Operativni sistemi

1 Uvod

Donedavno su se operativni sistemi vezivali isključivo za personalne računare. Kada vam neko pomene operativni sistem, prva asocijacija bi, uglavnom bio Microsoft Windows (10/7/Vista/XP/98..), možda neka distribucija Linux operativnog sistema ili eventualno macOS-X. Ali svakako neki personalni računar na kojem možete da u grafičkom okruženju pokrećete programe koji se izvršavaju (naizgled) istovremeno.

Sa druge strane, sve rasprostranjeniji su postajali embeded sistemi. Prvobitni embeded sistemi su bili prilično jednostavni: najčešće su sadržali neki mikrokontroler skromnih performansi. Oni nisu imali previše RAM memorije, uglavnom su imali skroman CPU, eventualno sa mogućnošću protočne obrade instrukcija-dok se jedna izvršava, druga se dekoduje, a treća prihvata iz memorije. Imali su relativno mnogo periferijskih jedinica koje su bile značajne za interakciju sa spoljašnjim svetom - A/D konvertori, tajmeri/brojači, komunikacioni interfejsi (UART, SPI, I2C), PWM izlazi, itd.

Tipičan program koji bi se izvršavao na jednom takvom embeded sistemu obavezno je sadržao jednu beskonačnu petlju, koja je obezbeđivala da programski brojač nikada ne dosegne „zabranjenu zonu“ tj. zonu u kojoj se ne nalazi nikakav program, odnosno zonu u kojoj se nalaze ilegalne instrukcije. Unutar petlje, sekvencijalno su se izvršavale operacije koje su uključivale interakciju sa periferijskim jedinicama i obradu podataka. Na primer, pseudo kod programa koji detektuje da li je pritisnut taster (koji ubrzava/usporava motor), a onda generiše PWM izlaz za kontrolu motora korišćenjem PID regulatora i eventualno prikazuje trenutnu brzinu na displeju:

```
while(true)
{
        ocitaj_trenutnu_brzinu_obrtanja()
        if pritisnut_taster()
        {
                ažuriraj_parametre_regulatora()
        }
        izračunaj_novi_izlaz_za_pwm()
        podesi_kontroler_za_pwm()
        prikaži_brzinu_na_displeju()
}
```

U pozadini, kompajler bi preveo ovaj kod u niz instrukcija iz instrukcijskog seta datog mikrokontrolera, a prilikom programiranja mikrokontrolera, ovaj niz instrukcija bi se upisao na unapred poznatu lokaciju u memoriji, najčešće na adresu 0, odakle bi krenulo izvršavanje programa jer je to podrazumevana početna vrednost programskog brojača. Podaci i promenljive bi se nalazile u RAM memoriji (koje je vrlo često bilo ne više od nekoliko stotina bajta), gde bi se uglavnom nalazio i stek koji se koristi prilikom poziva procedura ili izvršavanja rutina za obradu prekida.

Ovakvo sekvencijalno izvršavanje instrukcija je uglavnom prihvatljivo, sve dok se ne pojavi potreba za paralelnim izvršavanjem određenih delova programa. Na primer, osim kontrole motora, ukoliko bi trebalo komunicirati sa drugim mikrokontrolerom ili personalnim računarom putem serijske komunikacije:

```
while(true)
{
	ocitaj_trenutnu_brzinu_obrtanja()
	if pritisnut_taster()
	{
		ažuriraj_parametre_regulatora()
	}
	izračunaj_novi_izlaz_za_pwm()
	podesi_kontroler_za_pwm()
	prikaži_brzinu_na_displeju()
	if (pristigla_poruka())
	{
		dekoduj_pristiglu_poruku()
		spremi_odgovor()
		odgovori_na_zahtev()
	}
}
```

Međutim, ovde se već pojavljuju potencijalni problemi: šta ako se blok za komunikaciju dugo izvršava, pa usled toga regulacija brzine „zakasni“? Takođe, šta se dešava ako se tokom računanja izlaza za PWM modul, propusti poruka pristigla putem serijske komunikacije? Šta ako osim ove dve aktivnosti, treba dodatno vršiti složene proračune koji zahtevaju dosta vremena. Prekidi rešavaju problem brze reakcije na neki događaj, ali usložnjavaju hardver, programski kod i organizaciju prioriteta događaja na koje treba reagovati. Očigledno bi nam bilo zgodno da više poslova možemo da obavljamo istovremeno, ali kako da to uradimo?

Vremenom, embeded sistemi su postajali sve kompleksniji i kompleksniji. Umesto CPU sa jednim jezgrom, pojavili su se više-jezgarni procesori koji sadrže skrivenu (*cache*) memoriju i koji mogu da rade na znatno višim učestanostima. Količina dostupne RAM memorije je postajala sve veća i veća, hardver sve složeniji i složeniji i vremenom smo došli u situaciju da se na embeded sistemima mogu izvršavati znatno kompleksniji programi. Kao rezultat, današnji embeded sistemi su sposobni da pokrenu kompletan operativni sistem, gotovo identičan onome koji se pre mogao pokrenuti samo na personalnim računarima. Time se spektar aplikacija koje se mogu izvršavati na takvim platformama vrtoglavo proširio. Odjednom imamo mogućnost istovremenog izvršavanja više programa, mogućnost komunikacije složenijim komunikacionim kanalima kao što je Ethernet/Internet, možemo mnogo jednostavnije da skladištimo podatke korišćenjem fajl sistema, interfejs ka periferijskim jedinicama je znatno jednostavniji, ne moramo da vodimo računa o registrima periferijske jedinice jer to za nas radi operativni sistem koji pojednostavljuje situaciju toliko da slanje podatka ka uređaju svede na upis u takozvanu „datoteku uređaja“ (o ovome ćemo mnogo detaljnije pričati u nastavku), a čitanje podatka sa uređaja na čitanje „datoteke uređaja“, potpuno identično kao što bismo čitali/upisivali tekstualnu datoteku. Operativni sistem kao softver koje upravlja resursima računara, omogućio nam je da i u embeded svetu konstruišemo pouzdan, prenosiv, efikasan i siguran računarski sistem.

Stoga, u nastavku ovoga predmeta izučavaćemo brojne tehnike i koncepte koji se koriste u operativnim sistemima, kako bi vam približili svu moć i snagu koju sa sobom oni nose, pod uslovom da ih pravilno koristite.

Koncepti operativnih sistema su među najkompleksnijim temama u okviru

računarskih nauka. Savremeni operativni sistem opšte namene sastoji se od preko 50 miliona linija koda. Operativni sistemi se, sa druge strane, kontinualno razvijaju, a novi nastaju. Ukoliko se ovaj tekst čita na čitaču e-knjiga, tabletu ili pametnom telefonu, operativni sistem je taj koji upravlja uređajem. Pošto nećemo moći da pokrijemo sve, naš fokus će biti na suštinskim konceptima operativnih sistema.

Sa druge strane dobra vest je da su koncepti operativnih sistema takođe među dostupnijim temama u okviru računarskih nauka. Većina problema koje svakodnevno srećemo ima svoj analog u teoriji operativnih sistemima, što će nam omogućiti da možemo objasniti kako operativni sistemi rade svoj posao u okviru jednog udžbenika. Sve što se od čitatelja zahteva, to je osnovno razumevanje rada računara i mogućnost čitanja pseudo-koda.

Razumevanje načina na koji rade operativni sistemi je definitivno od suštinskog značaja za svakog studenta zainteresovanog za rad sa savremenim računarskim sistemima. Naravno, svako ko koristi računar ili pametni telefon (ili čak moderan toster) koristi operativni sistem, pa je razumevanje funkcionisanja operativnog sistema korisno svima. Naš cilj u ovom predmetu je da idemo mnogo dublje od toga, da objasnimo tehnologije koje se koriste u operativnim sistemima, tehnologije na koje se mnogi od nas oslanjaju svakodnevno, a da toga nisu ni svesni najverovatnije.

Softverski inženjeri često se susreću sa izazovima sličnim onima koji se rešavaju u okviru operativnih sistema, prilikom izgradnji drugih složenih sistema i koriste često iste tehnologije i dizajn. Bilo da je vaš cilj rad na kernelu operativnog sistema ili na razvoju nove generacije softvera za *Cloud* računanje, bezbednim i pouzdanim veb pretraživačima, konzolama za igranje, grafičkim korisničkim interfejsima, medijskim plejerima ili bazama podataka, koncepti i apstrakcije potrebne za pouzdan, prenosiv, efikasan i siguran softver su uglavnom isti. Možda i najbolji način da se nauče ovi koncepti je da se prouči kako se oni koriste u operativnim sistemima, ali će se njihova primena svakako veoma brzo prepoznati u mnogo širem spektru računarskih sistema.

Za početak, razmatraćemo veb server. Njegovo ponašanje je zapanjujuće jednostavno: on dobija paket koji sadrži ime veb stranice putem mreže. Veb server dekoduje paket, preuzima datoteku sa diska i vraća sadržaj preko mreže korisniku. Deo posla operativnog sistema je da olakšava pisanje aplikacija poput veb servera. Ali, ako zagrebemo malo dublje i pažljivije analiziramo ovaj jednostavan koncept, brzo se postavlja mnogo novih pitanja:

- Mnogi veb zahtevi uključuju kako same podatke tako i zahtevno i kompleksno računanje. Na primer, Google stranica predstavlja jednostavan tekstualni okvir, ali svaki upit za pretragu unešen u taj okvir, sadrži pretraživanje baza podataka koje se prenosi bukvalno na hiljade mašina. Da bi njihov softver bio upravljiv, veb serveri često pozivaju pomoćne aplikacije, npr. za kontrolu same funkcije pretraživanja. Ove pomoćne aplikacije moraju da komuniciraju sa glavnim veb serverom da bi čitav ovaj postupak bio uspešno obavljen. Kako operativni sistem omogućava više aplikacija da komuniciraju jedne sa drugima?

- Šta se dešava ako dva korisnika (ili milion njih) pokušaju da zatraže veb stranicu sa servera istovremeno? Jednostavan pristup bi mogao biti obrađivanje svakog zahteva po redu pristizanja. Međutim, ako svaki pojedinačni zahtev traje duže vremena, ovaj pristup bi značio da bi svi ostali morali da sačekaju da se on završi. Brže, ali složenije rešenje je *multitasking* obrada: istovremeno obrađivanje više zahteva u istom vremenskom intervalu. *Multitasking* je posebno važan na savremenim računarima sa više jezgara, jer pruža način da se više procesora koristi istovremeno. Kako operativni sistem omogućava aplikacijama da rade više stvari istovremeno?

- Radi boljih performansi, veb server će često kreirati kopiju, koja se naziva keš, nedavno zatraženih stranica, kako bi sledeći korisnik koji zahteva istu stranicu mogao dobiti rezultat iz keša, umesto započinjanja zahteva od početka. Za ovo je potrebno da aplikacija sinhronizuje pristup keš podacima od strane hiljade veb zahteva koji istovremeno pristižu. Kako operativni sistem podržava sinhronizaciju aplikacija sa deljenim podacima?

