Niti (*Threads*)

Model procesa predstavljen u prethodnom poglavlju pretpostavljao je da je proces izvršni program s jednom niti izvršavanja. Međutim, skoro svi moderni operativni sistemi pružaju infrastrukturu koja omogućava da proces sadrži više niti izvršavanja. U ovom poglavlju uvodimo koncepte računarskih sistema sa više niti, a prikazaćemo i programski intefejs za biblioteke koje omogućavaju rad sa višestrukim nitima. Proučavaćemo niz pitanja koja se odnose na programiranje sa višestrukim nitima, kao i njihov uticaj na dizajn operativnih sistema. Na kraju, videćemo kako Linux operativni sistem podržava niti na nivou kernela.

1 Pregled

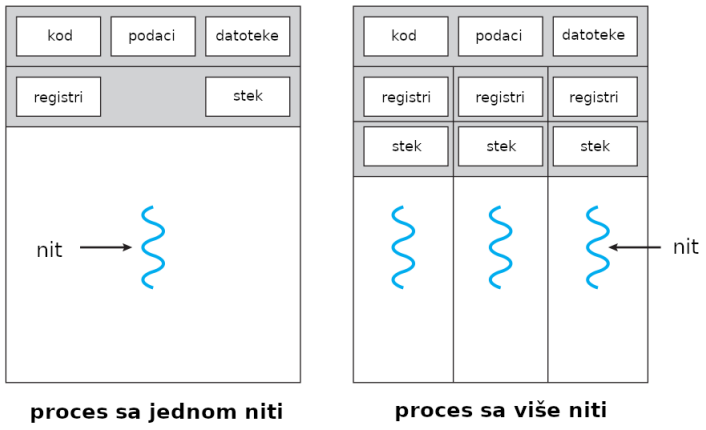
Nit je osnovna jedinica iskorišćenosti procesora. Ona sadrži ID niti, programski brojač, skup registara i stek. Sa drugim nitima koje pripadaju istom procesu deli sekciju sa kodom, podacima i druge resurse operativnog sistema, kao što su otvorene datoteke i signali. Tradicionalni proces ima jednu nit izvršavanja, ali proces koji sadrži više niti izvršavanja, može obavljati više zadataka istovremeno. Slika 1 prikazuje razliku između tradicionalnog procesa sa jednom niti (*single threaded process*) i procesa sa više niti (*multithreaded process*).

1.1 Motivacija

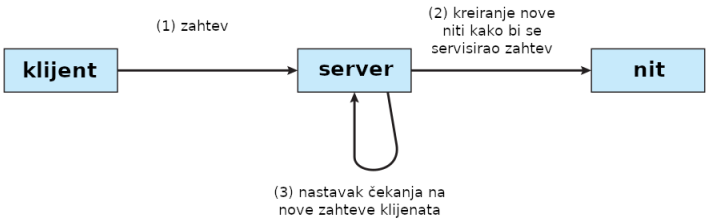
Većina softverskih aplikacija koje rade na savremenim računarima izvršavaju se u višestrukim nitima. Sama aplikacija se obično realizuje kao nezavisan proces sa nekoliko niti izvršavanja. Web pretraživač može prikazivati slike ili tekst u okviru jedne niti, dok drugu koristi za preuzimanje sadržaja sa Interneta. Program za obradu teksta može imati jednu nit za prikazivanje grafike, drugu nit za reagovanje na pritiske tastera od strane korisnika i treću nit za proveru pravopisa i gramatike u pozadini. U principu, sve aplikacije se, osim toga, mogu projektovati tako da se iskoriste maksimalno prednosti obrade na višejezgarnim sistemima. Takve aplikacije mogu da izvršavaju istovremeno nekolicinu intenzivnih zadataka u paraleli, pri čemu se svaki izvršava na zasebnom CPU jezgru.

U određenim situacijama može se od jedne aplikacije zahtevati izvršavanje nekoliko sličnih zadataka. Na primer, web server prihvata zahteve klijenta za prikaz web stranice i propratne slike, zvuka itd. Opterećen web server može imati nekoliko (možda i hiljade) klijenata koji mu istovremeno pristupaju. Ako bi se web server pokrenuo kao tradicionalni proces sa jednom niti izvršavanja, on bi mogao istovremeno da servisira samo jednog klijenta, dok bi ostali morali da čekaju veoma dugo da se njegov zahtev servisira.

Jedno rešenje je pokretanje servera kao jedinstvenog procesa koji prihvata zahteve. Kada server primi zahtev, on kreira zaseban proces da ga servisira. U stvari, ova metoda stvaranja procesa bila je u uobičajenoj upotrebi pre nego što su niti postale popularne. Međutim, stvaranje procesa zahteva mnogo vremena i zahteva mnogo resursa. Ako će novi proces obavljati iste zadatke kao i postojeći proces, zašto onda odabrati takav pristup? Generalno je efikasnije koristiti jedan



Slika 1: Proces sa jednom i više niti izvršavanja



Slika 2: Arhitektura servera sa više niti izvršavanja

proces koji sadrži više niti. Ako se proces web servera izvršava u više niti, server će kreirati zasebnu nit koja će osluškivati zahteve klijenata. Kada je zahtev pristigao, umesto da kreira novi proces, server stvara novu nit da bi se servisirao pristigli zahtev i nastavlja sa čekanjem na nove zahteve. To je prikazano na slici 2.

Niti takođe igraju značajnu ulogu u *pozivu udaljenih procedura* (RPC). Kao što smo pominjali u prethodnom poglavlju, RPC omogućava međuprocesnu komunikaciju pružajući mehanizam sličan uobičajenim pozivima funkcija ili procedura. RPC serveri su obično implementirani u više niti izvršavanja. Kada server primi poruku, on je servisira u okviru zasebne niti. Ovo omogućava serveru da istovremeno servisira nekoliko zahteva.

Konačno, većina kernela operativnog sistema je danas implementirana u više niti izvršavanja (*multithreaded kernel*). Nekolicina niti se izvršavaju u kernelu, pri čemu svaka nit obavlja određeni zadatak, poput upravljanja uređajima, upravljanja memorijom ili opsluživanjem prekida. Na primer, Linux koristi kernelsku nit za upravljanje količinom slobodne memorije u sistemu.

1.2 Prednosti

Prednosti programa sa višestrukim nitima mogu se podeliti na četiri glavne kategorije:

1. Odzivnost (*responsiveness*). Višestruke niti kod interaktivnih aplikacija mogu omogućiti programu da nastavi sa radom čak i ako je deo njega blokiran ili izvodi vremenski zahtevnu operaciju, čime se poboljšava odziv ka korisniku. Ovaj kvalitet je posebno značajan prilikom dizajniranja korisničkog interfejsa. Na primer, šta se dešava kada korisnik klikne na dugme što će rezultovati izvršavanjem dugotrajne operacije. Aplikacija sa jednom niti neće biti u mogućnosti da reaguje na nove interakcije sa korisnikom, sve dok se ta operacija ne završi. Nasuprot tome, ako se dugotrajna operacija izvršava u zasebnoj niti, aplikacija ostaje dostupna korisniku, u smislu da je i dalje moguća interakcija sa aplikacijom od strane korisnika.

