Glava 1

Sistemski softver

Podsetimo se da se računarski sistem sastoji iz *hardvera* i *softvera*. Hardver čine sve fizičke komponente računara, poput procesora, operativne memorije, magistrala, ulazno-izlaznih uređaja i spoljnih memorija. *Softver* čine računarski programi koji se mogu pokretati i izvršavati na tom računaru.

Hardver i softver su od jednake važnosti za računarski sistem, jer su jedan bez drugoga potpuno neupotrebljivi. Računar je, po definiciji, mašina koja izvršava programe – to je sve što računar ume i može da radi. Zbog toga računaru odmah po uključivanju mora biti dostupan program koji bi izvršavao, u suprotnom računar ne bi mogao da radi ništa.

Softver može biti *aplikativni* i *sistemski*. Pod aplikativnim softverom podrazumevamo programe koji su razvijeni sa ciljem primene u različitim oblastima. Drugim rečima, to su oni programi koji se koriste da korisniku reše neki problem ili mu pruže neku uslugu. U aplikativni softver spada npr. kancelarijski softver (editori i procesori teksta, programi za tabelarna izračunavanja i sl.), multimedijalni softver (reprodukcija i obrada slika, zvuka i videa, vektorska grafika i sl.), komunikacioni softver (veb pregledači, programi za udaljeni pristup, mejl klijenti i sl.), matematički i naučni softver (programi za numerička i simbolička izračunavanja, statistički softver i sl.), razvojni softver (programski prevodioci, debageri, profajleri, sistemi za održavanje i kontrolu verzija softvera i drugi programerski alati).

Dakle, aplikativni softver čine svi oni programi koji su nama potrebni da bismo mogli da primenjujemo računare u različitim oblastima. Ipak, pokretanje i zaustavljanje ovakvih programa, kao i bezbedna i kontrolisana upotreba različitih računarskih resursa od strane aplikativnih programa tokom njihovog izvršavanja nije ni malo jednostavan posao. Da bi korisnik mogao da pokreće svoje aplikativne programe, upravlja njihovim radom, bezbedno pristupa svojim podacima na spoljnim memorijama, kao i drugim resursima računara, potrebni su posebni programi koji mu to omogućavaju. Skup takvih programa nazivamo *sistemskim softverom*.

Najznačajnija komponenta sistemskog softvera je *operativni sistem*. Njega čini skup programa koji obezbeđuju efikasnu i bezbednu kontrolu računarskih resursa. U ove resurse ubrajamo kako hardverske komponente (procesor, memoriju, ulazno-izlazne uređaje, spoljne memorije, komunikacioni hardver i sl.), tako i podatke sačuvane u spoljnim memorijama. Pored upravljanja ovim resursima, operativni sistem obezbeđuje mehanizme korišćenja tih resursa od strane aplikativnih programa. Najzad, operativni sistem omogućava komunikaciju korisnika sa računarom podsredstvom ulazno-izlaznih uređaja. Ovo podrazumeva upotrebu podataka, kao i pokretanje i zaustavljanje aplikativnih programa.

Pored operativnog sistema, u sistemski softver spadaju i drugi programi poput antivirusnih programa, alata za održavanje, konfiguraciju i optimizaciju sistema, programi za učitavanje operativnog sistema, ugrađeni programi poput BIOS-a i različitih firmvera za uređaje i sl. Međutim, podeljena su mišljenja o tome koji od tih programa se mogu smatrati delom operativnog sistema, a koji predstavljaju izdvojene softverske komponente. Takođe, ponekad se i programerski alati poput previodica, linkera i debagera smatraju sistemskim softverom. Isto važi za različite mrežne servere, koji spadaju u komunikacioni softver, ali su obično u nadležnosti administratora sistema. Sa druge strane, pojedine komponente koje se isporučuju sa operativnim sistemom (poput editora teksta, jednostavnih igara, multimedijalnih programa i sl.) se obično smatraju delom operativnog sistema, iako ne spadaju u sistemski softver. Dakle, granica između sistemskog i aplikativnog softvera nije uvek tako oštra i postoje programi koji se mogu smatrati „sivom zonom" između ove dve vrste softvera.

1.1 Operativni sistem

Operativni sistem predstavlja skup programa koji omogućavaju efikasno i bezbedno upravljanje resursima računarskog sistema i stavljanje tih resursa na raspolaganje aplikativnim programima i, posredno, korisniku računara. Komponente operativnog sistema se mogu nalaziti u ROM memoriji (u novije vreme EEPROM) ili u

Elektronska verzija (2024)

spoljašnjim memorijama (danas tipično na hard ili SSD disku), odakle se po uključivanju računara automatski učitavaju u operativnu memoriju (RAM) i započinju sa izvršavanjem na procesoru računara. Operativni sistem dalje omogućava pokretanje aplikativnih programa od strane korisnika (podsredstvom dela operativnog sistema koji se zove *korisnički interfejs*). Ovo znači da je bez operativnog sistema nemoguće efektivno koristiti računar, što operativni sistem čini neophodnom softverskom komponentom svakog računara.

U neka ranija vremena, računari su imali fiksirani, fabrički ugrađeni operativni sistem koji se nalazio u ROM memoriji računara. Sa druge strane, moderni računari obično imaju operativni sistem koji je izmenjiv, jer se nalazi u spoljašnjoj memoriji čiji se sadržaj može menjati. Otuda na većini savremenih računara korisnik ima mogućnost izbora operativnog sistema koji će koristiti, kao i mogućnost nadogradnje verzije operativnog sistema i pratećeg softvera.

U najpoznatije operativne sisteme za desktop i laptop računare danas spadaju operativni sistemi iz serije *Windows*, razvijeni od strane kompanije Majkrosoft (engl. *Microsoft*). U pitanju je vlasnički softver za čiju se upotrebu plaća licenca, uz različita ograničenja koja su tom licencom nametnuta. Alternativno, korisnicima su na raspolaganju operativni sistemi koji spadaju u kategoriju *slobodnog softvera*, što znači da se mogu potpuno slobodno koristiti, distribuirati i modifikovati gotovo bez ikakvih ograničenja (a uz to su u većini slučajeva i potpuno besplatni). U ovakve operativne sisteme uglavnom spadaju sistemi zasnovani na operativnom sistemu *UNIX* (poput sistema *GNU/Linux* i *BSD UNIX*). Pored toga, postoji i aktivan razvoj slobodnih operativnih sistema kompatibilnih sa *Windows* operativnim sistemom (poput *ReactOS* sistema). Sa druge strane, i među UNIX-zasnovanim operativnim sistemima postoje oni sa ne-slobodnom licencom – najpoznatiji primer je *macOS* kompanije Epl (engl. *Apple*).

Kada su u pitanju mobilni uređaji (poput pametnih telefona i tableta), na tržištu dominiraju komercijalni proizvodi poput sistema *Android* kompanije Gugl (engl. *Google*) i *iOS* operativnog sistema kompanije Epl. Takođe postoji i veliki broj *GNU/Linux* distribucija prilagođenih za izvršavanje na ovakvim uređajima, ali je njihov udeo na tržištu veoma ograničen, pre svega zbog neadekvatne podrške proizvođača hardvera za ovakve sisteme.

