Sveučilište J. J. Strossmayera u Osijeku

Fakultet elektrotehnike, računarstva i informacijskih tehnologija

Kneza Trpimira 2b

HR-31000 Osijek

www.ferit.unios.hr

Priprema za laboratorijske vježbe:

Uvod u laboratorijske vježbe iz operacijskih sustava

Sadržaj

1. UVOD

Laboratorijske vježbe su o konceptima, strukturama i mehanizmima operacijskih sustava. Usprkos raznolikosti i brzini promjene modernih operacijskih sustava, određeni fundamentalni koncepti se primjenjuju konzistentno i danas. Namjera laboratorijskih vježbi je detaljno prikazati osnovne principe dizajna operacijskih sustava i na primjeru poznatijih sustava prikazati njihovu upotrebu.

Koristit ćemo operacijske sustave Linux i Windows:

- Windows – višezadaćni operacijski sustav za osobna računala, radne stanice i poslužitelje. Inkorporira mnoge najnovije značajke iz razvoja operacijskih sustava. Jedan od prvih sustava koji se značajno temelji na principima objektno-orijentiranog dizajna.
- Linux – *open-source* verzija UNIX-a; višezadaćni operacijski sustav implementiran na širokom rasponu računala, od mikroračunala do superračunala.

1.1. Sadržaj laboratorijskih vježbi

9 laboratorijskih vježbi:

- 3 laboratorijske vježbe za rad s ljuskom operacijskog sustava Linux
- 4 laboratorijskih vježbi za rad s jezgrom operacijskog sustava Linux
- 1 laboratorijska vježba za koncepte operacijskog sustava Windows
- 1 laboratorijska vježba za rad s operacijskim sustavom za ugradbene računalne sustave

1.2. Način provedbe laboratorijskih vježbi

Laboratorijske vježbe nose 30 bodova od ukupnih 100 bodova kolegija:

- 18 bodova za izvještaje
- 12 bodova izlazni kolokvij iz vježbi

Na svakoj laboratorijskoj vježbi studenti rješavaju zadatke za samostalni rad. Zadatke je potrebno riješiti tijekom vježbe.

Prije svake vježbe studenti su **dužni** pročitati predložak za laboratorijsku vježbu. Osobito je nužno pročitati **prvo poglavlje** koje predstavlja sažetak s predavanja te se ono **neće** dodatno pojašnjavati na vježbama!

2. POTREBNA PREDZNANJA

2.1. Programiranje u programskom jeziku C

Znanje stečeno na kolegijima Programiranje 1 i Programiranje 2.

2.2. Poznavanje arhitekture računala

Operacijski sustav iskorištava računalne resurse pomoću jednog ili više procesora kako bi pružio odgovarajući skup usluga korisnicima. Prema tome, važno je usvojiti osnovno znanje o arhitekturi računala prije proučavanja operacijskih sustava.

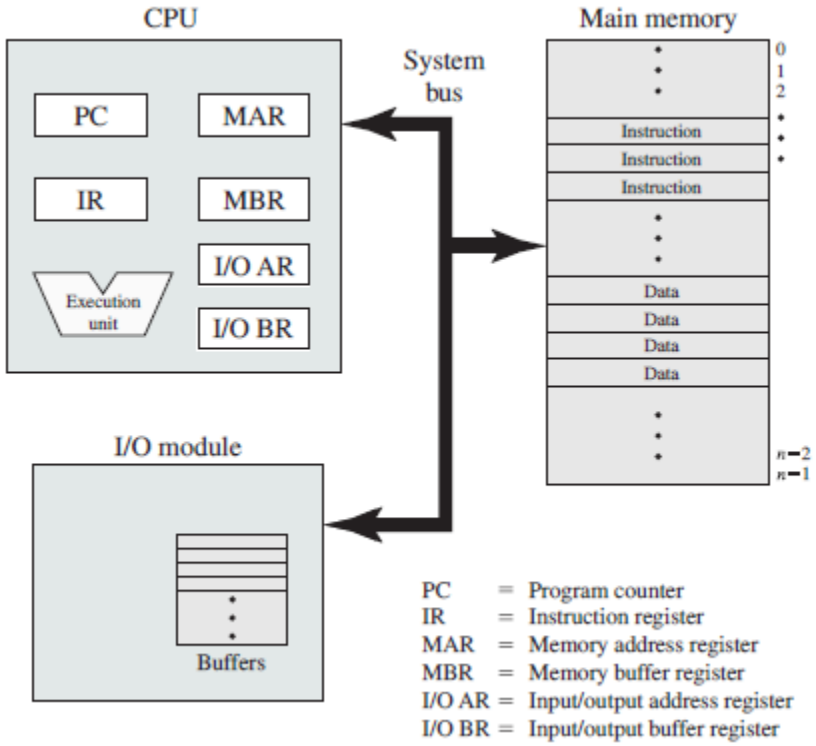
Kao osnovno predznanje o arhitekturi računalnih sustava, podrazumijevaju se sljedeći koncepti:

- Opisivanje osnovnih elemenata računala i njihovih međuovisnosti
- Poznavanje načina rada procesora
- Razumjeti koncepte prekida i zašto ih procesor koristi
- Navesti i opisati osnovne razine hijerarhije memorija računala
- Objasniti osnovne karakteristike višeprocesorskih i višejezgrenih arhitektura
- Razumjeti koncepte lokalnosti i analizirati performanse višerazinske hijerarhije memorije
- Razumjeti strukture podataka kao što su stog, povezani popis i sl.

2.2.1. Osnovni elementi računala

Na najvišoj razini, računalo se sastoji od procesora, memorije i ulazno-izlaznih komponenata. Komponente su međusobno povezane u svrhu ostvarivanja osnovnog cilja računala – izvršavanje programa. Četiri osnovna strukturna elementa računala su:

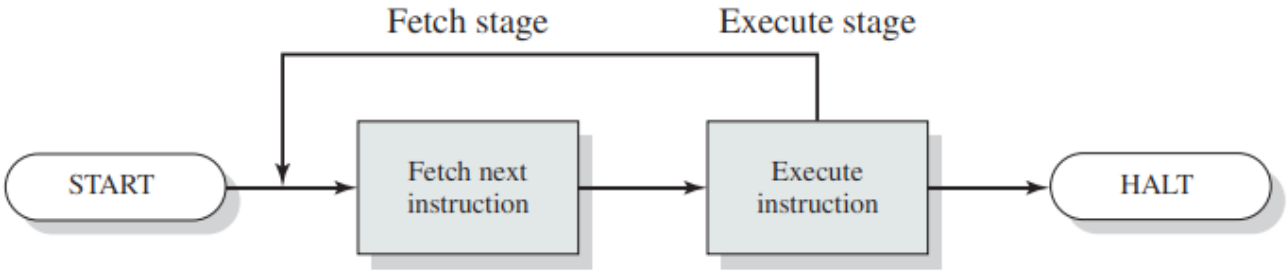
- **Procesor** – kontrolira rad računala i provodi operacije nad podacima. Kada postoji samo jedan procesor, često ga se zove i centralna jedinica (engl. *central processing unit (CPU)*).
- **Glavna memorija** – pohranjuje programe i podatke. Ova memorija je uglavnom promjenjiva, tj. sadržaji se brišu gašenjem računala. S druge strane, sadržaji na diskovnoj memoriji su sačuvani i nakon gašenja. Glavna memorija se često naziva i radna memorija, stvarna memorija (engl. *real memory*) ili primarna memorija.
- **Ulazno-izlazne komponente** (engl. *Input-Output(I/O)*): prenose podatke između računala i vanjskog okruženja. Vanjsko okruženje čine različiti uređaji, među kojima su sekundarni memorijski uređaji (npr. diskovi), komunikacijska oprema i terminali.
- **Sistemska sabirnica**: omogućuje komunikaciju između procesora, glavne memorije i ulazno-izlaznih komponenata.



Slika 1. Sažeti prikaz osnovnih komponenata računala

2.2.2. Instrukcijski ciklus procesora

Osnovna zadaća procesora, a i cijelog računala je izvođenje programa. Program se sastoji od niza instrukcija pohranjenih u memoriji. U najjednostavnijoj formi, upravljanje instrukcijama se sastoji od 2 koraka: dohvaćanje (engl. *fetch*) instrukcija iz memorije te njihovo izvršavanje (engl. *execute*).



Slika 2. Osnovni instrukcijski ciklus procesora

Na početku svakog instrukcijskog ciklusa, procesor dohvaća instrukcije iz memorije. Uobičajeno, registar za programski brojač (PC) sadrži adresu instrukcije koja se treba izvršiti sljedeća. Procesor inkrementira PC nakon svakog dohvaćanja instrukcije, a njih smješta u instrukcijski registar (IR). Procesor zatim interpretira instrukciju i izvodi odgovarajući akciju. U globalu, akcije se mogu svrstati u neku od sljedećih kategorija:

- Procesor – glavna memorija: podaci se prenose između ovih komponenata.
- Procesor – ulazno/izlazne komponente: podaci se prenose prema ili od perifernog uređaja putem I/O modula.
- Procesiranje podataka – procesor izvodi neke aritmetičke ili logičke operacije.