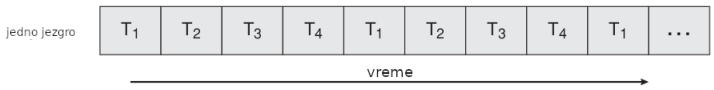
2. Deljenje resursa. Procesi mogu da dele resurse samo prethodno opisanim tehnikama kao što su deljena memorija i sistem za prenošenje poruka. Programer mora eksplicitno koristiti takve tehnike, uz popriličan dodatni napor. Sa druge strane, niti dele memoriju i resurse procesa kojem one podrazumevano pripadaju. Prednost deljenja koda i podataka je ono što omogućava aplikaciji postojanje nekoliko različitih aktivnih niti u okviru istog adresnog prostora.

3. Ekonomičnost. Alociranje memorije i resursa prilikom kreiranja procesa je skupa operacija. Obzirom na to da niti dele resurse procesa kojem pripadaju, ekonomičnije je kreirati nit i nju koristiti prilikom *zamene konteksta*. Proračun konkretnog poboljšanja na ovaj način nije jednostavan, ali, generalno, mnogo je više vremena potrebno za kreiranje i upravljanje procesima nego nitima. Na primer, u Solaris operativnom sistemu kreiranje procesa je trideset puta sporije nego što je kreiranje niti, a zamena konteksta je oko pet puta sporija.

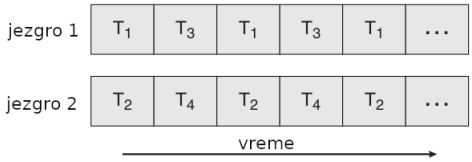
4. Skalabilnost. Prednosti višestrukih niti mogu biti još veće na multiprocesorskim arhitekturama, gde se niti mogu paralelno izvršavati na različitim CPU jezgrima. Proces sa jednom niti izvršavanja može da se izvršava samo na jednom procesoru, bez obzira koliko ih ima na raspolaganju i o ovome ćemo detaljnije pričati u sledećem odeljku.

2 Višejezgarno programiranje (*multicore programming*)

Tokom istorije razvoja računarskih sistema, kao odgovor na potrebu za poboljšanjem računarskih performansi, jednoprocesorski sistemi su evoluirali u multiprocesorske sisteme. Poslednji, sličan trend u dizajniranju sistema je uvođenje višejezgarnih arhitektura u okviru jednog čipa, gde se svako jezgro pojavljuje kao zaseban procesor u okviru operativnog sistema. Bez obzira da li se



Slika 3: Izvršavanje više niti na jednojezgarnom procesoru



Slika 4: Izvršavanje više niti na višejezgarnom procesoru

jezgra nalaze na zasebnim ili istim CPU čipovima, ove sisteme nazivamo višejezgarnim ili višeprocesorskim sistemima. Višejezgarno programiranje pruža mehanizam za efikasniju upotrebu ovih višejezgarnih arhitektura i poboljšanu *konkurentnost* (*concurrency*). Razmotrimo aplikaciju sa četiri niti. U sistemu sa jednim jezgrom, *konkurentnost* (istovremenost) znači samo da će se izvršavane niti smenjivati sa proticanjem vremena (slika 3), jer je jezgro za obradu sposobno da izvršava samo jednu nit istovremeno.

Međutim, u sistemu sa više jezgara, *konkurentnost* znači da niti mogu da se izvršavaju u paraleli, jer sistem može dodeliti zasebnu nit svakom jezgru (slika 4).

Sistem je *paralelan* ako može istovremeno da obavlja više zadataka. Suprotno tome, *konkurentni* sistem podržava više zadataka, omogućavajući svim zadacima da napreduju, ali ne nužno istovremeno. Dakle, moguće je imati *konkurentnost* bez *paralelizma*. Pre pojave multijezgarnih arhitektura, većina računarskih sistema imala je samo jedan procesor. CPU planeri dizajnirani su tada tako da omoguće iluziju paralelizma brzim prebacivanjem između procesa u sistemu, omogućavajući tako svakom procesu da napreduje. Takvi procesi su se odvijali *konkurentno*, ali ne i *paralelno*. Kako su softverski sistemi narasli sa desetina niti na hiljade niti, CPU dizajneri su poboljšali performanse sistema dodavanjem hardvera za podršku višestrukim nitima. Moderni Intel procesori često podržavaju dve hardverske niti po jezgru, dok Oracle T4 CPU podržava osam hardverskih niti po jezgru. Ova podrška znači da se u jezgro može učitati više softverskih niti u cilju brže zamene. Računari sa više jezgara bez sumnje će i dalje povećavati broj jezgara i poboljšavati hardversku podršku za niti.

2.1 Amdalov zakon (Ahmdal's law)

Amdalov zakon je predstavljen formulom koja definiše potencijalno poboljšanje performansi usled dodavanja jezgara za procesiranje, aplikaciji koja ima

kako sekvencijalne (serijske, ne-paralelne) blokove koda, tako i blokove koda koji se mogu paralelizovati.

Ako je S deo aplikacije koji se mora izvršavati sekvencijalno na sistemu sa N jezgara, formula definiše ubrzanje rada kao

$$Speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

Kao primer, pretpostavimo da imamo aplikaciju kod koje se 75% koda može izvršavati u paraleli (npr petlje sa poznatim brojem iteracija), a 25% koda se mora izvršavati sekvencijalno (serijski). Ako pokrenemo takvu aplikaciju na sistemu sa dva jezgra procesiranja, dobićemo ubrzanje od 1.6 puta. Ako bismo dodali još dva (ukupno četiri), ubzanje bi bilo 2.28 puta.

Interesantan detalj u vezi sa Amdalovim zakonom je taj da kako $N \to \infty$ ubrzanje konvergira ka $\frac{1}{S}$. Na primer, ako imamo 40% sekvencijalnog koda u okviru aplikacije, maksimalno ubrzanje koje možemo postići je 2.5 puta, nezavisno od broja jezgara koje dodajemo sistemu. Ovo je fundamentalni princip Amdalovog zakona: sekvencijalni deo aplikacije može imati disproporcionalan uticaj na performanse koje dobijamo dodavanjem jezgara za procesiranje.

2.2 Izazovi u programiranju

Trend u višejezgarnim sistemima i dalje vrši pritisak na dizajnere sistema i programere aplikacija kako bi bolje iskoristili više jezgara koje su im na raspolaganju. Dizajneri operativnih sistema moraju razviti algoritme raspoređivanja koji koriste više jezgara, kako bi omogućili paralelno izvršavanje prikazano na slici 4. Za aplikativne programere, izazov je da modifikuju postojeće programe, kao i da dizajniraju nove programe tako da se izvršavaju u višestrukim nitima.

Sve u svemu, možemo identifikovati pet oblasti koje predstavljaju izazove u programiranju višejezgarnih sistema:

1. Prepoznavanje zadataka. Ovo uključuje detaljno analiziranje aplikacija u cilju pronalaženja kritičnih putanja i područja koji se mogu podeliti na odvojene, konkurentne zadatke. U idealnom slučaju, zadaci su sami po sebi nezavisni jedni od drugih i mogu se paralelno izvršavati na zasebnim jezgrima.

2. Balans. Iako uspešno identifikuju zadatke koji se mogu izvršavati paralelno, programeri moraju takođe osigurati da zadaci obavljaju jednaku količinu posla. U nekim slučajevima određeni zadatak možda neće dodati dovoljno vrednosti celokupnom procesu kao ostali zadaci, u ovom kontekstu. Samim tim, upotreba zasebnog jezgra za izvršavanje tog zadatka možda nije vredna onoga što dobijamo.

3. Raspodela podataka. Baš kao što su aplikacije podeljene na odvojene zadatke, podaci kojima oni pristupaju i kojima manipulišu moraju se podeliti da bi se odvojeni zadaci mogli izvršavati na zasebnim jezgrima.