1.1.1 Struktura operativnog sistema

Operativni sistem je veoma složen softver koji se sastoji iz više celina. Glavni deo operativnog sistema naziva se *jezgro* (engl. *kernel*) koji je zadužen za upravljanje računarskim resursima, komunikaciju sa hardverom, kao i obezbeđivanje interfejsa ka korisničkim programima. Korisnički programi (u koje ubrajamo i aplikativne programe, ali i druge komponente operativnog sistema koje nisu deo jezgra) pristupaju resursima računara isključivo putem *interfejsa jezgra* operativnog sistema pomoću koga od jezgra zahtevaju odgovarajuće usluge. Interfejs jezgra se obično sastoji iz skupa funkcija koje se nazivaju *sistemski pozivi* (engl. *system call*). Sistemski poziv obezbeđuje bezbedan transfer kontrole jezgru operativnog sistema, kako bi ono moglo da korisničkom programu pruži traženu uslugu.

Sistemski pozivi su obično implementirani na nivou asemblerskog jezika. S obzirom da se softver danas uglavnom razvija koristeći programske jezike visokog nivoa, obezbeđuju se različite *sistemske biblioteke* koje omogućavaju programerima da upućuju zahteve jezgru operativnog sistema iz različitih programskih jezika.

Sa druge strane, da bi korisnik mogao da komunicira sa operativnim sistemom, obezbeđen je *korisnički interfejs*. U pitanju je korisnički program (ili skup programa) koji se pokreće odmah nakon što se korisnik prijavi na sistem, a koji omogućava korisniku da izdaje komande, pokreće korisničke programe i prati njihov rad, kao i da koristi podatke na spoljnim memorijama.

Pored toga, uz operativni sistem se obično isporučuju i različiti programi koji obezbeđuju različite servise, poput servisa štampe, mrežnih servisa, servisa za pokretanje periodičnih akcija za održavanje sistema, i sl. Ovi servisi su opcioni i korisnik može konfiguracijom sistema odabrati koji će servisi biti aktivni.

Najzad, uz operativni sistem se obično isporučuje i skup uslužnih programa koji omogućavaju korisniku da jednostavnije upravlja svojim sistemom, obavlja razne administrativne poslove, instalira dodatni softver i sl.

1.1.2 Programi i procesi

U terminologiji operativnih sistema, *proces* predstavlja program u izvršenju. Na taj način razlikujemo programe koji se izvršavaju na računaru od onih koji stoje negde zapamćeni u spoljnoj memoriji i nisu aktivni. Preciznije, proces predstavlja kontekst u kome se program izvršava, a koji kreira operativni sistem. Taj kontekst se sastoji iz različitih struktura podataka koje operativni sistem interno održava za svaki pokrenuti program, a koje sadrže informacije o trenutnom stanju u kome se program nalazi prilikom izvršenja, kao i informacije o resursima koji su programu dodeljeni (poput memorije, otvorenih fajlova, kanala za komunikaciju sa drugim procesima i sl.). Procesi se unutar operativnog sistema obično identifikuju brojevima procesa (engl. *process identifier* (PID)). Prilikom pokretanja programa, operativni sistem najpre inicijalizuje proces i dodeljuje mu

jedinstven PID. Nakon toga mu dodeljuje memoriju i u nju učitava programski kôd programa i inicijalizuje njegove podatke. Nakon toga mu (u nekom trenutku) dodeljuje procesor i program kreće da se izvršava.

Proces može biti kreiran od strane jezgra operativnog sistema ili od strane drugog procesa. Proces može kreirati druge procese i u okviru njih pokretati druge programe. Proces koji kreira drugi proces ćemo u tom slučaju nazivati *roditeljski proces* (engl. *parent process*), dok će pokrenuti proces biti *dete proces* (engl. *child process*). Posredno, procese može kreirati i korisnik tako što pokreće programe putem korisničkog interfejsa operativnog sistema. Sâm korisnički interfejs je takođe proces koji je upravo i namenjen za komunikaciju sa korisnikom (i za pokretanje procesa u ime korisnika).

Tokom svog rada, proces može zahtevati resurse od operativnog sistema. U cilju dodele tih resursa, operativni sistem može inicijalizovati dodatne strukture podataka. Na primer, ako proces želi da otvori neki fajl na disku koji želi da čita, operativni sistem će kreirati strukturu podataka koja će sadržati informacije o lokaciji otvorenog fajla na disku, broju bajta do koga se stiglo sa čitanjem, bafere koji sadrže pročitane bajtove i sl. Procesu će, po otvaranju fajla, biti prosleđena *ručka*, koja predstavlja neku vrstu identifikatora otvorenog fajla pomoću koje proces može u nastavku pristupati otvorenom fajlu i obavljati operacije sa njim. Po zatvaranju fajla, kreirane strukture podataka se uništavaju, a memorijski resursi se oslobađaju.

Proces može završiti svoj rad na više načina. Najprirodniji način je da se završi izvršavanje programskog kôda pokrenutog programa, čime program vraća kontrolu operativnom sistemu, a ovaj dealocira resurse dodeljene procesu i uklanja pridružene strukture podataka iz memorije. Drugi način je da program bude prekinut protiv svoje volje. To može uraditi ili operativni sistem (ako program pokuša da uradi nešto što nije dozvoljeno, poput izvršenja neke nedozvoljene instrukcije ili pristupa nedozvoljenoj zoni memorije), ili korisnik podsredstvom korisničkog interfejsa operativnog sistema i odgovarajućih alata za upravljanje procesima. Time se šalje signal operativnom sistemu da dati proces treba likvidirati, što operativni sistem radi oduzimajući mu resurse i uklanjajući ga iz evidencije aktivnih procesa.

1.1.3 Pokretanje i zaustavljanje operativnog sistema

Prilikom uključivanja računara, procesor odmah započinje sa radom. Njegov programski brojač (registar procesora koji sadrži adresu instrukcije koju sledeću treba izvršiti) je obično fabrički podešen na neku fiksiranu početnu vrednost, pa je potrebno obezbediti da počev od te adrese imamo program koji prvi treba da se izvrši po uključivanju računara. Ovo je odgovornost onoga ko ugrađuje procesor u svoj računar, a u slučaju modernih desktop i laptop računara, to su proizvođači matičnih ploča, koji u njih ugrađuju ROM memorije koje sadrže taj početni program koji se prvi izvršava. Tradicionalno, taj program se naziva BIOS (engl. *basic-input-output-system*).[1] Uloga BIOS programa je da inicijalizuje hardverske komponente, izvrši neke osnovne testove ispravnosti, a da zatim preda kontrolu sledećoj softverskoj komponenti u nizu, koja se naziva *punilac* (engl. *boot loader*). Ovu komponentu po pravilu obezbeđuje sâm operativni sistem koji je instaliran na računaru. Punilac se, kao i sâm operativni sistem, nalazi na nekoj od spoljnih memorija, tipično na hard ili SSD disku. Uobičajena konvencija je da se punilac (ili njegov početni deo, u slučaju složenijih punilaca), nalazi u okviru nultog sektora hard diska, koji je poznat pod nazivom MBR (engl. *master boot record*). BIOS učitava program iz MBR-a u memoriju i predaje mu kontrolu.