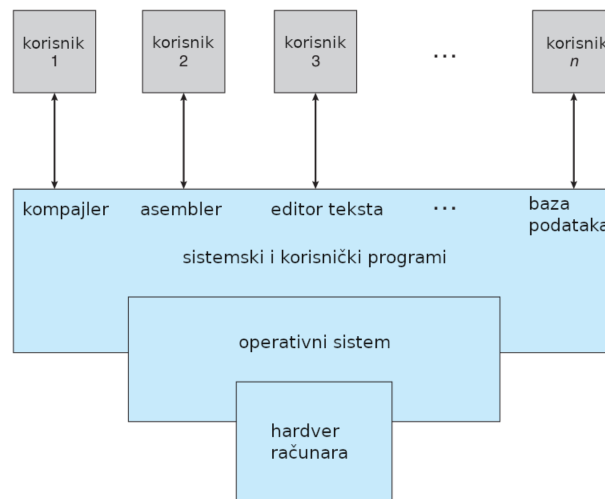
- Da biste prilagodili i obogatili korisničko iskustvo, uobičajeno je za veb servere da koriste klijentske skript programe, umetnute u sam sadržaj veb stranice. Ali to znači da klik na link može prouzrokovati pokretanje tuđeg koda na vašem računaru. Kako se klijentski operativni sistem štiti od toga da ga kompjuterski virus, eventualno umetnut u skript kod, na bilo koji način ugrozi?

- Pretpostavimo da administrator veb stranice koristi tekstualni editor za ažuriranje sadržaja veb stranice. Veb server tada mora biti u mogućnosti da pročita datoteku koju je administrator kreirao - kako operativni sistem čuva podatke na disku tako da veb server može da ih pronađe i pročita?

- Šta se događa kada veb pretraživač i veb server rade različitim brzinama? Ako server pokušava da pošalje veb stranicu klijentu brže nego što klijent može da prikaže stranicu, gde se nalazi sadržaj datoteke u međuvremenu? Može li operativni sistem potpuno razdvojiti klijenta i server tako da svaki može raditi svojom brzinom, a da se pritom drugi ne usporava?

- Kako saobraćaj ka veb serveru raste, administrator će verovatno želeti da pređe na bolji hardver, sa više memorije, više procesora, bržim mrežnim uređajima i bržim diskovima. Da biste iskoristili prednost ovog novog hardvera, da li je potrebno da veb server bude dizajniran ispočetka ili može biti napisan na hardverski-nezavisan način? Šta je sa operativnim sistemom - da li je njega potrebno razvijati od nule za svaki novi hardver koji se pojavi?

Mogli bismo u nedogled da nastavimo sa navođenjem ovakvih i sličnih problema, ali ideja je jasna: zadatak operativnog sistema jeste da ih razreši. U nastavku ćemo dati definiciju operativnog sistema i navesti izazove koji postoje u njihovom dizajniranju.



Slika 1: Komponente u računarskom sistemu

Operativni sistem je sloj softvera koji upravlja resursima računara za svoje korisnike i njihove aplikacije. Operativni sistemi rade u širokom rasponu računarskih sistema. Ponekad su krajnjem korisniku nevidljivi i kontrolišu embeded uređaje poput tostera, konzola za igre i mnogih računarskih sistema u okviru modernih automobila ili aviona. Operativni sistemi su takođe esencijalna komponenta sistema opšte namene kao što su pametni telefoni, desktop računari i serveri.

Kao što je prikazano na slici 1 računarski sistem se može grubo podeliti na četiri komponente: hardver, operativni sistem, aplikativni programi i korisnici. Hardver, koji čine centralna procesna jedinica (CPU), memorija i uređaji za ulaz/izlaz (U/I) - predstavljaju osnovne računarske resurse u uokviru sistema. Aplikativni programi - kao što su editori teksta, kompajleri, i veb pretraživači - definišu načine na koje se ovi resursi koriste za rešavanje računarskih problema korisnika. Operativni sistem kontroliše hardver i koordinira njegovu upotrebu putem različitih aplikativnih programa za brojne korisnike.

Sa računarske tačke gledišta, operativni sistem je program koji je najintimnije povezan sa hardverom. U tom kontekstu možemo videti operativni sistem kao *alokator resursa*. Računarski sistem ima mnogo resursa koji će biti korišćeni za rešavanje problema: centralnu procesorsku jedinicu, memoriju, prostor za skladištenje datoteka, U/I uređaje i tako dalje. Operativni sistem ponaša se kao menadžer ovih resursa. Suočavajući se sa brojnim i eventualno konfliktnim zahtevima za resursima, operativni sistem mora odlučiti kako ih dodeliti određenim programima i korisnicima kako bi računarski sistem funkcionisao efikasno i pošteno.

Sa druge strane, kod operativnog sistema naglašena je potreba za kontrolom različitih U/I uređaja i korisničkih programa. Operativni sistem je, u skladu

sa tim, *kontrolni program*. Kontrolni program kontroliše izvršavanje korisničkih programa radi sprečavanja grešaka i nepravilne upotrebe računara. Posebno se to odnosi na rad i kontrolu ulazno/izlaznih uređaja.

Operativni sistemi opšte namene i njihove tehnologije generalizacija su tehnologija potrebnih za embeded sisteme. Međutim, tehnologije opšte namene sve se češce razvijaju ka sferi embedded aplikacija. Na primer, rani mobilni telefoni imali su jednostavne operativne sisteme za upravljanje hardverom i za pokretanje nekoliko primitivnih aplikacija. Današnji pametni telefoni, sposobni za pokretanje eksternih aplikacija (*third party applications*), deo su najbrže rastućeg razvoja u svetu mobilne telefonije. Ovi novi uređaji zahtevaju mnogo složenije operativne sisteme, sa sofisticiranim upravljanjem resursima, dodatnim zadacima i zaštitom od otkaza.

U slučaju sistema opšte namene, korisnici komuniciraju sa aplikacijama, aplikacije se izvršavaju u okruženju koje obezbeđuje operativni sistem, a operativni sistem posreduje i kontroliše pristup osnovnom hardveru. Šta nam treba od operativnog sistema da bismo mogli pokrenuti grupu programa? Operativni sistemi imaju tri uloge:

1. Operativni sistemi igraju ulogu *sudije* - oni upravljaju zajedničkim resursima između različitih aplikacija koje rade na istoj fizičkoj mašini. Na primer, operativni sistem može zaustaviti jedan program i pokrenuti drugi. Operativni sistemi razdvajaju različite aplikacije jednu od druge, tako da, ako postoji greška u jednoj aplikaciji, ona ne ošteti druge aplikacije koje se pokreću na istoj mašini. Dodatno, operativni sistem mora zaštititi sebe i druge aplikacije od zlonamernih računarskih virusa. Na kraju, pošto aplikacije dele fizičke resurse, operativni sistem mora da odluči koje aplikacije dobijaju koji resurs.

2. Operativni sistemi igraju ulogu *iluzioniste* - oni pružaju apstrakciju fizičkog hardvera kako bi pojednostavili dizajn aplikacija. Da biste napisali "Hello world" program, ne trebate (i ne želite!) da razmišljate o tome koliko fizičke memorije ima sistem ili koliko drugih programa može da deli resurse vašeg računara. Umesto toga, operativni sistemi pružaju iluziju beskonačne memorije, kao apstrakciju ograničene količine fizičke memorije. Isto tako, operativni sistemi pružaju iluziju da svaki program ima na raspolaganju procesor računara alociran u potpunosti za njega. Očigledno je da je stvarnost sasvim drugačija! Ove iluzije omogućavaju da se aplikacije pišu nezavisno od količine fizičke memorije u sistemu ili fizičkog broja procesora. Pošto su aplikacije napisane na višem nivou apstrakcije, operativni sistem ima slobodu da menja resurse dodeljene svakoj pojedinačnoj aplikaciji, bilo u trenutku pokretanja ili zaustavljanja aplikacije.

3. Operativni sistemi se ponašaju kao *lepak* - posedujući skup zajedničkih servisa koji se dele između aplikacija. Korist postojanja zajedničkih servisa ogleda se u olakšavanju komunikacije i deljenja podataka između aplikacija. Tako, na primer, *cut* i *paste* deluje jednoliko u celom sistemu, a

datoteku koju kreira jedna aplikacija može da čita bilo koja druga. Dodatno, operativni sistemi obezbeđuju standardni korisnički interfejs kako bi aplikacijama pomogli da imaju uobičajeni izgled ("*look and feel*"). Možda najvažnije od svega jeste da operativni sistemi pružaju sloj koji razdvaja aplikacije od hardverskih ulazno-izlaznih uređaja, tako da se aplikacije mogu razvijati potpuno nezavisno od toga koja tastatura, miš ili hard disk se koristi u sistemu.

Prokomentarisaćemo svaku od tih uloga detaljnije u nastavku.

2 Uloge operativnog sistema

2.1 Deljenje resursa: Operativni sistem kao sudija

Deljenje resursa je centralni problem kod većine računara: u datom trenutku na jednom računaru može biti pokrenut veb pretraživač, uređivač teksta, klijent program za elektronsku poštu, i program za puštanje muzike. Operativni sistem mora podržavati sve ove aktivnosti odvojeno, ali dozvoliti svakoj puni kapacitet mašine ukoliko ostali ne rade. U najmanju ruku, kada se jedan program prestane izvršavati, operativni sistem bi trebao dozvoliti pokretanje drugog. Još bolje, operativni sistem bi trebao omogućiti istovremeno pokretanje više aplikacija, kao kad čitamo e-poštu dok preuzimamo sa interneta ažuriranja sistemskog softvera. Čak i pojedinačne aplikacije mogu biti dizajnirane da rade više stvari odjednom. Na primer, veb server će više odgovarati svojim korisnicima ako ima mogućnost da podnese više zahteva istovremeno, a ne da čekate da se svaki završi pre nego što počne sledeći. Isto važi i za veb pretraživač koji je korisniji i funkcionalniji ako može početi iscrtavati stranice dok se ostatak stranice još uvek prenosi i učitava. Na sistemima sa više procesora, računanje u okviru aplikacija koje podrazumevaju paralelno procesiranje se može podeliti na odvojene jedinice koje se mogu pokrenuti nezavisno, u cilju bržeg izvršenja. Sam operativni sistem je primer softvera u kome se može raditi više stvari odjednom. Kao što ćemo kasnije videti, sam operativni sistem je korisnik svojih apstrakcija.

Ipak, iako je ideja jednostavna, samo deljenje resursa vodi ka višestrukim izazovima kada dođe do implementacije.

1. Alokacija resursa: Operativni sistem mora da izvršava sve istovremene aktivnosti odvojeno, raspoređujući resurse po potrebi. Računar obično ima samo nekoliko procesora i ograničenu količinu memorije, kapacitet mreže i prostor na disku. Kada treba obaviti više zadataka u isto vreme, kako operativni sistem treba da odabere koliko resursa da dodeli svakom? Naizgled trivijalne razlike u načinu na koji se dodeljuju resursi mogu imati veliki uticaj na performanse koje percipiraju korisnici. Kao što ćemo videti kasnije, ako operativni sistem dodeli programu premalo operativne memorije, to neće samo usporiti taj određeni program, već može drastično usporiti rad čitave mašine. Kao dodatni primer, šta se dešava ako korisnički program ima beskonačnu petlju:

```
while (true){
        ;
}
```

Ako bi se programi izvršavali direktno na hardveru, ovaj fragment koda bi blokirao računar i on čak ne bi mogao ni da reaguje na unos od strane korisnika. Uz multipleksiranje resursa koje pruža operativni sistem, specifična aplikacija se može blokirati, ali ostali programi će se i dalje nesmetano izvršavati. Dodatno, korisnik može zatražiti od operativnog sistema da nasilno terminira program i na taj način izađe iz beskonačne petlje.