4. Zavisnost od podataka. Podaci kojima pristupaju zadaci se moraju ispitati u cilju provere da li postoje međuzavisnosti između dva ili više zadataka. Kada jedan zadatak koristi podatke koji se koriste i od strane drugog, programeri moraju osigurati da je izvršavanje zadataka sinhronizovano kako bi se prilagodili toj međuzavisnosti podataka. O ovome ćemo detaljno pričati na jednom od narednih predavanja.

5. Testiranje i uklanjanje grešaka. Kada se program pokreće paralelno na više jezgara, mnogo različitih putanja izvršavanja dolazi u obzir. Testiranje i uklanjanje grešaka (debagovanje) takvih konkurentnih programa inherentno je znatno teže u poređenju sa testiranjem i uklanjanjem grešaka kod aplikacija sa jednom niti.

Zbog svih ovih izazova, veliki broj programera i projektanata softvera tvrdi da će, kao posledica napretka višejezgarnih sistema u budućnosti, biti neophodan potpuno novi pristup u dizajniranju softverskih sistema. Slično tome, mnogi nastavnici informatike i programiranja smatraju da se razvoj softvera mora podučavati uz veći naglasak na paralelnom programiranju, što do sada svakako nije bio slučaj.

2.3 Tipovi paralelizma

Postoje dve vrste paralelizma: paralelizam podataka i paralelizam zadataka.

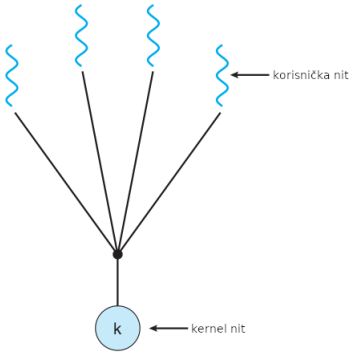
Paralelizam podataka fokusiran je na distribuciju podskupova datog skupa podataka na više računarskih jezgara i obavljanje iste operacije na svakom jezgru. Razmotrimo, na primer, sabiranje elemenata niza veličine N. U jednojezgarnom sistemu, jedna nit će prosto sabrati elemente [0, . . . , N - 1]. Na dvojezgarnom sistemu, međutim, nit A, koja se izvršava na jezgru 0, može izračunati zbir elemenata [0, . . . , N / 2 - 1], dok bi nit B, koja se izvršava na jezgru 1, mogla sabrati elemente [N / 2, . . ., N - 1]. Dve niti bi potpuno paralelno radile na zasebnim jezgrima.

Paralelizam zadataka uključuje distribuciju ne podataka, već zadataka (niti) na više jezgara. Svaka nit izvodi jedinstvenu operaciju. Različite niti mogu raditi na istim podacima ili mogu raditi na različitim podacima. Nasuprot primeru od gore, primer paralelizma zadataka može uključivati dve niti, od kojih svaka izvodi drugačije izračunavanje na datom nizu elemenata. Niti i tada rade paralelno na zasebnim računarskim jezgrima, ali svaka vrši drugačiju operaciju.

U osnovi, paralelizam podataka podrazumeva distribuciju podataka na više jezgara, dok paralelizam zadataka podrazumeva distribuciju zadataka na više jezgara. U praksi, međutim, malo aplikacija strogo sledi paralelizam podataka ili zadataka i u većini slučajeva aplikacije koriste hibrid ove dve strategije.

3 Modeli višestrukih niti

Naša dosadašnja diskusija tretirala je niti u generičkom smislu. Međutim, podrška za niti može se omogućiti kako na korisničkom nivou, za korisničke



Slika 5: Model „više na jedan“

niti, tako i u okviru kernela, za kernel niti. Korisničke niti su podržane povrh kernela i njima se upravlja bez podrške kernela, dok niti kernel-a podržava i direktno njima upravlja operativni sistem. Gotovo svi savremeni operativni sistemi - uključujući Windows, Linux, Mac OS i Solaris - podržavaju kernel niti.

Konačno, veza mora postojati između korisničkih niti i kernel niti, jer svaka korisnička nit može, u bilo kom trenutku, zatražiti intervenciju kernela putem sistemskog poziva. U ovom odeljku ćemo razmotriti tri uobičajena načina uspostavljanja takvog odnosa:

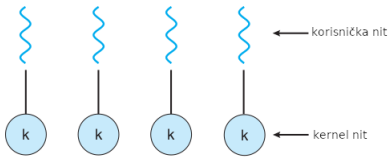
1. model „više na jedan“ (*many-to-one*),
2. model „jedan na jedan“ (*one-to-one*) i
3. model „više na više“ (*many-to-many*).

3.1 Model „Više na jedan“

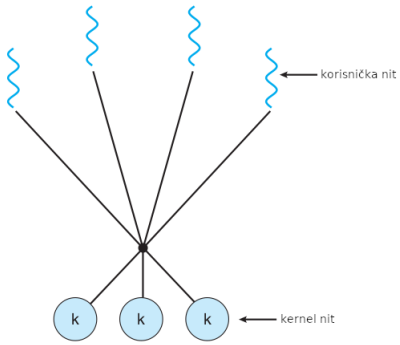
Model „više na jedan“ (slika 5) mapira višestruke niti korisničkog nivoa u jednu nit kernela. Kontrolu niti vrši biblioteka za podršku nitima u korisničkom prostoru, tako da je sama kontrola efikasna. Međutim, ceo proces će se blokirati ako nit pozove blokirajući sistemski poziv. Takođe, budući da samo jedna nit može pristupiti kernelu u datom trenutku, više niti se ne može paralelno izvršavati čak i na višejezgarnim sistemima. Vrlo mali broj sistema i danas koristi ovoj model zbog nemogućnosti iskorišćenja više jezgara za obradu.

3.2 Model „Jedan na jedan“

Model „jedan na jedan“ mapira svaku korisničku nit u odgovarajuću kernel nit. Ovakav pristup omogućava znatno više konkurentnosti u poređenju sa prethodnim modelom, omogućavajući drugoj niti da se izvršava čak i ako je prva



Slika 6: Model „jedan na jedan“

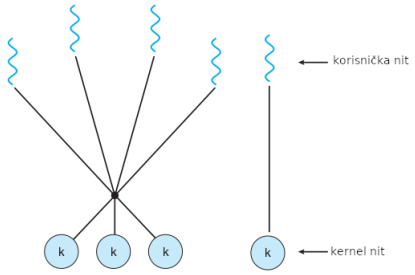


Slika 7: Model „više na više“

pozvala blokirajući sistemski poziv. Takođe, ovakav pristup omogućava paralelno izvršavanje više niti na višejezgarnim sistemima. Jedini nedostatak ovakvog pristupa leži u činjenici da kreiranje svake korisničke niti zahteva kreiranja kernel niti. Obzirom na to da „dodatna cena“ usled kreiranja kernel niti može znatno narušiti performanse aplikacije, većina implementacija ovog modela ograničava broj niti koje će biti podržane od strane sistema. Linux, kao i čitava familija Windows operativnih sistema, implementira ovakav model.