Uloga punioca je da pronađe ostale komponente operativnog sistema na disku i učita ih u operativnu memoriju. Kako ovaj postupak zavisi od vrste operativnog sistema, punilac mora biti kompatibilan sa operativnim sistemom i mora biti upoznat sa strukturom operativnog sistema. Otuda se punilac obično i isporučuje i instalira zajedno sa operativnim sistemom.[2] Nakon što punilac učita jezgro operativnog sistema u operativnu memoriju i preda mu kontrolu, njegova uloga se završava.

Jezgro operativnog sistema najpre inicijalizuje svoje strukture podataka, postavi procesor u odgovarajući režim rada, inicijalizuje različite sistemske atribute (poput sistemskog vremena, vremenskog intervala tajmera i sl.) i uspostavi okruženje za izvršavanje korisničkih procesa. Nakon toga, jezgro pokreće prvi korisnički program, čime se započinje proces inicijalizacije korisničkog okruženja i pokretanje sistemskih servisa. Na kraju tog procesa, korisniku se omogućava prijava na sistem, obično unosom korisničkog imena i lozinke. Ukoliko prijava bude uspešna, pokreće se korisnički interfejs operativnog sistema, što dalje omogućava korisniku da po želji pokreće svoje korisničke programe, upravlja svojim podacima ili izvršava različite administrativne poslove.

Na kraju rada sa računarom, korisnik pokreće proceduru isključivanja računara. Tom prilikom se zaustavljaju svi procesi koji trenutno rade na računaru, a zatim se kontrola predaje jezgru koje posebnim postupkom isključuje računar.

[1] U novije vreme, BIOS je ustupio svoje mesto programu koji se naziva UEFI (engl. *unified extensible firmware interface*), a koji služi istoj svrsi.

[2] Korisnici *Windows* operativnog sistema nisu ni svesni da punilac postoji, jer ih instalacioni program o tome ne informiše. Sa druge strane, *GNU/Linux* korisnici jako dobro znaju šta je punilac, čak uglavnom imaju i mogućnost izbora različitih punilaca prilikom instalacije (najpoznatiji su *Lilo* i *Grub*).

Elektronska verzija (2024)

1.1.4 Funkcije operativnog sistema

U ovom odeljku ukratko opisujemo osnovne funkcije operativnog sistema. Kao što ćemo videti, operativni sistem obavlja vrlo složene zadatke koji zahtevaju sa jedne strane interakciju sa hardverom na niskom nivou, a sa druge strane komunikaciju sa korisničkim programima i korisnikom na visokom nivou. Otuda je razvoj kvalitetnog operativnog sistema veoma zahtevan posao (po mišljenju mnogih, zahtevniji od razvoja većine aplikativnih programa).

Upravljanje procesorom

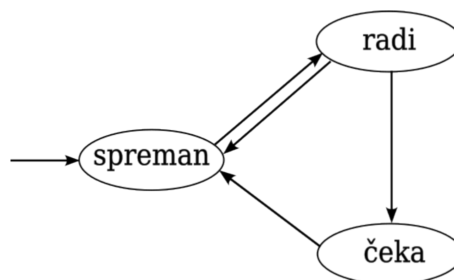
Centralni procesor (engl. *Central processing unit* (CPU)) je, uz operativnu memoriju, najznačajnija hardverska komponenta svakog računara. Njegova uloga je da izvršava mašinske programe koji se nalaze u operativnoj memoriji računara. Za izvršavanje programa procesoru je potrebno neko vreme. Ovo vreme, koje nazivamo i *procesorsko vreme*, najznačajniji je resurs računarskog sistema koji programi koriste. Ukoliko bi se u memoriji nalazio samo jedan program, tada bi upravljanje ovim resursom bilo trivijalno – prosto bismo svo procesorsko vreme dodelili tom jednom programu. Sa druge strane, treba imati u vidu da je procesor najbrža hardverska komponenta računara. Sve druge komponente, od operativne memorije, preko spoljašnjih memorija, do ulazno-izlaznih uređaja, značajno su sporije. Otuda se prilikom izvršavanja programa najveći deo vremena neće trošiti za izvršavanje mašinskih instrukcija u procesoru, već ce se tipično trošiti na interakciju sa drugim sporijim komponentama hardvera, poput ulazno-izlaznih uređaja. Dok program čeka na izvršenje spore ulazno-izlazne operacije, procesor stoji u mestu i ne radi ništa. Pritom, vreme izvršavanja ulazno-izlazne operacije iz ugla procesora traje veoma dugo – izraženo u broju instrukcija koje bi procesor mogao da izvrši za to vreme, to može biti i više hiljada instrukcija. To znači da će efikasnost korišćenja procesorskog vremena kao resursa biti veoma niska, naročito u slučaju programa koji intenzivno koriste ulaz i izlaz.

Kako bi se efikasnost korišćenja procesora povećala, u memoriju treba učitati više programa. U svakom trenutku, jedan program se izvršava na procesoru, dok ostali programi čekaju da dobiju procesor. Kada program koji se izvršava mora da se zaustavi da bi sačekao neku sporu ulazno-izlaznu operaciju, operativni sistem predaje procesor na korišćenje drugog programu koji nastavlja svoj rad. Na ovaj način se obezbeđuje da procesor stalno bude uposlen, što značajno povećava njegovu efikasnost.

Opisano povećanje efikasnosti iskorišćenja procesora je istorijski bilo inicijalni motiv za uvođenje *multiprocesiranja*, tj. držanja više programa u operativnoj memoriji koji se naizmenično izvršavaju na procesoru. U današnje vreme, dodatni motiv jeste i povećanje udobnosti korišćenja računara kroz mogućnost obavljanja većeg broja poslova istovremeno (npr. surfujemo internetom, a istovremeno nam u pozadini svira muzika ili nam je otvoren prozor u kome uređujemo neki dokument). Danas je potpuno uobičajeno da se u svakom trenutku na računaru izvršava više desetina programa istovremeno.

Sa druge strane, multiprocesiranje čini upravljanje procesorskim vremenom kao resursom računara mnogo komplikovanijim. Sada procesorsko vreme nije ekskluzivni resurs jednog programa, već se mora deliti između većeg broja programa, zadovoljavajući pritom više različitih zahteva, od efikasnosti iskorišćenja procesora, preko pravednosti raspodele procesorskog vremena između programa do udobnosti za korisnika koji zahtevaju visok stepen interaktivnosti (tj. žele brz odziv svojih aplikacija). Da bi se sve to postiglo, tokom prethodnih decenija razvijeni su različiti *algoritmi raspoređivanja procesa* koji pokušavaju da zadovolje sve ove zahteve. Ovi algoritmi su implementirani u okviru operativnog sistema.