2. Izolacija: Greška u jednoj aplikaciji ne bi trebalo da poremeti druge aplikacije, pa čak i sam operativni sistem. To se naziva izolacija grešaka. Svako ko je ikada išta programirao zna kolika je vrednost operativnog sistema koji može da zaštiti sebe i druge aplikacije od programskih grešaka. Otklanjanje pogrešaka bilo bi znatno teže ako greška u jednom programu može pokvariti strukture podataka u drugim aplikacijama. Isto tako, preuzimanje i instaliranje jedne aplikacije, nikako ne bi smelo da poremeti programe koji ne zavise od nje, niti bi takva instalacija trebala omogućiti „propratnu“ instalaciju računarskog virusa u sistemu. Takođe, jedan korisnik ne bi nikako trebao da ima pristup ili da promeni podatke drugog bez eksplicitno date dozvole. Izolacija grešaka ograničava aplikacije na ograničen skup hardverskih resursa. Ukoliko bi bio omogućen potpuni pristup hardveru, bilo koja aplikacije preuzeta sa interneta ili bilo koja skripta umetnuta u veb stranicu, imala bi potpunu kontrolu nad mašinom. Tako bi mogao da se jednostavno pokrene virus koji bi, na primer, pratio i beležio svaki taster koji korisnik pritisne ili pamtio lozinke na svakoj veb lokaciji koju je korisnik posetio. Bez izolacije greške koju obezbeđuje operativni sistem, bilo kakva greška u bilo kojem programu može nepovratno oštetiti disk. Pogrešne ili zlonamerne aplikacije prouzrokovale bi totalni haos.

3. Komunikacija: Mana izolacije od gore je potreba za komunikacijom između različitih aplikacija i između različitih korisnika. Na primer, veb stranica se može implementirati putem istovremenog izvršavanja čitavog skupa aplikacija: jedna za odabir banera, druga za keširanje nedavno primljenih podataka, treća za preuzimanje podataka sa diska i upisivanje podataka na disk, četvrta za pristup podacima u bazi podataka, itd. Da bi ovo funkcionisalo, potrebno je omogućiti svim tim programima da komuniciraju jedni s drugima. Ako su operativni sistemi dizajnirani za sprečavanje grešaka i zlonamernih korisnika i aplikacija koji utiču na druge korisnike i njihove aplikacije, kako operativni sistem može podržati komunikaciju u cilju deljenja rezultata? Prilikom postavljanja granica, operativni sistem, dakle, mora omogućiti da se te granice mogu preći pažljivo, na kontrolisani način, kada se za to ukaže potreba.

U svojoj ulozi sudije, operativni sistem balansira potrebe, razdvaja konflikte i olakšava deljenje. Jedan korisnik ne bi smeo nikako biti u mogućnosti da prisvoji sve resurse sistema, da pristupi ili promeni datoteke drugih korisnika bez

dozvole. Greška u aplikaciji ne bi smela biti u stanju da sruši operativni sistem ili druge aplikacije nezavisne od nje. A, ipak, aplikacije moraju da rade zajedno. Forsiranje i uravnoteženje ovih zahteva je uloga operativnog sistema.

2.2 Maskiranje limitacija hardvera: Operativni sistem kao iluzionista

Druga važna uloga operativnih sistema je da prikrivaju postojeća ograničenja u računarskom hardveru. Hardver je nužno ograničen fizičkim ograničenjima - računar ima samo ograničen broj procesora i ograničenu količinu fizičke memorije, kapacitet mrežnog interfejsa i diska. Nadalje, budući da operativni sistem mora odlučiti kako podeliti fiksni skup resursa između različitih aplikacija koje se pokreću, određena aplikacija može, u zavisnosti od trenutka, imati različite količine dodeljenih resursa, čak i kada se pokreće na istom hardveru. Iako je značajan broj aplikacija dizajniran tako da iskoristi prednost specifične konfiguracije hardvera i dodeljivanja specifičnih resursa, većina programera želi da koristi viši nivo apstrakcije.

Kao što smo pomenuli ranije, računar sa jednim procesorom može pokrenuti samo jedan program u datom trenutku. Ipak, većina operativnih sistema omogućava iluziju za korisnika da se više aplikacija istovremeno izvršava, čak i na sistemima sa jednim procesorom. Operativni sistem to čini kroz koncept koji se zove *virtualizacija*. Virtualizacija pruža aplikaciji iluziju resursa koji fizički nisu prisutni. Na primer, operativni sistem može svakoj aplikaciji predstaviti apstrakciju čitavog procesora posvećenog isključivo njoj, iako na fizičkom nivou može biti samo jedan procesor koji se deli između svih aplikacija koje se pokreću na računaru. Sa odgovarajućom hardverskom i podrškom operativnog sistema, većina fizičkih resursa može se virtualizovati. Primeri uključuju procesor, memoriju, prostor na ekranu, disk i mrežu. Čak se i tip procesora može virtualizovati, kako bi se omogućilo pokretanje iste, neizmenjene aplikacije na pametnom telefonu, tabletu i laptop računaru.

Ako odemo korak dalje, svi savremeni operativni sistemi virtualizuju ceo računar, kako bi se pokrenuo operativni sistem kao aplikacija koja se pokreće u okvoru drugog operativnog sistema. Ovakav koncept se naziva *virtualna mašina*. Operativni sistem koji je pokrenut u okviru virtualne mašine i koji se često naziva *gostujući* operativni sistem (*guest operating system*), izvršava se potpuno identično kao što bi se izvršavao na stvarnoj, fizičkoj mašini, što je iluzija predstavljena od strane operativnog sistema na kojem je pokretnuta virtualna mašina. Jedan od razloga zašto operativni sistem obezbeđuje virtualnu mašinu je prenosivost aplikacija. Ako se program pokreće samo na staroj verziji operativnog sistema, virtualizacija nam omogućava da možemo pokrenuti program i na novom sistemu koji pokreće samu virtualnu mašinu. Virtualna mašina pokreće aplikaciju na starom operativnom sistemu, a pokreće je izvršavajući se na novom operativnom sistemu. Drugi razlog za virtualne mašine je pomoć u otklanjanju pogrešaka - debagovanje. Ako se operativni sistem može pokrenuti kao aplikacija, tada programeri operativnog sistema mogu postaviti tačke prekida (*breakpoints*), zaustaviti i korak po korak prolaziti kroz kod kao u slučaju deba-

govanja aplikacija. Vrlo često se virtualne mašine koriste u slučajevima kada se želi izbeći migracija softvera zahtevana prelaskom na novi operativni sistem, takođe. Ako se koristi virtualna mašina, bilo koja izmena postojećih biblioteka ili zamena sistemskih resursa u novom operativnom sistemu neće zahtevati izmenu aplikacije, obzirom da se ona i dalje izvršava na starom operativnom sistemu.

Osim virtualizacije, operativni sistemi maskiraju mnoga druga ograničenja svojstvena hardveru, pružajući aplikacijama iluziju hardverskih mogućnosti koje fizički nisu prisutne. Na primer, na računaru sa više procesora koji dele memoriju svaki procesor može da ažurira samo pojedinačnu memorijsku lokaciju u datom trenutku. Memorijski sistem u hardveru osigurava da je svako ažuriranje date memorijske reči *atomička operacija* (eng. *atomic*), odnosno da je vrednost sačuvana u memoriji poslednja vrednost koju je sačuvao jedan od procesora, a ne mešavina ažuriranja različitih procesora. Iako se ovo može činiti dovoljnim, aplikacije (i sam operativni sistem) moraju biti u mogućnosti da ažuriraju veće strukture podataka, one koje zauzimaju širi opseg memorijskih lokacija. Šta se dešava kada dva procesora pokušaju ažurirati istu strukturu podataka otprilike u isto vreme? Kao što ćemo kasnije videti, rezultati mogu biti prilično neočekivani i drastično različiti od onoga što bi se dogodilo da su procesori ažurirali strukturu podataka sukcesivno, jedan za drugim. U idealnom slučaju, programer bi želeo da apstrakuje atomičko ažuriranje na celokupnu strukturu podataka, a ne samo na jednu reč u okviru memorije. Iluziju atomičkog ažuriranja struktura podataka obezbeđuje operativni sistem koristeći neke specijalizovane mehanizme hardvera. O tome ćemo pričati mnogo detaljnije u narednim predavanjima.

Uređaji za trajno skladištenje podataka, poput magnetnog diska ili flash RAM-a, pružaju još jedan primer. Na fizičkom nivou, ovaj sistem podržava upis podataka u blokovima, pri čemu veličina bloka zavisi od fizičkih karakteristika uređaja. Ako se, iz bilo kog razloga, rad računara prekine usred upisa u blok, disk bi mogao da ostane u nedefinisanom stanju, u kojem na toj lokaciji nije sačuvana ni stara ni nova vrednost. Naravno, aplikacije moraju biti u mogućnosti da upisuju podatke na disk različite veličine, pri čemu sam upis može da obuhvata više blokova. Pri svemu tome, korisnici svakako podrazumevaju da će im podaci biti sačuvani, čak (ili posebno) ako dođe do kvara na mašini dok se disk ažurira. Razgovaraćemo o tehnikama koje operativni sistem koristi da bi postigao ove i druge iluzije. U svakom od ovih slučajeva, operativni sistem pruža pogodniju i fleksibilniju apstrakciju programerima od one koja je data od strane osnovnog hardvera.

2.3 Zajednički servisi: Operativni sistem kao lepak

Operativni sistem takođe igra i treću ulogu, pružajući skup uobičajenih, standardnih servisa aplikacijama kako bi se pojednostavio i standardizovao njihov dizajn. Primer toga videli smo sa veb serverom navedenim na početku ovog poglavlja. Operativni sistem krije specifičnosti načina rada mreže i uređaja za skladištenje podataka, pružajući jednostavniju apstrakciju aplikacijama na osnovu prijema i slanja korišćenjem pouzdanih tokova bajtova, kao i čitanja i pisanja podataka korišćenjem imenovanih datoteka. To omogućava da se veb server fo-

kusira na svoj osnovni zadatak dekodiranja dolaznih zahteva i ispunjavanja istih, umesto da troši vreme na formatiranje podataka u pojedinačne mrežne pakete ili u blokove koji će se koristiti prilikom upisa na disk.

Važan razlog zašto operativni sistem pruža zajedničke usluge, umesto da ih prepusti svakoj aplikaciji, jeste omogućavanje deljenja između aplikacija. Veb server mora biti u mogućnosti da pročita datoteku koju je kreirao i popunio tekst editor. Ako aplikacije žele da dele datoteke, one se moraju čuvati u standardnom formatu, u okviru standardnog fajl sistema i uz standardizovano upravljanje direktorijumima datoteka. Slično tome, većina operativnih sistema pruža standardni način da aplikacije prenose poruke i razmenjuju memoriju kako bi olakšali deljenje podataka.

Izbor servisa koje bi operativni sistem trebao da pruža često je pitanje procene. Na primer, računari mogu da se konfigurišu sa čitavim spektrom različitih uređaja: različiti grafički koprocesori i formati piksela, različiti mrežni interfejsi (WiFi, Ethernet i Bluetooth), različiti diskovi (SCSI, IDE), različiti interfejsi uređaja (USB, Firewire) i različiti senzori (GPS, akcelerometri), da ne spominjemo različite verzije svakog od tih standarda. Većina aplikacija moći će da ignoriše ove razlike koristeći samo generički interfejs koji pruža operativni sistem. Za ostale aplikacije, kao što je baza podataka, određeni tip disk uređaja može da ima izuzetan značaj. Za one aplikacije koje mogu raditi na višem nivou apstrakcije, operativni sistem služi kao sloj interoperabilnosti, obezbeđujući da se i aplikacije i sami uređaji mogu nezavisno razvijati bez potrebe za istovremenim promenama sa druge strane.