- Kontroliranje – instrukcija može specificirati izmjenu slijeda izvršavanja nekog programa.

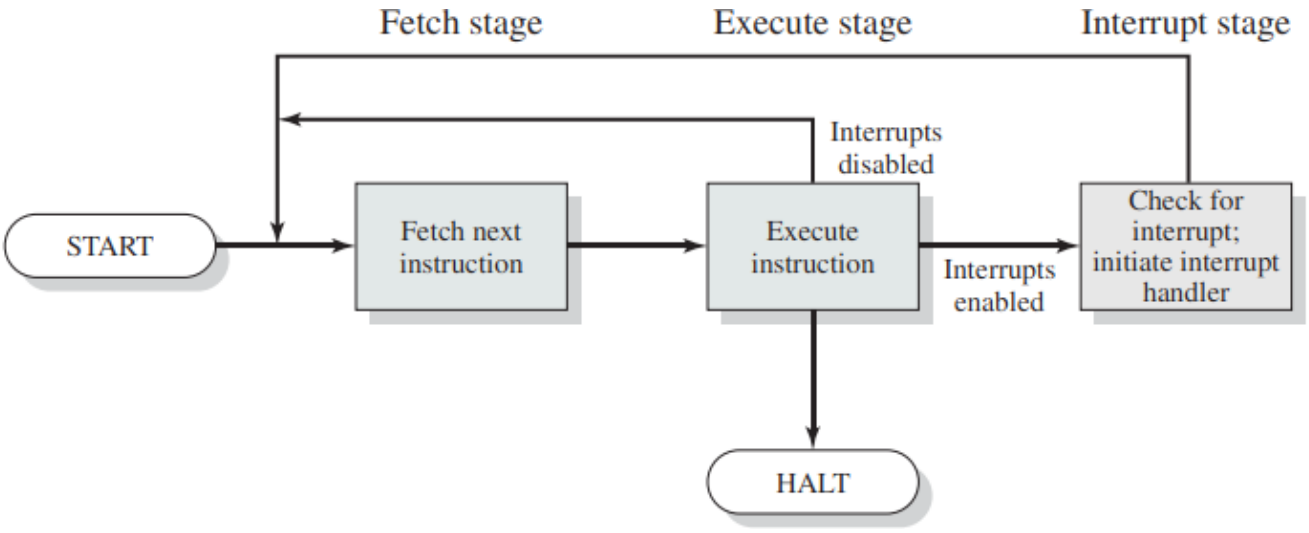
Ovakav rad procesora je unaprijed definiran i dokumentiran u apstraktnom modelu računala koji se naziva **arhitektura skupa instrukcija** (engl. *instruction set architecture (ISA)*). Procesor je zapravo implementacija ISA modela, jer striktno izvodi instrukcije koje su definirane modelom i u redoslijedu koji model nalaže. U globalu, ISA modelom su definirani tipovi podataka, nazivi i adrese registara, hardverski pristup glavnoj memoriji te osnovni principi poput virtualne memorije i načina adresiranja memorije. Može se reći da je ISA modelom definirano ponašanje strojnog koda koje se pokreće na računalu.

2.2.3. Prekidi

Gotovo sva računala pružaju mehanizam putem kojeg ostali moduli (I/O, memorija) mogu prekinuti normalno izvođenje instrukcijskog ciklusa procesora. Najčešće klase prekida su:

- Programski – generira ga neki uvjet koji se javlja kao rezultat izvođenja instrukcije, kao što je aritmetički preljev (engl. *arithmetic overflow*), dijeljenje s 0, pokušaj izvršavanja ilegalne instrukcije, referenciranje izvan dozvoljenog memorijskog prostora i sl.
- Vremenski – generira ga vremenski brojač računala, uglavnom za izvođenje nekog periodičkog posla.
- Ulazno-izlazni – generira ga I/O kontroler kako bi signalizirao završetak neke I/O operacije ili različite pogreške.
- Pogreška u sklopovlju – generira ju hardverski kvar, kao što je nestanak struje ili greška pariteta memorije.

Prekidi su značajke u arhitekturi računala kojima se nastoji povećati iskorištenje procesora (Primjer – rad pisača). Periferni uređaji su puno sporiji od procesora i kad bi procesor slijedno čekao završetak rada svakog pojedinog uređaja, velik dio takta procesora bi ostao neiskorišten. Pomoću prekida, procesor može izvršavati druge aktivnosti dok je neka ulazno-izlazna operacija u tijeku. Slika 3 prikazuje osnovni instrukcijski ciklus procesora proširen s prekidima.



Slika 3. Osnovni instrukcijski ciklus procesora s prekidima

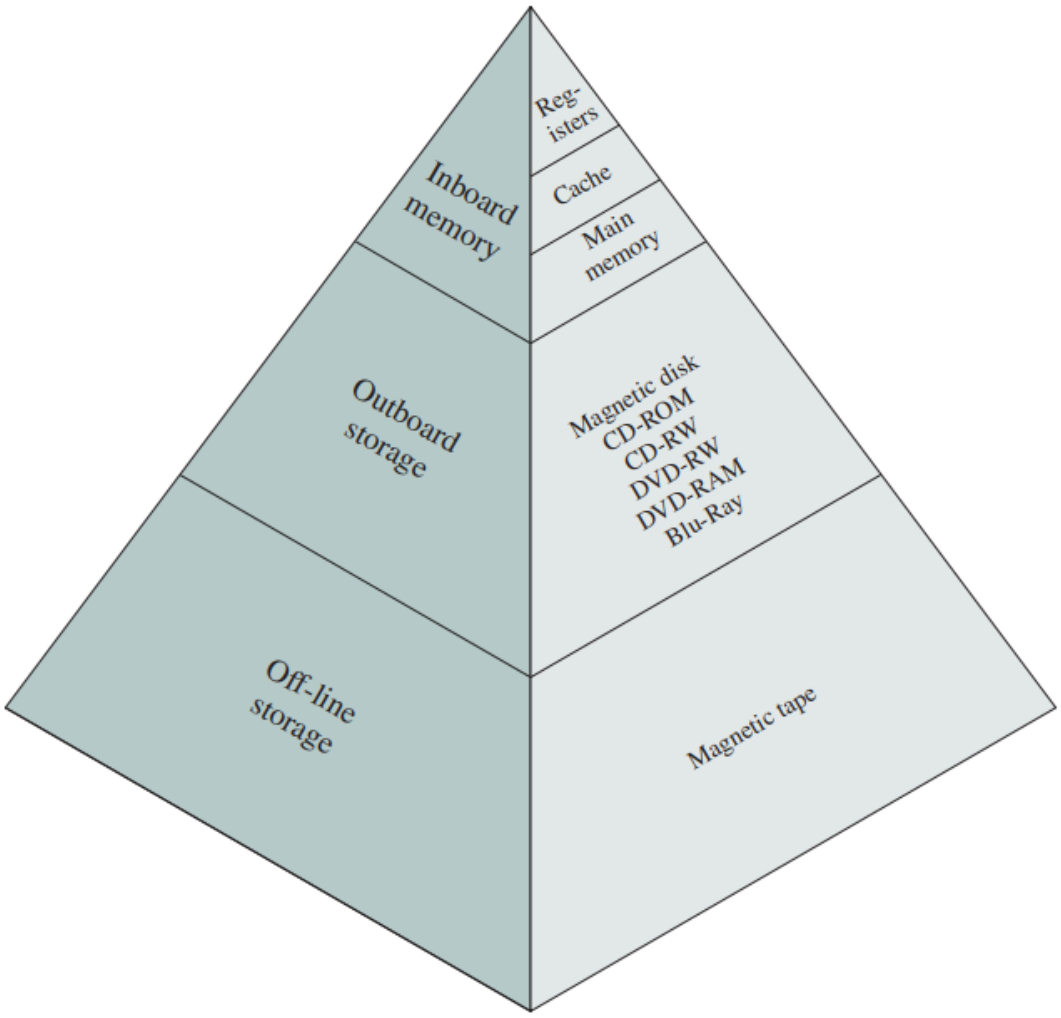
Rutina rukovanja s prekidima je uglavnom dio operacijskog sustava. Prekid okida više događaja i u hardveru i u softveru. Kada I/O uređaj završi I/O operaciju, nastupa sljedeći ciklus događaja:

1. Uređaj procesoru šalje signal za prekid.
2. Procesor završava s izvođenjem trenutne instrukcije prije odgovaranja na prekid.