Elektronska verzija (2024)



Slika 1.1: Graf stanja procesa

Da bismo bolje objasnili raspoređivanje procesa, razmotrimo najpre moguća *stanja* u kojima se može nalaziti neki proces tokom svog izvršenja. Graf stanja procesa i mogućih prelaza između njih dat je na slici 1.1. Prilikom kreiranja, proces dolazi u stanje *spreman* (engl. *ready*) koje ukazuje da procesor može započeti svoje izvršavanje na procesoru u proizvoljnom trenutku. U svakom trenutku možemo imati veći broj procesa koji su u ovom stanju. Sve procese koji su u stanju *spreman* operativni sistem drži u *redu spremnih procesa* (engl. *ready queue*). Iz ovog

reda operativni sistem uzima jedan proces (u skladu sa nekom politikom koja je implementirana u algoritmu raspoređivanja) i dodeljuje mu procesor, tj. preusmerava procesor na izvršenje odgovarajućeg programa. Ovaj proces prelazi iz stanja *spreman* u stanje *radi* (engl. *running*). Kada proces koji radi dođe do tačke kada mu je potrebna neka ulazno-izlazna operacija, on zahteva od operativnog sistema da tu operaciju za njega obavi (videćemo kasnije da procesi ne mogu sami baratati ulazno-izlaznim uređajima, već samo mogu da zamole operativni sistem da to za njih uradi). Operativni sistem tada proces prebacuje u stanje *čeka* (engl. *wait*), u kom će ostati sve dok ulazno-izlazni uređaj ne završi zahtevanu operaciju i o tome ne obavesti operativni sistem. Tada će operativni sistem informacije o obavljenoj operaciji dostaviti procesu i prebaciti ga iz stanja *čeka* u stanje *spreman*. Proces se dodaje u red spremnih procesa i čeka da dobije procesorsko vreme kako bi nastavio sa radom. Primetimo da samo procesi koji se nalaze u stanju *spreman* učestvuju u nadmetanju za dobijanje procesorskog vremena. Procesi u stanju *čeka* nisu kandidati za raspoređivanje u tom trenutku, s obzirom da oni ne mogu nastaviti svoj rad dok se ne završi zahtevana ulazno-izlazna operacija.

U najjednostavnijem scenariju raspoređivanja, proces prolazi kroz ciklus *spreman*-*radi*-*čeka*-*spreman*. Drugim rečima, proces koji radi može izgubiti pristup procesoru samo ako dođe do tačke u svom izvršenju u kojoj mu je potrebna ulazno-izlazna operacija. Tada on prelazi u stanje *čeka*. Sa stanovišta efikasnosti upotrebe procesora, ovo je najbolja moguća strategija, jer nema razloga prekidati rad procesa koji efektivno koristi procesor u punoj meri. Sa druge strane, ovakva strategija može da ne bude fer prema drugim procesima koji čekaju na procesor, ukoliko program previše dugo radi bez zahteva za ulazno-izlaznim operacijama (npr. programi koji vrše neka intenzivna izračunavanja i retko koriste ulaz i izlaz). Takođe, u moderno vreme, interaktivnost je veoma bitna karakteristika operativnih sistema. Korisnik želi da aktivno koristi više različitih programa i očekuje da ti programi brzo reaguju na njegove komande. Zbog toga je potrebno korisniku obezbediti utisak da se svi programi izvršavaju sve vreme, iako mi znamo da to nije moguće, s obzirom da imamo samo jedan procesor. Ovo se postiže tzv. *podelom vremena* (engl. *time sharing*). Ideja je da se svakom procesu odredi maksimalno vreme koje može da koristi u jednom krugu, pre nego što procesor prepusti drugom procesu. Ovo vreme nazivaćemo *vremenski kvantum* i obično je relativno kratko (npr. 50ms). Ukoliko proces koji je u stanju *radi* prekorači vremenski kvantum, a da pritom nije zahtevao ulazno-izlaznu operaciju, operativni sistem ga automatski prekida i prebacuje u stanje *spreman*, a iz reda spremnih procesa bira drugi proces koga prebacuje u stanje *radi* i prepušta mu procesor na korišćenje. Na ovaj način obezbeđujemo da se procesi dovoljno često smenjuju na procesoru, što ostavlja utisak korisniku da oni svi istovremeno rade. Efektivno, ako n procesa raspoređujemo na ovaj način na jednom procesoru date brzine, korisnik će imati utisak da računar poseduje n procesora koji su n puta manje brzine, pri čemu svaki od tih procesora sve vreme izvršava jedan od datih n procesa.

Osnovna operacija koju operativni sistem obavlja prilikom upravljanja procesorom jeste oduzimanje procesora jednom procesu i njegovo prepuštanje drugom procesu. Ovaj postupak se naziva *zamena konteksta* (engl. *context swich*). Pri svakoj zameni konteksta, potrebno je sačuvati stanje svih registara procesora (uključujući i programski brojač i pokazivač steka), kako bi proces kasnije mogao nastaviti tamo gde je stao. Vrednosti registara procesora se upisuju u posebnu zonu u memoriji koja se naziva *kontrolni blok procesa*. Svaki proces ima svoj kontrolni blok koji je deo struktura podataka operativnog sistema, što znači da se za svaki od procesa zna u kom su stanju ostavili registre u trenutku kada je njihov rad prekinut. Nakon što sačuva stanje registara prekinutog procesa, operativni sistem učitava u registre vrednosti sačuvane u kontrolnom bloku procesa koji je odabran da nastavi svoj rad. Učitavanjem njegovog programskog brojača kontrola se predaje tom procesu koji nastavlja sa radom.

Upravljanje operativnom memorijom

Operativna memorija (poznata i kao RAM memorija) je drugi najznačajniji resurs računara. Koristi se za čuvanje programskog kôda, podataka i ostalih informacija pridruženih aktivnim procesima, kao i programskog koda i struktura podataka samog operativnog sistema. Svaki proces konzumira određeni memorijski prostor u RAM-u. Otuda ovaj prostor smatramo resursom koji je dodeljen procesu. Kao i u slučaju procesora, i ovde bi upravljanje memorijskim prostorom kao resursom bilo trivijalno kada bismo imali samo jedan program u memoriji – prosto bismo svu memoriju dali na raspolaganje tom programu. Ukoliko pak želimo da u svakom trenutku imamo veći broj aktivnih procesa, moraćemo svakom od njih da dodelimo deo memorijskog prostora, što samo upravljanje ovim resursom čini komplikovanijim.