Druga standardna usluga u većini modernih operativnih sistema je biblioteka grafičkog korisničkog interfejsa. I Microsoft i Apple operativni sistemi pružaju set standardnih komponenti (*widget*) za korisnički interfejs. Ovo olakšava uobičajeni „izgled i osećaj“ za korisnike, tako da se česte operacije, kao što su „padajući“ meniji i „cut“ i „paste“, koriste na identičan način u različitim aplikacijama.

Većina koda operativnog sistema upravo ima ulogu da implementira ove zajedničke servise. Međutim, veliki deo složenosti operativnih sistema nastaje zbog potreba raspoređivanja resursa i maskiranja ograničenja hardvera. Budući da su zajednički servisi izgrađeni na apstrakcijama koje pružaju dve druge uloge operativnog sistema, u nastavku gradiva ćemo se uglavnom fokusirati na te dve teme.

3 Kriterijumi za poređenje operativnih sistema

Nakon što smo definisali šta radi operativni sistem, potrebno je prokomentarisati na koji način se može birati između alternativnih pristupa u izazovima dizajna operativnih sistema? U nastavku ćemo raspravljati o nekoliko kriterijuma za poređenje operativnih sistema. U mnogim slučajevima, kompromisi između ovih kriterijuma su neizbežni - poboljšanje sistema duž jedne dimenzije će dovesti do pogoršanja sistema u drugoj.

3.1 Pouzdanost

Možda i najvažnija karakteristika operativnog sistema je njegova pouzdanost. Pouzdanost podrazumeva da sistem radi upravo ono za šta je stvoren. Kao najniži nivo softvera koji radi na sistemu, greške u kodu operativnog sistema mogu imati pogubne i skrivene efekte. Ako se operativni sistem pokvari, korisnik uglavnom neće moći obaviti nikakav posao, a u nekim slučajevima može čak i izgubiti prethodno urađeni posao, na primer, ako novonastali kvar ošteti datoteke na disku. Nasuprot tome, kvarovi aplikacija mogu biti mnogo benigniji, upravo zato što operativni sistemi omogućavaju izolaciju grešaka i brzo i jednoznačno ponovno pokretanje nakon greške u aplikaciji.

Dizajnirati pouzdan operativni sistem je veliki izazov. Operativni sistemi često rade u neprijateljskom okruženju, gde računarski virusi i slični zlonamerni kodovi često pokušavaju da preuzmu kontrolu nad sistemom za sopstvene svrhe iskorišćavanjem grešaka u dizajnu ili implementaciji operativnog sistema.

Nažalost, tipične metode za poboljšanje pouzdanosti softvera, kao što su testiranje uobičajenih putanja koda, manje su efikasne kada se primenjuju na operativne sisteme. Budući da zlonamerni napadi najčešće ciljaju upravo izvršavanje retkih putanja koda, bukvalno sve mora ispravno raditi da bi operativni sistem bio pouzdan. Čak i bez zlonamernih napada koji namerno aktiviraju greške, mogu se javiti izuzetno retki slučajevi grešaka i bagova u kontekstu operativnog sistema (*corner cases*). Ako operativni sistem ima milion korisnika, događaj sa verovatnoćom jedan u milijardu će se, nažalost, dogoditi nekome od njih brže nego što je očekivano.

Srodni koncept pouzdanosti je *odzivnost*, procenat vremena u kojem je sistem upotrebljiv. Bagoviti operativni sistem koji se često ruši, što uzrokuje gubitkom rezultata rada korisnika, je istovremeno i nepouzdan je i neodzivan. Bagoviti operativni sistem koji se često ruši, ali nikada ne gubi rad korisnika i ne može ga upropastiti eventualnim virusima, bio bi pouzdan, ali neodzivan. Operativni sistem koji je nefunkcionalan, ali i dalje deluje normalno dok registruje korisničke pritiske tastera ili pomeraje miša, nepouzdan je, ali odzivan.

Stoga su jednako poželjna i pouzdanost i odzivnost. Na odzivnost utiču dva faktora:

- učestalost otkaza, koja se meri *srednjim vremenom do otkaza* (*Mean Time To Failure* - MTTF) i

- vreme potrebno za vraćanje sistema u funkcionalno stanje posle neuspeha (na primer, ponovno pokretanje), što se naziva *srednje vreme oporavka* (*Mean Time To Recover* - MTTR).

Odzivnost se može poboljšati povećanjem MTTF ili smanjenjem MTTR. U predavanjima koja slede, predstavićemo različite pristupe poboljšanju pouzdanosti i odzivnosti operativnog sistema. U mnogim slučajevima apstrakcije koje ćemo uvoditi mogu na prvi pogled izgledati prestrogo i preformalno. Međutim, važno je shvatiti da je to učinjeno namerno: samo precizne apstrakcije daju osnovu za konstrukciju pouzdanih i odzivnih sistema.

3.2 Sigurnost

Dva koncepta usko povezana sa pouzdanošću su *sigurnost* i *privatnost*. *Sigurnost* je svojstvo koje garantuje da napadač ne može ugroziti rad računara. *Privatnost* je deo sigurnosti - da su podaci sačuvani na računaru dostupni samo ovlašćenim korisnicima.

Nijedan upotrebljiv računar nije savršeno siguran! Bilo koji složeni deo softvera ima greške, a čak i inače neškodljivi bagovi mogu biti iskorišćeni od strane napadača da bi stekli kontrolu nad sistemom. Sa jedne strane, hardver računara može biti tako dizajniran da omogućava pristup potencijalnom napadaču, sa druge strane, možda je administrator računara nepouzdan, te koristeći privilegije omogućava krađu korisničkih podataka. Na kraju krajeva, programer operativnog sistema je možda namerno i svesno omogućio ulazne tačke kako bi napadač mogao pristupiti sistemu.

Ipak, operativni sistem može i treba da bude osmišljen tako da, koliko god je to moguće, minimizira svoju ranjivost na napade. Na primer, dobra izolacija grešaka može sprečiti eksterne aplikacije (*third party applications*) da preuzmu kontrolu nad sistemom. Ali čak i sa jakom izolacijom grešaka, sistem može biti nesiguran ako njegove aplikacije nisu dizajnirane imajući u vidu sigurnost. Na primer, Internet standard elektronske pošte ne omogućava pouzdan identitet pošiljaoca; moguće je formirati poruku elektronske pošte sa proizvoljnom adresom u polju „od“, koja ne odgovara adresi stvarnog pošiljaoca. Stoga se može pojaviti email od nekoga kome verujete, a u stvarnosti je od nekog drugog (i sadrži virus koji preuzima kontrolu nad računarom kada se otvori prilog). Ovaj problem bi mogao da se posmatra kao ograničenje interakcije između klijenta elektronske pošte i operativnog sistema - da je operativni sistem omogućio jeftin i jednostavan način da otvori prilog u izolovanom okruženju izvršenja sa ograničenim mogućnostima, problema garantovano ne bi bilo čak i ako prilog sadrži virus.

Komplikacija je svakako to što operativni sistem ne mora samo da sprečava neželjeni pristup deljenim podacima, već mora da omogući pristup u mnogim slučajevima. Želimo da korisnici i programi međusobno komuniciraju, da mogu da urade “cut” i “paste” teksta između različitih aplikacija i da čitaju ili upisuju podatke na disk ili putem mreže. Ako bi svaki program bio potpuno samostalan bez ikakve mogućnosti za interakciju s bilo kojim drugim programom, tada bi bila dovoljna samo izolacija grešaka. Međutim, ne samo da ne možemo da izolujemo programe jedne od drugih, već želimo da lako delimo podatke između programa i između korisnika.

Stoga su operativnom sistemu potrebni i *mehanizam podsticanja* (*enforcement mechanism*) i *sigurnosna politika* (*security policy*). *Mehanizam podsticanja* predstavlja način na koji operativni sistem omogućava izvršavanje samo dozvoljenih radnji. *Sigurnosna politika*, sa druge strane, definiše šta je dozvoljeno - kome je dozvoljen pristup podacima i ko može obavljati neke operacije.

Zlonamerni napadači mogu ciljati kako na ranjivosti u mehanizmima podsticanja, tako i u bezbednosnoj politici.

3.3 Prenosivost (Portabilnost)

Svi operativni sistemi pružaju apstrakciju osnovnog računarskog hardvera, a prenosiva apstrakcija je ona koja se ne menja u zavisnosti od promena hardvera. Program napisan za Microsoft-ov Windows 10 trebao bi ispravno da radi bez obzira da li se koristi određena grafička kartica, da li je obezbeđeno trajno skladištenje podataka putem SSD ili magnetnog diska ili da li je mreža Bluetooth, WiFi ili gigabitni Ethernet.

Prenosivost se odnosi i na sam operativni sistem. Operativni sistemi spadaju u najsloženije softverske sisteme ikada izumljene, tako da je nepraktično razvijati ih ispočetka svaki put kada se proizvede novi hardver ili svaki put kada se razvije nova aplikacija. Umesto toga, novi operativni sistemi često nasleđuju, bar delimično, od starih. Kao jedan primer, iOS, operativni sistem za iPhone i iPad, nasleđen je od OS-X, Apple Macintosh operativnog sistema. Kao rezultat toga, najuspešniji operativni sistemi imaju vek meren decenijama: početna primena Microsoft Windows 8 počela je razvojem Windows NT-a počev od 1990. godine, kada je tipični računar bio više od 10000 puta manje moćan i imao je 10000 puta manje memorije i prostora na disku, nego što je to danas slučaj. Operativni sistemi koji traju decenijama nisu anomalija: Microsoftov raniji operativni sistem, MS / DOS, prvi put je predstavljen 1981. Kasnije je evoluirao u ranim verzijama Microsoft Windows-a, a konačno je ukinut oko 2000.

Sve ovo gore navedeno znači da operativni sistemi moraju biti dizajnirani tako da podržavaju aplikacije koje još nisu napisane i da rade na hardveru koji tek treba razviti. Isto tako, ne želimo da razvijamo od nule aplikacije, samo zato što se operativni sistem, na kome se one izvršavaju, preneo sa jedne mašine na drugu.

Kako dizajnirati operativni sistem u cilju postizanja prenosivosti? Za prenosivost, izuzetno je značajno da postoji jednostavan, standardizovan način interakcije aplikacija s operativnim sistemom, putem tzv *apstraktnog mašinskog intrerfejsa* (*Abstract Machine Interface - AMI*). Apstraktni mašinski interfejs (AMI) je interfejs koji operativni sistemi pružaju aplikacijama. Ključni deo AMI je *interfejs za programiranje aplikacija* (*Application Programming Interface - API*), lista funkcija koje operativni sistem pruža aplikacijama. AMI takođe uključuje model pristupa memoriji i koje instrukcije se mogu legalno izvršiti. Na primer, instrukcija za promenu moda rada (kanije će biti više reči o tome), koja definiše da li procesor izvršava pouzdan kod u okviru kernela operativnog sistema ili nepouzdan kod iz korisničke aplikacije mora biti dostupna operativnom sistemu, ali ne i aplikacijama.

