Višedretveno programiranje u C++-u na bazi slanja poruka

Lasić, Anđela

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Anđela Lasić

VIŠEDRETVENO PROGRAMIRANJE U C++-U NA BAZI SLANJA PORUKA

Diplomski rad

Voditelj rada:
prof. dr. sc. Mladen Jurak

Zagreb, srpanj, 2022

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Zahvaljujem se svom mentoru prof. dr. sc. Mladenu Juraku na izdvojenom vremenu, konstruktivnim kritikama i savjetima, pomoći i strpljenju pri izradi ovog diplomskog rada. Hvala Jasmini, Ivani, Mariji, Ani i Kristini za sve lijepe i one manje lijepe trenutke koje smo skupa proživjele tijekom studija. Hvala mojoj obitelji na bezuvjetnoj ljubavi, podršci i vjeri u mene. Najveća hvala mužu Ivanu bez kojeg ovo što sam postigla ne bi bilo moguće.

Sadržaj

Uvod

Ogromni napredak u razvoju procesorske snage nalazi se u središtu gotovo svih najvećih postignuća u polju znanosti, interneta i zabave. Dekodiranje ljudskog genoma, visoko pouzdana kriptografska zaštita mobilnog prometa, 5G ili vrlo realistične kompjutorske igre ne bi bile moguće bez takvog napretka.

Do sredine 80-ih godina prošlog stoljeća rast performansi monolitnih procesora bio je vezan za razvoj tehnologije i kretao se oko 25% na godišnjoj razini. Od tada pa do 2002. godine taj je rast iznosio približno 52% godišnje čemu su doprinijele naprednije arhitekturalne i organizacijske ideje. Najveće zasluge u tome preuzima dizajn tranzistora, glavne gradivne jedinice integriranog kruga. Sve većim smanjivanjem tranzistora povećavala se njegova brzina. Međutim, povećanjem brzine došlo je i do povećanja potrošnje energije, najvećim dijelom u obliku zagrijavanja, što je u konačnici dovelo do povećanja nepouzdanosti integriranog kruga [1].

Tako je nakon 2002. godine rast ponovo usporio na 20% godišnje, što je dovelo do ideje o ulaganju u razvoj višejezgrenih procesora [1].

Ova promjena dovela je do prekretnice u razvoju programske podrške, gdje je odjednom trebalo početi razvijati programe za višejezgrenu okolinu, koja se uvelike razlikuje od slijedno izvođenih programa.

U ovom diplomskom radu pokušat ću dati pregled metoda za razvoj programa u višejezgrenoj okolini te usporediti višedretveno programiranje s dijeljenom memorijom i programiranje na bazi slanja poruka.

Poglavlje 1

Proces

1.1 Što je proces

U računarstvu, proces (engl. *process*) predstavlja instancu računalnog programa kojeg izvodi jedna ili više programskih niti (engl. *thread*). Drugim riječima, proces predstavlja izvršavanje programa što omogućuje izvođenje pripadnih akcija opisanih u programu. Operacijski sustav (engl. *Operational System*) kreira, vremenski raspoređuje i prekida proces kojeg koristi centralna procesorska jedinica (engl. *Central Processing Unit*).

Proces je izolirana izvedbena cjelina koja ne dijeli podatke i informacije s drugim procesorima unutar jednog sustava, pa tako svaki proces ima zaseban stog (engl. *stack*), heap memoriju i podatkovnu mapu (engl. *data map*).

Memorija koju operacijski sustav dodjeljuje procesu pripada samo tom procesu i drugi procesi nemaju pristup toj memoriji. Pokuša li neki drugi proces pristupiti toj memoriji, podsustav operacijskog sustava zadužen za upravljanje memorijom će to zabraniti i u pravilu će to završiti rušenjem programa koji je pokušao nedopušteni pristup.

Kako bi proces razmjenjivao podatke i informacije s drugim procesima u svojoj okolini koriste se mehanizmi međuprocesne komunikacije (engl. *Inter-Process Communication*) [4].

1.2 Međuprocesna komunikacija

Mehanizmi koji su u vlasništvu operacijskog sustava, a služe omogućavanju razmjene informacija među procesima nazivaju se međuprocesna komunikacija. Takvom komunikacijom jedan proces obavještava drugi proces o događaju (engl. *event*) koji se prethodno dogodio ili jedan proces prenosi podatke drugom kako bi se osiguralo sinkronizirano obavljanje zajedničke zadaće.

Tablica 1.1: Međuprocesna komunikacija u UNIX sustavima

Ime	**Opis**	**Doseg**	**Korištenje**
Datoteka	Podaci se upisuju i čitaju u tipičnu UNIX datoteka. Podžava komunkaciju među proizvoljnim brojem procesa.	Lokalno	Dijeljenje velikih podatkovnih skupova.
Cjevovod	Podaci se prenose između dva procesa dodijeljenim opisnicima (engl. *file descriptors*). Moguće je prenositi podatke samo između procesa roditelj-dijete.	Lokalno	Jednostavno dijeljnje podataka, kao u primjeru proizvođač - potrošač.
Imenovani cjevovod	Podaci se prenose preko dodijeljenih opisnika između bilo koja dva procesa koja se izvode na jednom domaćinu (engl. *host*).	Lokalno	Proizvođač - potrošač ili naredba - kontrola
Signal	Prekid obavještava aplikaciju o specifičnom događaju.	Lokalno	Upravljanje procesom
Dijeljena memorija	Podaci se razmjenjuju upisivanjem i čitanjem iz zajedničkog bloka memorije.	Lokalno	Obavljanje zajedničkih zadaća svih vrsta, pogotovo kad je bitna sigurnost.
Utičnica	Podaci se prenose korištenjem standardiziranih operacija za slanje i primanje poruka.	Lokalno i udaljeno	Mrežni servisi kao što su FTP, SSH, Apache Web Server, ...

Poglavlje 2

Programske niti

Programska nit (engl. *thread*) je slijedni skup instrukcija koje se izvode unutar jednog procesa. Time je programska nit sastavnica procesa te predstavlja jednu njegovu izvedbenu jedinicu. To znači da proces može biti sastavljen od jedne ili više programskih niti od kojih svaka može raditi paralelno ili konkurentno, ovisno o arhitekturi procesorske jedinice. Budući da svaki proces ima vlastiti adresni prostor, sve programske niti unutar jednog procesa dijele taj adresni prostor, što znači da je vrlo jednostavno dijeliti resurse među programskim nitima. Vidjet ćemo da postupak dijeljenja resursa među programskim nitima nije jednostavna operacija, pa će stoga biti potrebno razmisliti o različitim mehanizmima koji će nam osigurati sigurno dijeljenje resursa među više programskih niti. Druga važna zadaća bit će osiguravanje sinkronog rada među programskim nitima, kako bi se izvođenje složenijih poslova moglo rasporediti na više programskih niti.

2.1 Podaci u višedretvenom okruženju

Kada pokrenemo neki program, operacijski sustav će pokrenuti proces unutar kojeg će se program izvoditi. Operacijski sustav svakom procesu dodjeljuje određeni komad memorije te dodjeljuje vremenske okvire unutar kojih se oni izvode. Ti se okviri izmjenjuju vrlo brzo, što omogućava da istovremeno bude pokrenuto nekoliko programa i da se korisnik može jednostavno prebacivati između njih. Pritom svaki od programa može nešto raditi čak i kada ga korisnik ne drži aktivnim.

Unutar svakog procesa može se pokrenuti više programskih niti. Svaka od niti dobiva određeni vremenski okvir unutar kojeg se izvodi. Međutim, za razliku od procesa, različite programske niti unutar procesa koriste isti memorijski prostor. Ovo može biti poprilično nezgodno ako više programskih niti istovremeno pristupa nekom podatku. Ilustrirajmo to jednostavnim primjerom: pretpostavimo da imamo objekt koji sadrži podatak o nekoj osobi s imenom i prezimenom kao zasebnim članovima. U memoriji se trenutno nalaze

podaci za osobu ”Pero Perić”. U određenom trenutku, jedna programska nit pokušava učitati podatke o toj osobi, no istovremeno druga programska nit mijenja ime te osobe u ”Ivo Ivić”. Budući da se radi o složenom objektu koji se sastoji od niza podataka, procesi njegova zapisivanja, odnosno učitavanja se odvijaju u više taktova procesora. Zbog toga se može dogoditi da programska nit koja učitava podatak iz memorije pročita djelomično stari (npr. ime ”Pero”), ali i dio novog sadržaja (npr. prezime ”Ivić”) koji druga programska nit upravo zapisuje. Ovo će rezultirati pogrešnim rezultatom koji u konačnici može biti poguban za korisnika programa, ali sam program će vjerojatno nastaviti funkcionirati normalno. Međutim, ako je podatak, koji programske niti istovremeno obrađuju, pokazivač na neku adresu, može se dogoditi da očitana adresa bude složena od dijelova dviju potpuno različitih adresa te da zbog toga program pokuša pristupiti nedozvoljenoj adresi.

Napomenimo da je gornji opis značajno pojednostavljen. Naime, istovremeni pristup memoriji iz dvije različite programske niti nije moguć - to nije izvedivo na razini elektroničkih komponenti računala. Ono što se doista može dogoditi jest da programskoj niti, koja čita podatke i pročita ime osobe, a još nije pročitala njezino prezime, u tom trenutku završi vremenski okvir koji joj je operacijski sustav dodijelio. Pokreće se programska nit koja zapisuje podatke i ona prepisuje ime i prezime osobe. Nakon nekog vremena ponovo se pokreće programska nit koja čita podatke i nastavlja s čitanjem od mjesta gdje je stala, a to je prezime osobe koje je u međuvremenu druga programska nit promijenila [6].

Ovakve situacije se obično nazivaju stanjem natjecanja jer se dvije programske niti natječu koja će brže obraditi podatak, no više o tome u nastavku.

2.2 Problemi sa dijeljenjem podataka između programskih niti

Svi problemi koji nastaju prilikom dijeljenja podataka između programskih niti javljaju se zbog toga što programske niti izmjenjuju zajedničke podatke. Kada bi svi dijeljeni podaci bili samo za čitanje (engl. *read-only*), tada ne bi postojali nikakvi problemi, zato što podatak kojeg čita jedna nit ostaje nepromijenjen bez obzira čitala njega neka druga nit ili ne.

Dakle, ako postoji podatak koji je dijeljen između programskih niti, i ako jedna ili više programskih niti nastoje izmijeniti taj podatak, javlja se velika vjerojatnost za pojavu greške. U takvim slučajevima programer mora osigurati da se sve operacije točno izvrše.

Jedan koncept koji se široko primjenjuje kako bi se izbjegli problemi je koncept *nepromjenjivih iskaza* (engl. *invariant statements*). Takvi su iskazi uvijek istiniti kada se radi o određenoj strukturi podataka, npr., iskaz koji uvijek govori koliko članova ima određena struktura. Iskazi često postanu neistiniti u trenucima kada se struktura podataka mijenja, tj. kada se dodaje ili briše njezin član.

Uzmimo na primjer dvostruko povezanu listu, u kojoj svaki čvor drži pokazivač na sljedeći i prethodni član. Neka jedan od nepromjenjivih iskaza kaže da ako pratimo sljedbenika od čvora A do drugog čvora B, prethodnik od čvora B pokazuje nazad prema prvom čvoru A. Ako sada želimo izbrisati čvor iz liste, čvorovi na obje strane moraju biti izmijenjeni kako bi pokazivali jedan na drugoga. Jednom kada se jedan čvor izmjeni, nepromjenjivi iskaz nije istinit sve dok se i drugi čvor ne izmjeni. Nakon što je izmjena napravljena, nepromjenjivi iskaz je ponovo istinit.

2.3 Stanja natjecanja

Promotrimo još jedan fenomen koji se može javiti prilikom dijeljenja resursa između programskih niti.

Pretpostavimo da osoba kupuje ulaznice za kino. Kino dvorana je velika pa više blagajnika istovremeno prodaje ulaznice, što znači da više osoba može istovremeno kupiti ulaznice. Ako druga osoba kupuje ulaznicu kod drugog blagajnika u isto vrijeme kad i prva, skup dostupnih sjedala ovisit će o tome koja osoba prva kupi ulaznice. Pretpostavimo sada da je ostalo vrlo malo slobodnih sjedala, što znači da vremenska razlika između izbora sjedala može biti krucijalna - tj., može se doći u situaciju gdje se doslovno natječete kako bi pribavili zadnju ulaznicu. To je primjer stanja natjecanja (engl. *race condition*).

U konkurentnom programiranju, stanja natjecanja predstavljaju bilo koju situaciju gdje izlaz ovisi o relativnom kronološkom poretku izvršavanja operacija dvije ili više programske niti. U većini slučajeva rezultat je benigan jer svi mogući izlazi su prihvatljivi, čak iako se mogu promijeniti ovisno o poretku. Na primjer, ako dvije programske niti dodaju elemente u red za obradu, načelno nije bitno koji element ulazi u red kao prvi, a koji kao drugi.

Problematična stanja natjecanja tipično se pojavljuju kada je izvršenje operacije zahtjeva promjenu dva ili više potpuno nezavisna podatka, kao što su, na primjer, pokazivači u primjeru s dvostruko povezanom listom. Budući da operacija mora dohvaćati podatke s dva odvojena mjesta, podaci moraju biti izmijenjeni u zasebnim instrukcijama, i druga programska nit potencijalno može pristupiti strukturi u trenutku kad je samo jedan član izmijenjen. Ovakve slučajeve je vrlo teško otkriti i reproducirati jer je opseg mogućnosti vrlo sužen. Ako su promjene odrađene u slijednim CPU instrukcijama, šansa da se problem dogodi u bilo kojoj iteraciji izvršavanja programskog koda je mala, čak iako se struktura dohvaća iz druge konkurentne niti. Ako se povećava pritisak na sustav i broj izvođenja instrukcija se također povećava. Upravo iz tog razloga povećava se i vjerojatnost pojave greške zbog stanja natjecanja. Budući da su stanja natjecanja vremenski osjetljivi problemi, vrlo često nestanu tijekom pokretanja programa s *debuggerom* jer *debugger* utječe na vremensku komponentu izvođenja aplikacije.