3. Procesor provjerava nadolazeći zahtjev za prekidom, utvrđuje tip prekida te šalje signal o primitku tom uređaju. Time se uklanja i signal za prekid.
4. Procesor započinje s pripremom za prenošenje kontrole rutini za prekide. Prvo pohranjuje informacije potrebne za nastavljanje trenutnog programa i iduće instrukcije.
5. Procesor učitava programski brojač s ulaznom lokacijom rutine za upravljanje prekidima, te dohvaća prvu instrukciju uvjetovanu ovim prekidom.
6. Svi trenutni registri se spremaju na stog u odgovarajućem poretku.
7. Rukovatelj prekidima započinje s izvođenjem operacija koje odgovaraju prekidu.
8. Nakon što je procesiranje prekida završeno, spremljeni registri se dohvaćaju sa stoga i vraćaju nazad u registre procesora.
9. Zadnji korak je dohvaćanje programskog brojača i informacija spremljenih u 4., kako bi se odredili idući koraci procesora u nastavljanju izvršavanja prekinutog programa.

2.2.4. Hijerarhija računalne memorije

Tri osnovne karakteristike memorije su: kapacitet, brzina pristupa i cijena. Između tih karakteristika mora postojati kompromis, s obzirom da poboljšavanje jedne negativno utječe na drugu. Prilikom dizajna sustava, preporuka je ne oslanjati se na jedan tip memorije, nego upotrijebiti memorijsku hijerarhiju, koja je prikazana na Slici 4. Manje, skuplje i brže memorije se najčešće kombiniraju s velikim, jeftinim i sporim memorijama.



Slika 4. Hijerarhija računalne memorije

Kao što je već spomenuto, brzina rada procesora je značajno veća nego kod drugih komponenata. Iz tog razloga, razvijena je vrlo brza **priručna** (engl. *cache*) memorija, koja je smještena najbliže procesoru. Ona uglavnom sadrži kopije dijela glavne memorije koji je u tom trenutku potreban procesoru za čitanje ili pisanje, kako bi taj proces tekao brže. Kada je procesoru potreban neki podatak, on prvo provjerava postoji li u priručnoj memoriji, a ako ne postoji, on se u nju kopira iz glavne memorije. Zbog fenomena **lokalnosti** referenca, kada se blok podataka dohvaća u priručnu memoriju, velika je vjerojatnost da će i buduće memorijske reference upućivati na bajtove iz tog istog bloka podataka. Blok podataka je podatkovna jedinica koja se prenosi između procesora i priručne memorije. Prilikom razvoja programa, moguće je napisati programski kod koji dobro iskorištava princip lokalnosti, pa se time i brže izvodi na računalu.

Prilikom dizajna priručne memorije, potrebno je odrediti njenu ukupnu veličinu (engl. *cache size*) i veličinu bloka podataka (engl. *block size*). Kada se novi blok podataka učitava u priručnu memoriju, izvodi se funkcija mapiranja (engl. *mapping function*) koja određuje koju lokaciju u priručnoj memoriju će taj blok zauzeti. Ukoliko je na toj lokaciji već prisutan neki blok, izvodi se nekakav algoritam zamjene (engl. *replacement algorithm*), koji sadrži pravila o tome koji blok će se zamijeniti (npr. onaj koji stoji najduže u priručnoj memoriji). Današnja računala sadrži višestruke razine priručne memorije, označene s L1 (najbliža procesoru), L2, L3.

2.2.5. Tehnike izvođenja ulazno-izlaznih operacija

Postoje tri kategorije tehnika za izvođenje I/O operacija:

- **programirani I/O** (engl. *programmed I/O*) – I/O modul izvodi zahtijevanu akciju, postavlja odgovarajuće bite u I/O statusni registar te zatim ne obavještava procesor (ne izvodi prekid). Procesor mora imati mehanizam koji provjerava je li I/O operacija završila (npr. periodička provjera). Kod ovog načina je značajno narušena performansa procesora.
- **I/O vođen prekidima** (engl. *interrupt-driven I/O*) – procesor upućuje I/O operaciju nekom modulu na izvršavanje, dok on nastavlja sa svojim radom. Kada je I/O modul spreman za prijenos podataka prema procesoru, on šalje signal za prekid. Procesor izvodi operaciju zahtijevanu s prekidom te nastavlja sa svojim radom. Ovaj način rada je efikasniji od programiranog I/O, no zahtijeva aktivnu intervenciju procesora za prijenos podataka između memorije i I/O modula, čija je brzina ograničena I/O uređajem.
- **direktan pristup memoriji** (engl. *direct memory access(DMA)*) – za premještanje velike količine podataka, najefikasnija tehnika je DMA. DMA funkcija se može izvršiti kao zaseban modul na sistemskoj sabirnici ili može biti inkorporirana u I/O modul. Kada procesor želi čitati ili pisati blok podataka, šalje naredbu DMA modulu s podacima o tom bloku podataka, vrsti željene operacije te adresi I/O uređaja. Nakon toga procesor nastavlja sa svojim poslom a delegira I/O operaciju DMA modulu. DMA modul dobiva kontrolu nad sistemskog sabirnicom te prenosi cijeli blok podataka direktno iz memorije, bez potrebe za procesorom. Nakon što je prijenos završen, DMA šalje signal prekida procesoru. Iako je ova tehnika najefikasnija zbog male uključenosti procesora, njezin nedostatak se javlja ukoliko procesor i DMA moraju istovremeno koristiti sistemsku sabirnicu. U tom slučaju procesor pauzira s radom, sve dok DMA ne završi.

2.2.6. Višeprocesorska i višejezgrena organizacija računala

Tradicionalno računalo se promatra kao sekvencijalni uređaj, pri čemu se programi i algoritmi specificiraju kao slijed instrukcija. S napretkom računalnog sklopovlja, pojavljuje se sve veći oblik paralelizma u svrhu unaprjeđenja performansi te pouzdanosti. Najpoznatiji pristupi ostvarivanja paralelizma replikacijom procesora su :

- **Simetrična višeprocesorska obrada** (engl. *symmetric multiprocessing (SMP)*) – SMP se može definirati kao samostalno računalo sa sljedećim karakteristikama:
    - Sadrži 2 ili više slična procesora usporedivih mogućnosti.
    - Ti procesori dijele glavnu memoriju i I/O module, a povezani su sabirnicom (ili nekom drugom unutarnjom shemom) tako da je brzina pristupa podjednaka za svaki procesor.
    - Svi procesori dijele pristupni put prema I/O uređajima, ili preko istih kanala ili preko drugih kanala ka istom uređaju.
    - Svi procesori izvršavaju istu funkciju (stoga naziv „simetrični“).
    - Sustavom upravlja integrirani operacijski sustav koji omogućuje interakciju između procesora i programa na razini posla, zadataka, datoteke te struktura podataka.
    - Prednosti ovog pristupa su:
        - Performanse – više procesora rade na istoj funkciji.
        - Dostupnost – ispadom jednog procesora iz rada, sustav može nastaviti s radom samo s nižim performansama.
        - Inkrementalni rast – performanse se mogu povećati dodavanjem novog procesora
        - Skaliranje – različite vrste i broj procesora se mogu kombinirati.

- **Višejezgrena računala** (engl. *multicore computers*) – kombinacija 2 ili više procesora (u ovom kontekstu zvanih jezgrama) na jednom čipu. Svaka jezgra se sastoji od svih komponenata neovisnog procesora (registri, ALU, cjevovodi, CU, priručne memorije…), a čipovi s više takvih jezgri sadrže više razina priručnih memorija (L2 i L3). Motivacija za ovakvim računalima proizlazi iz limita trenutnog hardvera u smislu smanjivanja procesorskih čipova i ujedno povećavanja performansi neovisnih procesora. Tako se stvara ideja za objedinjavanjem više takvih procesora i priručnih memorija na jedan čip. Na primjer, Intel Core i7 se sastoji od 4 procesora x86 arhitekture (svaki sadrži L2 priručnu memoriju) sa dijeljenom L3 priručnom memorijom.

Oba pristupa paralelizacije procesorskog rada doprinose poboljšanju performansi računalnog sklopovlja. Višejezgrena računala ne zahtijevaju tako kompleksnu konfiguraciju kao kod SMP pristupa te izvršavaju jedan program brže. S druge strane, SMP je pouzdaniji i sposobniji u izvršavanju višestrukih programa odjednom.