Dobro dizajniran AMI operativnog sistema omogućava fiksnu tačku preko koje i kod aplikacije i hardver mogu evoluirati nezavisno. Takav koncept je sličan ulozi Internet Protocol (IP) standarda u umrežavanju - distribuirane aplikacije kao što su elektronska pošta i web, razvijane korišćenjem IP-a, izolovane su od promena u osnovnoj mrežnoj tehnologiji (Ethernet, WiFi, optička).

Jednako je važno da promene u aplikacijama ne zahtevaju istovremeno promene osnovnog hardvera.

Ovaj pojam prenosne hardverske apstrakcije je toliko moćan da operativni

sistemi koriste istu ideju za svoju implementaciju. Ovo omogućava da se sam operativni sistem može u velikoj meri implementirati nezavisno od specifičnosti hardvera. Ovaj interfejs naziva se *sloj hardverske apstrakcije* (*Hardware Abstraction Layer - HAL*). Na prvi pogled može izgledati da bi AMI i HAL operativnog sistema trebalo da budu identični - na kraju krajeva oba su prenosivi slojevi dizajnirani da sakriju nevažne detalje o hardveru. AMI je ipak znatno složeniji - kao što smo napomenuli, aplikacije se izvršavaju u ograničenom, virtualizovanom kontekstu uz pristup standardizovanim servisima visokog nivoa, dok se sam operativni sistem implementira pomoću apstrakcije koja je mnogo bliža stvarnom hardveru.

Linux je primer visoko prenosivog operativnog sistema. Linux se koristi kao operativni sistem za web servere, personalne računare, tablete, netbook-ove, čitače e-knjiga, pametne telefone, set-top-boksove, rutere, WiFi pristupne tačke i konzole za igre. Linux je zasnovan na operativnom sistemu zvanom UNIX, koji je prvobitno razvijen početkom 1970-ih. UNIX je razvio mali tim programera, te pošto nisu mogli da priušte razvoj obimnog koda, dizajniran je tako da bude veoma mali, jednostavan za programiranje i veoma prenosiv, uz određene cene u pogledu performansi. Tokom godina, prenosivost UNIX-a i Linux-a, kao i pogodne apstrakcije za programiranje bili su ključ za njihov uspeh.

3.4 Performanse

Za razliku od prenosivosti operativnog sistema, koja se može posmatrati samo u širem vremenskom okviru, performanse operativnog sistema su često odmah vidljive njegovim korisnicima.

Iako često povezujemo performanse sa svakom pojedinačnom aplikacijom, dizajn operativnog sistema može imati veliki uticaj na percipirane performanse aplikacije jer operativni sistem odlučuje kada se aplikacija može pokrenuti, koliko memorije može da koristi i da li se datoteke keširaju u memoriju ili se skladište na disku. Operativni sistem takođe posreduje prilikom pristupa aplikacije memoriji, mreži i disku. Performanse se ne mogu meriti jednoznačno, već isključivo korišćenjem višestrukih metrika. Jedna metrika performansi je efikasnost apstrakcije koja je predstavljena aplikacijama. Efikasnost je uvek u uskoj vezi sa dodatnim troškom resursa za sprovođenje same apstrakcije (eng *Overhead*). Jedan od načina za merenje efikasnosti je stepen do koga apstrakcija ometa performanse aplikacije. Pretpostavimo da je aplikacija dizajnirana da radi direktno na osnovnom hardveru, bez dodatnih troškova apstrakcije operativnog sistema; koliko bi to poboljšalo performanse aplikacije?

Operativni sistemi takođe moraju da raspoređuju resurse između aplikacija, a to može uticati na performanse sistema onako kako ih opaža krajnji korisnik. Jedno je pitanje pravičnost (*fairness*), između različitih korisnika iste mašine ili između različitih aplikacija koje se pokreću na toj mašini. Treba li resurse podeliti podjednako između različitih korisnika ili različitih aplikacija ili bi neki trebali imati povlašćen tretman? Ako je tako, kako operativni sistem odlučuje koji zadaci dobijaju prioritet?

Dva povezana koncepta su *vreme odziva* (*response time*) i *propusnost* (*throughput*). Vreme odziva, koje se ponekad naziva *kašnjenje*, pokazuje koliko traje određeni zadatak, od trenutka kada započinje do njegovog završetka. Na primer, visoko vidljivo vreme odziva za desktop računare je vreme od kada korisnik pomera hardverski miš, pa sve dok pokazivač na ekranu ne odreaguje na korisnikovu radnju. Operativni sistem koji pruža loše vreme odziva može biti neupotrebljiv. *Propusnost* je brzina kojom se može obavljati grupa zadataka. Propusnost je merilo efikasnosti za grupu zadataka a ne jedan pojedinačni. Iako se može činiti da će dizajni koji poboljšavaju vreme odziva takođe poboljšati propusnost, to nije slučaj, o čemu ćemo kasnije govoriti.

Predvidljivost performansi, pokazuje koliko su performanse sistema vremenom konzistentne, bez obzira da li je vreme odziva sistema ili neka druga metrika korišćena. Predvidljivost često može biti važnije od prosečnih performansi. Razmotrimo, na primer, dva sistema: u jednom su korisnikovi pritisci na tastere gotovo uvek trenutni, ali 1% vremena pritisak tastera zahteva 10 sekundi kašnjenja. U drugom sistemu korisnikovim pritiscima tastera uvek treba 0,1 sekunda da se odraze na ekranu. Prosečno vreme odziva može biti isto u oba sistema, ali drugi je predvidljiviji i samim tim korisniji (više *user friendly*).

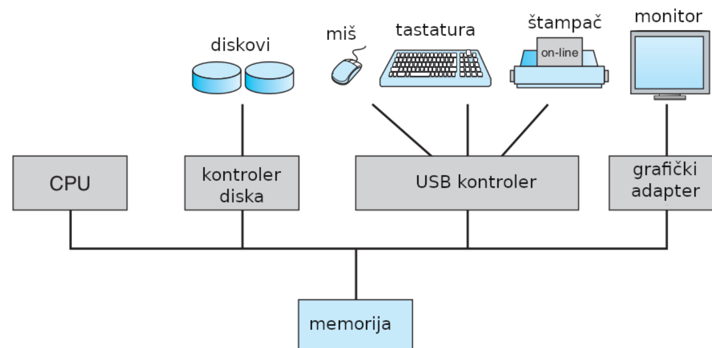
3.5 Dostupnost

Pored pouzdanosti, prenosivosti i performansi, uspeh operativnog sistema zavisi od dva faktora koji su van njegove neposredne kontrole: (široka) dostupnost aplikacija prenosivih u taj operativni sistem i (široka) dostupnost hardvera koji operativni sistem može da podrži. IPhone pokreće iOS, ali bez unapred instaliranih aplikacija i sadržaja App Store-a, iPhone bi bio samo mobilni telefon sa verovatno lošim prijemom kod korisnika.

Neto efekat nastaje kada vrednost neke tehnologije ne zavisi samo od njenih sopstvenih mogućnosti, već i od broja drugih ljudi koji su usvojili tu tehnologiju. Dizajneri aplikacija i hardvera troše svoje napore na platformama operativnog sistema s najviše korisnika, dok korisnici favorizuju one operativne sisteme sa najboljim aplikacijama ili najjeftinijim hardverom. Ako ovo zvuči kao kružni efekat, to i jeste tako: više korisnika podrazumeva više aplikacija i jeftiniji hardver; više aplikacija i jeftiniji hardver podrazumeva više korisnika u virtualnom ciklusu.

Izazov onda očigledno postaje dizajniranje operativnog sistema tako da iskoristi neto efekat ili bar da bude imun na njega. Očigledan korak u tom smeru bi bio dizajniranje sistema tako da se olakša dodatak novog hardvera i olakša prenos aplikacija u različitim verzijama istog operativnog sistema.

Suptilnije je pitanje izbora da li je interfejs za programiranje (API) operativnog sistema ili sam izvorni kod operativnog sistema *otvoren* (*open code*) ili *zatvoren* (*proprietary*). Softver zatvorenog koda je pod kontrolom jedne kompanije, pa ga vlasnik može u bilo kom trenutku promeniti kako bi zadovoljio potrebe svojih kupaca. Otvoreni sistem je onaj gde je izvorni kod sistema javni, omogućavajući svima da ga pregledaju i promene. Često će otvoreni sistem imati API koji se može menjati samo uz saglasnost tela za javne standarde. Pridržava-



Slika 2: Struktura računarskog sistema

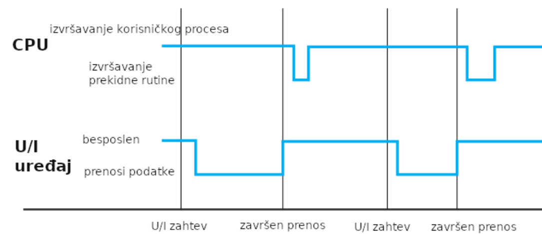
nje standarda pruža sigurnost programeru da se API neće menjati, osim opštim sporazumom; s druge strane, organi za standarde mogu otežati brzo dodavanje novih, željenih funkcija. Ni otvoreni ni zatvoreni sistemi nisu očigledno bolji za široko usvajanje. Windows 10 i MacOS su primeri zatvorenih operativnih sistema, Linux je primer otvorenog operativnog sistema i sva tri se široko koriste! Otvoreni sistemi se lakše prilagođavaju širokom rasponu hardverskih platformi, ali rizikuju fragmentaciju, što utiče na neto efekat. Tvorci zatvorenih operativnih sistema tvrde da su njihovi sistemi pouzdaniji i bolje prilagođeni potrebama kupaca. Problemi interoperabilnosti i kompatibilnosti su smanjeni ako i hardver i softver kontroliše ista kompanija, ali ograničenje operativnog sistema na jednu hardversku platformu umanjuje neto efekat.

Olakšavanje prenošenja aplikacija sa postojećih na novi operativni sistem može pomoći novom sistemu da se uspostavi, i obrnuto, dizajniranje API-ja operativnog sistema da otežava prenos aplikacija van operativnog sistema može sprečiti njegovo širenje i konkurentnost. Stoga često postoje komercijalni pritisci koji sprečavaju da interfejs operativnog sistema postane idiosinkratski. Iako ćemo u nastavku raspravljati o problemima operativnih sistema na konceptualnom nivou, važno je shvatiti da će sami detalji poprilično varirati za svaki određeni operativni sistem, zbog važnih, ali i pomalo haotičnih komercijalnih interesa.

4 Operativni sistemi sa stanovišta organizacije računarskog sistema

4.1 Upravljanje računarskim sistemom

Savremeni računarski sistem opšte namene sastoji se od jednog ili više procesora i brojnih kontrolera uređaja povezanih zajedničkom magistralom koja omogućava pristup deljenoj memoriji (slika 2). Svaki kontroler uređaja je za-



Slika 3: Vremenski dijagram kombinovanog izvršavanja korisničkog procesa i prekidne rutine

dužen za određenu vrstu uređaja (na primer, diskovi, audio uređaji ili video displeji). CPU i kontroleri uređaja rade paralelno, nadmećući se za memorijske cikluse, obzirom na činjenicu da je memorija deljena.

Da bi računar mogao da se pokrene (uključi ili nakon reseta), potrebno je da se pokrene inicijalni program. Ovaj inicijalni program, ili program za pokretanje sistema (*bootstrap program*), obično je jednostavan. Uglavnom se čuva u računarskom hardveru u memoriji isključivo za čitanje (ROM) ili u električno izbrisivoj programibilnoj memoriji za čitanje (EEPROM). On inicijalizuje sve aspekte sistema, od registara CPU-a, preko kontrolera uređaja do memorijskog sadržaja. Program za pokretanje sistema mora znati kako učitati operativni sistem i kako započeti njegovo izvršavanje. Da bi postigao ovaj cilj, program za pokretanje sistema mora locirati jezgro (*kernel*) operativnog sistema i učitati ga u memoriju.