2.4 Izbjegavanje problematičnih stanja natjecanja

Postoji nekoliko načina kako se nositi s problematičnim stanjima natjecanja. Najjednostavnija opcija je omotati podatkovnu strukturu zaštitnim mehanizmom kako bi se osiguralo da samo programska nit koja provodi izmjenu može vidjeti međukorake gdje nepromjenjivi iskazi nisu istiniti. Tada se sa stajališta drugih niti, koje pristupaju toj podatkovnoj strukturi, vidi da operacije promjena ili nisu nikako počele ili su završile. U ovu svrhu C++ standardna biblioteka nudi nekoliko takvih mehanizama.

Druga opcija je modificirati dizajn podatkovne strukture i njezinih nepromjenjivih iskaza tako da su promjene napravljene kao niz nedjeljivih promjena, kojom prilikom svaka od njih čuva istinitost iskaza. Ovaj pristup se naziva programiranje bez korištenja lokota (engl. *lock-free programming*) i teško ga je implementirati.

Još jedna opcija kako se nositi sa stanjima natjecanja je izvršavanje promjena nad podatkovnom strukturom unutar jedne transakcije (engl. *transaction*), na isti način na koji su izvedene operacije promjene nad bazom podataka. Relevantni niz promjena i čitanja podataka je spremljen u zapis (engl. *transaction log*) i izvršava se u jednom koraku. Ako se promjene ne mogu izvršiti jer je struktura modificirana, transakcija se ponavlja ispočetka. Ovaj pristup se naziva *software transactional memory* (STM) i još uvijek se nalazi u fazi istraživanja.

Najosnovniji mehanizam zaštite dijeljenih podataka, dostupan iz standardne C++ biblioteke, je lokot (engl. *mutex*).

2.5 Mehanizam zaključavanja

Osnovna ideja sinkronizacije programskih niti jest sprječavanje da više programskih niti istovremeno izvodi problematični dio koda koji se naziva kritični odsječak (engl. *critical section*). To se postiže pomoću globalnog objekta kojim se kontrolira pristup kritičnom odsječku iz različitih programskih niti i lokota koji se postavlja na početak kritičnog odsječka. Prva programska nit koja naiđe na lokot na početku kritičnog odsječka će zaključati globalni objekt, tj. preuzet će vlasništvo nad njim. Time će spriječiti ostalim programskim nitima ulazak u taj dio koda. Naiđe li neka druga programska nit na dio koda koji je zaključan, njeno izvođenje će se zaustaviti sve dok prva programska nit ne završi s izvođenjem kritičnog odsječka, otključa globalni objekt i time dozvoli pristup drugim programskim nitima [6].

Biblioteka jezika C++ u zaglavlju `mutex` definira nekoliko klasa koje se mogu koristiti za sinkronizaciju. Te klase su tipa muteks [1]. Muteksi su objekti koji omogućavaju da samo programska niti koja ima vlasništvo nad njim može izvršiti kritični odsječak koda.

[1] Složenica od engl. *mutually exclusive* - uzajamno isključiv

2.6 Zaštita dijeljenih podataka lokotima

Pretpostavimo da postoji struktura podataka, kao prethodno opisana dvostruko povezana lista, koju želimo zaštiti od stanja natjecanja ili mogućih kršenja nepromjenjivih iskaza. Zamislimo kako bi bilo lijepo označiti sve dijelove koda, koji pristupaju našoj strukturi, kao međusobno isključive takve da niti jedna druga nit ne može pristupiti strukturi ako je već obrađuje jedna od programskih niti. Takvim pristupom bi osigurali da nit ne može vidjeti postojanje neistinitog nepromjenjivog iskaza, osim ako bi struktura bila modificirana baš u toj niti.

Upravo ovaj primjer precizno opisuje osnovni sinkronizacijski model koji se naziva lokot (engl. *mutex*). Prije pristupa dijeljenom podatku potrebno je zaključati lokot, i kad se završi s operacijom nad podatkom isti lokot je potrebno otključati. `thread` biblioteka osigurava da kad jedna programska nit zaključa određeni lokot sve druge niti koje žele zaključati isti lokot moraju čekati sve dok nit koja je uspješno zaključala taj lokot ne otključa isti. To osigurava da sve programske niti uvijek pristupaju konzistentnim podacima.

Listing 2.1: Osiguranje kritičnog odsječka lokotima

```
#include <mutex>

std::mutex mtx;

void foo(void)
{
  ...
  // ulazak u kriticni odsjecak - zakljucavam lokot
  mtx.lock();

  // uradi posao

  // izlazak iz kriticnog odsjecka - otkljucavam lokot
  mtx.unlock();
  ...
}
```

Bitno je napomenuti da se zaključani lokot ne smije zaboraviti otključati na kraju kritičnog odsječka.

2.7 Automatsko otključvanje

Potreba da se svaki puta mora ekplicitno pozvati funkcijski član `unlock()` pogoduje pogreškama, bilo zato što možemo zaboraviti navesti naredbu ili zato što unutar kritičnog odsječka može biti bačena iznimka zbog koje se naredba za otključvanje neće izvesti. Želimo li se osigurati da će se lokot otključati po izlasku iz kritičnog odjsečka, klasu

`mutex` trebamo zapakirati u drugu klasu koja bi u destruktoru automatski pozvala član `unlock()`. Standardna C++ biblioteka definira takav predložak klase `std::lock_guard`. Ta klasa tijekom izvođenja destruktora poziva funkcijski član `unlock()` objekta koji joj je proslijeđen kao argument konstruktoru. Predložak je parametriziran tipom muteksa koji je zadužen za sinkronizaciju.

Listing 2.2: Automatsko otključavanje lokota

```
#include <mutex>

std::mutex mtx;

void foo(void)
{
  ...
  {
    // ulazak u kriticni odsjecak
    std::lock_guard<mutex> lg(mtx);

    // uradi posao
  }
  // izlazak iz kriticnog odsjecka - lokot automatski otkljucan
  ...
}
```

2.8 Potpuni zastoj - problem i rješenje

Pretpostavimo da imamo igračku koja se sastavlja iz dva dijela, od kojih su oba neophodna za ispravno funkcioniranje igračke. Neka to budu bubanj i udaraljke. Pretpostavimo sada da imamo i dvoje djece koja žele svirati bubanj. Ako jedno od njih ima i bubanj i udaraljke, može se neometano igrati sve dok želi. A ako se drugo dijete želi igrati u tom trenutku, mora čekati sve dok prvo dijete ne ostavi i udaraljke i bubanj. Pretpostavimo sada da se oba dijela igračke nalaze zakopani u kutiji s igračkama i djeca počinju prekopavati po njoj kako bi ih pronašli. Neka je jedno dijete našlo bubanj, a drugo udaraljke. Tada će se naći u situaciji gdje ni jedno neće moći svirati sve dok onaj drugi ne popusti i ne preda svoj dio drugom.

Zamijenimo sada u ovoj priči djecu i igračke s programskim nitima i lokotima. Obje programske niti moraju uzeti vlasništvo nad oba lokota i zaključati ih kako bi uspješno izvršile operaciju, i svaka programska nit ima po jedan lokot i čeka onu drugu kako bi predala vlasništvo i nad drugim lokotom. U ovoj situaciji niti jedna programska niti ne može nastaviti obavljati posao jer čeka da druga programska niti otpusti svoj lokot. Ta situacija se zove potpuni zastoj i najveći je problem u situacijama kad je za izvođenje operacije potrebno dva ili više lokota.

Jedan od savjeta za izbjegavanje potpunog zastoja je zaključavanje lokota uvijek u istom redoslijedu. Ako uvijek zaključamo lokota A prije lokota B, nikad nećemo doći u situaciju potpunog zastoja. Ponekad je ovo prilično jednostavno jer se lokoti koriste u različite svrhe, ali ponekad i nije tako jednostavno, kao na primjer u situaciji kad lokoti štite odvojene instance iste klase. Promotrimo operaciju koja razmjenjuje podatke između dvije instance iste klase. Kako bi osigurali ispravnu razmjenu, bez posljedica uzrokovanih konkurentnim pristupima, lokoti na obje instance moraju biti zaključani. Ako fiksiramo redoslijed zaključavanja lokota po redoslijedu ulaznih parametara, završit ćemo u potpunom zastoju ako se dogodi da netko zamijeni redoslijed ulaznih parametara - što nije malo vjerojatno [5].

Na sreću, C++ standardna biblioteka nudi rješenje za takve probleme kroz objekt `std::lock` koji istovremeno zaključava dva ili više lokota bez rizika od potpunog zastoja. Primjer ispod pokazuje kako objekt iskoristiti za jednostavnu zamjenu.

Listing 2.3: Istovremeno zaključavanje dva lokota

```
#include <mutex>

class neki_veliki_objekt;

void zamijeni(neki_veliki_objekt& lhs, neki_veliki_objekt& rhs);

class X
{
private:
  neki_veliki_objekt objekt;
  std::mutex m;
public:
  X(neki_veliki_objekt const& obj) : objekt(obj){}
  friend void zamijeni(X& lhs, X& rhs)
  {
    if(&lhs == &rhs)
      return;
    std::lock(lhs.m,rhs.m);
    std::lock_guard<std::mutex> lock_a(lhs.m,std::adopt_lock);
    std::lock_guard<std::mutex> lock_b(rhs.m,std::adopt_lock);
    zamijeni(lhs.objekt, rhs.objekt);
  }
};
```

Prvo je potrebno provjeriti da su objekti različiti zbog toga što pokušaj preuzimanja vlasništva nad lokotom, koji je već u vlasništvu te programske niti, rezultira nedefiniranim ponašanjem[2] (engl. *undefined behavior*). Nakon toga se poziva `std::lock()` koji

[2]Lokot koji dozvoljava višestruko zaključavanje je lokot tipa `std::recursive_mutex`.

zaključava oba lokota. Potom se konstruiraju dva objekta tipa `std::lock_guard`, po jedan za svaki lokot. Dodatni parametar `std::adopt_lock`, koji se predaje konstruktoru, naglašava da su lokoti već zaključani i da je samo potrebno prilagoditi vlasništvo nad lokotima.

To osigurava da se lokoti ispravno otključaju prilikom izlaska iz funkcije u općenitom slučaju, što uključuje situaciju gdje štićena operacija izbacuje iznimku te na taj način završava funkciju.

Funkcija `std::lock()` će spriječiti pojavu potpunog zastoja samo u slučajevima kad je oba lokota potrebno zaključati u istom trenutku, dok ne pomaže u slučajevima kad je lokote potrebno zaključati odvojeno. U tom slučaju se moramo pouzdati u vlastitu disciplinu kako bi izbjegli potpune zastoje - što nije nimalo lak posao [5].

2.9 Smjernice za izbjegavanje potpunog zastoja

Korištenje lokota je najučestaliji razlog pojave potpunog zastoja, ali nikako nije jedini razlog zbog kojeg se taj problem može dogoditi. Potpuni zastoj je moguće vrlo jednostavno reproducirati bez korištenja ijednog lokota. Ako imamo dvije programske niti, gdje svaka programska nit zove funkciju `join()` na objektu `std::thread` koji predstavlja onu drugu programsku nit, dolazimo u situaciju gdje niti jedna nit ne može završiti jer obje čekaju onu drugu da završi.

Ova jednostavna situacija može se dogoditi gdje jedna programska nit može čekati drugu da odradi svoj posao, a istovremeno druga programska nit čeka prvu kako bi odradila taj isti posao. Glavna smjernica za izbjegavanje potpunog zastoja se svodi na ideju - ne čekaj drugu programsku nit, ako postoji mogućnost da ona čeka tebe.

Izbjegavanje ugnježđivanja lokota

Ne preuzimaj vlasništvo nad lokotom ako već imaš vlasništvo nad jednim. To je najjednostavnija ideja za izbjegavanje potpunog zastoja, i zaista je nemoguće upasti u potpuni zastoj zbog korištenja lokota, jer će svaka programska nit u svakom trenutku zaključavati samo jedan lokot. Još uvijek se može upasti u potpuni zastoj zbog drugih razloga, ali najčešći uzrok, a to je korištenje lokota, će ovime biti izbjegnut. Ako je potrebno preuzeti vlasništvo nad više lokota, preporuka je koristiti funkciju `std::lock()` koja će osigurati izbjegavanje potpunog zastoja.

Izbjegavanje poziva korisničkog koda dok programska nit drži lokot

Ova smjernica nastavak je na prethodnu. Generalno je nemoguće znati koje operacije obavlja korisnički odrezak koda, pa između ostalih, moguće je da obavlja i zaključavanje lokota.

Ako pozovemo takav dio koda iz programske niti koja je već zaključala lokot, prekršit ćemo smjernicu o ugnježđivanju lokota. Nažalost, nije uvijek moguće ispoštovati ovu smjernicu, pa zbog toga trebamo uvesti novu.

Zaključavanje lokota u fiksnom poretku

Ako je potrebno zaključati dva ili više lokota, a to nije moguće učiniti koristeći funkciju `std::lock()`, sljedeće najbolje rješenje je zaključavanje lokota uvijek istim redoslijedom u svim programskim nitima. Ključni dio je definirati redoslijed koji mora biti konzistentan među svim programskim nitima. Neki od primjera zaključavanja u fiksnom poretku opisani su u [5] (vidi str. 49 i 50).