3. STRUKTURA I IZVRŠAVANJE PROGRAMSKOG KODA

3.1. Prikazivanje programskog koda u strojnom obliku

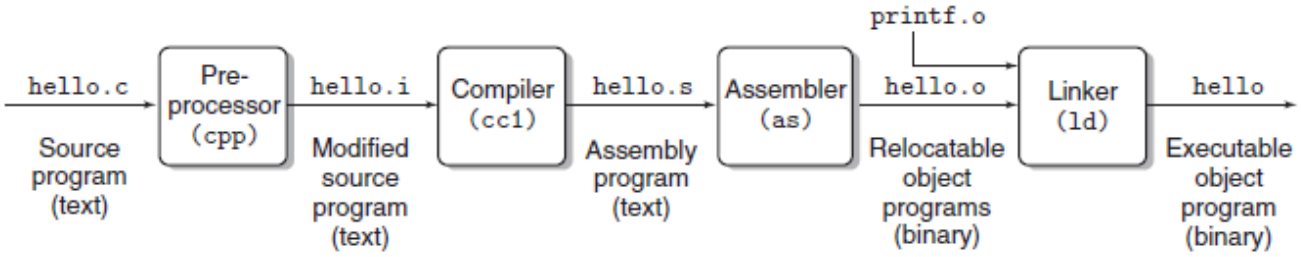
Računala izvršavaju **strojni kod**, tj. slijed bajtova koji predstavljaju kodirane operacije niske razine za manipuliranje podacima, upravljanje memorijom, čitanje i pisanje na pohranu računala te komunikaciju putem mreže. **Prevođenje ili kompajliranje** je postupak kojim se izvorni kod, napisan u određenom programskom jeziku, pretvara u strojni kod za specifičnu računalnu arhitekturu i operacijski sustav.

Današnji razvoj softvera se većinom izvodi programiranjem u višim programskim jezicima (C/C++, Java, C#, ...) koji skrivaju implementacije strojnih instrukcija unutar računala. Međutim, ne tako davno, programiranje se većinom izvodilo u **asembleru** u kojem je bilo obavezno specificirati instrukcije niske razine kako bi procesor izvršio program. Ipak, prednosti kompajlera poput provjere sintaksnih pogrešaka, optimizacija koda, te provjera tipova podataka su gotovo u potpunosti istisnule programiranje u asembleru.

3.1.1. Postupak prevođenja programskog koda napisanog u programskom jeziku C

Prevođenje ili kompajliranje (engl. *compiling*) je proces konverzije izvornog koda u objektni kod. Izvorni kod se sastoji od naredbi programskog jezika i biblioteka treće strane. Prevoditelj (kompajler) provjerava izvorni kod za sintaksne ili strukturne pogreške, a ako je izvorni kod bez pogrešaka, generira objektni kod. Proces prevođenja koda napisanog u programskom jeziku C može se podijeliti u četiri dijela:

1) Predobrada (engl. *preprocessing*) - Pretprocesor prima izvorni kod kao ulaz i uklanja sve komentare. Izvorni kod početno ima nastavak .c. Zatim pronalazi pretprocesorske direktive (#include, #define) unutar koda i tumači ih. Primjerice, ako je *<stdio.h>* biblioteka uključena u izvornom kodu, pretprocesor tumači direktivu #*include* te ju zamjenjuje sa sadržajem datoteke *stdio.h*. Nakon predobrade, izvorni kod se pretvara u obrađeni tekstualni kod s nastavkom *.i*.
2) Prevođenje (engl. *compiling*) - Kompajler pretvara obrađenu tekstualnu datoteku u kod napisan u asembleru s nastavkom .*s*.
3) Sastavljanje (engl. *assembling*) - Asembler pretvara datoteku nastavka .*s* u objektni kod. Objektni kod je napisan u strojnom jeziku (kombinacija nula i jedinica). Nastavak objektnog koda je *.o*. Primjerice, ako je naziv datoteke u kojoj je pohranjen izvorni kod napisan u programskom jeziku C bio *hello.c*, nakon izvođenja ovog koraka datoteka će imati naziv *hello.o*.
4) Povezivanje (engl. *linking*) - Većina programa napisana u programskom jeziku C koristi funkcije raznih biblioteka. Te funkcije su već unaprijed kompajliranje, a njihov objektni kod je spremljen u datotekama s nastavkom *.lib* ili *.a*. Glavni zadatak linkera je povezati objektni kod tih biblioteka s objektnim kodom izvornog programa. Kao rezultat ovog koraka, stvara se izvršna datoteka istog imena kao i izvorni kod, ali nastavka *.exe (Windows)* ili *.out (UNIX)*. Izvršna datoteka se sastoji od strojnog koda kojeg procesor može izvršiti.



Slika 5. Proces prevođenja izvornog koda napisanog u C programskom jeziku

Još jedna grafička reprezentacija:

hello.c

```
#include <stdio.h>
int main() {
        printf ("hello, world\n");
        return 0;
}
```

hello.i

#	i	n	c	l	u	d	e	SP	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	SP	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	SP	SP	SP	SP	p	r	i	n	t	f	(	h e l			
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	SP	w	o	r	l	d	\	n	) ; \n SP				
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	32
SP	SP	SP	r	e	t	u	r	n	SP	0	;	\n	}	\n	
32	32	32	114	101	116	117	114	110	32	48	59	10	125	10	

hello.s

```
main:
        subq $8, %rsp
        movl $.LCO, %edi
        call puts
        movl $0, %eax
        addq $8, %rsp
        ret
```

hello.o

53 48 89 d3 e8 00 00 00
00 48 89 03 5b c3

Slika 6. Proces prevođenja izvornog koda napisanog u C programskom jeziku

3.1.2. Strojni kod

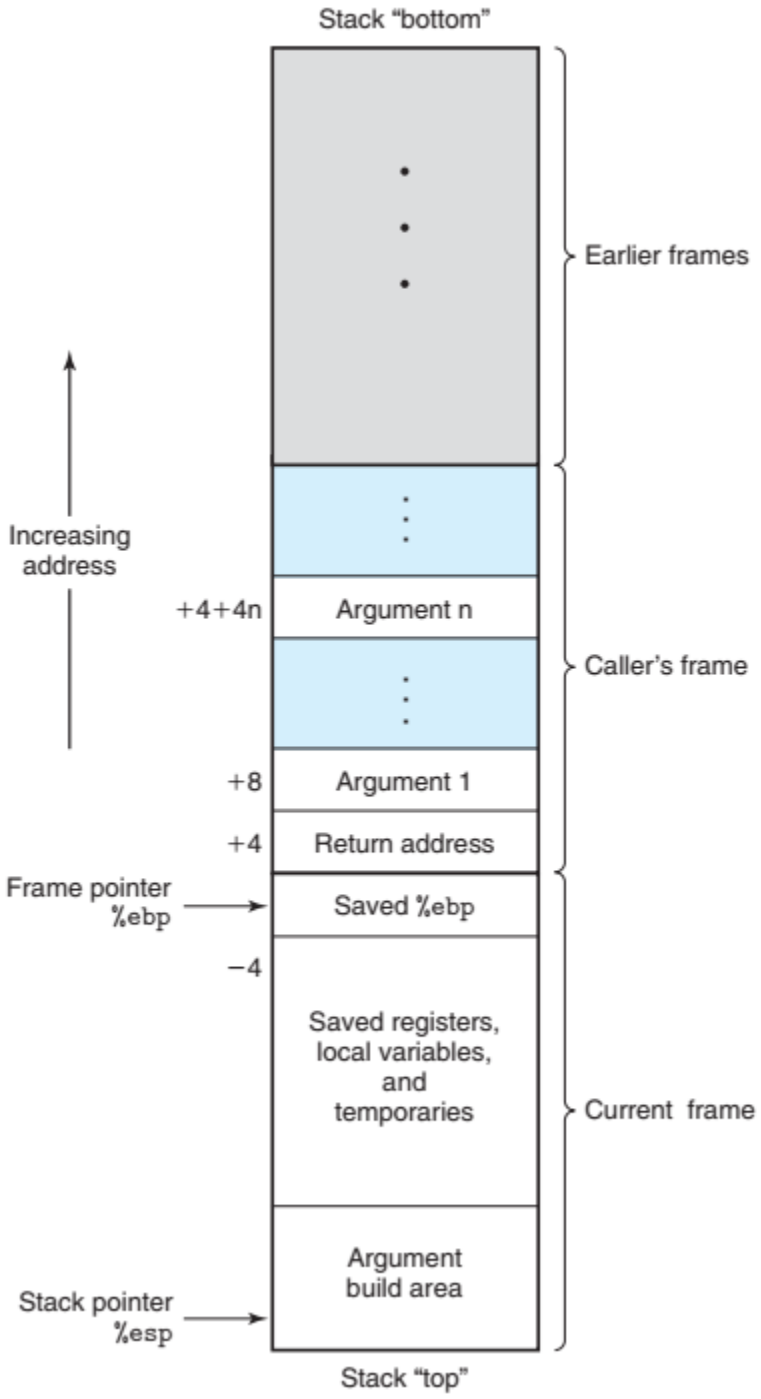
Kompajler prevodi izvorni program napisan u programskom jeziku više razine u strojni kod kojeg računalo razumije. Taj postupak pretvaranja je specifičan za ciljanu računalnu

arhitekturu na kojoj se program treba izvršavati. Iako kompajler od programera skriva detalje sklopovlja, dobiveni strojni kod može se izvršiti samo na računalnim arhitekturama za koje je provedeno prevođenje. Stoga se nameće logično pitanje - kako kompajler generira strojni kod prikladan baš za traženi procesor?