Prvi problem koji nastaje u slučaju koegzistencije više programa u memoriji je mogućnost da jedan proces slučajno ili namerno pristupi podacima drugog procesa ili samog operativnog sistema, što stvara ozbiljne funkcionalne i bezbednosne rizike. Da bismo to sprečili potrebno je da na neki način izolujemo memorijske prostore pojedinačnih procesa i sprečimo adresiranje memorijskih lokacija izvan tog prostora. Ovo se postiže upotrebom *virtuelne memorije*. Ideja je da svaki proces ima svoj zasebni *virtuelni adresni prostor* određene veličine. U pitanju je fiktivni adresni prostor koji ne postoji zaista, već se njegova egzistencija emulira od strane procesora, u saradnji sa operativnim sistemom. Virtuelni adresni prostor je jedini prostor koji je vidljiv procesu

tokom izvršavanja. On sadrži kompletan mašinski program, kao i sve podatke koje proces koristi tokom svog rada. Sve instrukcije programa referišu isključivo na adrese iz tog virtuelnog adresnog prostora. Takođe, adresa tekuće instrukcije programa koja se nalazi u programskom brojaču procesora je takođe adresa iz tog virtuelnog prostora. Adrese lokacija unutar virtuelnog adresnog prostora nazivaju se *virtuelne adrese*. Na primer, ako je veličina virtuelnog prostora 1MB (tj. 2^{20} bajtova), tada je opseg virtuelnih adresa kojima proces može pristupati od 0 do $2^{20} - 1$. Instrukcije programa mogu koristiti samo adrese iz ovog opsega i sve takve adrese se smatraju adresama unutar tog virtuelnog prostora. Na ovaj način se sprečava pristup podacima drugih procesa, s obzirom da proces može pristupati samo lokacijama u svom virtuelnom prostoru.

Kao što je već rečeno, virtuelni adresni prostor ne postoji zaista. Ono što zaista postoji jeste *fizički adresni prostor*, koji čini fizička RAM memorija koja je instalirana na našem računaru. Za razliku od virtuelnog prostora koji je zaseban za svaki proces, fizički prostor je jedinstven i zajednički za sve procese. Virtuelnim adresama svakog procesa pridružujemo fizičke adrese u RAM-u gde će odgovarajući podaci ili instrukcije zaista biti smeštene. Na koji način se vrši ovo pridruživanje zavisi od konkretne implementacije virtuelne memorije. Ono što je jedino važno je da se taj proces obavlja automatski i potpuno nevidljivo za sam proces koji fizičkih adresa uopšte nije ni svestan. Kada proces želi da pristupi nekoj adresi u svom virtuelnom prostoru, ta adresa se propušta kroz deo procesora koji se zove *memorijska jedinica* (engl. *memory unit*) koja uz pomoć informacija koje je unapred obezbedio operativni sistem automatski prevodi virtuelnu adresu u fizičku adresu na kojoj se traženi podatak zaista nalazi, a zatim procesor pristupa toj fizičkoj adresi u RAM memoriji.

Još jedan problem koji može nastati prilikom upravljanja memorijom jeste problem nedostatka prostora u fizičkoj memoriji, do koga može doći u slučaju pokretanja velikog broja programa i njihovog učitavanja u operativnu memoriju. Ovaj problem je naročito bio prisutan u nekom ranijem periodu, kada su RAM memorije bile relativno male. Problem nedovoljne količine memorije se tada rešavao tako što se programi nisu učitavali celi u memoriju, već samo oni delovi koji se trenutno koriste. Programer je morao sam da vodi računa o tome koji su delovi programa trenutno učitani i da po potrebi zahteva od operativnog sistema da iz memorije izbaci neke delove i da ubaci neke druge. Ova tehnika bila je poznata i kao *overlay* tehnika, i bila je prilično zahtevna i komplikovana za programera. U novije vreme, i ovaj problem se elegantno rešava pomoću tehnike virtuelne memorije. Naime, ne moraju sve virtuelne adrese nekog procesa u svakom trenutku imati pridružene fizičke adrese. Obično su samo neki delovi virtuelnog adresnog prostora „učitani" u fizičku memoriju, dok se ostali delovi virtuelnog prostora mogu čuvati u spoljnoj memoriji (tipično na hard ili SSD disku). Ovakvo učitavanje virtuelnog adresnog prostora nazivaćemo *parcijalno učitavanje*. Kada proces pristupi nekoj virtuelnoj adresi za koju se ispostavi da nema pridruženu fizičku adresu, dolazi do automatskog prekida rada programa i predaje kontrole operativnom sistemu. Operativni sistem razrešava ovu situaciju tako što sa hard diska učita deo virtuelnog prostora procesa koji sadrži traženu adresu u neki slobodni deo operativne memorije i u skladu sa tim ažurira svoje strukture podataka koje čuvaju informacije o odgovarajućem pridruživanju virtuelnih i fizičkih adresa. Nakon toga operativni sistem vraća kontrolu prekinutom procesu koji, ne znajući šta se uopšte dogodilo, ponovo pokušava da izvrši istu instrukciju. Ovog puta će proces prevođenja virtuelne adrese u fizičku biti uspešan, s obzirom da je odgovarajući deo virtuelnog prostora učitan u fizičku memoriju. Proces će uspešno izvršiti instrukciju i nastaviti sa daljim radom. Važno je razumeti da se ceo ovaj postupak obavlja potpuno nevidljivo za sam proces, pri čemu se odgovarajući delovi virtuelnog adresnog prostora učitavaju u memoriju automatski, onda kada su potrebni. Otuda, programer koji piše program može smatrati da je ceo njegov virtuelni prostor sve vreme dostupan i ne mora preduzimati nikakve akcije da bi to obezbedio. Ukupna veličina svih virtuelnih adresnih prostora svih aktivnih procesa je obično značajno veća od veličine fizičke memorije računara, ali to ne predstavlja problem, zato što se ne čuvaju celi virtuelni prostori u RAM-u, već samo delovi koji se trenutno koriste. Ovo korisniku računara ostavlja utisak da ima znatno više memorije nego što je zaista ima. Ipak, treba imati u vidu da preveliki broj aktivnih procesa može značajno pogoršati performanse računara, ukoliko to dovede do prepunjenosti fizičke memorije. Ukoliko se ispostavi da za učitavanje zahtevanog dela virtuelnog prostora nekog procesa nema dovoljno slobodnog mesta u fizičkoj memoriji, operativni sistem će morati da izbaci deo virtuelnog prostora nekog drugog procesa iz fizičke memorije i prebaci ga na hard disk, kako bi oslobodio prostor. Ovo se takođe obavlja automatski, od strane operativog sistema i za posledicu ima značajno usporavanje rada računara u slučaju prepunjenosti operativne memorije. U tom slučaju će operativni sistem morati neprestano da prebacuje podatke sa hard diska i na hard disk, koji je značajno sporiji od operativne memorije.

