock issue in ccr? Dostopno na: http://social.msdn.microsoft.com/Forums/en-US/roboticsccr/thread/fc2e890d-4013-4941-aece-e8dbab407fb9

[6] (2009) Erlang programming language, official website. Dostopno na: http://erlang.org/

[7] (2009) Executor (Java platform SE 6). Dostopno na: http://java.sun.com/javase/6/docs/api/java/util/concurrent/Executor.html

[8] (2009) Future (Java platform SE 6). Dostopno na: http://java.sun.com/javase/6/docs/api/java/util/concurrent/Future.html

[9] (2009) Haskell - HaskellWiki. Dostopno na: http://haskell.org/

[10] (2009) Intel threading building blocks. Dostopno na: http://www.threadingbuildingblocks.org/

[11] (2009) io. Dostopno na: http://www.iolanguage.com/

[12] (2009) java.util.concurrent (Java platform SE 6). Dostopno na: http://java.sun.com/javase/6/docs/api/java/util/concurrent/package-summary.html

[13] (2009) JTC1/SC22/WG21 - the C++ standards committee. Dostopno na: http://www.open-std.org/jtc1/sc22/wg21/

[14] (2009) Najboljši rezultat na testu c't. Dostopno na: http://islonline.com/ct/

[15] (2009) The OpenMP API specification for parallel programming. Dostopno na: http://openmp.org/

[16] (2009) Synchronization and overlapped input and output. Dostopno na: http://msdn.microsoft.com/en-us/library/ms686358%28VS.85%29.aspx

[17] (2009) TBB documentation. Dostopno na: http://www.threadingbuildingblocks.org/documentation.php

[18] (2009) zlib home site. Dostopno na: http://www.zlib.net/

[19] H. J. Boehm. (2008, apr.) Getting C++ threads right. Dostopno na: http://www.hpl.hp.com/personal/Hans_Boehm/

[20] W. Chen, Z. Ying, in Z. Defu, "An efficient method for expressing active object in C++," *SIGSOFT Softw. Eng. Notes*, zv. 25, št. 3, str. 32–35, 2000.

[21] E. G. Coffman, M. Elphick, in A. Shoshani, "System deadlocks," *ACM Comput. Surv.*, zv. 3, št. 2, str. 67–78, 1971.

[22] U. Drepper. (2007, nov.) What every programmer should know about memory. Dostopno na: http://people.redhat.com/drepper/cpumemory.pdf

[23] G. Duarte. (2009, jan.) Anatomy of a program in memory. Dostopno na: http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory

[24] F. Faruqui. (2002, jul.) Introduction to interprocess communication using named pipes. Dostopno na: http://developers.sun.com/solaris/articles/named_pipes.html

[25] S. Franklin. (2007) XML parsers: DOM and SAX put to the test. Dostopno na: http://www.devx.com/xml/Article/16922/1954

[26] J. W. Havender, "Avoiding deadlock in multitasking systems," *IBM Systems Journal*, zv. 7, št. 2, str. 74–84, 1968.

[27] D. Leijen in J. Hall. (2007, okt.) Optimize managed code for multi-core machines. Dostopno na: http://msdn.microsoft.com/en-us/magazine/cc163340.aspx

[28] C. E. Leiserson, "The Cilk++ concurrency platform," v *DAC '09: Proceedings of the 46th Annual Design Automation Conference*. New York, NY, USA: ACM, 2009, str. 522–527.