Arhitektura skupa instrukcija (ISA) je apstrakcija računala (procesora) u kojoj su definirani podržani tipovi podataka, registri i razne značajke procesora (virtualna memorija, načini adresiranja, …). ISA specificira ponašanje strojnog koda pružajući binarnu kompatibilnost između različitih implementacija. Operacijski sustavi održavaju standardno i kompatibilno aplikacijsko binarno sučelje (engl. *application binary interface (ABI)*) za pojedinačnu ISA-u. ISA je definirana logičkim sklopovima unutar računala, ne postoji specifična lokacija u kojoj se taj skup instrukcija drži.

Izvorni kod napisan u programskom jeziku C razlikuje se u odnosu na strojni kod po tome što sadrži dijelove stanja procesora, poput programskog brojača (koji ukazuje na adresu iduće naredbe u memoriji), datoteke s registrima (16 lokacija registara opće namjene), adrese registara uvjetnog koda (sadrže statusne informacije o nedavno evaluiranim aritmetičkim i logičkim operacijama) te adrese vektorskih registara koji sadrže cjelobrojne ili realne vrijednosti. Dok u C-u definiramo i alociramo različite tipove podataka bez saznanja o stvarnoj strukturi memorije, strojni jezik memoriju promatra kao jednostavno polje bajtova od kojih svaki ima specifičnu adresu. Agregirani tipovi podataka iz C-a (polja, strukture) se u strojnom jeziku predstavljaju kao uzastopne kolekcije bajtova.

Strukture programa koje se smještaju u radnu memoriju sastoje se od izvršne datoteke programa, nekih informacija od operacijskog sustava, *run-time* stoga za upravljanje pozivima procedura i njihovim povratnim vrijednostima te adrese blokova memorije koje je alocirao korisnik (npr. blokovi alocirani funkcijom *malloc*). Memorija programa se adresira preko koncepta **virtualne memorije**. U 64-bitnom računalu, adresa se sastoji od 64 bita. Čitanjem i prevođenjem virtualne adrese dolazi se do prave fizičke adrese koja upućuje gdje su navedeni elementi smješteni u računalu. **Operacijski sustav** je taj koji upravlja virtualnim adresnim prostorom i vrši prevođenje virtualnih adresa u fizičke adrese. Poziv procedura unutar programa uključuje prosljeđivanje podataka u obliku parametara te kontrole s jednog dijela programa na drugi. Dodatno, potrebno je alocirati prostor za lokalne varijable procedure te dealocirati ga nakon završetka. Prosljeđivanje podatka, alokaciju i dealokaciju lokalnih varijabli manipulira **programski stog**. Dio stoga alociran za jednu proceduru se naziva okvir stoga (engl. *stack frame*), a njegova struktura je prikazana na Slici 7.



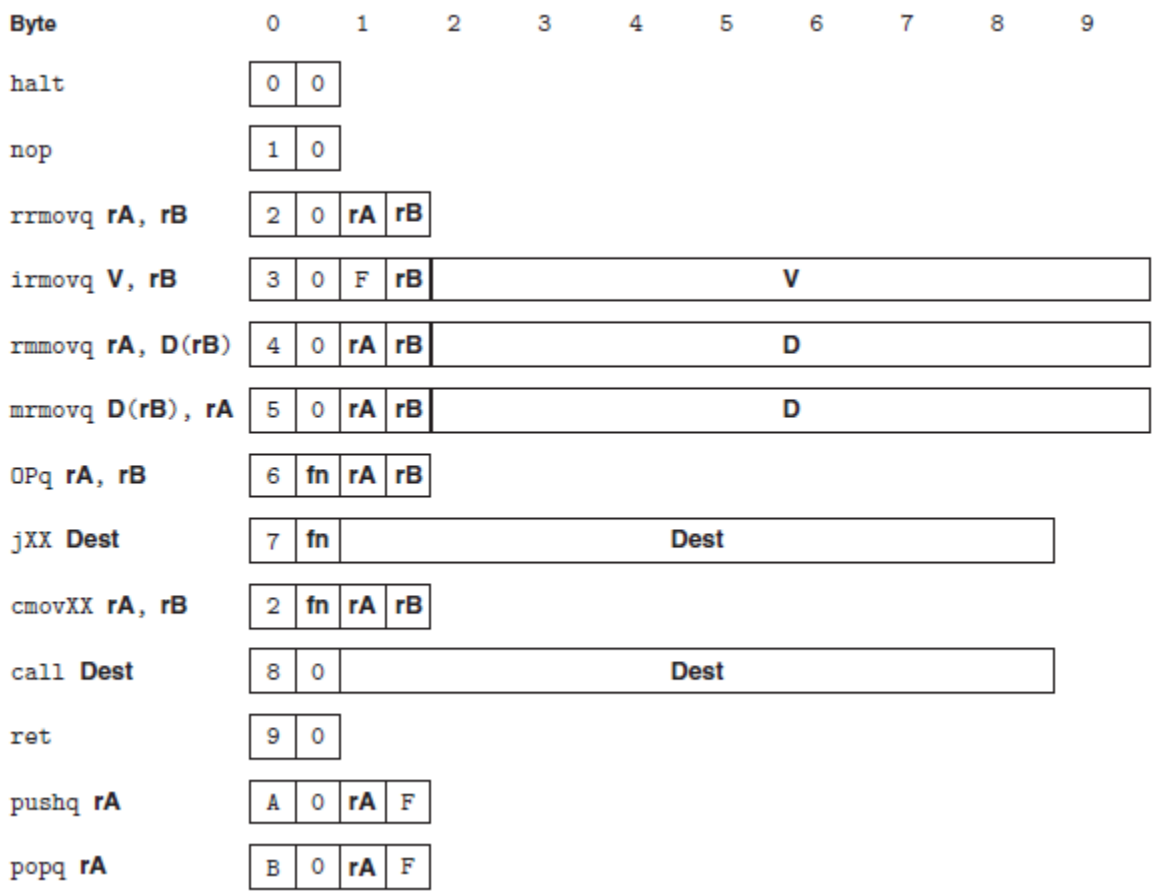
Slika 7. Struktura jednog okvira stoga

3.1.3. Pristupanje informacijama

Uobičajeni (Intel) procesor arhitekture x86-64 sadrži 16 registara opće namjene koji mogu pohraniti 64-bitne vrijednosti. Ti se registri koriste za pohranu cjelobrojnih vrijednosti te sadržaja pokazivača. Slika 8 prikazuje strukturu 16 registra opće namjene kod Intel x86-64 procesora.

Slika 8. Registri opće namjene unutar Intel procesora x86-64 arhitekture

Naredbe u asembleru uglavnom imaju jedan ili više operanda koji specificiraju vrijednosti nad kojima se izvode naredbe te lokacije na koje treba pohraniti rezultat. Vrijednosti mogu biti konstante ili se čitaju iz drugih registara ili radne memorije. Slika 9 prikazuje podskup ISA modela za Inter procesor Y86-64, u kojem su kodirane instrukcije veličine od 1 do 10 bajtova. Tipična instrukcija sadrži inicijalni bajt koji identificira tip instrukcije. Taj bajt se dijeli na dva dijela od 4 bita, pri čemu gornjih 4 bita označavaju kod, a donjih 4 bita funkciju. Idući bajtovi se odnose na argumente strojne instrukcije te podatke nad kojima se instrukcija izvršava. Procesor je dizajniran tako da za svaki bit točno zna što on predstavlja s obzirom na njegovu lokaciju, a ti bitovi su zapravo binarni oblik strojnog koda koji se treba izvršiti.



Slika 9. ISA model za procesor Y86-64

3.2. Izvršavanje programa na operacijskom sustavu

3.2.1. Povezivanje (Linking)

Povezivanje je proces prikupljanja i kombiniranja različitih dijelova koda i biblioteka u jedinstvenu datoteku koja se potom učitava u glavnu memoriju i izvršava. Povezivanje se može izvršiti tijekom prevođenja, kada se izvorni kod prevodi u strojni kod ili tijekom učitavanja, kada se program učitava u glavnu memoriju i ga izvršava ga dio operacijskog sustava zvan *loader*. *Linker* omogućuje razvoj softvera kao kolekcije malih modula koji se mogu modificirati i kompajlirati zasebno.