Hijerarhija lokota

Iako je ovo poseban slučaj definiranja redoslijeda zaključavanja, hijerarhija zaključavanja može osigurati način provjere da se konvencija poštuje tijekom izvođenja. Ideja je da aplikaciju podijelite na slojeve i identificirate sve lokote koji mogu biti zaključani u bilo kojem sloju. Kada kod pokuša zaključati lokot, nije mu dopušteno zaključavanje ako taj lokot već ima u vlasništu lokot s nižeg sloja. To možete provjeriti tijekom izvođenja dodjeljivanjem brojeva slojevima i vođenjem evidencije o tome koji su lokoti zaključani u svakoj programskoj niti [5].

Poglavlje 3

Sinkronizacija rada programskih niti

Osim što je potrebno zaštiti dijeljene podatke između programskih niti, potrebno je i sinkronizirati njihov rad. Moguće je da jedna programska nit treba čekati drugu nit da završi svoj posao prije nego što ona sama završi svoj. Načelno, programska nit dosta često zahtjeva čekanje na određeni događaj ili na istinit uvjet. Iako je moguće periodički provjeravati zastavicu ili neku drugu informaciju spremljenu u dijeljenoj memoriji kako bi se napravila sinkronizacija, taj pristup je daleko od idealnog. Potrebu za sinkronizacijom rada programskih niti omogućuje standardna C++ biblioteka sa značajkama *conditional variables* i *futures* [5].

3.1 Čekanje na događaj ili drugi uvjet

Pretpostavite da putujete noćnim vlakom. Jedan od načina na koji možete osigurati da izađete na vašoj stanici je da ostanete budni cijelu noć i da konstantno pratite gdje vlak staje. Tako ne bi propustili stanicu, ali bi bili umorni od neprospavane noći. Druga mogućnost je namjestiti alarm po rasporedu dolazaka vlaka na stanice i otići na spavanje. Taj bi pristup bio dobar, s tim da bi ste se probudili prerano u slučaju kašnjenja. Isto tako postoji mogućnost da se alarm ne aktivira uslijed različitih problema pa bi u tom slučaju propustili stanicu. Najidealnija mogućnost bi bila kad bi imali nekoga ili nešto pored sebe što bi vas probudilo kadgod bi vlak stigao na stanicu.

Ovakvu situaciju možemo povezati i sa sinkronizacijom programskih niti. Nit koja čeka drugu nit da završi posao ima nekoliko mogućnosti na raspolaganju kako bi saznala taj trenutak. Kao prvo, prva programska nit može kontinuirano provjeravati zajedničku zastavicu, osiguranu lokotima, koju bi druga niti postavila po završetku posla. Ovakav pristup nepotrebno iskorištava resurse, neprestano provjeravajući zastavicu, i također zaključava lokot čekajući na zastavicu. Ako se programska nit, koja čeka, izvodi, oduzimaju se resursi potrebni drugoj programskoj niti za rad. Isto tako, dok čekajuća nit drži lokot

zaključanim, programska nit koja treba postaviti zastavicu ne može je postaviti jer joj je lokot nedostupan. Bespotrebno korištenje resursa može se usporediti sa zapričavanjem strojovođe iz vlaka s početka priče. Ako zapričavate strojovođu i smanjujete mu koncentraciju, on mora voziti sporije i u konačnici na odredište stižete sa zakašnjenjem. Slično tome, čekajuća programska nit nepotrebno iskorištava resurse, čekajući na zastavicu te time u konačnici prolongira vrijeme završetka posla druge niti [5].

Druga mogućnost je osigurati da čekajuća programska nit spava i budi se u kraćim periodima kako bi provjerila zastavicu. To se može osigurati korištenjem standardne C++ biblioteke na sljedeći način:

Listing 3.1: Provjeravanje zastavice u jednakim vremenskim intervalima

```
#include <mutex>
#include <chrono>
#include <thread>

bool flag;
std::mutex m;
void wait_for_flag()
{
    std::unique_lock<std::mutex> lk(m);
    while(!flag)
    {
        lk.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        lk.lock();
    }
}
```

Lokot se otključava u petlji prije spavanja, te se zaključava nakon buđenja kako bi se provjerilo stanje zastavice. Prije spavanja se lokot otključava kako bi se radnoj programskoj niti dala mogućnost postavljanja zastavice po završetku posla.

U usporedbi s prvom mogućnosti, gdje je čekajuća programska nit neprekidno provjeravala zastavicu, ovo je poboljšanje u smjeru nepotrebnog iskorištavanja resursa. Ali isto je tako teško odrediti periode buđenja. Prekratki periodi između provjera ponovo bespotrebno troše resurse, dok predugi periodi mogu zakašnjelo reagirati na postavljenju zastavicu, unoseći tako kašnjenje u sustav.

Treća, i najbolja mogućnost je čekanje na konkretan događaj. Najosnovniji mehanizam za tu uporabu, a koju nudi standardna C++ biblioteka, je uvjetna varijabla (engl. *conditional variable*). Uvjetna varijabla je vezana za događaj ili za drugi uvjet, i na istu uvjetnu varijablu može čekati jedna ili više programskih niti. Kada radna programska nit shvati da je uvjet ispunjen, ona može objaviti (engl. *notify*) jednoj ili više čekajućih programskih niti da je uvjet zadovoljen i tako ih probuditi i dopustiti im daljnji rad.

3.2 Čekanje na uvjet pomoću uvjetne varijable

Standardna C++ biblioteka pruža dvije implementacije uvjetne varijable. Obje se nalaze u istom zaglavlju:

Listing 3.2: Implementacije uvjetne varijable

```
#include <condition_variable>

std::condition_variable prva_implementacija;
std::condition_variable_any druga_implementacija;
```

U oba slučaja uvjetne varijable moraju raditi u sprezi s lokotima kako bi pružile ispravnu sinkronizaciju, s tim da prva implementacija može raditi samo s običnim lokotom (*std::mutex*), dok je druga fleksibilnija i može raditi s lokotima koji ispunjavaju minimalne kriterije za lokot. Ali isto tako, druga implementacija je puno skuplja po pitanju veličine, performansi ili resursa operacijskog sustava, pa je zbog toga preporučljivo koristiti prvu implementaciju kad god je to moguće.

Listing 3.3: Primjer korištenja uvjetne varijable

```
#include <condition_variable>
#include <mutex>
#include <thread>

std::mutex m;
std::condition_variable condVar;
bool dataReady;

void foo2()
{
  std::unique_lock<std::mutex> lck(m);
  condVar.wait(lck, []{return dataReady;});
  ...
}
void foo1()
{
  // radi neki dulji posao
  ...
  std::lock_guard<std::mutex> lck(m);
  dataReady = true;
  condVar.notify_one();
  ...
}

int main()
{
  std::thread t1(foo1);
```

```
  std::thread t2(foo2);
  t1.join();
  t2.join();
}
```

Ovo je jednostavni primjer korištenja uvjetne varijable u svrhu sinkronizacije rada dvije programske niti, `foo1` i `foo2`. Glavna programska nit pokrene te dvije programske niti u isto vrijeme, te se obje niti krenu izvoditi usporedno. Dok programska nit `foo1` obavlja dulji posao, druga programska nit `foo2` će zaključati lokot te provjeriti dostupnost dijeljenog podatka funkcijom `wait()`. Funkcija će provjeriti stanje zastavice `dataReady`, pa kad utvrdi da je njezino stanje `false`, otključat će lokot te će staviti drugu programsku nit u blokirajuće stanje. Druga nit će ostati u blokirajućem stanju sve dok prva programska nit ne odradi svoj posao i ne pozove funkciju `notify_one()`, čime će obavijestiti druge programske niti [1] da je uvjet za odblokiranje ispunjen, a odblokirat će samo jednu. Postoji i funkcija `notify_all()` koja će odblokirati sve programske niti koje čekaju na uvjet isti uvjet [2].

Druga će nit, u tom trenutku, ponovo odblokirati, zaključati lokot te provjeriti stanje zastavice. U ovom slučaju stanje će biti `true`, pa će onda i druga nit nastaviti s poslom za koji je prethodno morala dobiti odobrenje od prve programske niti. Bitno je naglasiti da je uvijek potrebno dodatno provjeravati uvjet, kako se ne bi dogodilo neočekivano ponašanje uslijed slučajnog buđenja programske niti.

Kombinacija korištenja uvjetne varijable i lokota bit će iskorištena u nastavku dokumenta za dizajn strukture podataka, koja će predstavljati siguran medij za razmjenu podataka u višedretvenom okruženju u modelu sustava s raspodijeljenom memorijom.

3.3 Sinkronizacija programa slanjem poruka

Glavna ideja sinkronizacije je jednostavna - ako dijeljeni podatak ne postoji, svaka programska nit može se sagledati potpuno neovisno o drugim programskim nitima, isključivo na temelju ponašanja koje opisuje način obrade primljene poruke.

Stoga se gotovo svaka programska nit može modelirati automatom (engl. *state machine*). Po primitku poruke, programska nit ažurira svoje stanje i, po potrebi, šalje jednu ili više poruka drugim programskim nitima. Jedan od načina kako implementirati takve programske niti je implementacijom konačnog automata (engl. *Finite State Machine*). Međutim, kako postoje i druge metode, na programeru ostaje odabir pristupa za implementaciju programskih niti u višedretvenoj okolini.

Bitno je napomenuti da se podaci u sustavu sa slanjem poruka dijele samo kroz poslanu poruku, a nikako kroz adresni prostor. Međutim, kako C++ programske niti dijele adresni

[1] U ovom slučaju samo će programska nit `foo2` biti obaviještena.

prostor, nemoguće je prisiliti programera na pristup dijeljenja podataka samo kroz poruku. Tu ipak disciplina dolazi na vidjelo, pa se autori biblioteka za protokole razmjene poruka između programskih niti moraju pobrinuti za takav zahtjev. Najrašireniji pristup je dijeljenje poruka kroz red (engl. *queue*), koji je jedini objekt kojeg je potrebno dijeliti među programskim nitima. Red je uvijek dobro "sakriti" u biblioteku, te ga ne izlagati krajnjem korisniku biblioteke.

U konstrukciji višedretvenih programa koji se zasnivaju na bazi slanja poruka, okosnicu čini upravo taj dijeljeni red (engl. *shared queue*) kroz koji se šalju poruke, u kojima se nalaze svi objekti koji se žele podijeliti između programskih niti. Osim što je implementaciju za dijeljeni red potrebno sakriti od krajnjeg korisnika biblioteke, potrebno se pobrinuti da je takav red sigurno koristiti u višedretvenom okruženju. Pa se tako umetanje i skidanje poruke iz reda može shvatiti kao kritični odsječak, te je te dvije akcije potrebno zaštiti lokotima. Potrebno je primijetiti da je u svrhu osiguranja kritičnih odsječaka potrebno iskoristiti samo jedan lokot, koji je u konačnici i jedini potrebni lokot za implementaciju kompletne biblioteke, kao i krajnjeg programa.

Jedna potpuno fleksibilna i funkcionalna biblioteka trebala bi krajnjem korisniku pružiti mogućnost dijeljenja poruka različitih tipova, tj. poruka s različitim sadržajem za koje je moguće definirati različite obradne metode. Kako bi se osigurala takva fleksibilnost, potrebno je napraviti parametrizaciju po tipu poruke i po tipu metode za obradu primljene poruke, korištenjem C++ `template` značajke.

Poglavlje 4

Konkurentni programi

Razjasnimo na početku terminologiju vezanu za paralelne programe. Tipično je da, kad želimo izvršiti program s dijeljenom memorijom (engl. *shared memory*), pokrenemo jedan proces (engl. *process*) i iz njega izvedemo više programskih niti (engl. *threads*). U tom slučaju govorimo o programskim nitima kao izvršiteljima zadaća. S druge strane, kad želimo izvršiti program u sustavu s raspodijeljenom memorijom (engl. *distributed memory*), pokrenemo više procesa i u tom slučaju govorimo o procesima kao izvršiteljima zadaća.

4.1 Koordinacija programskih niti

U vrlo malo slučajeva pronalaženje optimalnih performansi paralelnih programa je trivijalno. Pretpostavimo da imamo dva skupa koja želimo zbrojiti:

```
double x[n], y[n];
...
for (int i=0; i<n; i++)
    x[i] += y[i];
```

Kako bi paralelizirali ovu zadaću, potrebno je samo ispravno raspodijeliti članove skupova odvojenim programskim nitima koje će zbrojiti dodijeljeni podskup. Neka imamo t programskih niti, pa onda niti s rednim brojem 1 možemo dodijeliti podskup elemenata $\{0, ..., \frac{n}{t} - 1\}$, niti sa rednim brojem 2 podskup $\{\frac{n}{t}, ..., \frac{2n}{t} - 1\}$, i tako dalje.

Kako bi se osigurala takva raspodjela posla, potrebno je napraviti:

- Podijeliti posao među programskim nitima
  - na način da svaka nit dobije otprilike jednaku veličinu podskupa
  - tako da se optimizira broj komunikacija među nitima

Pristup podjele posla na jednake dijelove se naziva balansiranje opterećenja (engl. *load balancing*).

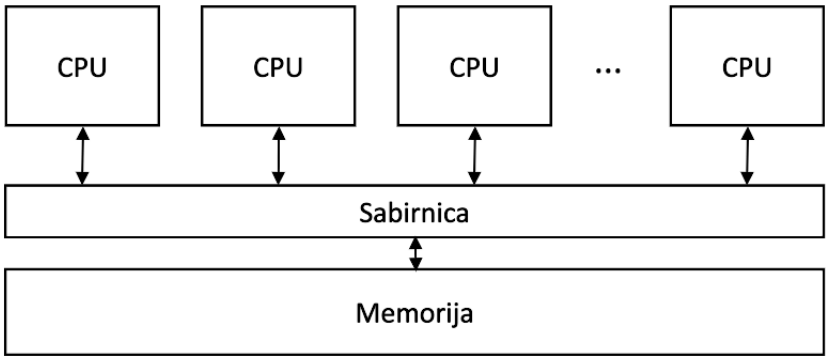
- Osigurati sinkronizaciju među nitima
- Osigurati komunikaciju među nitima

Posljednje dva problema su gotovo uvijek međusobno zavisna. Na primjer, u sustavima s raspodijeljenom memorijom, često se sinkronizacija procesa obavlja komunikacijom među njima, dok u sustavima s dijeljenom memorijom često se komunicira među nitima sinkronizirajući ih.