3.3 Model „Više na više“

Model „više na više“ (Slika 7) multipleksira višestruke niti na nivou korisnika na manji ili jednak broj kernel niti. Broj kernel niti koji se koristi može biti specifičan bilo za određenu aplikaciju ili za određenu mašinu (aplikaciji se može dodeliti više kernel niti u višejezgarnom sistemu nego u slučaju jednog CPU). Razmotrimo uticaj ovog dizajna na konkurentnost. Dok model „više na jedan“ omogućava programeru da kreira onoliko korisničkih niti koliko želi, to ne rezultuje istinskom konkurentnošću, jer kernel može biti korišćen samo od strane jedne niti u datom trenutku. Model „jedan na jedan“ omogućava veću konkurentnost, ali programer mora biti oprezan da ne kreira previše niti unutar aplikacije (a u nekim slučajevima može biti ograničen i broj niti koje može da kreira). Model „više na više“ neće biti pogođen ni jednim od ovih nedostataka:



Slika 8: Dvostepeni model povezivanja

programeri mogu kreirati onoliko korisničkih niti koliko je potrebno, a odgovarajuće kernel niti mogu se aktivirati paralelno na multiprocesorskom sistemu. Takođe, kada nit obavlja sistemski poziv koji blokira, kernel može aktivirati drugu nit za izvršenje.

Jedna varijacija modela „više na više“ i dalje multipleksira većinu korisničkih niti na manji ili jednak broj kernel niti, ali takođe omogućava da se korisnička nit veže za jednu eksplicitnu nit kernela. Ova varijacija se ponekad naziva i dvostepenim modelom (slika 8). Operativni sistem Solaris podržavao je dvostepeni model u verzijama operativnog sistema starijim od Solaris 9. Međutim, počevši od Solaris 9, ovaj operativni sistem koristi model jedan na jedan, takođe.

4 Biblioteke za podršku nitima

Ove biblioteke pružaju programeru API (*Application Programming Interface*) za kreiranje i upravljanje nitima. Postoje dva osnovna načina implementacije biblioteke za podršku nitima. Prvi pristup je implementacija biblioteke u potpunosti u korisničkom prostoru bez podrške od strane kernela. Sav kod i strukture podataka koji se koristi u biblioteci nalaze se u korisničkom prostoru. To znači da pozivanje funkcije unutar biblioteke rezultuje pozivom lokalne funkcije u korisničkom prostoru, a ne sistemskim pozivom.

Drugi pristup je implementacija biblioteke na nivou kernela, podržane direktno od strane operativnog sistema. U ovom slučaju, kod i strukture podataka za biblioteku postoje u kernel adresnom prostoru. Pozivanje API funkcije za biblioteku obično rezultuje sistemskim pozivom ka kernelu.

Tri glavne biblioteke za podršku nitima su danas u upotrebi: POSIX Pthreads, Windows i Java. Pthreads, proširena implementacija POSIX standarda, može biti implementirana ili na nivou korisnika ili kao biblioteka na nivou kernela. Windows biblioteka za poršku nitima je biblioteka na nivou kernela i dostupna je na Windows sistemima. *Java Thread API* omogućava da se niti kreiraju i da se njima upravlja direktno iz Java programa. Međutim, s obzirom da se u većini slučajeva JVM pokreće iz konteksta glavnog operativnog sistema,

Java Thread API se obično implementira koristeći biblioteku za podršku nitima koja je dostupna na datom sistemu. To znači da se u Windows sistemima Java niti obično implementiraju pomoću Windows API-ja, dok Unix i Linux sistemi uglavnom koriste Pthreads.

Za POSIX i Windows niti, svi podaci deklarisani globalno - tj. deklarisani izvan bilo koje funkcije — dele se između svih niti koje pripadaju istom procesu. Pošto u Javi ne postoji svest o *globalnim podacima*, pristup deljenim podacima mora biti eksplicitno dogovoren između niti. Podaci deklarisani kao lokalni za funkciju obično se čuvaju na steku. Pošto svaka nit ima svoj stek, svaka nit ima svoju kopiju lokalnih podataka.

U ostatku ovog odeljka opisujemo osnovne principe stvaranje niti pomoću POSIX Pthreads biblioteke i Java Threads API. Kao ilustrativan primer, dizajniramo program sa više niti koji vrši sabiranje ne-negativnih celih brojeva u zasebnoj niti:

$sum = \sum_{i=0}^{N} i$

Na primer, ako je N=5, ova funkcija bi računala zbir celih brojeva od 0 do 5, što je 15. Oba programa će se pokrenuti sa prosleđenim N kao argumentom u komandnoj liniji. Dakle, ako korisnik unese broj 8, biće dobijen zbir celih vrednosti od 0 do 8. Pre nego što nastavimo sa našim primerima stvaranja niti, uvodimo dve tipične strategije za stvaranje višestrukih niti:

1. asinhrone niti i
2. sinhrone niti.

Sa asinhronim nitima, nakon što roditelj stvori nit, roditelj nastavlja svoje izvršavanje, tako da se roditelj i dete izvršavaju istovremeno. Svaka nit radi nezavisno od svih ostalih niti, a roditeljska nit ne mora znati kada se njegovo dete nit završava. Pošto su niti nezavisne, obično je u takvim slučajevima malo ili nimalo deljenja podataka između samih niti. Asinhrone niti su strategija koja se, na primer, koristi kod servera sa više niti izvršavanja ilustrovanom na slici 2.

Sinhrone niti se, sa druge strane, koriste kada roditeljska nit stvori jedno ili više dece niti, nakon čega mora čekati da sva deca niti završe izvršavanje pre nego što nastavi sa izvršavanjem. Ovo se naziva *fork-join* strategija. U ovom scenariju, niti koje je stvorio roditelj (*fork*) rade konkurentno, ali roditelj nit ne može nastaviti dok ovaj posao ne bude završen. Nakon što svaka nit završi sa radom, završava se i pridružuje se roditelju (*join*). Tek nakon što se sva deca niti pridruže, roditelj može nastaviti sa svojim izvršavanjem. Tipično, sinhrone niti uključuju značajno deljenje podataka između samih niti. Na primer, roditeljska nit, da bi uopšte nastavila sa radom, mora koristiti povratne vrednosti koje su izračunale i vratile deca niti. Oba primera koja ćemo pokazati koriste sinhrone niti.

4.1 Pthreads

Pthreads biblioteka se bazira na POSIX standardu (IEEE 1003.1c) definišući API za kreiranje i sinhronizaciju niti. To je specifikacija ponašanja niti i

ne odnosi se na samu implementaciju. Dizajneri operativnog sistema mogu implementirati datu specifikaciju na koji god način da požele to da urade. Brojni sistemi implementiraju Pthreads specifikaciju, većina od njih su sistemi bazirani na Unix operativnom sistemu (Linux, Mac OS X i Solaris). Iako Windows ne podržava Pthreads podrazumevano, postoje implementacije i za Windows (*third-party implementations*).

C program prikazan ispod demonstrira osnove Pthreads API-ja za konstruisanje programa sa više niti koji računa sumu ne-negativnih celih brojeva u zasebnim nitima. U Pthreads programu, zasebne niti izvršavaju datu funkciju. U programu ispod, to je funkcija *runner*(). Nakon inicijalizacije *main*() kreira drugu nit (*runner*), pri čemu obe niti koriste globalnu promenljivu *sum*.

Svi programi koji koriste Pthreads moraju uključiti *pthread.h* datoteku zaglavlja (*header file*). Iskaz

```
pthread_t tid;
```

deklariše identifikator niti koja se kreira. Svaka nit ima skup atributa, uključujući veličinu odgovarajućeg steka i podatke vezane za respoređivanje niti. Deklaracija

```
pthread_attr_t attr;
```

predstavlja atribute za datu nit.

Atributi se postavljaju pozivom funkcije *pthread_ attr_ init*. Ako se atributi ne podese eksplicitno, koriste se podrazumevani atribute. Nova nit se kreira pozivom funkcije

```
pthread_create()
```

Osim prosleđivanja adrese identifikatora niti (pošto će ona biti postavljena od strane funkcije *pthread_ create*), i postavljenih atributa, prilikom kreiranja niti prosleđujemo i naziv funkcije kroz koju nit treba da se izvršava, u ovom slučaju *runner*. Na kraju, prosleđuje se i parametar koji je prosleđen iz komandne linije prilikom poziva programa, *argv*[1].