Dodeljivanje fizičke memorije procesu može biti *kontinualno* i *diskontinualno*. U prvom slučaju, svakom procesu se dodeljuje odgovarajući broj *susednih* adresa fizičke memorije, tj. dodeljuje mu se fizički adresni prostor „u komadu". Ovakav pristup je veoma jednostavan, jer se učitavanje i uklanjanje procesa iz memorije jednostavno implementira. Takođe, implementacija virtuelne memorije u slučaju kontinualnog dodeljivanja fizičke memorije je veoma jednostavna. Pretpostavimo da je veličina virtuelnog prostora procesa 64KB, kao i da je procesu dodeljen kontinualni prostor u fizičkoj memoriji veličine 64KB, počev od fizičke adrese 1024. Za implementaciju virtuelne memorije, potrebno je da procesor sadrži dva posebna registra. Prvi registar, označimo ga sa B, predstavlja

bazni registar koji sadrži fizičku adresu od koje počinje kontinualni prostor u fizičkoj memoriji koji je dodeljen procesu (u našem primeru, u taj registar treba učitati 1024). Drugi registar, označimo ga sa L, predstavlja *granični registar* koji sadrži prvu fizičku adresu nakon završetka kontinualnog prostora koji je dodeljen procesu (tj. prvu adresu koja *ne* pripada našem procesu). U našem primeru, u registar L biće učitan zbir $1024+64\cdot 1024$. Translacija virtuelne adrese va u fizičku adresu fa vrši se veoma jednostavno, po formuli $fa = va + B$. Prilikom izračunavanja fizičke adrese, memorijska jedinica mora da proveri da li je dobijena fizička adresa u opsegu koji je dodeljen našem procesu. U tu svrhu je potrebno samo ispitati da li je $fa < L$. Dakle, cela memorijska jedinica će se sastojati iz jednog binarnog sabirača i jednog komparatora. Vrednosti bazne i granične adrese za svaki od aktivnih procesa se čuvaju u strukturama podataka operativnog sistema. Prilikom promene konteksta, registri B i L će biti od strane operativnog sistema automatski postavljeni na baznu i graničnu adresu procesa koji je raspoređen na izvršavanje u procesoru. Na taj način će svaki proces pristupati isključivo adresama iz opsega koji mu je dodeljen.

Dva su glavna nedostatka kontinualnog dodeljivanja fizičke memorije procesu. Prvi nedostatak je to što ovakav način dodeljivanja memorije podrazumeva da se ceo virtuelni prostor nalazi u fizičkoj memoriji u kontinualnom opsegu memorijskih adresa. To znači da će parcijalno učitavanje virtuelnog prostora procesa u cilju uštede fizičke memorije biti otežano, jer će različiti delovi virtuelnog adresnog prostora koji se zasebno po potrebi učitavaju morati da budu učitavani u susedne zone u memoriji, što često neće biti moguće jednostavno realizovati. Drugi nedostatak je pojava *fragmentacije* fizičke memorije. Naime, kada se neki proces završi, memorijski prostor koji je on zauzimao se oslobađa. Na tom mestu u memoriji sada imamo slobodan prostor koji je veliki onoliko koliko je prostora zauzimao proces koji se upravo završio. Operativni sistem će taj slobodan prostor moći da dodeli nekom novom procesu samo ako taj novi proces ne zahteva više memorije. U suprotnom, operativni sistem će morati da pronađe neki drugi slobodan prostor u memoriji, a taj prostor će ostati neupotrebljen. Kako se tokom rada računara procesi stalno pokreću i završavaju sa radom, nakon nekog vremena možemo imati veliki broj „rupa” u memoriji koje predstavljaju slobodni prostor. Sada se može lako dogoditi da za neki novi proces mi imamo dovoljno slobodnog prostora posmatrano ukupno, ali ga nemamo „u komadu”, s obzirom da je slobodni prostor fragmentisan na veliki broj delova.

Oba ova nedostatka se rešavaju upotrebom nekontinualnog dodeljivanja fizičkog memorijskog prostora procesu. U ovom scenariju, nije neophodno da se ceo virtuelni prostor procesa preslika u kontinualni deo fizičkog prostora, tj. nije neohodno da uzastopne virtuelne adrese budu preslikane u uzastopne fizičke adrese. Na ovaj način mi možemo virtuelni adresni prostor izdeliti na delove i svaki od delova učitati u posebne zone fizičke memorije. Sada naknadno učitavanje dela virtuelnog adresnog prostora u memoriju ne predstavlja problem, jer je dovoljno pronaći bilo koji deo memorije koji je slobodan. Takođe, fragmentacija više neće postojati, jer će svi slobodni delovi memorije moći da budu popunjeni delovima virtuelnog adresnog prostora odgovarajuće veličine. Sa druge strane, nekontinualno dodeljivanje fizičke memorije je znatno komplikovanije za implementaciju, s obzirom da operativni sistem mora za svaki proces čuvati informacije o svim delovima virtuelnog adresnog prostora (da li su učitani u RAM i počev od koje adrese). Memorijska jedinica procesora mora da učitava ove informacije i na osnovu njih preračunava fizičke adrese, što samu memorijsku jedinicu čini kompleksnijom.

Jedan od najčešćih načina upravljanja memorijom kod savremenih računara je zasnovan na *straničenju* (engl. *paging*). U pitanju je sistem virtuelne memorije zasnovan na nekontinualnom dodeljivanju fizičkih adresa. U ovom sistemu, svi procesi imaju virtuelni adresni prostor iste veličine. U cilju ilustracije, pretpostavimo da imamo 32-bitni virtuelni adresni prostor. To znači da su sve virtuelne adrese 32-bitne, pa je veličina virtuelnog adresnog prostora svakog procesa 4GB. Virtuelni prostor procesa se logički deli na *stranice* (engl. *page*) jednake veličine. Određenosti radi, neka su sve stranice veličine 4KB (što je 2^{12} bajtova). To znači da se ceo virtuelni prostor procesa sastoji iz 2^{20} stranica od po 2^{12} bajtova (ukupno 2^{32} bajtova). Možemo smatrati da se sada 32-bitna virtuelna adresa (va) sastoji iz dva dela: viših 20 bitova predstavljaju redni broj stranice (označimo ga sa ha), dok nižih 12 bitova (označimo ih sa la) predstavljaju *pomeraj* (engl. *offset*) u okviru stranice.