4.2 Dijeljena memorija

Dijeljena memorija je memorija kojoj istovremeno može pristupati više procesa ili procesora u višejezgrenim okolinima. Pružanjem pristupa zajedničkom adresnom prostoru smanjuje se potreba za komunikacijom među procesima, što olakšava implementaciju, ali s druge strane unosi dodatne probleme koji se javljaju prilikom istovremenog pristupa podacima.

Slika 4.1: Sustav s dijeljenom memorijom



Ako sustav dijeljene memorije želimo staviti u kontekst programskih niti, bitno je napomenuti da programske niti po svojoj definiciji dijele adresni prostor, te se zbog toga ovaj pristup često prihvaća kao prvi izbor prilikom implementacije višedretvenih programa.

U takvim sustavima varijable mogu biti dijeljene (engl. *shared*) ili privatne (engl. *private*). Dijeljenu varijablu može pročitati ili izmijeniti svaka programska nit, dok privatne varijable može kontrolirati samo jedna programska nit. Komunikacija među programskim nitima uobičajeno se obavlja kroz dijeljene varijable [4].

Dinamičke i statičke programske niti

U većini sustava s dijeljenom memorijom programi koriste dinamičke programske niti (engl. *dynamic threads*). U ovom pristupu postoji glavna programska nit (engl. *master thread*) i neodređena skupina programskih niti radnika (engl. *worker threads*). Glavna nit obično sjedi i čeka zahtjev za rad, koji tipično dolazi preko mreže. U trenutku kada glavna nit primi takav zahtjev, kreira se radna programska nit koja obavlja traženi posao, i gasi se nakon što obavi posao. Ovaj pristup optimizira korištenje resursa, jer se resursi zapravo koriste samo kad se obavlja koristan posao.

Alternativa dinamičkom pristupu je pristup statičkih programskih niti (engl. *static threads*). Glavna programska nit stvara (engl, *fork*) sve dodatne niti nakon što za to dođe zahtjev. Svaka programska nit radi sve dok ne obavi kompletan posao. Nakon što se pokrenute niti spoje (engl. *join*) s glavnom programskom niti, glavna nit može odraditi dodatno čišćenje, npr. memorije, i onda i ona završava. U kontekstu korištenja resursa, ovaj pristup je manje učinkovit. Kada je programska nit besposlena (engl. *idle*), njezini resursi, kao što su stog, programsko brojilo, itd., ne mogu biti oslobođeni. Također, operacije stvaranja i spajanja programskih niti mogu biti vremenski značajno zahtjevne operacije. Ako na raspolaganju imamo sve potrebne resurse, pristup sa statičkim programskim nitima potencijalno može imati bolje performanse od dinamičkog pristupa. Također, statički pristup je bliži široko rasprostranjenom pristupu s raspodijeljenom memorijom, pa je stoga ovaj pristup češće korišten u sustavima s dijeljenom memorijom.

Problemi s funkcijama

U velikoj većini slučajeva paralelni programi mogu pozivati funkcije razvijene za korištenje u serijskim programima i neće se dogoditi nikakav problem. Međutim, postoje neke značajne iznimke. Najvažnija iznimka za C programere dogodi se u funkcijama koje koriste lokalne statičke (engl. *static*) varijable. Prisjetimo se da se obične lokalne varijable u C programskom jeziku alociraju iz sistemskog stoga. Budući da svaka programska nit ima svoj stog, obične lokalne varijable su privatne (engl. *private*). Također, prisjetimo se da statičke varijable deklarirane unutar funkcije očuvaju svoju vrijednost iz poziva u poziv. Zbog toga, statičke varijable su zapravo dijeljene među svim programskim nitima koje pozivaju tu funkciju, i ovaj efekt može imati neočekivane i neželjene posljedice.

Na primjer, C biblioteka za znakovne nizove (engl. *string library*) implementira funkciju `strtok()` koja razdvaja dani znakovni niz u podnizove. Kada se funkcija pozove prvi put, predan joj je znakovni niz i prilikom sljedećih poziva funkcija vraća podnizove. To se može odraditi korištenjem statičkog pokazivača na znakovni niz `(static char *)` koji pokazuje na znakovni niz koji je predan u funkciju prilikom prvog poziva. Sada pretpostavimo da dvije programske niti razdvajaju znakovni niz u podnizove. Jasno je da ako prva nit prvi put pozove funkciju `strtok()`, a onda druga nit napravi isto prvi poziv funkcije,

prije nego što je prva nit završila razdvajanje svog niza, prva će nit izgubiti svoj rezultat ili će on biti prepisan rezultatom druge niti.

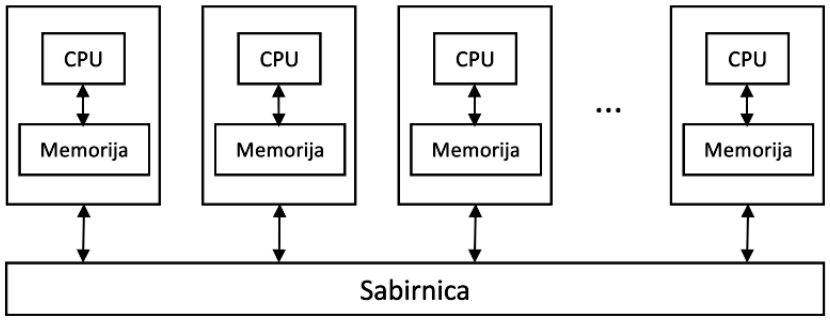
Stoga, funkcija `strtok()` nije sigurna za korištenje u programskim nitima (engl. *not thread safe*). To znači da se mogu dogoditi greške ili neočekivani rezultati kad je koristimo u programima s više programskih niti. Kada odsječak koda nije siguran u okolini programske niti, obično je to rezultat toga što različite niti pristupaju dijeljenim podacima. Zbog toga je potrebno biti oprezan prilikom korištenja funkcija koje su napisane isključivo za upotrebu u serijskim programima.

4.3 Raspodijeljena memorija

U sustavima s raspodijeljenom memorijom, svaki procesor ima svoj memorijski prostor i jedino njemu može direktno pristupati. Ne postoji mogućnost da jedan procesor direktno pristupi memorijskom prostoru drugog procesora. Ako se pojavi potreba za razmjenom informacija među procesorima, oni moraju razmijeniti poruke međusobno kako bi zahtijevali odnosno podijelili informaciju jedan drugome. Postoji nekoliko pristupa razmjene poruka među procesorima, a najčešće se koristi protokol slanja poruka (engl. *message passing protocol*).

Kao što je već spomenuto, programi s raspodijeljenom memorijom obično se izvode pokretanjem više procesa. To je zato što se "programske niti" u sustavima s distribuiranom memorijom češće izvode na nezavisnim procesorskim jedinicama s nezavisnim operacijskim sustavima.

Slika 4.2: Sustav s raspodijeljenom memorijom



Prebacivanjem konteksta na višedretvene programe dolazimo do zaključka kako je potrebno ograničiti dijeljenje adresnog prostora među programskim nitima te implementirati mehanizme slanja poruka, koji bi simulirali protokol slanja poruka.

4.4 Usporedba pristupa u višedretvenom okruženju

Po svojoj definiciji programske niti u C++ okruženju izvode se unutar jednog procesa te dijele adresni prostor. Zbog toga svaka programska nit ima pristup svim globalnim varijablama definiranim unutar programa i svim statičkim varijablama definiranim unutar metoda. Na prvi pogled, pristup definiranja globalnima onih varijabli, koje želimo dijeliti među programskim nitima, se čini idealnim, jer nije potrebno implementirati dodatne mehanizme za dijeljenje tih varijabli. Međutim, zbog ranije opisanih problema koji se javljaju prilikom višestrukog pristupa istim podacima, taj pristup je potrebno ograničiti na samo jedan u jednom trenutku. Osim što smo serijalizirali konkurentni program, uveli smo mehanizam ograničavanja koji može postati vrlo složen u sustavima s većim brojem programskih niti i s većim brojem dijeljenih podataka. Upravo ta složenost donosi dodatne probleme zbog kojih se javljaju fenomeni stanja natjecanja i potpunog zastoja, koji predstavljaju najveću glavobolju programerima koji se bave višedretvenim programima.

S druge strane, sustav s raspodijeljenom memorijom u startu zahtjeva veće ulaganje zbog potrebe za implementacijom mehanizama za razmjenu poruka među programskim nitima. Uzimajući u obzir dobru implementaciju mehanizama za razmjenu poruka, vjerojatnost pojave stanja natjecanja je neutralizirana, dok je vjerojatnost pojave potpunog zastoja svedena na minimum [1]. U konačnici, negativne strane ovog pristupa su dulje vrijeme izvođenja istog programa u usporedbi s dijeljenom memorijom, te veća potreba za resursima kao što su procesorska moć i memorija.

Pristup s dijeljenom memorijom bolje je izabrati ako je potrebno implementirati program u kojem mehanizmi zaštite dijeljenih podataka nisu složeni, dok je u drugom slučaju bolje žrtvovati hardverske resurse, te time izbjeći neočekivane greške, koristeći pristup raspodijeljene memorije.

4.5 Konkurentni programi - zaključak

S jedne strane, višedretveno izvođenje na današnjim višeprocesorskim sustavima može ubrzati izvođenje programa u slučaju kada se niz operacija raspodijeli na različite jezgre. S druge strane, višedretveno izvođenje može poboljšati odziv aplikacije, posebice kada se radi o aplikacijama s grafičkim sučeljem, kada bi neka spora operacija blokirala izvođenje glavne programske niti u kojoj se obično nalaze elementi grafičkog sučelja. Treba ipak biti svjestan da često dobici zbog višedretvenog izvođenja neće biti spektakularni. Program koji se pokrene u 20 programskih niti neće raditi 20 puta brže.

Pretpostavimo da se nalazimo na šalteru banke i čekamo u redu. U toj analogiji jednodretveno izvođenje odgovara slučaju kada svaki šalter ima zaseban red. Klijenti se obrađuju

[1] Još uvijek postoji mogućnost pojave potpunog zastoja zbog neispravnog završetka rada programske niti (vidi poglavlje 2.9)

prema redoslijedu kako stoje u redu i svaki klijent mora čekati da se završi obrada prethodnog klijenta. Ako dođe do zastoja što neki klijent obavlja složenu transakciju, cijeli red će morati čekati dok se ta transakcija ne izvrši. S druge strane, višedretveno izvođenje odgovara slučaju kad je jedan red za dva ili više šaltera. Sljedeći klijent uvijek ide na prvi slobodni šalter. Ako na jednom od šaltera dođe do zastoja, red neće stajati već se klijenti preusmjeravati na druge šaltere. Ukupno gledano, zbroj klijenta koji će se obraditi u slučaju kada je na svakom šalteru po jedan red bit će isti kao u slučaju kada je jedan red za sve šaltere. Taj je broj ograničen mogućnostima šalterskih službenika - na računalu bi to odgovaralo ukupnoj procesorskoj moći svih jezgri. Međutim, kod višešalterskog pristupa postoji privid bolje protočnosti, jer su zastoji manje uočljivi.

Poglavlje 5

Dizajn dijeljenih struktura podataka

Izbor strukture podataka za implementaciju programa može biti ključni dio kompletnog rješenja, pri čemu paralelni programi nisu iznimka. Ako će se strukturi podataka pristupati iz više programskih niti, ili struktura mora biti potpuno nepromjenjiva, što znači da sinkronizacija nije potrebna, ili program mora biti dizajniran tako da se osigura da promjene nad strukturom podataka moraju biti sinkronizirane među programskim nitima. Jedna opcija je korištenje posebnog lokota i obavljanje zaključavanja izvan strukture podatke, a druga opcija je dizajniranje strukture takve da nije potrebno raditi vanjsko zaključavanje. Prilikom dizajna, moguće je koristiti osnovne elemente za višedretvene aplikacije kao što su lokoti i uvjetne varijable.

5.1 Koja struktura podataka je sigurna za dijeljenje

Dizajnirati strukturu podataka za višedretvene programe znači da više programskih niti (dretvi) može pristupati strukturi konkurentno, bilo za izvršavanje istih ili različitih operacija, i da svaka nit ima konzistentan pogled na strukturu podataka. Ako su uvjeti ispunjeni, neće biti izgubljenih podataka, svi nepromjenjivi iskazi bit će istiniti i neće biti problematičnih stanja natjecanja. Takva struktura podataka je sigurna za korištenje u višedretvenim programima. Općenito, struktura je sigurna samo za posebne vrste konkurentnih pristupa. Moguće je imati više programskih niti koje izvode samo jednu vrstu operacija nad strukturom podataka usporedno, gdje druga operacija jedne niti zahtjeva isključivi pristup. Alternativno, istovremeni pristup više programskih niti, koje izvode različite operacije, prema istoj strukturi podataka može biti siguran, dok istovremeni pristup može biti problematičan ako programske niti izvode istu operaciju.

Doista dizajniranje za konkurentnost znači omogućivanje konkurentnosti programskim nitima koje pristupaju strukturi podataka. U svojoj naravi, lokot pruža međusobnu isključivost, što znači da samo jedna nit može zaključati lokot. Lokot štiti strukturu tako

da zapravo sprječava pravi konkurentni pristup podacima koje štiti. Taj postupak se zove serijalizacija (engl. *serialization*). Programske niti se međusobno izmjenjuju prilikom pristupa podatku koji je pod zaštitom lokota, i u konačnici pristup se zapravo obavlja serijski umjesto konkurentno [5].