Na ovom mestu, program se sastoji iz dve niti: inicijalna (roditeljska) nit koja izvršava *main* funkciju, i (dete) nit koja će vršiti sumiranje brojeva kroz *runner* funkciju. Ovaj program koristi strategiju sinhronih niti (*fork-join*) koju smo opisali iznad: nakon što se kreira dete nit, roditeljska nit će čekati da se dete nit terminira pozivom *pthread_ exit* funkcije. Rezultat sumiranja (vrednost deljene promenljive *sum*) je prikazan korisniku od strane roditeljske niti. U realnim aplikacijama, roditelj nit bi nastavio svoje proračune koristeći dobijeni rezultat od dete niti - nije neophodno da roditelj samo čeka kako bi prikazao rezultat korisniku.

```
#include <pthread.h>
#include <stdio.h>
int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */
int main(int argc, char *argv[])
{
        pthread_t tid; /* the thread identifier */
        pthread_attr_t attr; /* set of thread attributes */
        if (argc != 2)
        {
                fprintf(stderr,"usage: %s <integer value> \n", argv[0]);
                return -1;
        }
        if (atoi(argv[1]) < 0)
        {
                fprintf(stderr,"%d must be >= 0 \n", atoi(argv[1]));
                return -1;
        }

        /* get the default attributes */
        pthread_attr_init(&attr);

        /* create the thread */
        pthread_create(&tid, &attr, runner, argv[1]);
        /* wait for the thread to exit */
        pthread_join(tid, NULL);

        printf("sum = %d \ n",sum);
}
/* The thread will begin control in this function */
void *runner(void *param)
{
        int i, upper = atoi(param);
        sum = 0;

        for (i = 1; i <= upper; i++)
                sum += i;

        pthread_exit(0);
}
```

Dati primer programa kreira samo jednu nit. Sa razvojem višejezgarnih sistema, razvoj softvera koji koristi višestruke niti izvršavanja postaje sve češći. Jednostavan metod za čekanje višestrukih niti, takođe korišćenjem *pthread_join* funkcije prikazan je na primeru ispod, gde se čeka na terminaciju 10 dete niti.

```
#define NUM THREADS 10
/* an array of threads to be joined upon */
pthread_t workers[NUM THREADS];

for (int i = 0; i < NUM THREADS; i++)
        pthread_join(workers[i], NULL);
```

Sledeći primer programa koristi dve niti kako bi izračunao parcijalne sume u obe niti: u prvoj 1 do N/2, a u drugoj N/2+1 do N.

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>

//struktura koju koristimo da prenesemo niti višestruke ulazne parametre
struct data {
        int start;
        int stop;
};
```

```
static void* sumfnc(void* arg)
{
        //alocira se na heap-u povratna vrednost funkcije (može biti i struktura
        //ukoliko funkcija treba da vrati više vrednosti)
        int *sum = (int *) malloc(sizeof(int));

        //pročitaj ulazne parametre iz primljene strukture struct data
        struct data inp = *(struct data *) arg;
        //saberi brojeve počevši od inp.start do inp.stop
        for (int i = inp.start; i < inp.stop; i++)
        {
                *sum += i;
        }
        //vrati vrednost koja će biti prihvaćena od strane thread_join
        //funkcije pozvane iz osnovne (roditelj) niti
        pthread_exit((void*)sum);
}

//glavni program očekuje dodatni parametar N koji se koristi prilikom sabiranja
int main (int argc, char *argv[])
{
        int ret;
        //povratne vrednosti iz dve niti
        int *sum1;
        int *sum2;
        //identifikatori niti
        pthread_t thread1;
        pthread_t thread2;

        //proveri da li je prosleđen dodatni parametar prilikom poziva programa
        if (argc<2)
        {
                printf("Usage:%s N\n", argv[0]);
                exit(1);
        }

        //konvertuj string u celobrojni podatak koji će se koristiti
        int N = atoi(argv[1]);

        //napravi strukturu koja će biti prosleđena prvoj niti
        struct data str1;
        str1.start = 1;
        str1.stop= N/2 + 1;

        //napravi strukturu koja će biti prosleđena drugoj niti
        struct data str2;
        str2.start = N/2 + 1;
        str2.stop = N+1;

        //kreiraj niti
        ret = pthread_create(&thread1, NULL, sumfnc, &str1);
        if(ret!=0)
        {
                printf("Failed to create thread1 ret=%d\n",ret);
                exit(1);
        }

        ret=pthread_create(&thread2, NULL, sumfnc, &str2);
        if(ret!=0)
        {
                printf("Failed to create thread2 ret=%d\n",ret);
                exit(1);
        }

        //čekaj da se niti završe i preuzmi povratne vrednosti
        pthread_join(thread1,(void **)&sum1);
        pthread_join(thread2,(void **)&sum2);
```

```
        //prikaži rezultat
        printf("Rezultat sabiranja je %d\n",*sum1+*sum2);

        //oslobodi memoriju rezervisanu u svakoj od niti
        free(sum1);
        free(sum2);
        return 0;
}
```

Za povratne vrednosti računanja u samoj niti, mogao se rezervisati prostor i u samoj strukturi struct data. U tom slučaju, nije potrebno rezervisati prostor za povratnu vrednost u samoj niti:

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>

//struktura koju koristimo da prenesemo niti višestruke ulazne parametre
//i da dobijemo rezultat izračunavanja
struct data {
        int start;
        int stop;
        int sum;
};

static void* sumfnc(void* arg)
{
        //pročitaj ulazne parametre iz primljene strukture struct data
        struct data *inp = (struct data *) arg;

        //inicijalizuj sumu na 0 (ne mora biti podrazumevano kao polje strukture)
        inp->sum = 0;

        //saberi brojeve počevši od inp.start do inp.stop
        for (int i = inp->start; i < inp->stop; i++)
        {
                inp->sum += i;
        }
        //ne vraća se ništa, rezultat je deo strukture struct data
        pthread_exit(NULL);
}

//glavni program očekuje dodatni parametar N koji se koristi prilikom sabiranja
int main (int argc, char *argv[])
{
        int ret;

        //identifikatori niti
        pthread_t thread1;
        pthread_t thread2;

        //proveri da li je prosleđen dodatni parametar prilikom poziva programa
        if (argc<2)
        {
                printf("Usage:%s N\n", argv[0]);
                exit(1);
        }

        //konvertuj string u celobrojni podatak koji će se koristiti
        int N = atoi(argv[1]);

        //napravi strukturu koja će biti prosleđena prvoj niti
        struct data str1;
        str1.start = 1;
        str1.stop= N/2 + 1;

        //napravi strukturu koja će biti prosleđena drugoj niti
```

```
        struct data str2;
        str2.start = N/2 + 1;
        str2.stop = N+1;

        //kreiraj niti
        ret = pthread_create(&thread1, NULL, sumfnc, &str1);
        if(ret!=0)
        {
                printf("Failed to create thread1 ret=%d\n",ret);
                exit(1);
        }

        ret = pthread_create(&thread2, NULL, sumfnc, &str2);
        if(ret!=0)
        {
                printf("Failed to create thread2 ret=%d\n",ret);
                exit(1);
        }

        //čekaj da se niti završe bez preuzimanja povratnih vrednosti
        pthread_join(thread1, NULL);
        pthread_join(thread2, NULL);

        //prikaži rezultat
        printf("Rezultat sabiranja je %d\n", str1.sum + str2.sum);

        return 0;
}
```

4.2 Java niti

Niti predstavljaju osnovni model izvršavanja programa u Java programskom jeziku, a Java jezik i njegov API pružaju raznovrstan skup funkcija za kreiranje i upravljanje nitima. Svi Java programi sadrže najmanje jednu nit izvršavanja - čak i jednostavan Java program koji se sastoji od samo *main*() metode izvršava se u okviru jedne niti u JVM-u (Java Virtualnoj Mašini). Java niti su dostupne na bilo kojem sistemu koji podržava JVM, a koji uključuje Windows, Linux i Mac OS. *Java Thread API* je dostupan i za Android aplikacije.