Sa druge strane, fizički adresni prostor se takođe deli na delove koji su jednake veličine kao i stranice (u našem primeru 4KB). Nazivamo ih *okviri stranica* (engl. *page frames*). Primera radi, pretpostavimo da imamo 16GB (2^{34}) fizičkog adresnog prostora. Ovaj prostor se deli na 2^{22} okvira od po 2^{12} bajtova. U svaki okvir može biti učitana jedna stranica virtuelnog adresnog prostora nekog od aktivnih procesa. Pritom, svaka stranica može biti učitana u bilo koji slobodan okvir, tj. ne postoji zahtev da susedne stranice u virtuelnom adresnom prostoru nekog procesa budu u susednim okvirima fizičkog adresnog prostora. Ovo znači da prilikom učitavanja programa u memoriju njegove stranice mogu biti razbacane po celom fizičkom adresnom prostoru na proizvoljan način. Kako bismo mogli da izvršimo prevođenje virtuelne adrese u fizičku, neophodno je da za svaku stranicu znamo u kom se okviru nalazi. U ovu svrhu, operativni sistem za svaki proces održava *tabelu stranica* (engl. *page table*). U pitanju je tabela koja sadrži po jednu stavku za svaku od 2^{20} stranica datog procesa. Stavke su jednake veličine (npr. 8 bajta) i nalaze se jedna za drugom u memoriji, počev od neke fizičke adrese koja je poznata operativnom sistemu. U procesoru postoji poseban registar (označimo ga sa PTA - *page table address*) koji sadrži *fizičku* adresu tabele stranica za proces koji se trenutno izvršava u procesoru (prilikom zamene konteksta,

u PTA registar se učitava odgovarajuća adresa). Svaka stavka sadrži, između ostalog, informaciju o tome da li je odgovarajuća stranica prisutna u fizičkoj memoriji i, ako jeste, u kom okviru. Proces prevođenja virtuelne u fizičku adresu se sada obavlja na sledeći način: najpre se na osnovu višeg dela virtuelne adrese ha odredi redni broj stranice virtuelnog adresnog prostora. Zatim se pomoću PTA registra pristupa stavci tabele stranica sa tim rednim brojem (tj. pristupa se 8-bajtnom prostoru na adresi $PTA + 8 \cdot ha$). Ova stavka se učitava u memorijsku jedinicu i iz nje se utvrđuje da li je stranica učitana u fizičku memoriju ili ne. Ako jeste, tada se iz stavke pročita 22-bitni redni broj okvira u fizičkoj memoriji gde se stranica nalazi. Na ovo 22-bitno polje se dopisuje niži deo virtuelne adrese la, čime se dobija 34-bitna fizička adresa podatka u memoriji (ovo je zato što je pomeraj u okviru jednak pomeraju u stranici). Ukoliko se ispostavi da stranica nije učitana u fizičku memoriju, dolazi do prekida rada programa (tzv. *greška straničenja*, engl. *page fault*). Kontrola se predaje operativnom sistemu koji sa hard diska učitava traženu stranicu u neki slobodan okvir u fizičkoj memoriji. Ažurira se odgovarajuća stavka u tabeli stranica, a zatim se kontrola vraća procesu koji ponovo pokušava da izvrši istu instrukciju. Ovom prilikom prevođenje uspeva, jer se stranica sada nalazi u memoriji. Ako prilikom učitavanja nove stranice nema slobodnih okvira, tada operativni sistem bira „žrtvu" – stranicu koju će izbaciti iz svog okvira i iskopirati je na hard disk, kako bi se okvir oslobodio za novu stranicu. Pritom, izbačena stranica ne mora pripadati istom procesu. Odabir prave „žrtve" nije ni malo jednostavan, jer je cilj smanjiti broj skupih grešaka straničenja u budućnosti, što zahteva anticipaciju budućeg korišćenja stranica koje se nalaze u okvirima u memoriji.

Upravljanje ulazno-izlaznim uređajima

Ulazno-izlazni uređaji omogućavaju komunikaciju računara sa korisnikom, kao i sa drugim uređajima i računarima. Tradicionalni ulazni uređaji za komunikaciju sa korisnikom su tastatura i miš, a izlazni je monitor. U novije vreme, sve se češće koriste i ekrani osetljivi na dodir, kao vrsta ulazno-izlaznog uređaja. Pored toga, u ulazno-izlazne uređaje spadaju različite vrste štampača i skenera, kao i multimedijalni uređaji poput zvučnika, mikrofona, kamera i sl. Za komunikaciju sa drugim računarima koriste se različite vrste mrežnih uređaja, poput mrežnih i bežičnih (engl. *wireless*) kartica. Najzad, posebnu vrstu ulazno-izlaznih uređaja čine i *spoljne memorije* koje se koriste za skladištenje podataka i softvera (poput hard i SSD diskova, CD/DVD diskova, fleš memorija i sl.).

Procesor računara komunicira sa ulazno-izlaznim uređajima podsredstvom *ulazno-izlaznih kontrolera*. Kontroler se povezuje na magistralu i procesor sa njim komunicira na veoma sličan način kao i sa operativnom memorijom, razmenjujući *podatke*, *komande* i *statusne informacije* putem magistrale. Kontroler se, sa druge strane, povezuje sa samim uređajem na način koji je prilagođen prirodi tog uređaja.

Skup komandi koje razume ulazno-izlazni kontroler veoma zavisi od vrste uređaja, proizvođača, pa čak i samog modela uređaja. Otuda bi direktna komunikacija sa ulazno-izlaznim kontrolerima od strane korisničkih programa bila veoma teška, jer bi programer morao da poznaje detalje komandnog jezika, kao i da obezbedi podršku za veliki broj različitih tipova i modela uređaja. Takođe, direktna komunikacija sa kontrolerima predstavlja i bezbednosni izazov, jer bi pogrešnom upotrebom uređaj mogao i da se ošteti ili zloupotrebi. Zbog toga se komunikacija sa ulazno-izlaznim uređajima prepušta operativnom sistemu, a njegov deo zadužen za taj posao naziva se *ulazno-izlazni podsistem*. Ovaj sistem se sastoji iz dva sloja. Na nižem sloju nalaze se upravljački programi zaduženi za direktnu komunikaciju sa kontrolerima. Ovi programi su poznati i kao *drajveri* (engl. *device drivers*) i u njima su implementirani svi detalji komandnog jezika konkretnog uređaja. Za svaki tip i model uređaja operativni sistem mora biti snabdeven odgovarajućim drajverom. Vrlo često, drajveri se isporučuju od strane proizvođača hardvera, zajedno sa samim uređajem. Na višem sloju implementiran je interfejs ka drugim delovima operativnog sistema i korisničkim programima, a koji je nezavisan od konkretnog tipa i modela uređaja. Ovim interfejsom se obezbeđuje *apstrakcija hardvera*, jer se drugim delovima operativnog sistema i korisničkim programima obezbeđuje unifikovani pristup različitim uređajima, što značajno olakšava korišćenje samih uređaja.

U okviru ulazno-izlaznog podsistema implementirani su različiti *baferi*. Bafer predstavlja strukturu podataka koja može skladištiti podatke koji se prenose između dve strane u komunikaciji – u ovom slučaju između ulazno-izlaznog kontrolera i operativnog sistema. Uloga bafera je da amortizuje razliku u brzini između dve strane koje komuniciraju. Na primer, kada neka aplikacija šalje podatke izlaznom uređaju (npr. štampaču), ona ih prosleđuje operativnom sistemu koje te podatke dalje prosleđuje drajveru za odgovarajući uređaj, a ovaj ih šalje samom kontroleru, putem magistrale, koristeći odgovarajući komandni jezik. Međutim, ako je brzina slanja podataka prevelika, kontroler neće moći da ih prihvati, te postoji opasnost od gubitaka podataka. Da se to ne bi desilo, operativni sistem podatke privremeno smešta u bafer. Podaci se zatim iz bafera prosleđuju drajveru onom brzinom kojom uređaj može da ih prihvati. Ukoliko se bafer prepuni, tada će operativni sistem prestati da prihvata podatke od aplikacije, dok se u baferu ne napravi dovoljno mesta.