5.2 Smjernice za dizajn dijeljenih struktura

Prilikom dizajna dijeljenih struktura potrebno je brinuti o dvije stvari:

- Osiguravanje da je pristup strukturi siguran.
- Osiguravanje da je pristup u svojoj naravi konkurentan.

Kako bi ispunili gornje uvjete potrebno je:

- Osigurati da niti jedna programska nit ne može vidjeti stanje strukture kad je nepromjenjivi iskaz narušen zbog aktivnosti druge programske niti.
- Voditi računa o stanjima natjecanja.
- Obratiti pažnju na ponašanje dijeljene strukture prilikom bacanja iznimki, kako se ne bi narušio nepromjenjivi iskaz.
- Svesti mogućnost potpunog zastoja na najmanju razinu, ograničavanjem opsega dosega lokota i izbjegavanjem ugnježđivanja lokota kad god je to moguće.

5.3 Strukture podataka na bazi lokota

Prilikom dizajna dijeljenih struktura podataka na bazi lokota sve se svodi na osiguravanje da je pravi lokot zaključan prilikom pristupa podacima i da je lokot aktivan minimalno potrebno vrijeme. Potrebno je osigurati da prilikom korištenja jednog lokota nije omogućeno pristupati podacima izvan dosega osiguranog lokotom i da ne postoje problematična stanja natjecanja. Iako se pristup sa korištenjem samo jednog lokota na prvu čini dosta nezgodan, dodavanjem dodatnih lokota otvaramo put ka dodatnim problemima. Pa tako u situaciji sa korištenjem više lokota potrebno je osigurati da se ne dogodi potpuni zastoj ako postoje operacije koje zahtijevaju zaključavanje više od jednog lokota. Upravo zbog problema potpunog zastoja, dizajn dijeljenih struktura dosta je problematičniji prilikom korištenja više lokota nego samo jednog [5].

Korištenjem samo jednog lokota izbjegli smo neočekivane probleme nastale uslijed potpunog zastoja, ali nismo riješili problem prekomjerne potrošnje procesorske snage uslijed neprekidnog provjeravanja stanja zastavice.

5.4 Strukture podataka na bazi lokota i uvjetne varijable

Prisjetimo se uvjetne varijable iz poglavlja 3.2. Kombiniranjem jednog lokota i jedne uvjetne varijable moguće je izgraditi strukturu podataka koja će biti sigurna za dijeljenje informacija među programskim nitima, što znači da su neočekivani problemi uslijed stanja natjecanja i potpunog zastoja svedeni na minimum.

Osim što će biti sigurna, takva struktura će biti vremenski i procesorski optimizirana upravo zbog uključivanja uvjetne varijable u implementaciju.

Najčešća struktura podataka za razmjenu informacija među programskim nitima je red. Onaj gotovi C++ red iz biblioteke `queue` potrebno je omotati u klasu koja će zaštititi operacije stavljanja i dohvaćanja elemenata iz reda od usporednog pristupa u višedretvenom okruženju koje može dovesti do kršenja istinitosti nepromjenjivih iskaza, tj. do nekonzistentnosti podataka.

Pogledajmo primjer implementacije sigurnog reda za razmjenu podataka među programskim nitima na bazi lokota i uvjetne varijable na sljedećem odsječku koda.

Listing 5.1: Dijeljena struktura na bazi lokota i uvjetne varijable - *thread safe queue*

```
#include <queue>
#include <mutex>
#include <condition_variable>

template <class Element>
class SafeQueue
{
public:
    SafeQueue() : q(), m(), c() {}
    ~SafeQueue() {}

    void stavi(Element elem)
    {
        {
            std::lock_guard<std::mutex> lk(m);
            q.push(elem);
        }
        c.notify_one();
    }

    Element dohvati()
    {
        std::unique_lock<std::mutex> lk(m);
        c.wait(
            lk, []{return !q.empty();});
        Element val = q.front();
        q.pop();
        return val;
```

```
    }

private:
    std::queue<T> q;
    mutable std::mutex m;
    std::condition_variable c;
};
```

Operacija stavljanja elementa u red `stavi()` osigurana je lokotom. Kad je podatak za stavljanje u red spreman, radna programska nit, prije nego stavi podatak u red, zaključa lokot kako bi ga zaštitila od nekonzistentnog stanja u slučaju da postoji još jedna radna programska nit koja stavlja ili dohvaća element iz reda u istom trenutku.

```
std::lock_guard<std::mutex> lk(m);
```

Nakon što radna programska nit uspješno stavi element u red, preko uvjetne varijable šalje obavijest drugim programskim nitima koje čekaju, tj., nalaze se u blokiranom stanju čekajući podatak.

```
c.notify_one();
```

Primijetite da se posao stavljanja elementa u red nalazi u ograničenom dosegu (engl. *scope*) kako bi se obavijest kroz uvjetnu varijablu poslala neposredno nakon stavljanja elementa u red, tj. nakon otključavanja lokota[1]. Time je osigurana vremenska optimizacija izvođenja, odnosno kašnjenje slanja obavijesti svedno je na najmanju moguću vrijednost.

```
{
    std::lock_guard<std::mutex> lk(m);
    q.push(elem);
}
c.notify_one();
```

S druge strane, programska nit koja dohvaća element iz reda prvo zaključava lokot sa:

```
std::unique_lock<std::mutex> lk(m);
```

Nakon čega programska nit poziva funkciju `wait()` nad uvjetnom varijablom, predajući joj lokot i uvjet koji se čeka. Funkcija `wait()` će blokirati programsku nit sve dok traženi uvjet nije zadovoljen. U primjeru ispod, uvjet je izražen lambda izrazom.

```
c.wait(
    lk, []{return !q.empty();});
```

Lambda funkcija provjerava je li red prazan, tj. postoje li u njemu elementi spremni za obradu. Funkcija `wait()` implementirana je tako da provjerava uvjet, te ako je on zadovoljen funkcija završava. U ovom primjeru uvjet će biti zadovoljen kad lambda funkcija vrati

[1] Kad se kreira objekt `lock_guard` on preuzima vlasništvo nad predanim lokotom. Kad kontrola napusti doseg u kojem je objekt kreiran, `lock_guard` objekt se uništava i lokot se otključava [2]

istinu (engl. *true*). Ako lambda vrati neistinu (engl. *false*), funkcija `wait()` otključava lokot i stavlja programsku nit u blokirajuće stanje. Kad je uvjetna varijabla objavljena iz druge programske niti pozivom `notifiy_one()`, programska nit se odblokira, zaključa lokot i ponovo provjeri stanje uvjetne varijable. Ako je stanje uvjetne varijable zadovoljeno, funkcija `wait()` završava s još uvijek zaključanim lokotom. Ako stanje nije zadovoljeno, programska niti otključava lokot i opet počinje čekati. Ovdje je bitno napomenuti da je lokot neophodno otključati prije ponovnog odlaska na čekanje, jer inače druga programska nit nikad ne bi došla u situaciju da može staviti element u red, te bi tako bio prouzrokovan potpuni zastoj. Upravo iz tog razloga se na ovom mjestu koristi `std::unique_lock` umjesto `std::lock_guard` lokota koji dozvoljava višestruko otključavanje i zaključavanje lokota.

5.5 Iznimke prilikom kopiranja elemenata

Pretpostavimo sada da su elementi reda opisanog u prethodnom poglavlju objekti tipa `std::vector<int>`. Vektor je dinamička kolekcija proizvoljne duljine, prilikom čijeg je kopiranja potrebno alocirati određenu memoriju s hrpe (engl. *heap*). Ako je sustav previše opterećen, moguće je da kopiranje vektora neće uspjeti pa će kopirni konstruktor (engl. *copy constructor*) izbaciti iznimku `std::bad_alloc`. Ovakva iznimka je više vjerojatna ako se radi o kolekciji `vector` koja sadrži veći broj elemenata.

Ako je funkcija za skidanje elemenata iz reda definirana tako da vrati (kopira) element te ujedno i ukloni taj element iz reda, javlja se problem prilikom izbacivanja iznimke iz kopirnog konstruktora. Zbog iznimke nećemo imati kopiju elementa, a on će zauvijek biti izgubljen iz reda, pa skidanje nećemo moći ponoviti ako se u međuvremenu oslobodi dodatna memorija.

Srećom, programeri biblioteke `queue` su mislili o tom problemu pa se skidanje elemenata iz reda odvija u dvije odvojene operacije. Prvom operacijom će se dohvatiti kopija elementa, a tek potom će se element ukloniti iz reda. Ako je operacija kopiranja bila neuspješna, element će ostati u redu, pa je kopiranje moguće pokušati kasnije.

Međutim, razdvajanje operacija utječe na pojavu problem stanja natjecanja jer je moguće da dvije programske niti paralelno skidaju elemente iz reda, pa se može dogoditi da jedan element bude obrađen dva puta, dok se onaj drugi zauvijek izgubi.

Kako bi se pojava problema stanja natjecanja minimizirala, potrebno je skratiti vrijeme kopiranja većih elemenata iz reda. To se postiže tako da se red konstruira od pokazivača tipa `std::shared_ptr`, čime se veći objekti u redu zamjenjuju samo pokazivačima. Ovaj tip pokazivača je dobar izbor jer smanjuje probleme curenja memorije (engl. *memmory leak*) jer se objekt uvijek uništava kad se uništi zadnji pokazivač na taj objekt. Također, olakšava procese alokacije memorije jer nije potrebno koristiti `new` i `delete`.

Poboljšana struktura za dijeljenje podataka među programskim nitima prikazana je u programskom odsječku ispod.

Listing 5.2: Potpuno sigurna struktura za dijeljenje podataka među programskim nitima

```
#include <queue>
#include <mutex>
#include <condition_variable>

template <class Element>
class SafeQueue
{
public:
    SafeQueue() : q(), m(), c() {}
    ~SafeQueue() {}

    void stavi(Element elem)
    {
        {
            std::lock_guard<std::mutex> lk(m);
            q.push(std::make_shared<T>(elem));
        }
        c.notify_one();
    }

    Element dohvati()
    {
        std::unique_lock<std::mutex> lk(m);
        c.wait(
            lk, []{return !q.empty();});
        Element val = q.front();
        q.pop();
        return val;
    }

private:
    std::queue<std::shared_ptr<T>> q;
    mutable std::mutex m;
    std::condition_variable c;
};
```

Poglavlje 6

MPI

Budući da smo u prethodnim poglavljima opisali sustave s raspodijeljenom memorijom, i povukli paralelu s modelom istog sustava u višedretvenom okruženju, informativno sada pokažimo kako izgleda općeprihvaćeni standard za razmjenu informacija među čvorovima u sustavima s raspodijeljenom memorijom.

6.1 Message Passing Interface

MPI (engl. *Message-Passing Interface*) jest sučelje specifikacije biblioteke za mehanizam razmjene poruka koji programeru pruža potpunu kontrolu nad međuprocesnom komunikacijom. MPI ponajviše predstavlja paralelni programski model razmjene poruka, gdje se podaci prebacuju iz adresnog prostora jednog procesa u adresni prostor drugoga, kroz surađujuće operacije na svakom procesu. MPI je specifikacija, a ne implementacija. Postoji više MPI implementacija s obzirom na programski jezik. Sam MPI standard je specifikacija sučelja biblioteke koje će se koristiti kao funkcije, podrutine ili metode s obzirom na implementaciju MPI-a u nekom programskom jeziku. C, C++, Fortran-77 i Fortran-95 su dio MPI standarda. Cilj MPI-ja je razvoj široko upotrebljivog standarda za pisanje programa s principom razmjena poruka. Trenutna aktivna specifikacija standarda MPI je 4.0. Neke od gotovih implementacija MPI specifikacije su MPICH, OpenMPI te MS-MPI.

6.2 Nastanak i svojstva standarda

U razvoju paralelnih aplikacija javlja se potreba za mehanizmima razmjene poruka (engl. *message passing*) za uporabu na računalima s raspodijeljenom memorijom. Tako su se u ranom periodu razvili sustavi kao što su Express, p4, PICL, PARMACS te PVM, ali oni, zbog manjih sintaktičkih i funkcionalnih razlika, nisu bilo lako prenosivi.

Tako je 1993. godine osnovan Message Passing Interface Forum koji je okupio 40-ak industrijskih i istraživačkih organizacija. Osnovani forum je tako već 1994. godine donio prvi MPI standard koji je imao verziju 1.1. Standard se nastavio razvijati pa je već 1997. godine razvijen MPI 2.0, 2012. godine MPI 3.0, 2015. godine MPI 3.1. Posljednju preinaku MPI standard doživio je 2021. godine te ta izmjena nosi verziju 4.0.

MPI standard definira komunikaciju porukama, odnosno razmjenu podataka među procesima. Broj procesa tijekom izvođenja MPI programa je po definiciji konstantan, jer u osnovnoj inačici standarda nisu predviđeni mehanizmi stvaranja odnosno gašenja procesa.

Načini komunikacije u MPI standardu:

- komunikacija od-točke-do-točke (engl. *point-to-point*) između dva određena procesa
- grupna (engl. *collective*) komunikacija unutar grupe procesa
- ispitne (engl. *probe*) funkcije za asinkronu komunikaciju
- komunikator (engl. *communicator*) mehanizam za razvoj modularnih paralelnih programa

6.3 MPI principi za model razmjene poruka

Prvi pojam u MPI principima je pojam komunikatora (engl. *communicator*). Komunikator definira grupu procesa koji su u mogućnosti komunicirati s drugim procesom. U toj grupi svaki proces ima dodjeljen svoj jedinistveni identifikator, rang (engl. *rank*), te procesi eksplicitno komuniciraju jedni s drugima preko ranga.