Kada se kernel učita i krene izvršavati, on može početi da pruža servise sistemu i svojim korisnicima. Neki servisi su obezbeđeni izvan kernela, od strane sistemskih programa koji se učitavaju u memoriju tokom pokretanja sistema, a zovu se *sistemski procesi* (*system daemons*). Sistemski procesi se izvršavaju svo vreme dok se izvršava kernel operativnog sistema.

Na Unix-u (i sistemima nasleđenim od njega), prvi sistemski proces se naziva "init", a pokreće mnoge druge sistemske „daemon“ procese. Kada se ova faza završi, sistem je potpuno pokrenut i čeka da se dogodi neki događaj.

Pojava događaja obično se signalizira *prekidom* bilo od strane hardvera ili softvera. Hardver može u bilo kom trenutku izazvati prekid slanjem signala ka CPU. Softver može izazvati prekid izvršavanjem posebne operacije koja se zove *sistemski poziv* (*system call*, *monitor call*).

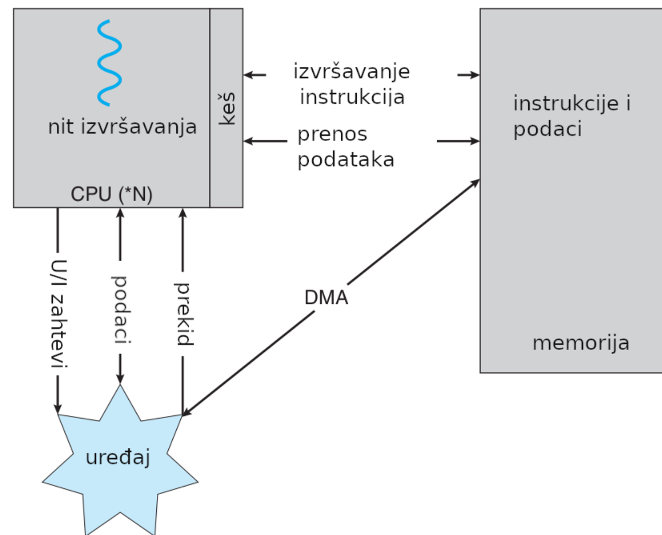
Kad se CPU prekine na ovaj način, on zaustavlja ono što radi i odmah prelazi na izvršavanje koda sa fiksne lokacije. Fiksna lokacija obično sadrži početnu adresu na kojoj se nalazi prekidna rutina. Prekidna rutina se izvršava, a po završetku, CPU nastavlja izvršavanje na mestu gde je stao kada je bio prekinut. Vremenski dijagram ove operacije prikazan je na slici 3. Prekidi su izuzetno važan deo računarske arhitekture. Svaki dizajn računara ima svoj mehanizam prekida, ali nekoliko funkcija je zajedničko. Prekid mora preneti kontrolu odgovarajućoj

rutini za obradu prekida. Najjednostavniji metod bi bio poziv generičke rutinu za ispitivanje informacija o prekidu. Ona bi, zauzvrat, pozvala rutinu specifičnu za dati prekid. Međutim, prekidi se moraju rešiti brzo, tako da se ovakav pristup ne koristi. Budući da je moguć samo unapred definisani broj prekida, može se koristiti tabela pokazatelja na prekidne rutine, u cilju znatno bržeg rada. Prekidna rutina se poziva indirektno kroz tabelu, bez posredne rutine. Tabela pokazatelja se obično čuva na najnižim adresama u memoriji (prvih par stotina lokacija). Ove lokacije sadrže adrese prekidnih rutina za različite uređaje. Ovaj niz ili tzv. *vektor prekida* se zatim indeksira jedinstvenim brojem uređaja, datim zahtevom za prekid, radi pružanja adrese početka prekidne rutine u cilju obrade prekida za uređaj koji je izazvao prekid. Različiti operativni sistemi kao što su Windows i Linux, obrađuju prekide na gore pomenuti način.

Sistem prekida mora takođe da sačuva adresu prekinute instrukcije. Mnogi stariji dizajni jednostavno su smeštali adresu prekida na fiksnu lokaciju ili na lokaciju koja je indeksirana brojem uređaja. Novije arhitekture čuvaju povratnu adresu u okviru sistemskog steka. Ako rutina za obradu prekida treba da promeni stanje procesora - na primer, promenom vrednosti registara - ona mora sačuvati trenutno stanje, a zatim ga rekonstruisati pre nego što se prekidna rutina završi. Nakon servisiranja prekida, sačuvana povratna adresa se učitava u programski brojač, a prekinuto izvršavanje nastavlja se kao da do prekida nije ni došlo.

4.2 Struktura ulazno/izlaznog (U/I) podsistema

Veliki deo koda operativnog sistema namenjen je upravljanju ulazno/izlaznim sistemom, kako zbog njegove važnosti za pouzdanost i performanse sistema, tako i zbog različite prirode uređaja. Računarski sistem opšte namene sastoji se od procesora i više kontrolera uređaja koji su povezani preko zajedničke magistrale. Svaki kontroler uređaja je zadužen za određenu vrstu uređaja. Zavisno od kontrolera, može biti priključeno više uređaja. Na primer, sedam ili više uređaja može se priključiti na mali kontroler interfejsa računarskog sistema (SCSI). Kontroler uređaja sadrži lokalni bafer i skup registara posebne namene, kojima se upravlja njime. Kontroler uređaja je odgovoran za premeštanje podataka između perifernih uređaja koje kontroliše i lokalnog bafera. Obično operativni sistemi imaju upravljački program uređaja (*device driver*) za svaki kontroler uređaja. Ovaj upravljački program razume kontroler uređaja i pruža operativnom sistemu jednoličan interfejs ka tom uređaju. Da bi pokrenuo U/I operaciju, drajver uređaja inicijalizuje odgovarajuće registre unutar kontrolera uređaja. Kontrolor uređaja, zauzvrat, ispituje sadržaj ovih registara kako bi utvrdio šta treba preduzeti (poput „čitanja znaka sa tastature“). Kontroler zatim započinje prenos podataka sa uređaja u njegov lokalni bafer. Nakon što je prenos podataka završen, kontroler uređaja putem prekida obaveštava upravljački program (drajver) uređaja da je završio svoj rad. Drajver uređaja potom vraća kontrolu operativnom sistemu, uglavnom obezbeđujući podatke ili pokazivač na podatke ako je bilo zahtevano čitanje podataka iz uređaja. Za ostale operacije, drajver uređaja vraća informacije o statusu.



Slika 4: Prenos podataka kod modernih računarskih sistema

Ovaj oblik U/I koji se bazira na prekidu je prihvatljiv za premeštanje malih količina podataka, ali može biti izuzetno neefikasan ako se koristi za prenos veliki količine podataka, kao što je slučaj kod upisa/čitanja sa diska. Da bi se ovaj problem rešio efikasno, koristi se *direktan pristup memoriji* (*Direct Memory Access* - DMA). Nakon podešavanja bafera, pokazivača i brojača za U/I uređaj, kontroler uređaja prenosi čitav blok podataka direktno iz (ili u) internog bafera u memoriju, bez posredovanja CPU-a. Samo jedan prekid se u ovom slučaju generiše za prenos celog bloka, kako bi se obavestio upravljački program (drajver) uređaja da je operacija završena, nasuprot prekidu za svaki bajt za uređaje male brzine. Dok kontroler uređaja izvodi ove operacije, CPU je dostupan za obavljanje drugih poslova (slika 4).

4.3 Struktura operativnog sistema

Nakon predstavljanja osnovne organizacije i arhitekture računarskog sistema, spremni smo da razgovaramo o operativnim sistemima. Operativni sistem pruža okruženje u kojem se programi izvršavaju. Interno se operativni sistemi uveliko razlikuju po svojoj strukturi, jer su organizovani sa različitim ciljevima dizajna i optimizacije. Međutim, postoje mnoge zajedničke stvari koje ćemo razmatrati u ovom odeljku.

Jedan od najvažnijih aspekata operativnih sistema je mogućnost multi-programiranja. Jedan program ne može za stalno da zauzme ni CPU ni U/I uređaje. Takođe, pojedinačni korisnici često istovremeno koriste više programa. Multi-programiranje povećava iskorišćenost CPU-a organizovanjem zadataka (koda i podataka) tako da CPU uvek ima jedan za izvršenje.

Ideja je sledeća: Operativni sistem istovremeno čuva nekoliko zadataka u memoriji, a pošto je, generalno, glavna memorija premala za prihvat svih tih zadataka, oni se u početku čuvaju na disku u okviru *grupe zadataka*. Ova grupa se sastoji od svih procesa koji ostaju na disku dok čekaju dodelu glavne memorije. Skup zadataka u memoriji može biti samo podskup zadataka koji se čuvaju u grupi zadataka. Operativni sistem bira i započinje izvršavanje jednog od zadataka u memoriji. Tokom izvršavanja tog zadatka, on će eventualno morati da sačeka da se završi neki drugi ili da se desi neki događaj (kao što je U/I operacija). U ne multi-programskom sistemu, CPU bi bio besposlen u tom periodu. U multi-programskom sistemu operativni sistem se jednostavno prebacuje na drugi zadatak i izvršava ga. Kada taj zadatak treba da sačeka (opet neki događaj ili drugi zadatak), CPU prelazi na treći i tako dalje. Vremenom, prvi zadatak završava svoje čekanje i vraća mu se CPU na raspolaganje. Sve dok barem jedan zadatak treba da se izvrši, CPU nikada neće biti besposlen.

Multi-programirani sistemi pružaju okruženje u kojem se različiti resursi sistema (na primer, CPU, memorija i periferni uređaji) efikasno koriste, ali ne obezbeđuju interakciju korisnika sa računarskim sistemom. Deljenje vremena (*time-sharing*, *multitasking*) je logična nadogradnja multi-programskog sistema. U multitasking sistemima, CPU izvršava više zadataka zamenjujući jednim druge, ali se zamene događaju toliko često da korisnici mogu da komuniciraju sa svakim programom dok je on aktivan. Za multitasking je potreban interaktivni računarski sistem koji omogućava direktnu komunikaciju između korisnika i sistema. Korisnik interaguje sa operativnim sistemom ili programom direktno, koristeći uređaj za unos kao što je tastatura, miš, ili ekran osetljiv na dodir i čeka trenutne rezultate na izlaznom uređaju. Prema tome, vreme odziva trebalo bi da bude kratko - obično kraće od jedne sekunde, često i kraće.

Multitasking operativni sistem omogućava većem broju korisnika istovremeno deljenje računara. Pošto je svaka naredba ili instrukcija u sistemu koji podržava deljenje vremena obično kratka, potrebno je veoma malo CPU vremena za svakog korisnika. Kako sistem brzo prelazi sa jednog na drugog korisnika, svakom se korisniku pruža utisak da je ceo računarski sistem posvećen njegovoj upotrebi, iako se deli između višestrukih korisnika.