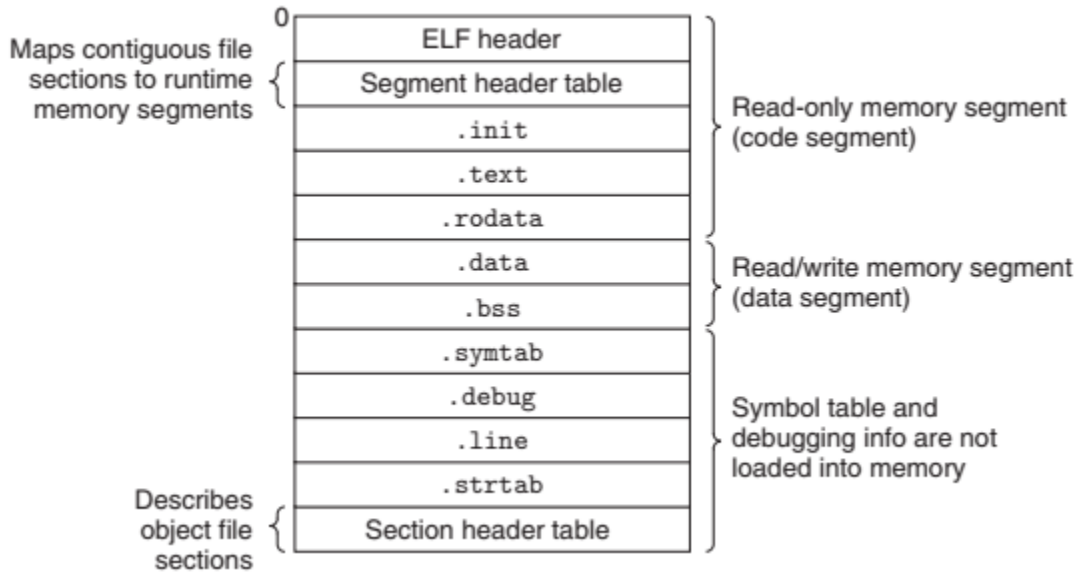
U suštini postoje dvije vrste povezivanja: statičko i dinamičko. **Statičko povezivanje** (engl. *static linking*) uzima kao ulaz kolekciju objektnih datoteka te argumente komandne linije te generira izvršnu datoteku koja povezuje sve navedene podatke i koja se može izvršiti i pokretati. Kako bi izgradio izvršnu datoteku, *linker* provodi dva zadatka:

- Rezoluciju simbola - objektni kod sadrži simbole (adrese funkcija, varijabli, …). Svrha rezolucije simbola je asocirati svaku referencu simbola sa svakom definicijom simbola.
- Relokacija - kompajleri i asembleri generiraju kod čiji podaci kreću od adrese 0. *Linker* mora realocirati te podatkovne sekcije asocirajući memorijske lokacije sa svakom definicijom simbola te modificira svaki poziv na simbol tako da pokazuje na prave lokacije.

Linker ima minimalno znanje o ciljanom stroju na kojem će izvršavati kod. Kompajler i asembler su programi koji su zaduženi za prevođenje izvornog koda u strojni kod prikladan za ciljanu računalnu arhitekturu. Objektne datoteke dolaze u tri oblika:

- **Premjestive** (engl. *relocatable*) - sadrže binarni kod i podatke u obliku u kojem se mogu kombinirati s drugim objektnim datotekama tijekom kompajliranja u svrhu stvaranja izvršne datoteke. Stvaraju ih kompajler i asembler.
- **Izvršne** (engl. *executable*) - sadrže binarni kod i podatke u obliku u kojem se direktno mogu kopirati u memoriju i izvršiti. Stvara ih *linker*.
- **Zajedničke** (engl. *shared*) - posebni tipovi premjestivih objektnih datoteka koje se mogu učitati u memoriju i povezati dinamički s drugim objektnim datotekama. Stvaraju ih kompajler i asembler.

Struktura izvršne datoteke vezana je uz operacijski sustav na kojem se pokreće. *Razmisliti o tome kako se pokreću i kompajliraju programi koji se izvršavaju na računalu koje ne sadrži operacijski sustav?* Moderni Unix operacijski sustavi (Linux, System V Unix, BDS Unix, Sun Solaris) koriste ELF (*Executable and Linkable Format*) format. Zaglavlje ELF datoteke opisuje cjelokupni format izvršne datoteke. Također uključuje i početnu točku programa, tj. adresu prve instrukcije koju treba pokrenuti kada se program izvršava. Sekcije *.text*, *.rodata* i *.data* sadrže redom strojni kod kompajliranog programa, podatke samo za čitanje poput formata printf naredbi te inicijalizirane globalne varijable. Sekcija *.init* sadrži malu funkciju zvanu *_init*, koju poziva inicijalizacijski kod programa. ELF izvršne datoteke su dizajnirane radi lakog učitavanja u memoriju, tako da se kontinuirani blokovi datoteke jednostavno mapiraju u blokove u memoriju. Način mapiranja je definiran u *segment header table* bloku.

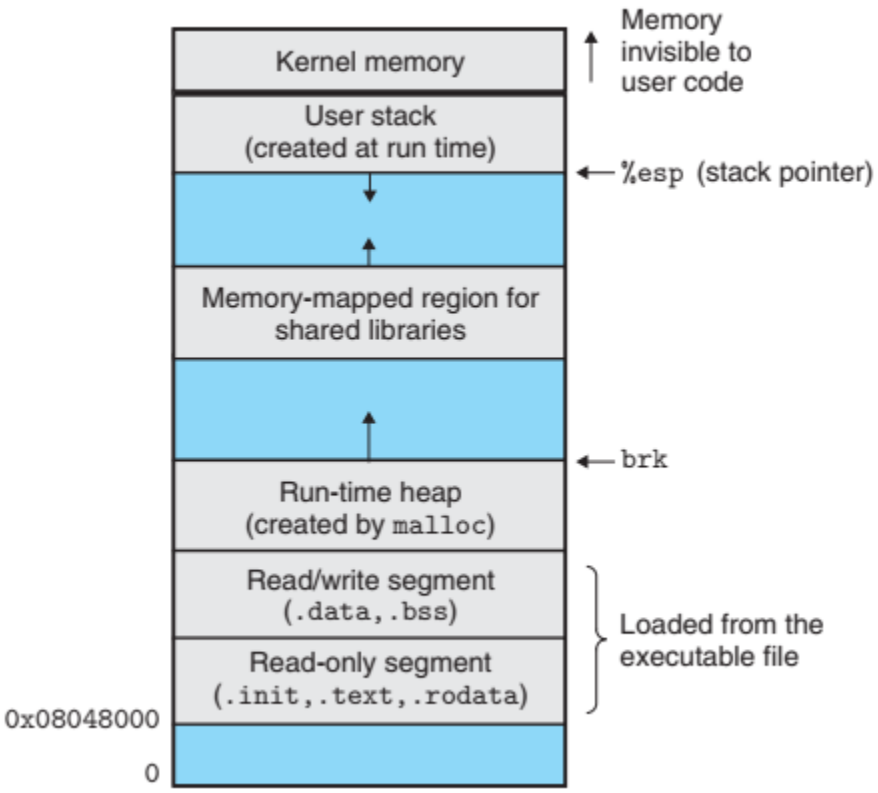


Slika 10. Struktura ELF izvršne datoteke

3.2.2. Učitavanje programa u radnu memoriju

Dio operacijskog sustava za upravljanje memorijom, *loader*, kopira kod i podatke izvršne datoteke s diska u glavnu memoriju te pokreće program tako što za iduću instrukciju računala pokazuje na početnu točku programa. Ovaj proces se naziva **učitavanje** programa u memoriju računala. Svaki program na UNIX-u ima *run-time* memorijsku sliku, koja je prikazana na slici 11. Na 32-bitnom operacijskom sustavu Linux, segment koda počinje na adresi *0x08048000*. Nakon njega slijedi segment podataka na idućih 4KB. Na sljedećih 4KB

nalazi se *run-time* hrpa koja se može i povećati pozivanjem *malloc* funkcija. Zatim slijedi segment za dijeljene biblioteke. Korisnički stog uvijek počinje na najvećoj mogućoj korisničkih adresi i raste prema nižim adresama. Segment koji započinje iznad stoga je rezerviran za kod i podatke dijela operacijskog sustava koji se naziva **jezgra (engl. *kernel*).**



Slika 11. *Run-time* memorijska slika Linux programa

Statičke biblioteke imaju neke značajne nedostatke. Kao i većina softvera, moraju se ažurirati i održavati periodično. Ako aplikacijski programeri žele koristiti najnovije verzije biblioteke, moraju na neki način eksplicitno ponovno povezati njihove programe s tim verzijama biblioteka. Osim toga, gotovo svi programi napisani u programskom jeziku C koriste standardne I/O funkcije poput *printf* te *scanf*. Tijekom izvršavanja tih programa, čitav kod tih funkcija se duplicira u memoriji za svaki od procesa u programu. Ako se na operacijskom sustavu vrti pedesetak takvih programa, značajna količina memorije bi bila neefikasno iskorištena. Dijeljene biblioteke su moderna inovacija koja nastoji prevladati ove nedostatke statičkih biblioteka. One se tijekom izvršavanja programa učitavaju u radnu memoriju te tek zatim vežu s programom u memoriji. Ovaj proces se naziva **dinamičkim povezivanjem** te ga izvodi *dynamic linker.* Dijeljene biblioteke se na operacijskom sustavu UNIX najčešće nalaze u datotekama s nastavkom *.so*, a u operacijskom sustavu Windows *.dll*.