Temelj komunikacije je izgrađen na operacijama za slanje (engl. *send*) i primanje (engl. *receive*) poruka među procesima. Proces može poslati poruku drugom procesu navodeći rang procesa i pridjeljujući jedinstvenu oznaku (engl. *tag*) poruci. Proces primatelj po primanju poruke može objaviti primitak sukladno oznaci poruke ili čak ni ne mora brinuti o jedinstvenoj oznaci.

Ovakva komunikacija, koja uključuje jednog pošiljatelja (engl. *sender*) i jednog primatelja (engl. *receiver*), poznata je pod nazivom komunikacija od-točke-do-točke *point-to-point*.

Postoje i razni slučajevi gdje procesi moraju komunicirati sa svim ostalim procesima. Na primjer, kad glavni proces mora svim procesima radnicima (engl. *workers*) objaviti informaciju (engl. *broadcast*). U takvoj situaciju bi bilo nezgrapno pisati programski kod koji odrađuje sva primanja i slanja, što bi također utjecalo na performanse mreže. Međutim,

MPI standard donosi široku kolekciju operacija za grupnu komunikaciju (engl. *collective*) koja uključuje sve dostupne procese.

Operacija suprotna operaciji koja svima objavljuje informaciju (engl. *broadcast*) je operacija skraćivanja (engl. *reduce*), koja se koristi za izračunavanje jednog podatka na temelju podataka od svih procesa. Funkcija skuplja podatke od svih procesa, uključujući i korjenski proces (engl. *root process*), nad njima izvršava neku operaciju te rezultat sprema na određenu adresu u korjenskom procesu. Tako je operacija skraćivanja klasični koncept funkcionalnog programiranja gdje programer, u paralelnoj aplikaciji, može skratiti skup nekih brojeva u manji skup korištenjem dostupnih funkcija.

Kombiniranjem od-točke-do-točke i grupnih komunikacijskih metoda mogu se modelirati vrlo složeni paralelni programi.

6.4 Osnovne MPI funkcije

Bilo koja tehnika paralelnog programiranja razmjenom poruka mora za svaki proces omogućiti barem sljedeće mehanizme:

- otkriti ukupan broj procesa
- identificirati vlastiti proces u grupi procesa
- poslati poruku određenom procesu
- primiti poruku od određenog procesa

MPI uključuje preko 150 funkcija, no većina funkcionalnosti može se postići sa mnogo manjim skupom. Osnovna ideja MPI protokola je pružiti gotovo rješenje krajnjem korisniku koji implementira algoritam koji se izvodi u sustavu s raspodijeljenom memorijom. Osim što takvi sustavi mogu predstavljati neovisne procesorske jedinice na jednom hardverskom sklopovlju, mogu također predstavljati i potpuno fizički dislocirane procesorske jedinice, koje onda preko mreže mogu razmjenjivati informacije, i tako optimizirati korištenje procesorskih potencijala samo jednog računala.

6.5 Primjer implementacije - OpenMPI

Jedna od gotovih i spremnih za korištenje implementacija komunikacije na bazi slanja poruka je OpenMPI. To je otvorena (engl. *open source*) implementacija MPI standarda koju razvija i održava udruženje sastavljeno od članova akademske, istraživačke i industrijske zajednice.

Slanje i primanje poruka pomoću OpenMPI

Message-passing API minimalno daje funkciju za slanje (engl. *send*) i primanje (engl. *receive*). Procesi se obično identificiraju prema poretku (engl. *rank*) u rasponu $\{0, 1, ..., p-1\}$, gdje je p ukupan broj procesa. Na primjer, proces 1 može poslati poruku procesu 0 korištenjem sljedećeg pseudo koda:

Listing 6.1: Primjer korištenja OpenMPI biblioteke

```
char poruka [100];
...
moj_rang = MPI_Comm_rank();
if (moj_rank == 1)
{
  sprintf(poruka, "Pozdrav od procesa 1");
  MPI_Send(poruka, MPI_CHAR, 100, 0);
}
else if (moj_rank == 0)
{
  MPI_Receive(poruka, MPI_CHAR, 100, 1);
  printf("Proces0 > Primljeno: %s\n", poruka);
}
...
```

Ovdje funkcija `MPI_Comm_rank()` vraća rang procesa koji poziva funkciju. Proces 1 stvara poruku korištenjem standardne funkcije iz C biblioteke `sprintf()`, i onda tako stvorenu poruku šalje procesu 0 pozivanjem funkcije `MPI_Send()`. Argumenti poziva su poruka, tip elemenata od kojih je sastavljena poruka (`MPI_CHAR`), broj elemenata u poruci (`100`) i rang odredišnog procesa (`0`). S druge strane, proces 0 poziva funkciju `MPI_Receive()` sa sljedećim argumentima: varijabla u koju će se spremiti primljeni sadržaj (`poruka`), tip elemenata od kojih je sastavljena poruka (`MPI_CHAR`), broj dostupnih mjesta u odredišnoj varijabli (`100`) i rang procesa koji šalje poruku (`1`). Nakon uspješnog primitka, proces 0 ispiše primljenu poruku.

Na ovaj način OpenMPI pruža gotovu infrastrukturu za razmjenu informacija među čvorovima koji izvode algoritam.

Za korištenje OpenMPI biblioteke, na računalo je potrebno instalirati OpenMPI programski paket. Osim što programski paket pruža infrastrukturu za dizajniranje složenih sustava s raspodijeljenom memorijom, pruža i mogućnost simuliranja složenijih modela, koji se sastoje od većeg broja procesorskih jedinica koje izvode jedan algoritam, na samo jednom računalu (jezgri).

Poglavlje 7

Implementacija modela protokola za slanje poruka

U ovom poglavlju dat ću pregled dvije implementacije biblioteka koje pružaju sigurnu komunikaciju među programskim nitima u modelu sustava s raspodijeljenom memorijom. Biblioteke predstavljaju simulaciju protokola slanja poruka (engl. *message-passing protocol*) u svrhu sinkronizacije programskih niti, te u svrhu izbjegavanja dijeljenja memorije među programskim nitima u višedretvenim programima.

Prva implementacija koja će biti prikazana, razvijena je tijekom izrade ovog diplomskog rada, dok je druga preuzeta iz [5] dodatak C.

Obje biblioteke su implementirane i testirane u dodatku ovom diplomskom radu, dok je u samom radu dan pregled najznačajnijih dijelova obiju biblioteka.

7.1 Prva implementacija

Okosnicu biblioteke čini implementacija dijeljenog resursa, koji je zapravo medij kroz koji se razmjenjuju poruke među programskim nitima. Implementacijski, taj resurs je predstavljen strukturom podataka red (engl. *queue*) u koji se spremaju pokazivači na poruke tipa bazne klase. Kako bi se osigurala tražena fleksibilnost svaki zasebni tip poruke predstavljen je predloškom (engl. `template`) klase koji je izveden iz bazne klase. Stavljanjem poruke u red, konstruira se predložak klase te se pokazivač na taj objekt sprema u dijeljeni red kao poruka koju će konzumirati programska nit koja očekuje taj tip poruke. Skidanjem tog elementa iz reda, druga programska nit dohvaća pokazivač na prethodno konstruirani objekt. Bitno je napomenuti da je bazna klasa potpuno virtualna klasa, te kao takva ne sadržava metode članice, što znači da nije moguće direktno pristupiti sadržaju skinutog elementa, prije nego se tip elementa ne pretvori u specifični predložak (isti onaj koji se iskoristio prilikom konstrukcije objekta).

Struktura za razmjenu poruka

U dijeljenom resursu kritični odsječci osigurani su lokotom, dok je blokiranje programske niti, koja čeka poruku, izvedeno uvjetnom varijablom, tako da će čekajuća programska nit blokirati sve dok je red prazan.

Listing 7.1: Implementacija klase za razmjenu poruka

```
namespace MPI
{
    class safeQueue
    {
    public:
        template <typename Message>
        void push(Message const &msg)
        {
            std::lock_guard<std::mutex> lk(m);
            q.push(std::make_shared<wrappedMessage<Message>>(msg));
            c.notify_all();
        }
        std::shared_ptr<messageBase> waitAndPop()
        {
            std::unique_lock<std::mutex> lk(m);
            c.wait(lk, [&]
                   { return !q.empty(); });
            auto res = q.front();
            q.pop();
            return res;
        }

    private:
        std::mutex m;
        std::condition_variable c;
        std::queue<std::shared_ptr<messageBase>> q;
    };
}
```

Konstrukcija poruke

Implementacije klase za poruke i njezinih izvedenih klasa prikazne su u programskom odsječku ispod. Izvođenjem predloška klase (`struct wrappedMessage`) iz bazne klase osigurana je fleksibilnost po tipu poruke, tako da svaka poruka ima vlastitu specijalizaciju. Bitno je naglasiti da se sadržaju može pristupiti samo poznavajući predložak klase `struct wrappedMessage` koji se definira prilikom konstrukcije poruke. Prije nego je željena poruka konstruirana i poslana, u drugoj programskoj niti, koja očekuje tu poruku, bitno je kreirati objekt za obradu te poruke, te tom prilikom obradnom objektu predati predložak

kojim je konstruirana poruka kako bi obradni objekt znao pristupiti sadržaju primljene poruke.

Listing 7.2: Bazna i izvedena klasa za konstrukciju poruke

```
namespace MPI
{
    struct messageBase
    {
        virtual ~messageBase() {}
    };

    template <typename Msg>
    struct wrappedMessage : messageBase
    {
        Msg contents;
        explicit wrappedMessage
                (Msg const &contents_) : contents(contents_) {}
    };
}
```

Slanje poruke

Implementacija dijeljenog resursa skrivena je od krajnjeg korisnika tako da su implementirane dvije dodatne klase - po jedna za slanje poruka (`class sender`) i za primanje poruka (`class receiver`).

Implementacija klase za slanje poruka je prilično jednostavna, budući da ona samo pruža metodu za stavljanje poruka u dijeljeni red, na koji klasa ima samo pokazivač. Metoda za stavljanje poruka u red omogućuje stavljanje svih specijaliziranih poruka u red.

Listing 7.3: Implementacija klase za slanje poruka

```
namespace MPI
{
    class sender
    {
    public:
        sender() : q(nullptr) {}

        explicit sender(safeQueue *q_) : q(q_) {}

        /// @brief Send message to queue
        template <typename Message>
        void send(Message const &msg)
        {
            if (q)
            {
```

```
                q->push(msg);
            }
        }

    private:
        safeQueue *q;
    };
}
```

Primanje poruke

Klasa za primanje poruka je ipak malo složenija od klase za slanje zbog toga što klasa za primanje mora čekati na poruke, prepoznavati tip poruke, i održavati vektor registriranih metoda za obradu poruka. Prije nego objekt `receiver` počne čekati dolazne poruke, potrebno je registrirati sve obradne metode koristeći metodu `registerMsgHandlers()`. Budući da se aplikacijski program, koji će koristiti ovu biblioteku, mora modelirati kao konačni automat, za svaku programsku nit u svakom trenutku će biti poznato koje poruke ona očekuje. Na osnovu tog znanja, prilikom promjena stanja konačnog automata, potrebno je registrirati obradne metode za poruke koje se očekuju u tom trenutku. Prilikom registracije, stvara se objekt tipa `msgHandlerBase` te se pokazivač na taj objekt sprema u privatni vektor klase `receiver`. Kad su sve metode uspješno registrirane, korisnik biblioteke mora pozvati metodu `waitForMsg()` koja će čekati na dolazne poruke tako da će blokirati čekajuću programsku nit sve dok je dijeljeni resurs prazan. U trenutku kad radna programska nit stavi poruku u dijeljeni resurs, metoda će odblokirati programsku nit te će pokušati obraditi primljenu poruku. Iteracijom kroz privatni vektor, u kojem su registrirane metode za obradu poruke, pokušat će se pronaći odgovarajuća obradna metoda za tip dolazne poruke. Po uspješnom pronalasku, poziva se odgovarajuća registrirana metoda te se prekida iteracija kroz vektor. Nakon što se iteracija prekine ili se ne pronađe odgovarajuća metoda, uništavaju se svi elementi vektora kako bi se oslobodio prostor u vektoru za primanje i obradbu sljedeće poruke prilikom sljedeće promjene stanja konačnog automata.

Kako je već spomenuto, klase `receiver` je u uskoj sprezi s klasom `msgHandlerBase` koja je zapravo predložak klasa koja poznaje tip svake metode za obradu, a ujedno i svaki tip primljene poruke.

Biblioteka je implementirana tako da se poruka i njezina obradna metoda konstruiraju koristeći isti tip, a to je tip poruke koja se šalje. Upravo je taj tip način na koji klasa `msgHandlerBase` prepoznaje koju obradnu metodu je potrebno pozvati za primljenu poruku. Budući da ova klasa poznaje i tip poruke, moguće je dinamički promijeniti tip bazne klase `messageBase` u izvedenu klasu koja specijalizira poruku. Promjenom tipa, omogućuje se pristup sadržaju primljene poruke, te se upravo taj sadržaj predaje obradnoj metodi kao ulazni argument.

Dodatno, ako klasa `msgHandlerBase` prepozna tip poruke kao `class close_queue`, tada klasa izbacuje iznimku čime prekida izvođenje programske niti te tako programska niti završava. Korisniku biblioteke savjetuje se slanje poruke tog tipa u trenutku kada je potrebno završiti programsku nit.