Postoje dve tehnike za kreiranje niti u Java programu. Prvi pristup je kreiranje klase koja nasleđuje od klase *Thread* i reimplementira njenu metodu *run*(). Alternativna i češće korištena tehnika je definisanje klase koja implementira *Runnable* interfejs. *Runnable* interfejs je definisan na sledeći način:

```
public interface Runnable
{
        public abstract void run();
}
```

Kada klasa implementira *Runnable*, mora definisati metodu *run*(). Kod koji implementira metodu *run*() je ono što se pokreće kao zasebna nit. Na listingu ispod prikazana je Java verzija programa sa višestrukim nitima koj određuje sumu ne-negativnih celih brojeva od gore.

```
class Sum
{
  private int sum;
  public int getSum()
  {
    return sum;
  }
```

```
  public void setSum(int sum)
  {
    this.sum = sum;
  }
}

class Summation implements Runnable
{
  private int upper;
  private Sum sumValue;
  public Summation(int upper, Sum sumValue)
  {
    this.upper = upper;
    this.sumValue = sumValue;
  }
  public void run()
  {
    int sum = 0;
    for (int i = 0; i <= upper; i++)
      sum += i;

    sumValue.setSum(sum);
  }
}

public class Driver
{
  public static void main(String[] args)
  {
    if (args.length > 0)
    {
      if (Integer.parseInt(args[0]) < 0)
        System.err.println(args[0] + " must be >= 0.");
      else
      {
        Sum sumObject = new Sum();
        int upper = Integer.parseInt(args[0]);
        Thread thrd = new Thread(new Summation(upper, sumObject));
        thrd.start();
        try
        {
          thrd.join();
          System.out.println ("The sum from 0 to "+ upper + " is "
                              + sumObject.getSum());
        }
        catch (InterruptedException ie)
        {
        }
      }
    }
    else System.err.println("Usage: Summation <integer value>");
  }
}
```

Klasa *Sumation* implementira interfejs *Runnable*. Stvaranje niti vrši se kreiranjem instance objekta klase *Thread* i prosleđivanjem konstruktoru objekta *Runnable*.

Stvaranje *Thread* objekta ne stvara samo po sebi novu nit. Umesto toga, metoda *start*() pozvana na tom objektu kreira novu nit. Pozivanje metode *start*() čini dve stvari:

1. raspoređuje memoriju i inicijalizuje novu nit u JVM.

2. Poziva metodu *run*(), čineći nit adekvatnom za pokretanje unutar JVM-a.

> Obratiti pažnju da se metoda *run*() nikada ne poziva direktno. Umesto toga, zovemo metodu *start*(), a ona u naše ime poziva metodu *run*().

Kada se program za sumiranje pokrene, JVM kreira dve niti. Prva je roditeljska nit, koja započinje izvršavanje u *main*() metodi. Druga nit se stvara kada se pozove metoda *start*() na *Thread* objektu. Ova (dete) nit započinje svoje izvršavanje metodom *run*() klase *Sumation*. Nakon što izračuna vrednost zbira, ova nit se terminira u momentu kada izlazi iz metode *run*().

Deljenje podataka između niti se lako rešava u Pthreads biblioteci, pošto se deljeni podaci jednostavno deklarišu kao globalne promenljive. Kao strogo objektno orijentisani jezik, Java ne podržava globalne podatke. Ako dve ili više niti trebaju deliti podatke u Java programu, deljenje se vrši prenošenjem referenci na zajednički objekat odgovarajućim nitima. U Java programu prikazanom na listingu iznad, glavna nit i nit za sumiranje dele instancu objekta klase *Sum*. Na ovaj zajednički objekat se referencira odgovarajućim *getSum*() i *setSum*() metodama.

Roditeljske niti u biblioteci Pthreads koriste *pthread_join*() da bi sačekale da se niti za sabiranje završe pre nego što nastave sa svojim izvršavanjem. Metoda *join*() u Javi omogućava sličnu funkcionalnost. Pri tome, treba imati na umu da *join*() može izbaciti *InterruptedException* izuzetak, koji mi u ovom slučaju zanemarujemo. U slučaju kada roditelj mora sačekati da se završi nekoliko dece niti, metoda *join*() može se pozivati unutar petlje, slično kao što je prikazano na primeru sa Pthreads pozivom *pthread_join*.

Varijanta java programa koji u dve niti sabira brojeve od 1 do N/2 (u prvoj), i od N/2+1 do N (u drugoj) je:

```
class Sum
{
  private int sum;
  public int getSum()
  {
    return sum;
  }
  public void setSum(int sum)
  {
    this.sum = sum;
  }
}

class Summation implements Runnable
{
  private int upper;
  private int lower;
  private Sum sumValue;
  public Summation(int lower, int upper, Sum sumValue)
  {
    this.lower = lower;
    this.upper = upper;
    this.sumValue = sumValue;
  }

  public void run()
  {
    int sum = 0;
    for (int i = lower; i <= upper; i++){
      sum += i;
    }
```

```
    synchronized(sumValue){
      sumValue.setSum(sumValue.getSum() + sum);
    }
  }
}

public class Driver
{
  public static void main(String[] args)
  {
    if (args.length > 0)
    {
      if (Integer.parseInt(args[0]) < 0)
        System.err.println(args[0] + " must be >= 0.");
      else
      {
        Sum sumObject = new Sum();
        int upper = Integer.parseInt(args[0]);
        int lower = upper/2 - 1;
        Thread thrd1 = new Thread(new Summation(0, lower, sumObject));
        Thread thrd2 = new Thread(new Summation(lower + 1, upper, sumObject));
        thrd1.start();
        thrd2.start();
        try
        {
          thrd1.join();
          thrd2.join();
          System.out.println ("The sum from 0 to "+ upper + " is "
                              + sumObject.getSum());
        }
        catch (InterruptedException ie)
        {
        }
      }
    }
    else System.err.println("Usage: Summation <integer value>");
  }
}
```

Obratite pažnju na blok

```
synchronized(sumValue){
  sumValue.setSum(sumValue.getSum() + sum);
}
```

koji obezbeđuje da se svim linijama unutar bloka (u ovom slučaju samo jedna) pristupa sukcesivno, najpre od strane prve niti, a zatim od strane druge niti. Drugim rečima, ovaj blok ne dozvoljava istovremeni pristup deljenim promenljivama od strane višestrukih niti izvršavanja. U suprotnom, dobijali bi se neočekivani rezultati. O ovome ćemo pričati detaljnije na narednim predavanjima.

4.3 Problemi u radu sa višestrukim nitima

4.3.1 *Fork*() i *Exec*() sistemski pozivi

U jednom od prethodnih poglavlja smo opisali kako se sistemski poziv *fork*() koristi za kreiranje nezavisnog, dupliranog procesa. Semantika sistemskih poziva *fork*() i *exec*() se menja u programu koji koristi više niti. Postavlja se pitanje: ako jedna nit u programu poziva *fork*(), da li u novonastalom procesu trebaju biti klonirane sve niti, ili novi proces treba da sadrži jednu nit (*single-threaded*)?