Multitasking operativni sistem koristi dodeljivanje CPU-a i multi-programiranje kako bi svakom korisniku pružili mali deo računara. Svaki korisnik ima najmanje jedan program u memoriji. Program učitan u memoriju i određen za izvršavanje naziva se ***proces***. Kada se proces izvršava, on se obično izvršava samo kratko vreme pre nego što završi ili mora da zatraži interakciju U/I jedinicom. Ulaz/izlaz može biti interaktivni; to jest, izlaz ide na ekran za korisnika, a ulaz dolazi od korisničke tastature, miša ili drugog uređaja. Budući da interaktivni U/I obično radi "brzinom čoveka", potrebno je dugo vremena da se dovrši. Na primer, unos može biti ograničen brzinom unosa korisnika; sedam znakova u sekundi je brzo za ljude, ali neverovatno sporo za računare. Umesto da dozvoli da CPU miruje u toku rada ovog interaktivnog ulaza, operativni sistem će brzo prebaciti CPU u posed procesa nekog drugog korisnika.

Deljenje vremena i multi-programiranje zahteva da se nekoliko zadataka održava istovremeno u memoriji. Ako je više zadataka spremno za upis u memoriju,

a u slučaju da nema dovoljno prostora za sve, sistem mora da bira između njih. U tom cilju, sistem vrši raspoređivanje zadataka (*job scheduling*). Kada operativni sistem izabere zadatak iz grupe zadataka sa diska, on učitava taj zadatak u memoriju radi izvršavanja. Da bismo smestili više programa u memoriju istovremeno, potrebno je implementirati neki oblik upravljanja memorijom, ali to je zaseban problem o kojem ćemo takođe detaljno pričati. Kada su jednom odabrani zadaci koji će se upisati u memoriju, ako je nekoliko njih spremno istovremeno da se izvršava, sistem mora odabrati koji će se zadatak prvi izvršavati. Donošenje ove odluke je vezano za raspoređivanje CPU-a (*CPU scheduling*) o kome ćemo takođe opširnije govoriti kasnije. Na kraju, izvršavanje više zadataka istovremeno zahteva da njihova sposobnost da utiču jedni na druge bude ograničena u svim segmentima operativnog sistema, uključujući raspoređivanje procesa, korišćenje diska i upravljanje memorijom.

U sistemu za deljenje vremena, operativni sistem mora da obezbedi razumno vreme odziva. Ovaj cilj se ponekad postiže *zamenom* (*swapping*), pri čemu se procesi prebacuju u glavnu memoriju sa diska i obrnuto. Češća metoda za obezbeđivanje razumnog vremena odziva je virtualna memorija, tehnika koja omogućava izvršenje procesa koji nije u potpunosti u memoriji i o ovome ćemo detaljno pričati u jednom od završnih predavanja. Glavna prednost virtualne memorije je u tome što omogućava korisnicima da pokreću programe koji su čak i veći od stvarne fizičke memorije. Dodatno, mehanizam virtualizacije memorije abstrakuje glavnu memoriju u veliki, uniformni niz lokacija, odvajajući *logičku memoriju*, kako je korisnik vidi, od fizičke memorije. Ovaj aranžman oslobađa programere od brige zbog ograničenja memorije.

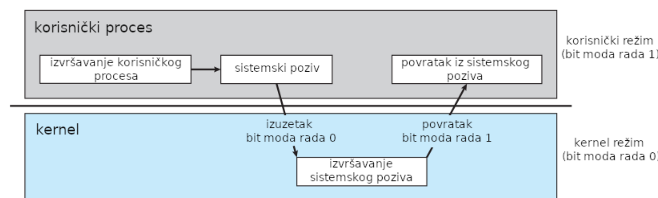
Multitasking sistem, takođe, mora da obezbedi sistem datoteka (*file system*). Sistem datoteka se nalazi na disku (ili više njih); stoga se mora obezbediti upravljanje diskom od strane operativnog sistema.

Za kraj, neophodan je mehanizam za zaštitu resursa od neprimerene upotrebe, a operativni sistem mora obezbediti i mehanizme za sinhronizaciju zadataka i njihovu međusobnu komunikaciju. O svemu ovome ćemo detaljnije pričati na predavanjima koja dolaze.

4.4 Funkcionisanje operativnog sistema

Kao što je ranije spomenuto, savremeni operativni sistemi su bazirani na prekidima. Ako nema procesa koji se izvršavaju, nema U/I uređaja koji bi se servisirali i nema korisnika sa kojima bi bilo interakcije, operativni sistem će biti besposlen i čekati da se nešto dogodi. Događaji su gotovo uvek signalizirani pojavom prekida ili izuzetka. Izuzetak (*trap*) je softverski generisan prekid izazvan ili greškom (na primer, deljenjem sa nulom ili pristupom nedozvoljenoj memorijskoj lokaciji) ili određenim zahtevom korisničkog programa da se izvrši usluga od strane operativnog sistema.

Budući da operativni sistem i korisnici dele hardverske i softverske resurse računarskog sistema, moramo biti sigurni da će eventualna greška u datom korisničkom programu stvoriti probleme samo za taj konkretan program koji se izvršava. Uz gore pomenuto deljenje resursa, bag u jednom programu bi mogao



Slika 5: Tranzicija između korisničkog i kernel moda

da ima negativan uticaj na ostale procese. Na primer, ako se proces zaglavi u beskonačnoj petlji, ova petlja može da spreči ispravan rad mnogih drugih procesa. Suptilnije greške mogu se pojaviti u sistemu s više programa, pri čemu greška u jednom programu može modifikovati drugi program, podatke drugog programa ili čak i sam operativni sistem.

Bez zaštite od ove vrste grešaka, računar bi morao ili da izvršava samo jedan program u datom trenutku ili je potrebno da svi rezultati i sve operacije budu preispitivane i proveravane. Pošto prvi pristup uopšte nije opcija, zbog svega gore navedenog, pravilno dizajniran operativni sistem mora osigurati da pogrešan (ili zlonamerni) program ne može nikako uzrokovati greške u izvršavanju drugih programa.

Dualni mod izvršavanja upravo obezbeđuje ovakvu zaštitu. U dualnom modu izvršavanja, razlikujemo *izvršavanje koda od strane operativnog sistema* i *izvršavanje korisničkog koda*. Većina računarskih sistema obezbeđuje odgovarajuću hardversku podršku, koja je neophodna za implementaciju dualnih (ili čak višestrukih) modova izvršavanja.

Da bismo ovo implementirali, potrebna su nam dva odvojena režima rada: korisnički režim i kernel režim (koji se takođe zove i nadzorni režim - *supervisor mode*, sistemski režim - *system mode* ili privilegovani režim - *privileged mode*). Zaseban bit, nazvan *bit moda rada* (eng. *mode bit*), dodaje se hardveru računara da bi označio trenutni režim izvršavanja: kernel (0) ili korisnički (1). Pomoću bita moda rada možemo razlikovati zadatke koji se izvršavaju u kontekstu operativnog sistema i zadatke koji se izvršavaju u kontekstu korisnika. Kada se izvršava kod u kontekstu korisničke aplikacije, sistem je u korisničkom režimu. Međutim, kada korisnička aplikacija zatraži uslugu od operativnog sistema (putem *sistemskog poziva*), sistem mora da pređe iz korisničkog u kernel režim da bi ispunio zahtev. To je prikazano na slici 5. Kao što ćemo videti, ovo arhitekturalno unapređenje pokazaće se korisno i za mnoge druge aspekte rada sistema.

Prilikom pokretanja sistema, hardver se pokreće u kernel režimu. Operativni sistem se zatim učitava i pokreće korisničke aplikacije u korisničkom režimu. Kad god se dogodi izuzetak ili prekid, hardver prelazi iz korisničkog režima u režim kernela (tj. menja stanje bita moda rada u 0). Stoga, svaki put kada operativni sistem preuzme kontrolu nad računarom, on se nalazi u kernel režimu. Sistem uvek prelazi u korisnički režim (postavljanjem bita moda rada na 1) pre prenosa

kontrole na korisnički program.

Dualni način rada pruža nam sredstva za zaštitu operativnog sistema od štetnih korisnika i štetnih korisnika jednih od drugih. Ovu zaštitu ostvarujemo definisanjem skupa *privilegovanih instrukcija*, u koji spadaju sve instrukcija koje mogu na bilo koji način naškoditi sistemu. Sa druge strane, hardver obezbeđuje da se *privilegovane instrukcije* uvek izvršavaju isključivo u kernel režimu. Ako se pokuša izvršiti privilegovana instrukcija u korisničkom režimu, hardver ne izvršava instrukciju, već je tretira kao nelegalnu i generiše izuzetak za operativni sistem. Instrukcija za prelazak u režim kernela je primer privilegovane instrukcije. Slične su instrukcije koje kontrolišu U/I uređaje, upravljaju tajmerom ili prekidima. Kao što ćemo videti u nastavku, postoji mnogo privilegovanih instrukcija koje ćemo koristiti.

Sada kada smo definisali privilegovane instrukcije, *sistemski pozivi* pružaju mogućnost korisničkom programu da od operativnog sistema zatraži da u ime njih obavlja zadatke rezervisane za operativni sistem. Sistemski poziv se vrši na različite načine, zavisno od funkcionalnosti koju pruža procesor. U svim varijantama, to je metoda koju koristi proces da bi zatražio akciju od strane operativnog sistema. Sistemski poziv obično ima oblik *izuzetka* kojem je dodeljena određena lokacija u *vektoru prekida*.

Kada se izvrši sistemski poziv, hardver ga obično tretira kao softverski prekid. Kontrola prelazi preko vektora prekida do servisne rutine u operativnom sistemu, a mod rada je postavljen na kernel režim. Rutina za obradu sistemskog poziva je deo operativnog sistema. Kernel proverava instrukciju koja je dovela do prekida kako bi utvrdio koji sistemski poziv se dogodio: parametar uz instrukciju ukazuje na vrstu usluge koju korisnički program traži od operativnog sistema. Dodatne informacije eventualno potrebne za zahtev mogu se proslediti kroz registre, na steku ili u memoriji (s pokazivačima na memorijske lokacije prosleđenim u registrima). Kernel proverava da li su parametri tačni i legalni, izvršava zahtev ako jesu i vraća kontrolu instrukciji koja sledi nakon instrukcije sistemskog poziva. Pričaćemo i o ovome detaljnije u nastavku.

4.5 Bezbedan transfer kontrole

Nakon što kernel kreira korisnički proces, postavlja se pitanje kako bezbedno prelaziti sa izvršenja korisničkog procesa ka izvršenju u okviru kernela i obrnuto. Ovi prelazi nisu retki događaji. Web server, na primer, može da obavlja ovu tranziciju između korisničkog i kernel režima stotine ili hiljade puta u sekundi. Dakle, mehanizam treba da bude i brz i siguran, ne ostavljajući prostora ni jednom programu da namerno ili nenamerno naškodi kernelu.

4.5.1 Transfer iz korisničkog u kernel režim

Prvo se fokusiramo na prelaze sa korisničkog režima na kernel režim. Kao što ćemo videti, prelazak u drugom smeru funkcioniše tako što „poništava" prelazak iz korisničkog procesa u kernel. Postoje tri razloga zbog kojih će kernel preuzeti kontrolu nad korisničkim procesom: izuzeci, prekidi i sistemski pozivi.