Listing 7.4: Implementacija klasa potrebnih za primanje i obradu poruke

```
namespace MPI
{
    class receiver
    {
    public:
        ~receiver() {}

        /// @brief Register new message handler to vector
        /// @param handler pointer to new message handler
        /// @return Status::None if send is successful otherwise error
        void registerMsgHandler(class msgHandlerBase *handler)
        {
            handlers.push_back(handler);
        }

        /// @brief Wait for messages and if received
        /// check for message validity with handlers
        void waitForMsg()
        {
            if (handlers.empty())
                return;

            auto msg = q.waitAndPop();
            for (auto i : handlers)
            {
                if (i->checkMsg(msg) == true)
                {
                    break;
                }
            }
            unregisterHandlers();
        }

        operator sender()
        {
            return sender(&q);
        }

    private:
        void unregisterHandlers()
        {
```

```
            for (auto i : handlers)
            {
                // Remove all handlers
                if (i != nullptr)
                {
                    delete i;
                    i = nullptr;
                }
            }
            // clear vector after removal
            handlers.clear();
        }

        safeQueue q;
        std::vector<msgHandlerBase *> handlers;
    };

    class close_queue {};

    class msgHandlerBase
    {
    public:
        virtual bool checkMsg(std::shared_ptr<messageBase> const &msg);
    };

    template <typename MsgType>
    class msgHandler : public msgHandlerBase
    {
    public:
        /// @brief Constructor
        msgHandler()
        {
        }

        /// @brief Constructor
        /// @param callback function for handler
        msgHandler(std::function<void(MsgType &)>
                   callback_) : callback(callback_) {}

        /// @brief Check for message validity
        /// @param msg recieved message
        /// @return true if handler received valid message
        /// and called callback otherwise false
        bool checkMsg(std::shared_ptr<messageBase> const &msg)
        {
            // If close_queue message is received,
            // throw exception to stop the infinite thread loop
```

```
            if (dynamic_cast<wrappedMessage<close_queue> *>(msg.get())
                != nullptr)
            {
                throw close_queue();
            }
            if (wrappedMessage<MsgType> *wrapper =
                    dynamic_cast<wrappedMessage<MsgType> *>(msg.get()))
            {
                callback(wrapper->contents);
                return true;
            }
            else
            {
                return false;
            }
        }

    private:
        std::function<void(MsgType &)> callback;
    };
```

Primjer registracije obradne metode u aplikacijskom programu

Kako bi korisnik ispravno registrirao metodu za obradu dolazne poruke, potrebno je konstruirati objekt klase `receiver` te pozvati njegovu metodu članicu `registerMsgHandler`. Ulazni argument metodi je novi objekt tipa `msgHandler` kojemu se mora predati tip poruke za koju će konstruirana obradna metoda biti valjana.

Prilikom konstrukcije objekta `msgHandler` potrebno je predati i pokazivač na obradnu metodu. U ovom primjeru u svrhu implementacije obradne metode iskorišten je lambda izraz, koji je jedna od značajki C++11 standarda.

Obratite pozornost na argument koji prima lambda izraz - to je sadržaj poruke koju je poslala radna programska nit.

Listing 7.5: Primjer registracije obradne metode (prva implementacija)

```
struct poruka1
{
    std::string tekst;
    explicit poruka1(std::string const &tekst_) :  tekst(tekst_) {}
};

struct poruka2
{
    int broj;
    float f;
```

```
    explicit poruka2(int broj_, float f_) :  broj(broj_), f(f_) {}
};

MPI::receiver incoming;

incoming.registerMsgHandler(
            new MPI::msgHandler<poruka1>(
                [&](poruka1 const &msg)
                {
                    // radi
                }));

incoming.registerMsgHandler(
            new MPI::msgHandler<poruka2>(
                [&](poruka2 const &msg)
                {
                    // radi
                }));

incoming.waitForMsg();
```

U ovom odsječku vidimo primjer registracije obradnih metoda za dvije poruke tipa `poruka1` i `poruka2`. Obje će obradne metode imati pristup sadržaju poruke, jer će se klasa `msgHandlerBase`, implementirana u biblioteci, pobrinuti o tome.

Prikažimo sada i dio aplikacijskog programa koji će poslati poruku tipa `poruka1`, koju će onda obraditi prethodno pokazani primjer.

```
struct poruka1
{
    std::string tekst;
    explicit poruka1(std::string const &tekst_) :  tekst(tekst_) {}
};

MPI::receiver incoming;
MPI::sender s = incoming;

s.send(poruka1("Pozdrav!"));
```

Ovdje je bitno naglasiti da je klasa za primanje poruka vlasnik dijeljenog resursa, dok klasa za slanje ima samo pokazivač na taj isti resurs. Kako bi osigurali da programske niti koriste isti komunikacijski kanal prilikom razmjene poruka, tj. isti red, klasu `sender` je potrebno inicijalizirati implicitnom pretvorbom tipa iz klase `receiver` [1].

[1] Pogledati implementaciju operatora pretvorbe u odsječku 7.4

7.2 Druga implementacija

Ova implementacija se razlikuje od implementacije prve biblioteke samo u dijelu koji obrađuje primljene poruke, a samim time drugačija je i konstrukcija glavnog programa koji koristi biblioteku.

U ovom slučaju primanje poruke započinje pozivanjem metode `wait()` koja je članica klase `receiver`.

Listing 7.6: Implementacija klase za primanje poruke

```
// Receiver.h
#pragma once

#include "SafeQueue.h"
#include "Sender.h"
#include "Dispatcher.h"

namespace MPI
{
    class Receiver
    {
        SafeQueue q;

    public:
        // operator castanja u klasu Sender
        operator Sender()
        {
            return Sender(&q);
        }
        Dispatcher wait()
        {
            return Dispatcher(&q);
        }
    };
}
```

Ta metoda konstruira objekt tipa `class Dispatcher` kojeg vraća pozivatelju, a on ima svoju metodu članicu `handle()`. Toj metodi se predaje funkcijski pokazivač na obradnu metodu poruke, vrlo slično kao i u prvoj implementaciji. Pozivanjem metode `handle()` i predavanjem specifične obradne metode, konstruira se novi objekt tipa `class Handler` koji ponovo ima istu metodu `handle()`. Uzastopnim pozivanjem metoda `handle()` dobiva se ulančani niz objekata tipa `class Handler` od kojih svaki ima pokazivač na obradnu metodu. Bitno je naglasiti da nije moguće ulančati samo objekte tipa `Dispatcher` zbog toga što primljenu poruku nije moguće raspakirati unutar tog objekta, nego je potrebno dodatno kreirati parametrizirani objekt tipa `Handler` koji će poznavati tip poruke, te će moći pristupiti njezinom sadržaju.

Listing 7.7: Klasa `Dispatcher` koja konstruira ulančane klase tipa `Handler`

```
// Dispatcher.h
#pragma once

#include "Handler.h"

namespace MPI
{
    class close_queue {};

    class Dispatcher
    {
        SafeQueue *q;
        bool chained;
        Dispatcher(Dispatcher const &) = delete;
        Dispatcher &operator=(Dispatcher const &) = delete;
        template <
            typename Dispatcher,
            typename Msg,
            typename Func>
        friend class Handler;
        void wait_and_dispatch()
        {
            for (;;)
            {
                auto msg = q->wait_and_pop();
                dispatch(msg);
            }
        }
        bool dispatch(
            std::shared_ptr<MessageBase> const &msg)
        {
            if (dynamic_cast<WrappedMessage<close_queue> *>(msg.get())
                != nullptr)
            {
                throw close_queue();
            }
            return false;
        }

    public:
        Dispatcher(Dispatcher &&other) : q(other.q), chained(other.
            chained)
        {
            other.chained = true;
        }
```

```
        explicit Dispatcher(SafeQueue *q_) : q(q_), chained(false)
        {
        }

        template <typename Message, typename Func>
        Handler<Dispatcher, Message, Func>
        handle(Func &&f)
        {
            return Handler<Dispatcher, Message, Func>(
                q, this, std::forward<Func>(f));
        }
        ~Dispatcher() noexcept(false)
        {
            if (!chained)
            {
                wait_and_dispatch();
            }
        }
    };
}
```

Za razliku od prve implementacije, nije potrebno dodatno pozivati metodu `waitForMessage()`, jer će slična metoda biti pozvana u destruktorima objekata koji su ulančani. Prilikom poziva destruktora zadnjeg objekta u ulančanom nizu, pozvat će se metoda koja će blokirati izvođenje sve dok je dijeljeni red prazan. Odrađivanje posla u destruktoru objekta je vrlo zanimljiv pristup, a u ovom nam slučaju posebno pojednostavljuje implementaciju, jer nije potrebno ručno pozivati metodu koja će čekati na primanje poruke. Ako bismo išli pozivati tu metodu izvan objekta, došli bismo u situaciju gdje bi morali poznavati parametrizaciju svakog objekta tipa `Handle` što postaje nepraktično jer rezultirala vrlo složenom ili ponekad nemogućom implementacijom.

U trenutku kad se pojavi poruka u redu, metoda će odblokirati programsku nit te će na isti način kao i prva implementacija obraditi primljenu poruku. U drugoj je implementaciji privatni vektor pokazivača na obradne metode zamijenjen ulančanim nizom objekta koji imaju pokazivače na obradne metode. Potrebno je iterirati kroz ulančani niz kako bi se pronašla odgovarajuća obradna metoda za primljeni tip poruke. Iteriranje kroz ulančane objekte pokrene destruktor samo zadnjeg u nizu ulančanih objekata. To je osigurano održavanjem varijable `chained` i provjeravanjem njezinog stanja u svakom destruktoru. Svaki objekt ima varijablu `chained` čija će imati vrijednost `true` ako se na objekt naslanja odnosno ulančava drugi objekt. Što znači da će samo zadnji objekt u ulančanom nizu imati varijablu `chained` s vrijednošću `false`.

Slično kao i kod prve implementacije, nakon iteracije kroz sve objekte uništava se posljednji u nizu ulančanih objekta, čime njegov destruktor završava s izvođenjem. Tada se nastavljaju izvršavati destruktori svih ostalih objekta, ali ovaj put bez blokiranja, jer

varijabla `chained` ima vrijednost `true`.

Isto tako, varijabla `chained` nam služi da, u slučaju da imamo samo jednu poruku koju čekamo, odgovornost za pozivanje obradne metode prebacimo s objekta tipa `Dispatcher` na objekt tipa `Handler`, kako bi uspjeli pristupiti sadržaju primljene poruke.

Listing 7.8: Isječak iz klase `Handler` koji čeka i obrađuje dolaznu poruku

```
void wait_and_dispatch()

{
    for (;;)
    {
        auto msg = q->wait_and_pop();
        if (dispatch(msg))
            break;
    }
}
bool dispatch(std::shared_ptr<MessageBase> const &msg)
{
    // Downcasting iz BaseMessage u WrappedMessage
    WrappedMessage<Msg> *wrapper = dynamic_cast<WrappedMessage<
        Msg> *>(msg.get());

    if (wrapper != nullptr)
    {
        // Ako je downcasting bio uspjesan, imamo pristup
            sadrzaju poruke
        f(wrapper->content);
        return true;
    }
    else
    {
        // Inace pokusaj sa prethodnim Handlerom
        return prev->dispatch(msg);
    }
}
```

Primjer registracije obradne metode u aplikacijskom programu

Kao što je u uvodu rečeno, druga implementacija koristi mehanizam ulančavanja objekata tipa `Handler`, od koji svaki zna koji tip poruke treba obrađivati kojom obradnom metodom. Konstrukcija ulančanih objekata započinje pozivom funkcijskog člana `wait()` klase `Reciver`, koji vraća objekt klase `Dispatcher`, te svakim daljnjim pozivanjem funkcijskog člana `handle()` konstrira se ulančani objekt tipa `Handler`.

Primjetite da druga implementacija biblioteke pruža jednostavniji način registracije obradnih metoda. Isto tako, zbog čekanja na dolaznu poruku u destruktoru obradnih objekata, nije potrebno raditi dodatni poziv funkcije za čekanje nakon registracije obradnih metoda, kao što je to bio slučaj kod prve implementacije biblioteke.

Listing 7.9: Primjer ulančavanja objekta za obradu poruke (druga implementacija)

```
struct poruka1
{
    std::string tekst;
    explicit poruka1(std::string const &tekst_) :  tekst(tekst_) {}
};

struct poruka2
{
    int broj;
    float f;
    explicit poruka2(int broj_, float f_) :  broj(broj_), f(f_) {}
};
struct poruka3
{
};

MPI::Receiver incoming;

incoming.wait()
            .handle<poruka1>(
                [&](poruka1 const &msg)
                {
                    // radi
                })
            .handle<poruka2>(
                [&](poruka2 const &msg)
                {
                    // radi
                })
            .handle<poruka3>(
                [&](poruka3 const &msg)
                {
                    // radi
                });
```

U ovom primjeru vidimo primjer registracije obradnih metoda za tri poruke tipa `poruka1`, `poruka2` i `poruka3`. Vidimo da su objekti ulančani pozivima metode `handle()`.

Ulančavanje se izvodi u sljedećem formatu:

```
incoming.wait().handle<type1>(arg1).handle<type2>(arg2)...
```

Ovdje će također sve tri obradne metode imati pristup sadržaju primljenih poruka, jer će se klasa `Handler`, implementirana u biblioteci, pobrinuti o tome.

Dio aplikacijskog koda koji šalje poruku jednak je onome prikazanom za prvu implementaciju biblioteke (pogledajte odsječak 7.6).

Poglavlje 8

Algoritmi

U ovom poglavlju dat ću pregled dva algoritma koja su implementirana koristeći gore opisane biblioteke u svrhu njihovog testiranja. Prvi algoritam je simulacija rada bankomata, dok je drugi klasični problem poznat kao problem pet filozofa.

U oba primjera nastojat ću prikazati način ispravnog korištenja biblioteka kako bi se implementirao malo složeniji program. Obje implementacije dostupne su u dodatku ovom diplomskom radu.