Neki Unix sistemi imaju dve verzije *fork*(): onu koja klonira sve niti i drugu koja klonira samo nit koja je pozvala sistemski poziv *fork*().

Sistemski poziv *exec*() obično radi kao što je opisano u jednom od ranijih poglavlja. To jest, ako nit poziva sistemski poziv *exec*(), program naveden u parametru *exec*() zameniće celi proces - uključujući i sve niti.

Koju od dve verzije *fork*() sistemskog poziva treba koristiti zavisi od aplikacije. Ako se *exec*() poziva odmah nakon grananja, tada je dupliranje svih niti nepotrebno, jer će program naveden u parametru poziva *exec*() svakako zameniti ceo proces. U ovom slučaju je pogodno kloniranje samo niti koja je pozvala sistemski poziv. Ako, međutim, kreirani proces ne poziva *exec*() nakon što je nastao sistemskim pozivom *fork*(), tada treba klonirati sve niti.

4.3.2 Rukovanje signalima

Signali se koriste u Unix sistemima za obaveštavanje procesa o određenom događaju koji se desio. Signal se može primati *sinhrono* ili *asinhrono*, u zavisnosti od izvora signala kao i događaja koji se signalizira. Svi signali, bili *sinhroni* ili *asinhroni*, slede isti obrazac:

1. Signal nastaje kao posledica pojave određenog događaja;
2. Signal se dostavlja procesu,
3. Nakon dostavljanja signala procesu, određena akcija se mora preduzeti.

Primeri *sinhronog* signala su ilegalni pristup memoriji, kao i deljenje broja sa nulom. Ako pokrenuti program izvede bilo koju od ovih radnji, generiše se signal. Sinhroni signali se dostavljaju procesu koji je izveo operaciju koja je izazvala signal (i to je razlog zbog kojeg se nazivaju *sinhronim*).

Kada signal generiše događaj izvan procesa koji se pokreće, kaže se da taj proces prima signal *asinhrono*. Primer takvih signala je terminiranje procesa pritiskom na tastere (kao što je <Ctrl>+<C>). Obično se *asinhroni* signal šalje drugom procesu.

Signalom se može upravljati na jedan od dva moguća načina:

1. Podrazumevana rutina za obradu signala;
2. Korisnički definisana rutina za obradu signala

Za svaki signal postoji podrazumevana rutina za obradu signala koju kernel pokreće u cilju obrade tog signala. Ovu predefinisanu rutinu može zameniti korisnički definisana rutina za obradu signala koja će biti pozvana kada se desi signal.

Samim signalima se rukuje na različite načine. Neki signali (poput promene veličine prozora) se jednostavno zanemaruju, dok drugi (kao što je ilegalan pristup memoriji) dovode do prekida programa.

Rukovanje signalima u slučaju programa s jednom niti izvršavanja je jednostavno: signali su uvek isporučeni toj niti. Međutim, isporuka signala je složenija u programima sa više niti izvršavanja. Gde bi u tom slučaju trebao da bude dostavljen signal? Generalno, postoje sledeće mogućnosti:

1. Dostaviti signal niti na koju se signal odnosi;
2. Dostaviti signal svakoj niti u okviru procesa;
3. Dostaviti signal određenim nitima u okviru procesa;
4. Postaviti određenu nit da prima sve signale za taj proces.

Na koji način se isporučuje signal zavisi od vrste generisanog signala. Na primer, sinhroni signali moraju da se isporučuju nitima koje su uzrokovale signal, a ne drugim nitima u okviru procesa. Međutim, situacija sa asinhronim signalima nije tako jednostavna. Neki asinhroni signali, poput signala koji terminira proces, trebali bi da se pošalju svim nitima u okviru procesa.

U slučaju Unixa, standardna funkcija za isporuku signala je:

```
kill(pid_t pid, int signal)
```

Ova funkcija određuje proces (*pid*) kojem se treba isporučiti određeni signal (*signal*).

Sa druge strane, većina verzija Unix-a omogućava svakoj pojedinačnoj niti da odredi koje će signale prihvatiti, a koje će blokirati. Zbog toga se u nekim slučajevima asinhroni signal može isporučiti samo onim nitima koje ga ne blokiraju. Međutim, obzirom na to da se obrada signala mora izvršiti samo jednom, signal se obično isporučuje samo prvoj pronađenoj niti koji ga *ne blokira*. POSIX *Pthreads* sadrži funkciju, koja omogućava isporuku signala određenoj niti (određenoj sa *tid*):

```
pthread_kill (pthread_t tid, int signal)
```

4.3.3 Otkazivanje niti (*Thread Cancellation*)

Otkazivanje niti podrazumeva zaustavljanje i terminiranje niti pre nego što se ona završi. Na primer, ako više niti istovremeno pretražuje bazu podataka i jedna nit vrati rezultat, preostale niti mogu biti otkazane. Može se dogoditi slična situacija kada korisnik pritisne dugme na web pretraživaču koje zaustavlja dalje učitavanje web stranice. Web stranica se, tipično, učitava pomoću nekoliko niti - svaka slika ili deo stranice se učitava u zasebnoj niti. Kada korisnik pritisne dugme *stop* u pretraživaču, sve se niti koje učitavaju stranicu otkazuju. Nit koju treba otkazati često se naziva *ciljna nit* (*target thread*). Otkazivanje ciljne niti može se pojaviti u dva različita scenarija:

1. Asinhrono otkazivanje: Jedna nit trenutno prekida ciljnu nit.

2. Odloženo otkazivanje: Ciljna nit povremeno proverava da li treba da se prekine, pružajući sebi mogućnost da se prekine, kad za to dođe vreme, na adekvatan način.

Poteškoće u vezi sa otkazivanjem nastaju u situacijama kada su resursi dodeljeni ciljnoj niti ili kada se nit otkaže dok se nalazi usred ažuriranja podataka koje deli sa drugim nitima. Ovo postaje posebno problematično kod asinhronog otkazivanja. Operativni sistem će uglavnom preuzeti sistemske resurse od otkazane niti, ali neće preuzeti sve resurse. Usled toga, otkazivanje niti asinhronim

putem može da ne oslobodi neophodan sistemski resurs. Za razliku od ovoga, kod odloženog otkazivanja, jedna nit označava da se ciljna nit treba otkazati, ali otkazivanje se dešava tek nakon što ciljna nit proveri zastavicu kako bi utvrdila da li treba ili ne treba da se otkazuje. Nit može izvršiti ovu proveru u tački u kojoj se može bezbedno otkazati.

U *Pthreads* biblioteci se otkazivanje niti vrši pomoću funkcije *pthread_cancel*(). Identifikator ciljne niti se prosleđuje kao parametar funkciji. Sledeći kod ilustruje kreiranje i potom otkazivanje niti:

```
pthread_t tid;
/* kreiraj nit */
pthread_create(&tid, 0, worker, NULL);
. . .
/* otkaži nit */
pthread_cancel(tid);
```

Pozivanje *pthread_cancel*() predstavlja samo zahtev za otkazivanjem ciljne niti; stvarno otkazivanje zavisi od toga na koji način je ciljna nit podešena za pristizanje zahteva za otkazivanjem. *Pthreads* podržava tri načina otkazivanja. Svaki mod je definisan preko *stanja* i *tipa*, kao što je prikazano u tabeli ispod.