3.2.3. Kontrola toka programa

Moderni sustavi moraju pravodobno reagirati na promjene u sustavu koje nisu zahvaćane internim programima i nisu nužno vezane uz izvođenje koda. Primjerice, mrežni paketi su

pristigli putem mrežnog adaptera i moraju se pohraniti u memoriji. Također, program zahtijeva podatke s diska i odlazi na spavanje sve dok podaci nisu dostupni. Svi dijelovi računala na svoj način reagiraju na takve događaje. Na razini sklopovlja, događaji otkriveni okidačem hardvera prenose nadzor na rukovatelj iznimkama. Na razini operacijskog sustava, jezgra prenosi kontrolu s jednog procesa na drugi putem kontekstnog preklopnika (engl. *context switch*). Na razini primjenskih programa, proces može poslati signal drugom procesu koji izaziva pozivanje rukovatelja signalima. Individualni program može reagirati na pogreške u kodu pozivajući rutinu za upravljanje iznimkama.

Kada procesor detektira događaj koji predstavlja iznimku u sustavu, stvara indirektni poziv procedure preko tablice iznimka koja poziva rutinu operacijskog sustava za upravljanje iznimkama. Kada rukovatelj iznimkama završi, jedna od tri moguće situacije nastupa:

- Rukovatelj vraća kontrolu instrukciji koja se izvršava prije nastanka iznimke
- Rukovatelj vraća kontrolu instrukciji koja je iduća na redu nakon instrukcije koja je izazvala iznimku
- Rukovatelj prekida program u kojem je nastala iznimka

U današnjim modernim računalima, postoje četiri vrste iznimka:

- **Prekidi** (engl. *interrupt*) - pojavljuju se asinkrono kao rezultati I/O operacija s vanjskim uređajima. I/O uređaji poput mrežnih adaptera, upravljača diskova ili vremenskih sklopova okidaju prekid signalizirajući pin na čipu procesora i postavljajući na sistemsku sabirnicu broj iznimke koja identificira uređaj koji je uzrokovao prekid. Nakon što rukovatelj prekida obradi iznimku, proces nastavlja s izvođenjem sljedeće naredbe koja je na redu nakon one koja je uzrokovala prekid.
- **Zamke ili sustavski pozivi** - zamke su namjerno uzrokovane iznimke koje nastaju kao rezultat izvođenja instrukcije. Kao i rukovatelji prekida, rukovatelji zamki također vraćaju kontrolu instrukciji koja bi se sljedeća trebala izvršiti. Najvažnija svrha zamki je omogućiti poziv sučelja između korisničkih programa i jezgre operacijskog sustava koje se naziva **sustavski pozivi**.
- **Greške** - greške nastaju zbog uvjeta u programu koji se izvršava. Kada nastaje greška, proces predaje kontrolu rukovatelju greškama. Ako on uspije identificirati stanje pogreške vraća kontrolu instrukciji koja ju je uzrokovala, a u suprotnom pokreće *abort* rutinu u jezgri operacijskog sustava koja nastoji ubiti proces u kojem je došlo do pogreške. Najpoznatije greške u operacijskom sustavu Linux su: dijeljenje s 0, referenciranje nepostojeće adrese u memoriji i greška stranice.
- **Fatalne greške (engl. *abort*)** - pogreške od kojih se ne može oporaviti. Obično su to hardverske greške, poput grešaka pariteta koje se javljaju kada su oštećeni DRAM ili SRAM bitovi. Rukovatelji fatalnih grešaka nikada ne vraćaju kontrolu primjenskom programu nego pozivaju *abort* rutinu.

4. OPTIMIZACIJA PERFORMANSI PROGRAMSKOG KODA (*Za one koji žele znati više)

Ovo poglavlje nije uvjet za razumijevanje laboratorijskih vježbi iz kolegija Operacijski sustavi. Međutim, sadržaj poglavlja je usko vezan uz arhitekturu računala te funkcioniranje svih programa koji se izvršavaju na računalu, pa tako i operacijskih sustava. Osim toga, prikazani savjeti za optimiranje programskog koda su korisni za svakog studenta iz područja računarstva koji se planira baviti programiranjem različitih računalnih sustava.

Pisanje efikasnog koda podrazumijeva izvršavanje slijeda različitih aktivnosti. Prvo je potrebno odabrati odgovarajuće strukture podataka i pripadne algoritme za izvršavanje željenog ponašanja programa. Zatim slijedi pisanje izvornog koda kojeg kompajler pretvara u objektni kod za izvršavanje na računalu. Tijekom ovog drugog dijela razvoja softvera, važno je razumjeti ograničenja i mogućnosti kompajlerske optimizacije koda. Naizgled male razlike u izvornom kodu mogu doprinijeti znatnom poboljšanju performansi programa koji se izvršava slijedom tog koda. Moderni kompajleri iskorištavaju sofisticirane tehnike optimizacije programskog koda, no i oni nisu imuni na pojedine aspekte ponašanja programa koji se teško mogu poboljšati. Kako bi se maksimizirale performanse specifičnog programa, kompajler i programer moraju primijeniti napredne tehnike optimizacije koda.

4.1. Ograničenja i mogućnosti modernog kompajlera

Moderni kompajleri iskorištavaju sofisticirane algoritme koji pojednostavljuju izraze programskog koda i skraćuju broj potrebnih instrukcija procesora. Većina kompajlera, uključujući i *gcc*, dozvoljavaju programeru kontrolu nad stupnjem optimizacije kojeg primjenjuju tijekom procesa prevođenja izvornog koda. Primjerice, kod *gcc* kompajlera postoje *-0g*, *-01*, *-02* i *-03* razine optimizacije, pri čemu se većina razina podrazumijeva i više izmijenjeni izvorni kod, kojeg je tada teže debugirati. Navesti ćemo najpoznatije optimizacijske blokatore:

- *Memory aliasing* – kada dva pokazivača pokazuju na istu memorijsku lokaciju. Tijekom izvođenja sigurnih optimizacija, kompajler mora pretpostaviti da je moguća situacija da dva različita pokazivača pokazuju na istu lokaciju. Ako iz ostatka koda ne može odrediti je li to istina, smanjuje se skup mogućih optimizacija.
- Pozivi funkcija – kompajler ne može biti siguran uzrokuje li poziv funkcije određene vanjske efekte, poput promjene vrijednosti globalne varijable. Pretpostavljajući najgore, ta funkcija se izuzima iz optimizacije.

4.2. Određivanje performansi programa

Standardna metrika za izražavanje performansi nekog programskog koda je broj ciklusa (procesora) po elementu (engl. *cycles per element (CPE)*). Slijed aktivnosti unutar procesora određen je frekvencijom takta procesora koja se najčešće izražava u GHz, tj. milijardi ciklusa u sekundi. Ovisno o programskom jeziku, operacijskom sustavu i arhitekturi računala svaka naredba u programskom kodu ima određenu vrijednost CPE metrike. Ako se naredba izvodi unutar petlje, onda se ta linija koda ponavlja onoliko puta koliko se izvršava i petlja, pa je ukupan CPE umnožak CPE vrijednosti te naredbe i broja njenog ponavljanja.