- Izuzeci: Izuzetak je reakcija na svako neočekivano stanje uzrokovano ponašanjem korisničkog programa. Prilikom izuzetka, hardver će zaustaviti proces koji se trenutno izvršava i pokrenuti posebno namenjen program za obradu izuzetka u okviru kernela. Kao što smo ranije spomenuli, izuzetak se generiše svaki put kada proces pokušava da izvrši privilegovanu instrukciju ili pristupi memoriji izvan memorije dodeljene tom procesu. Ostali izuzeci se dešavaju kada proces deli ceo broj sa nulom, pristupa memorijskoj reči sa neporavnatom adresom (*non-aligned address*), pokušava da upisuje u memoriju koja je rezervisana samo za čitanje i tako dalje. U tim slučajevima, operativni sistem jednostavno zaustavlja proces i korisniku vraća kod greške. Međutim, izuzeci mogu biti aktivirani i brojnim drugim, bezazlenijim, događajima u okviru programa. Na primer, da bi postavio tačku prekida u programu (*breakpoint*), kernel zamenjuje mašinsku instrukciju u memoriji posebnom instrukcijom koja će izazvati izuzetak. Kada program dođe do te tačke tokom izvršavanja, hardver se prebacuje u kernel režim. Kernel vraća prvobitnu instrukciju i prenosi kontrolu debageru. Pomoću debagera, korisnik može pregledati stanje promenljivih u okviru programa, postaviti novu tačku prekida i nastaviti program počevši od instrukcije koja je uzrokovala izuzetak.

- Prekidi: Prekid je asinhroni signal procesoru da se dogodio neki spoljni događaj koji može zahtevati njegovu pažnju. Prekid funkcioniše u velikoj meri kao izuzetak: procesor zaustavlja proces koji se trenutno izvršava i započinje izvršavanje posebno namenjene rutine za obradu prekida u okviru kernela. Svaka različita vrsta prekida zahteva svoju rutinu za obradu prekida. Posebno je specifičan prekid generisan od strane tajmera, uvek prisutan u operativnom sistemu. Kao prva uloga prekida tajmera, prekidna rutina proverava da li trenutni proces reaguje na unos korisnika, kako bi otkrila da li je proces eventualno ušao u beskonačnu petlju. Dodatno, prekidna rutina tajmera može takođe periodično prebaciti procesor nekom drugom procesu kako bi se osiguralo da se svaki proces izvršava. Ako ne postoji drugi proces koji bi se mogao izvršavati, nakon završetka prekidne rutine tajmera, nastavlja se izvršavanje na mestu prekinute instrukcije, potpuno transparentno sa stanovišta korisničkog procesa. Prekidi se takođe koriste za obaveštavanje kernela o pristiglim ulazno/izlaznim zahtevima. Na primer, hardver uređaja miša izazvaće prekid svaki put kada korisnik pomeri miša ili klikne mišem. Kernel će, zauzvrat, obavestiti o tome odgovarajući korisnički proces - onaj koji je bio aktivan u trenutku dok se miš pomerao. Skoro svaki ulazno/izlazni uređaj (Ethernet, WiFi, hard-disk, tastatura, miš) izaziva prekid kad god se pojavi neki događaj na koji procesor treba da odreaguje i kad god se ispuni neki zahtev. Alternativa prekidima je *prozivanje* (*polling*): kernel može prozivati i na taj način proveravati svaki ulazno/izlazni uređaj, kako bi video da li se dogodio neki događaj koji zahteva obrađivanje. Naravno, ukoliko kernel proziva dati uređaj, u tom periodu vremena nije moguće izvršavati nijedan proces iz korisničkog režima. Među-procesorski prekidi su takođe jedan od izvora

prekida sa kojima se srećemo. Na multiprocesorskim sistemima, procesor može izazvati prekid na koji će odreagovati bilo koji od ostalih procesora. Kernel koristi ove prekide za koordinaciju akcija u okviru multiprocesorskog sistema. Na primer, ako se na jednom procesoru desi fatalan izuzetak, kernel će, tipično, generisati prekid kako bi zaustavio izvršavanje eventualno kritičnog koda na bilo kojem drugom procesoru.

- Sistemski pozivi: Korisnički procesi takođe mogu dobrovoljno preći u kernel režim operativnog sistema i zatražiti da kernel izvrši neku radnju u ime korisnika. Sistemski poziv je svaka procedura koju pruža kernel a koja se može pozvati od strane korisničkog procesa. Većina procesora implementira sistemske pozive koristeći posebnu instrukciju *zamke* (*trap*). Međutim, postojanje instrukcija zamki nije obavezno; može se izazvati zamka izvršavajući bilo koju instrukciju koja rezultuje izuzetkom (npr. ona sa nepostojećim kodom operacije). Kao i kod prekida ili izuzetka, instrukcija zamke menja režim rada procesora iz korisničkog u kernel i započinje izvršavanje u kernelu, koristeći unapred definisanu rutinu. U cilju zaštite kernela od lošeg ponašanja korisničkih programa, od suštinskog je značaja da korisničke aplikacije nakon generisanja zamke, uvek dospeju na unapred definisanu adresu - nikako se ne sme dozvoliti skok na proizvoljnu adresu unutar kernela. Operativni sistemi omogućavaju znatan broj sistemskih poziva. Primeri uključuju one za uspostavljanje veze sa web serverom, slanje ili primanje paketa preko mreže, kreiranje ili brisanje datoteka, čitanje ili upisivanje podataka u datoteke i kreiranje novog korisničkog procesa. U korisničkom programu sistemski pozivi se pozivaju baš kao i obične procedure, sa parametrima i povratnim vrednostima. Kao i kod svake dobre apstrakcije, pozivalac se bavi samo interfejsom - on ne treba znati da rutinu zapravo izvršava kernel, a ne biblioteka u okviru korisničkog prostora. Kernel vodi računa o svim proverama i kopiranju argumenata, izvođenju zahtevane operacije i kopiranja svih povratnih vrednosti nazad u memoriju procesa. Kada je kernel završio sa obradom sistemskog poziva, nastavlja se izvršenje procesa u korisničkom režimu, počevši od instrukcije koja sledi odmah nakon poziva instrukcije *zamke*.

4.5.2 Transfer iz kernel režima u korisnički režim

Kao što postoji nekoliko povoda za prelazak iz korisničkog režima u kernel režim, tako postoji i nekoliko slučajeva kod kojih će doći do prelaska sa kernel režima na režim korisnika.

- Novi proces: Da bi se pokrenuo novi proces, kernel kopira program u memoriju, postavlja programski brojač na prvu instrukciju procesa, postavlja pokazivač steka na početnu adresu korisničkog steka i prebacuje se u korisnički režim.
- Nastavak rada nakon izuzetka, prekida ili sistemskog poziva: Kada kernel završi obradu zahteva, nastavlja izvršavanje prekinutog procesa vraćanjem

njegovog sačuvanog programskog brojača, kao i vraćanjem njegovih vrednosti registara i ponovnim aktiviranjem korisničkog režima.

- Prebacivanje na drugi proces: U nekim slučajevima, na primer nakon prekida tajmera, kernel će odlučiti da li treba da pređe na drugi proces, umesto da nastavi izvršavanje onog koji se izvršavao pre prekida. Budući da će kernel, pre ili kasnije, želeti da nastavi prekinuti proces, on mora da sačuva stanje procesa (njegov programski brojač, njegove registre, ...) u kontrolnom bloku procesa. Kernel može zatim nastaviti sa izvršavanjem drugog procesa, učitavanjem u procesor sačuvanog stanja (programskog brojača, registara, ...) iz kontrolnog bloka novog procesa, nakon čega se prebacuje u režim korisnika.

- Asinhrono obaveštavanje korisničkih procesa: Mnogi operativni sistemi pružaju korisničkim programima mogućnost da primaju asinhrono obaveštavanje o događajima. Mehanizam, koji ćemo kasnije detaljnije opisati, vrlo je sličan obrađivanju prekida u režimu kernela, sa izuzetkom što se ovi događaji obrađuju u korisničkom režimu.

4.5.3 Bezbedna zamena režima rada

Bez obzira da li se radi o prelasku iz korisničkog u kernel režim, ili suprotno, treba voditi računa da nijedan korisnički program ne može oštetiti kernel. Iako je osnovna ideja jednostavna, implementacija ovoga na nižem nivou može biti veoma zahtevna. Želimo da procesor sačuva svoje stanje i zameni ono što radi, a sve to dok nastavlja da izvršava instrukcije koje potencijalno mogu promeniti stanje koje se upravo čuva. Analogija ovome bi bila popravka menjača automobila dok se on kreće brzinom od 100 km/h.

Ovaj prelazak iz jednog u drugi režim rada mora biti pažljivo projektovan, oslanjajući se na određenu hardversku podršku. Da bi smanjili mogućnost greške, većina operativnih sistema ima zajedničku sekvencu instrukcija za ulaz u kernel režim (bilo zbog prekida, izuzetaka ili sistemskih poziva) i zajednički niz instrukcija za povratak u korisnički režim, opet nezavisno od uzroka. U najmanju ruku, ova zajednička sekvenca mora da obezbedi:

- Ograničen ulaz (*Limited entry*) - Da biste preneli kontrolu na kernel operativnog sistema, hardver mora osigurati da je ulazna tačka u kernel upravo ona koja je određena i postavljena od strane kernela. Korisničkim programima nije dozvoljeno da skaču na proizvoljne lokacije u okviru kernela. Na primer, kod kernela za rukovanje sistemskim pozivom za čitanje datoteke prvo će proveriti da li korisnički program ima dozvolu za to, a ako ne, kernel vraća grešku. Bez ograničenih ulaznih tačaka u kernel, zlonamerni program bi mogao jednostavno preskočiti odmah nakon koda za proveru prava, omogućavajući svakom korisniku pristup bilo kojoj datoteci u okviru fajl sistema.

- Atomičke promene stanja procesora - U korisničkom režimu, programski brojač i pokazivač na stek pokazuju na memorijske lokacije u korisničkom

procesu. Zaštita memorije sprečava korisnički proces da pristupi bilo kojoj memoriji izvan svoje zone memorije. U kernel režimu, programski brojač i stek pokazivač pokazuju na memorijske lokacije u okviru kernel-a. U ovom slučaju, zaštita memorije se menja kako bi se kernelu omogućio ne samo pristup sopstvenim podacima, već i podacima korisničkog procesa. Prelaz između ova dva mora biti *atomički*, obezbeđujući istovremenu promenu režima, programskog brojača, steka i zaštite memorije.

- Transparentno izvršenje koje se može ponovo pokrenuti - Proces koji se izvršava na korisničkom nivou (*user-level process*) može biti prekinut u bilo kojem trenutku, između bilo koje dve instrukcije. Na primer, procesor je mogao izračunati memorijsku adresu, učitati je u registar i spremiti se za čuvanje podatka na toj adresi. Ključno za sistem prekida je da operativni sistem mora biti u stanju da rekonstruiše stanje korisničkog programa upravo onako kako je bilo pre prekida. Za korisnički proces, prekid je potpuno transparentan, osim što on privremeno odlaže izvršavanje programa. Pisanje „Hello world!" programa ne bi trebalo da vodi računa o prekidima koji bi se mogli desiti dok se on izvršava, iako će se prekidi vrlo verovatno desiti dok se on izvršava. Stoga, prilikom prekida, procesor čuva trenutno stanje procesa u memoriji, privremeno odlaže sve ostale događaje i postavlja procesor da se izvršava u režimu kernela, pre nego što preskoči na obradu prekidne rutine ili rutine za obradu izuzetka. Kada se rutina završi, koraci se vrše u suprotnom smeru: procesor se vraća u prethodno stanje u skladu sa sačuvanim podacima, i prekinuti program nastavlja da se izvršava tamo gde je stao.