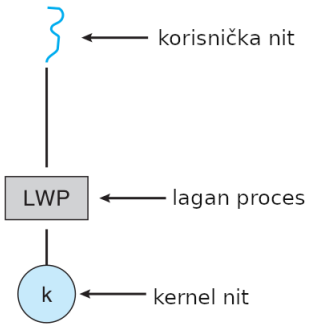
Mod	Stanje	Tip
Isključeno	Onemogućeno	-
Odloženo	Omogućeno	Odloženo
Asinhrono	Omogućeno	Asinhrono

Nit može postaviti svoje stanje i tip otkazivanja pomoću API-ja predviđenog za to. Kao što tabela ilustruje, *Pthreads* omogućava nitima da onemoguće ili omoguće otkazivanje. Očigledno je da se nit ne može otkazati ako je otkazivanje onemogućeno. Međutim, zahtevi za otkazivanje ostaju aktivni i u tom slučaju, tako da nit kasnije može omogućiti otkazivanje i odgovoriti na zahtev. Podrazumevani tip otkaza je odloženi otkaz, kod koga se otkazivanje dešava samo kada nit dostigne tačku otkazivanja.

Jedna tehnika za postavljanje tačke otkazivanja je pozivanje funkcije *pthread_testcancel*(). Ako se ustanovi da zahtev za otkazivanje čeka, poziva se funkcija za oslobađanje svih resursa koje je nit mogla posedovati pre nego što se prekine. Sledeći kod ilustruje kako nit može da odgovori na zahtev za otkazivanje koristeći odloženo otkazivanje:

```
while(1)
{
        /* do some work for awhile */
        /* . . . */
        /* check if there is a cancellation request */
        pthread_testcancel();
}
```

Zbog prethodno opisanih problema, u *Pthreads* dokumentaciji se ne preporučuje asinhrono otkazivanje. Ukoliko je onemogućeno otkazivanje ili zahtev za otkazivanjem nije pristigao, poziv funkcije *pthread_testcancel*() nema nikakvog efekta. Takođe, ova funkcija nema povratnu vrednost: ako je nit otkazana kao rezultat ove funkcije, funkcija se ne „vraća“ odnosno eventualni deo koda iza nje se neće nikada izvršiti.



Slika 9: Lagan proces

4.3.4 Lokalno skladište niti

Niti koje pripadaju procesu dele podatke i memorijski prostor sa procesom u okviru kojeg su kreirane. Zapravo, ova mogućnost razmene podataka pruža jednu od prednosti programiranja sa višestrukim nitima. Međutim, u nekim situacijama, svaka pojedinačna nit će, eventualno, imati potrebu za svojom lokalnom kopijom određenih podataka. Nazvaćemo takve podatke *lokalno skladište niti* (*Thread Local Storage - TLS*).

Na prvi pogled, postoji izvesna sličnost između TLS i lokalnih promenljivih. Ipak, lokalne promenljive su vidljive samo tokom jednog poziva funkcije koja se izvršava u okviru niti, dok su TLS podaci perzistentni i vidljivi kroz više sukcesivnih poziva funkcija. Na neki način, TLS je sličan statičkim podacima (promenljivama deklarisanim kao *static*), a razlika je u tome što su TLS podaci jedinstveni za svaku nit. Većina biblioteka za podršku radu sa nitima, uključujući Windows, Pthreads i Java Thread biblioteku, pruža neku vrstu podrške za lokalno skladište niti.

4.3.5 Aktivacija planera

Konačno pitanje koje treba razmotriti kod programa sa višestrukim nitima odnosi se na komunikaciju između kernela i biblioteke niti, što može biti potrebno kod modela „više na više“ kao i kod dvostepenog modela koje smo razmatrali ranije. Takva koordinacija omogućava da se broj kernel niti dinamički podešava kako bi se osigurale najbolje performanse.

Mnogi sistemi koji implementiraju „više na više“ ili dvostepeni model koriste dodatnu strukturu podataka koja se nalazi između korisničkih i kernel niti. Ova struktura podataka, tipično poznata kao *lagan proces* (LightWeight Process, LWP), prikazana je na slici 9.

Sa strane korisničke biblioteke za rad sa nitima LWP predstavlja virtualni procesor na kojem aplikacija može rasporediti korisničku nit za izvršavanje. Svaki LWP je vezan za kernel nit, a kernel niti su te koje operativni sistem izvršava na fizičkim procesorima. Ako se kernel nit blokira (npr usled čekanja

dok se U/I operacija završi), LWP se takođe blokira. Na vrhu cele ove hijerarhije, korisnička nit dodeljena tom LWP se takođe blokira.

Aplikacija može zahtevati proizvoljan broj LWP-a kako bi bila efikasna. Aplikaciju koja je CPU-kontrolisana (CPU-bound) i koja se izvršava na jednom procesoru, na kome se u datom trenutku može izvršavati samo jedna nit, zahteva jedan LWP. Međutim, aplikacija koja je U/I-kontrolisana može zahtevati više LWP-ova da bi se efikasnije izvršavala. Obično je potreban poseban LWP za svaki konkurentni blokirajući sistemski poziv. Pretpostavimo, na primer, da se pet različitih zahteva za čitanje datoteka pojavljuju istovremeno. Tada je potrebno pet LWP-ova, jer bi svi mogli čekati na završetak U/I operacije u okviru kernela. Ako dati proces ima samo četiri LWP-a, peti zahtev mora sačekati da se jedan od LWP-ova vrati od strane kernela.

Jedna šema za komunikaciju između korisničke biblioteke za rad sa nitima i kernela poznata je kao *aktivacija planera* (*scheduler activation*). Ona funkcioniše na sledeći način: kernel obezbeđuje aplikaciji skup virtualnih procesora (LWP-ova), a aplikacija može rasporediti korisničke niti na dostupnom virtualnom procesoru. Dodatno, kernel mora obavestiti aplikaciju o određenim događajima. Ovaj postupak je poznat i kao *poziv na gore* (eng. *upcall*). Pozivi na gore se obrađuju od strane biblioteke niti kroz specijalne *rutine za obradu poziva na gore*, pri čemu se ove rutine moraju izvršavati na virtualnom procesoru. Jedan od događaja koji pokreće poziv na gore dešava se kada nit u sklopu aplikacije treba da bude blokirana. U ovom scenariju, kernel izvršava poziv na gore kojim šalje informaciju aplikaciji da će se nit blokirati i prosleđuje joj odgovarajući identifikator konkretne niti. Kernel potom aplikaciji dodeljuje novi virtualni procesor. Aplikacija pokreće rutinu za poziv na gore na novo-dodeljenom virtualnom procesoru, što rezultuje čuvanjem stanja niti koja će se blokirati i oslobađanjem virtualnog procesora na kojem se ona izvršavala. Rutina za obradu poziva na gore tada raspoređuje drugu nit koja ispunjava uslove za pokretanje na novom virtualnom procesoru. Kada se desi događaj na koji je blokirana nit čekala, kernel vrši ponovni poziv na gore kako bi obavestio biblioteku za rad sa nitima da je prethodno blokirana nit sada spremna za izvršavanje. Rutina za obradu ovog poziva na gore takođe zahteva virtualni procesor, a kernel u tu svrhu može dodeliti ili novi virtualni procesor ili može prekinuti neku od korisničkih niti koja se izvršava pa pokrenuti rutinu na tom virtualnom procesoru. Nakon što je odblokirana nit označena kao spremna za izvršavanje, aplikacija raspoređuje spremnu nit za izvršavanje na dostupnom virtualnom procesoru.