4.3. Otklanjanje neučinkovitosti petlja

Standardna petlja je definirana kroz nekoliko aspekata, poput inicijalnog stanja brojača, uvjeta petlje te vrste inkrementa brojača. Slika 8 prikazuje dvije varijacije iste petlje, pri čemu je u drugoj iteraciji provedena jednostavna optimizacija koda. Ona uključuje izračun uvjeta petlje prije definicije petlje te se na taj način smanjuje broj poziva funkcije *vec_length()*.

```
for (i = 0; i < vec_length(v); i++){
        …
}
```

```
long length = vec_length(v);
for (i=0; i < length; i++) {
        …
}
```

Slika 8. Optimizacija uvjeta petlje

4.4. Smanjenje poziva procedura

Pozivi vanjskih procedura mogu negativno utjecati na performanse programskog koda. Osim toga, njihov unutarnji mehanizam može izazvati štetne vanjske efekte, poput promjene vrijednosti nekih globalnih varijabli. Generalno govoreći, smanjenje poziva procedura je jedan od koraka transformacije programskog koda koji može poboljšati performanse programa. Slika 9 prikazuje dvije varijacije iste petlje, pri čemu je u drugoj varijaciji poziv funkcije *get_vec_element* zamijenjen direktnim pristupom elementu vektora. Umjesto izvođenja poziva funkcije za dohvaćanje svakog elementa zasebno, druga petlja direktno pristupa polju.

```
long length = vec_length(v);
for (i = 0; i < length; i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest + *val;
}
```

```
long length = vec_length(v);
data_t *data = get_vec_start(v);
for (i = 0; i < length; i++) {
        *dest = *dest + *data[i];
}
```

Slika 9. Primjer optimizacije petlje smanjenjem poziva procedure

4.5. Uklanjanje nepotrebnih referenci na memoriju

Programski kod sa slike 9 čita element po element vektora *val* te ih zbraja, a ukupnu sumu pohranjuje u vektor *dest*. Stoga, u svakoj iteraciji petlje program čita memoriju na koju pokazuje referenca *dest* te piše na taj blok. Jedan od načina optimizacije ovog programskog koda je uvođenje lokalne varijable u koju se pohranjuje rezultat zbrajanja i koja se tek na izlazu iz petlje sprema na adresu na koju pokazuje *dest*. Iako je broj zbrajanja ostao nepromijenjen, broj pristupa točno određenoj adresi u memoriji (ili registru procesora) je smanjen na 1, dok se lokalnoj varijabli brže pristupa s obzirom da operacijski sustav i kompajler vode računa o njenoj adresi te je ona nepoznata programeru. Optimizirani kod se nalazi na slici 10.

```
long length = vec_length(v);

data_t *data = get_vec_start(v);

data_t temp = 0;

for (i = 0; i < length; i++) {
        temp = temp + *data[i];

}

*dest = temp;
```

Slika 10. Primjer optimizacije petlje smanjenjem poziva procedure

4.6. Razumijevanje modernih procesora

Prethodno prikazane tehnike optimizacije programskog koda se nisu oslanjale na značajke arhitekture procesora, već su obuhvatile općenite savjete vezane uz programski jezik. Kako bi se dodatno unaprijedile performanse programa, potrebno je razumjeti ponašanje modernih procesora. Na razini programskog koda, čini se da procesor slijedno izvršava instrukcije. Međutim, u modernim procesorima nekoliko instrukcija se izvršava istovremeno, a taj fenom se naziva **paralelizmom na razini instrukcija** (engl. *instruction-level paralelism*). Takvi procesori se u literature nazivaju superskalarnim procesorima. Uz to, redoslijed izvršavanja instrukcija unutar procesora ne mora odgovarati redoslijedu naredbi unutar programskog koda. Općenito, superskalarni procesor se sastoji od dva dijela: jedinice za kontrolu instrukcije (engl. *instruction control unit (ICU)*) te jedinice za izvršavanje (engl. *execution unit (EU)*). ICU čita instrukcije iz priručne memorije namijenjene za pohranu instrukcija (engl. *instruction cache*), no ne čeka da EU završi s izvođenjem neke instrukcije da bi dohvatila i dekodirala sljedeću instrukciju iz priručne memorije. Jedan od problema koji stoga nastaje je kada u programu dolazi do grananje, ICU ne može znati koja grana programa se treba izvršiti sljedeća, jer EC još nije stigla izvršiti uvjet grananja. Moderni procesori koriste tehniku predviđanja grane, prema kojoj predviđaju koja će se grana izvršiti. Takvo izvođenje se još naziva i spekulativno izvođenje. Ako se u kasnijem trenutku zaključi da se izvela pogrešna grana, resetira se stanje sve do uvjeta grananje i izvršava ispravna grana. EU prima operacije od ICU svakim ciklusom takta procesora te zatim prosljeđuje te operacije funkcionalnim jedinicama za čitanje, pisanje, računske i logičke operacije u procesoru. Spekulativnim izvođenjem instrukcija, procesor nikad ne pohranjuje konačne vrijednosti u memoriju ili na disk, sve dok ne bude siguran da su te vrijednosti ispravne.

Funkcionalne jedinice procesora vrše računske i logičke operacije, a svaka od operacija ima unaprijed definirano trajanje izvođenja te kapacitet memorije koji je za nju potreban. Znajući te podatke, moguće je provesti analizu programskog koda tako da se za svaku liniju definira njeno ukupno trajanje u smislu CPE vrijednosti. Zatim je moguće svesti programski kod na grafičku reprezentaciju toka programa raznim alatima za analizu programa. Na taj način se lakše stječe uvid u „problematične“ programske linije koje je zatim moguće zamijeniti efikasnijim naredbama.

4.7. Odmotavanje programske petlje

Odmotavanje programske petlje je transformacija programskog koda koja smanjuje broj iteracija petlje povećavajući broj elemenata koji se računaju unutar svake iteracije. Odmotavanje petlje smanjuje broj operacija poput indeksiranja petlje i grananja. Na slici 11 dodatno je optimiziran programski kod iz prethodnih primjera. Broj iteracija petlje je prepolovljen, dok se u svakoj iteraciji u ukupni zbroj dodaju dva elementa, umjesto jednog u prethodnim primjerima. Ovakvo odmotavanje petlje se naziva *2 x 1* odmotavanjem, jer se broj iteracija prepolovljuje. Međutim, potrebno je pripaziti da je veličina polja paran broj, te primijeniti drugu vrstu odmotavanja ukoliko nije.

```
long length = vec_length(v);

data_t *data = get_vec_start(v);

data_t temp = 0;

for (i = 0; i < length; i+=2) {
        temp = temp + *data[i] + *data[i+1];

}

*dest = temp;
```

Slika 11. Primjer optimizacije petlje smanjenjem poziva procedure

4.8. Povećanje broja paralelnih operacija

Funkcionalne jedinice procesora mogu izvoditi računske operacije svakim novim taktom procesora. Međutim, programski kod uglavnom ne može u potpunosti iskoristiti svaki takt, jer izvodi operaciju spremanja vrijednosti. Uvođenje dodatnog paralelizma u programski kod, omogućilo bi paralelno izvođenje operacija za zbrajanje i spremanje, čime bi se bolje iskoristio računalni kapacitet procesora. Na slici 12 prikazana dodatna optimizacija programskog koda iz prethodnih primjera prema kojoj se paralelno izvode dvije računske operacije zbrajanja pomoću dvije lokalne varijable. Ovakvo odmotavanje petlje se naziva *2 x 2* odmotavanje, jer se broj iteracija petlje prepolovljuje, ali i paralelno se izvode dvije operacije unutar iteracije.

```
long length = vec_length(v);

data_t *data = get_vec_start(v);

data_t temp = 0;

data_t temp1 = 0;

for (i = 0; i < length; i+=2) {
        temp = temp + *data[i];

        temp1 = temp1 + *data[i+1];

}

*dest = temp0 + temp1;
```

Slika 11. Primjer optimizacije petlje smanjenjem poziva procedure

4.9. Tehnike optimizacije performansi programskog koda u stvarnom svijetu

U prethodnim poglavljima opisano je samo nekoliko tehnika za optimizaciju performansi programskog koda. U suštini, strategije optimizacije se mogu podijeliti na sljedeće:

- Dizajn visoke razine – odabir prikladnih algoritama i struktura podataka za problem koji se rješava. Primjerice, odabir odgovarajućeg algoritma za sortiranje, za pristup polju, množenje matrica,…
- Osnovni principi kodiranja – Izbjegavati blokatore optimizacije:
    - Eliminirati višak poziva funkcija
    - Premjestiti računske operacije izvan petlje
    - Eliminirati nepotrebne reference na memoriju
- Optimizacije niske razine – strukturirati kod tako da iskoristi mogućnosti sklopovlja:
    - Odmotavanje programskih petlji
    - Uvođenje paralelizma na razini instrukcija
    - Transformirati uvjete grananja tako da se izvodi manji broj računalnih instrukcija (premještanje zagrada kod OR ili AND operatora i sl.)

Primjena ovih optimizacijskih praksi lako je ostvariva kod malih programa. Međutim, pri razvoju velikih softvera, teško je prepoznati moguće optimizacijske blokatore i identificirati linije koda je potencijalno moguće optimizirati. U tu svrhu koriste se **profilatori koda** (engl. *code profilers*), alati za analizu performansi programa koji se izvršava.

5. LITERATURA

[1] Stallings, W., 2012. Operating systems: internals and design principles. Boston: Prentice Hall,.

[2] Tanenbaum, A.S. and Bos, H., 2015. Modern operating systems. Pearson.

[3] https://gcc.gnu.org/

[4] Bryant, R. and O'Hallaron, D., 2016. Computer Systems: A Programmer's Perspective. Carnegie Mellon University