8.1 Bankomat

Simulaciju rada bankomata možemo podijeliti u poslove koji moraju osigurati:

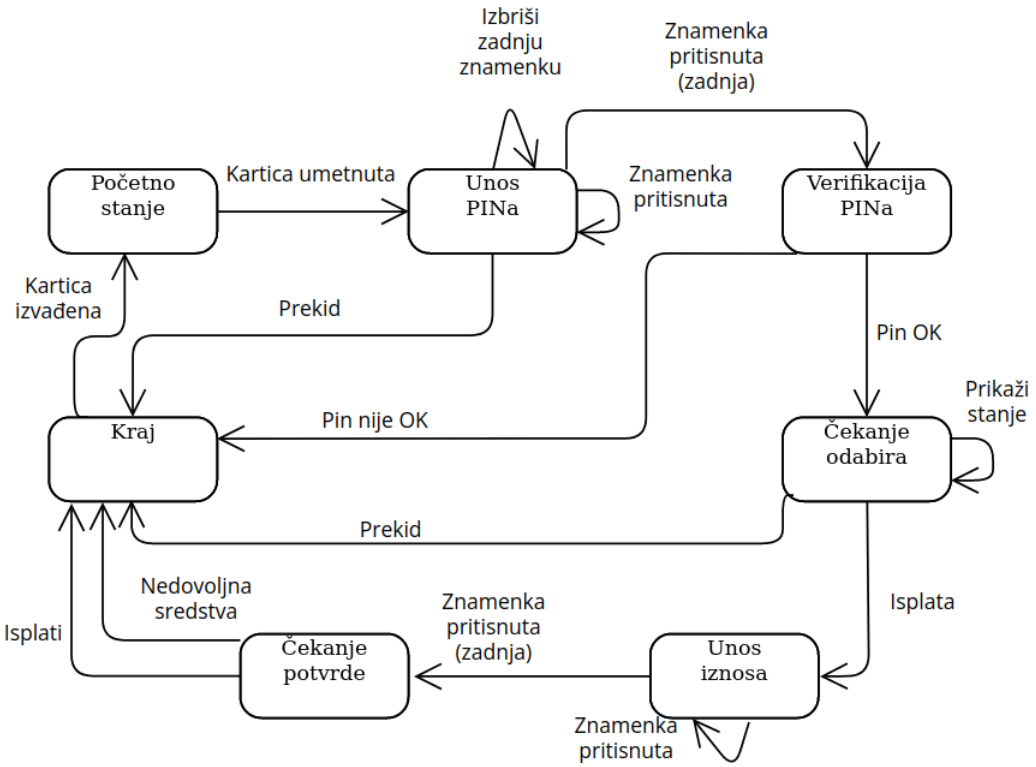
- komunikaciju bankomata kao fizičkog uređaja s korisnikom
- komunikaciju s bankom u svrhu verifikacije PIN-a i provjere stanja na bankovnom računu
- kontrolu ispisa poruka, umetanja i vraćanja kartice, i kontrolu pritiskanja tipki.

Taj rad implementiran je koristeći tri nezavisne programske niti, od kojih jedna upravlja bankomatom kao fizičkim uređajem, druga koja upravlja bankomatom kao logičkim uređajem i treća koja osigurava komunikaciju prema banci. Na primjer, programska nit koja upravlja bankomatom kao fizičkim uređajem poslat će poruku prema programskoj niti koja upravlja bankomatom kao logičkim uređajem kada korisnik umetne bankovnu karticu u bankomat ili kad pritisne tipku. Potom će logička programska nit poslati fizičkoj poruku sa sadržajem, npr. stanja računa i tako u krug.

Model bankomata opisan je automatom stanja. U svakom stanju, programska nit čeka na odgovarajuću poruku, koju potom obrađuje. U nekim slučajevima primljena poruka može rezultirati promjenom stanja u automatu.

Sva moguća stanja u kojima se program može naći opisana su modelom automata stanja na slici 8.1.

Slika 8.1: Jednostavni model automata stanja za bankomat

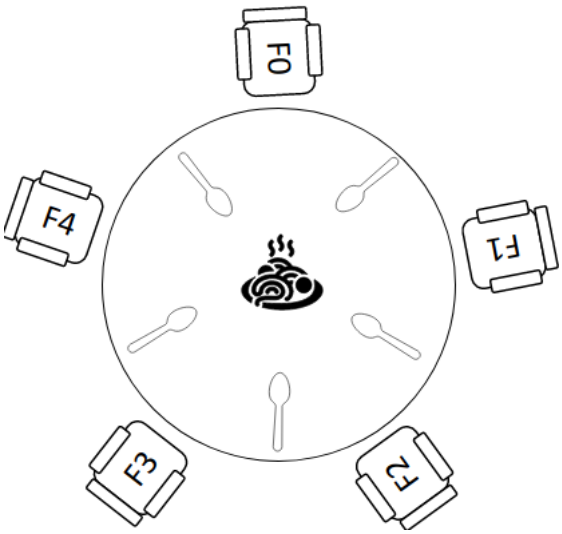


U početnom stanju model čeka da korisnik umetne karticu. Jednom kad je kratica umetnuta, model čeka korisnika da unese četveroznamenkasti PIN, znamenku po znamenku. Kad su unesene sve četiri znamenke, model će provjeriti ispravnost PIN-a. Ako PIN nije točan, bankomat će korisniku vratiti karticu. A ako je točan, model će korisniku ponuditi opciju provjere stanja na računi i opciju isplate. Ako je korisnik odabrao prikaz stanja, bankomat će stanje prikazati, te korisnika ponovo vratiti na prethodni odabir. Ako je korisnik odabrao isplatu, model će čekati da korisnik unese troznamenkasti iznos zaokružen na 50. Jednom kad je iznos unesen, model će provjeriti dostupnost traženih sredstava. Ako sredstva nisu dostupna, bankomat će odbiti isplatu i korisniku vratiti karticu. Ako je iznos dostupan, bankomat će isplatiti traženi iznos te vratiti karticu i otići ponovo u početno stanje.

8.2 Problem pet filozofa

Problem pet filozofa koji večeraju je klasični problem sinkronizacije koji kaže da 5 filozofa sjedi za okruglim stolom koji misle i jedu naizmjenično. Naime, na stolu se nalazi 5 štapića za jelo i na sredini stola je jedna zdjela s hranom. Kako bi jeo, svaki filozof treba i desni i lijevi štapić. Dakle, svaki pojedinačni štapić mogu koristiti samo dva susjedna filozofa. Filozof može jesti samo kad su u istom trenutku dostupni i lijevi i desni štapić. U slučaju kada jedan od štapića nije dostupan, filozof spušta onaj drugi (lijevi ili desni) na stol i ponovo odlazi misliti [3].

Slika 8.2: Filozofi za stolom



Prikažimo sada kako okvirno izgleda jedno od pojednostavljenih rješenja, a koje je implementirano u dodatku ovom diplomskom radu korištenjem obje verzije biblioteke.

Svaki od n štapića može biti čist ili prljav, te se u jednom trenutku može nalaziti samo kod jednog filozofa. Na početku, svi su štapići prljavi. Također, štapići su na početku podijeljeni tako da se svaki štapić, koji mogu dijeliti dva susjedna filozofa, nalazi kod filozofa s nižim rednim brojem (indeksom programske niti). Slijedom navedenog, filozof s indeksom 0 na početku ima dva štapića, a filozof s indeksom $n - 1$ niti jedan. Svi filozofi slijede ova pravila:

1. filozof jede ako je gladan i ako ima oba štapića (bez obzira jesu li čisti ili prljavi).
2. nakon jela, oba korištena štapića filozof šalje svojim susjedima.
3. ako filozof želi jesti, pričeka dok ne dobije oba štapića.

Iz navedenih pravila je vidljivo da filozof ne udovoljava zahtjevu za čistim štapićem, tj. filozof će poslati svoj štapić susjedu tek nakon jela. Isto tako, ako filozof misli (trenutno nije gladan), nije obvezan odmah udovoljiti zahtjevima drugih filozofa.

```
Nit(i)
{  misli (slucajan broj sekundi);

   dok (nemam oba stapica) {

      ponavljaj {
         cekaj lijevi stapic;
         cekaj desni stapic;
      } dok ne dobijes trazene stapice;
   }
   jedi;
   posalji oba stapica susjedima;
}
```

Implementirani algoritam pokazuje na koji način je moguće koristiti razvijene biblioteke u slučaju komunikacije više programskih niti koje obrađuju objekte koji su predstavljeni istim tipom klase. Prisjetimo se da su u slučaju bankomata bila tri objekta različitih klasa.

Bibliografija

[1] http://www.cs.columbia.edu/~martha/courses/3827/au14/advanced-topics.pdf.

[2] https://cppreference.com.

[3] https://www.fer.unizg.hr/_download/repository/vjezba1[3].html.

[4] P. Pacheco, *Parallel Programming*, Morgan Kaufman, 2011.

[5] A. Williams, *C++ Concurrency in action*, Manning, 2019.

[6] J. Šribar i B. Motik, *Demistificirani C++*, Element d.o.o., Zagreb, 2014.

Sažetak

U ovom diplomskom radu dan je pregled razvoja programske podrške za višedretvene aplikacije na principima dijeljene memorije i slanja poruka među programskim nitima. Obje metode promatrane su u C++ programskoj okolini.

Kao prirodni način implementacije višedretvenih programa u C++ programskoj okolini nameće se princip dijeljene memorije, upravo jer programske niti međusobno dijele adresni prostor te stoga nije potrebno implementirati posebne mehanizme za dijeljenje resursa među njima. Međutim, u tom se slučaju javlja problem prilikom zaštite dijeljenih resursa od nekonzistentnosti, pa je potrebno koristiti veći broj lokota i njihovo međusobno ugnježđivanje. Stoga su složeniji programi podložniji pogreškama uzrokovanim potpunim zastojem ili stanjima natjecanja.

S druge strane, princip koji se zasniva na bazi slanja poruka manje je sklon ovakvim pogreškama, ali se potrebno dodatno pobrinuti o mehanizmima razmjene poruka i o broju dijeljenih resursa. Broj dijeljenih resursa potrebno je svesti na minimum, po mogućnosti na samo jedan, koji bi zapravo predstavljao medij za razmjenu poruka među programskim nitima - najčešće red. Ovaj pristup zahtjeva dodatne procesorske i memorijske resurse u usporedbi s pristupom s dijeljenom memorijom.

U diplomskom radu razvijena je biblioteka za razmjenu poruka među programskim nitima te su, korištenjem razvijene biblioteke, implementirani algoritmi simulacije rada bankomata i problem pet filozofa. Medij za razmjenu poruka upravo je red koji je osiguran sa samo jedim lokotom. Tim lokotom je osigurana konzistentnost elemenata u redu, pa je stoga takav red sigurno koristiti u višedretvenoj okolini. Okosnicu biblioteke čini taj red, koji je upakiran u klase za slanje i primanje poruka, te je time njegova implementacija skrivena od korisnika biblioteke. Dodatno je osigurana fleksibilnost razmjene poruka tako da se različiti tipovi poruka mogu slati i primati kroz jedan te isti medij.

Na kraju rada zaključujem da je pristup s dijeljenom memorijom bolji prilikom razvoja jednostavnih sustava kojima mehanizmi zaštite dijeljenih resursa nisu složeni. Dok s druge strane, pristup s razmjenom poruka ima bolje performanse u složenijim aplikacijama iz perspektive otpornosti na nepredviđene pogreške, ali zahtjeva veće ulaganje na početku u razvoj mehanizma razmjene poruka i skuplji je po pitanju hardverskih resursa u usporedbi s prethodnim pristupom.

Summary

In this master's thesis, the outline of multi-threaded application development based on shared memory and message passing methods, is given. Both methods were observed in C++ environment.

Straightforward way of multi-threaded application development in C++ is shared memory approach, due to the fact that threads share the address space, hence it is not necessary to implement further sharing mechanisms. However, in that case many issues related to the data integrity can occur, so the programmers have to use a lager number of locks and their nesting. The outcome of such approach, in more complex solutions, is a higher risk of having more issues caused by deadlocks or race conditions.

On the other hand, development approach based on message passing between threads is less likely to suffer from a higher number of such issues, but it is necessary to take care of additional data sharing mechanisms and of number of shared data. The number of shared data should be reduced as much as possible, preferably to only one, which will then serve as a medium for message passing. The most usual data structure for this solution is the queue. Compared to shared memory, this approach requires additional CPU and memory resources.

In this master's thesis, the library for message passing was developed, and the algorithms for cash machine simulation and Dining Philosophers issue was implemented using that library. The medium for message passing is the queue which is secured using a single lock. That lock assures the data integrity in the queue in the concurrent program, so the queue can be considered as a thread safe. The central part of the library is right that queue, which is wrapped in the higher classes for sending and receiving messages. That way the queue implementation is hidden for the library users. Additionally, the flexibility of message sharing is ensured in a way that different message structures can be shared through the same media.

The conclusion, at the end of the thesis, is that approach with shared memory is better to be used when locking mechanisms requires simple solutions. While, the message passing approach has better performance in more complex solutions from the undefined behavior point of view, but requires big initial invest to develop message passing interface, and it is also more expensive in terms of hardware resources, compared to the first approach.

Životopis

Rođena sam u Ljubuškom, u Bosni i Hercegovini 04. listopada 1995.. Osnovnoškolsko obrazovanje završila sam u Osnovnoj školi Marka Marulića u Ljubuškom. Nakon toga upisujem Matematičku gimnaziju u Dubrovniku, gdje maturiram sa odličnim uspjehom. Preddiplomski studij Matematike na matematičkom odsjeku Prirodoslovno-matematičkog fakulteta u Zagrebu, upisala sam 2014., a završila 2020. godine, te time stekla titulu prvostupnice matematike. Na istom odsjeku, 2020. godine, upisala sam i diplomski studij Računarstva i matematike.