Upravljanje podacima

Podaci se na računaru skladište u spoljnim memorijama (poput hard i SSD diskova, fleš memorija i sl.). Računar ove uređaje vidi kao ulazno-izlazne uređaje i sa njima komunicira putem odgovarajućih kontrolera (npr. SATA kontroler se koristi za komunikaciju sa hard i SSD diskovima, dok se USB kontroler koristi za komunikaciju sa USB fleš memorijama). Svaka spoljna memorija se može logički posmatrati kao niz adresibilnih lokacija fiksirane veličine. Na primer, hard diskovi se mogu posmatrati kao nizovi *sektora*, tipične veličine 512 bajtova, čije adrese počinju od nule. Putem ulazno-izlaznog podsistema, moguće je vršiti očitavanje ili upis pojedinačnih sektora hard diska i na taj način baratati sa skladištenim podacima.

Iako bi, teorijski, korisnički programi mogli da sa spoljnim memorijama na ovaj način razmenjuju sirove podatke (tj. konkretne bajtove, brojeve, tekst i sl.), tako nešto bi vrlo brzo dovelo do haosa, jer bi vrlo teško bilo pronaći podatke, kao i sprečiti da aplikacije greškom oštete podatke upisane na uređaju.

Zbog toga se podaci na spoljnim memorijama organizuju na način koji obezbeđuje konzistentan, efikasan i bezbedan pristup. Ovo se postiže podsredstvom specijalnih struktura podataka koje čine *sistem datoteka* (engl. *file system*). Pod *datotekom* ili *fajlom* (engl. *file*) podrazumevamo skup logički povezanih podataka koji čine jednu celinu. Na primer, fajl može biti neki tekstualni dokument, slika, video zapis, ali i računarski program. Svaki fajl ima svoje ime koje ga identifikuje, kao i sadržaj koji je predstavljen nizom bajtova, u formatu koji zavisi od specifičnog tipa fajla. Na nekim fajl sistemima, tip fajla može biti određen *ekstenzijom*, tj. specifičnim nastavkom imena (npr. *.txt*, *.doc*, *.jpg*, *.exe*, i sl.), dok na drugim sistemima naziv fajla nema uticaja na tip, već se tip prepoznaje na drugačiji način. Pored imena, tipa i sadržaja, fajl može imati i niz drugih karakteristika poput veličine (u bajtovima), vlasništva, prava pristupa, datuma poslednje modifikacije i sl.

Sistem datoteka obično podrazumeva dva aspekta – *logički* i *fizički*. Logički aspekt predstavlja način na koji se podaci koji se nalaze skladišteni u okviru sistema datoteka predstavljaju korisniku. Danas su uobičajeni *hijerarhijski* sistemi datoteka, kod kojih se fajlovi organizuju u *direktorijume* ili *fascikle*. Svaki direktorijum može sadržati fajlove, kao i druge direktorijume (koje nazivamo *poddirektorijumi*). Na vrhu hijerarhije nalazi se *koreni direktorijum* (engl. *root directory*) koji sadrži (posredno) sve podatke koji se nalaze u okviru tog sistema datoteka. Direktorijumi kao i fajlovi imaju svoja imena. Ovakav pristup omogućava korisniku da svoje podatke organizuje tako što ih razvrstava na kategorije i podkategorije, čime je omogućeno lakše pronalaženje fajlova.[3]

Fizički aspekt podrazumeva na koji način se podaci fizički skladište u spoljnoj memoriji. Ovo je obično sakriveno od korisnika i odgovornost je operativnog sistema. Setimo se da se memorijski prostor spoljne memorije može posmatrati kao niz adresibilnih jedinica. Uobičajeno je da sistem datoteka na početku tog memorijskog prostora skladišti meta-informacije o sebi (tip sistema datoteka, verzija, i sl.) kao i strukture podataka koje sadrže informacije o fajlovima i direktorijumima i omogućavaju pronalaženje sadržaja konkretnih fajlova u okviru fajl sistema. U nastavku prostora nalaze se sami fajlovi, tj. njihovi sadržaji.

Postoji više tipova sistema datoteka koji se danas aktivno koriste. Neki od njih su *FAT*, *NTFS*, *ext4*, *XFS*, *ReiserFS* i sl. Svaki operativni sistem podržava određeni skup tipova sistema datoteka i može pristupati samo sistemima datoteka tih tipova. Sistemi *FAT* i *NTFS* su razvijeni za operativne sisteme *DOS* i *Windows* od strane kompanije Majkrosoft, ali su podržani i od strane *GNU/Linux* operativnog sistema. Sistemi *ext4* i *ReiserFS* su razvijeni za *Linux* i podrazumevano nisu podržani od strane *Windows* operativnog sistema (mada postoji mogućnost instalacije dodatnog sistemskog softvera koji omogućava pristup ovakvim sistemima datoteka od strane *Windows* korisnika).

Da bi spoljna memorija mogla da se koristi za skladištenje podataka, potrebno je da se na njoj inicijalizuje sistem datoteka. Ovo se radi pomoću sistemskih alata koji se isporučuju sa samim operativnim sistemom. Postupak kreiranja sistema datoteka poznat je i kao *formatiranje*. Nakon formatiranja, korisnik dobija pristup sistemu datoteka koji je inicijalno prazan, tj. u njemu nema fajlova niti direktorijuma, izuzev korenog direktorijuma. Nakon toga, korisnik može kreirati fajlove i direktorijume i na taj način organizovati svoje podatke.

Pojedini uređaji poput hard i SSD diskova se mogu od strane operativnog sistema logički posmatrati kao više nezavisnih spoljnih memorija. Disk se može logički podeliti na *particije* – međusobno disjunktne celine, pri čemu se svaka particija sastoji iz skupa susednih sektora. Informacije o particionisanju (tj. od kog do kog sektora se prostire koja particija) se obično nalaze u početnim sektorima diska koji nisu deo ni jedne particije. Sada se sistemi datoteka mogu nezavisno kreirati u svakoj od particija.

Lokacija svakog fajla ili direktorijuma u okviru hijerarhijskog sistema datoteka se opisuje *putanjom*, koja može biti *apsolutna* i *relativna*. Apsolutna putanja se dobija navođenjem svih direktorijuma u stablu prateći put od korenog direktorijuma do datog fajla ili direktorijuma. Na primer, na operativnom sistemu *UNIX*, putanja `/etc/apache/httpd.conf` predstavlja apsolutnu putanju koja određuje lokaciju fajla `httpd.conf` (u okviru direktorijuma `apache` koji je poddirektorijum direktorijuma `etc` koji se nalazi u okviru korenog direktorijuma

[3] Zanimljivo je da mnogi korisnici računara ne koriste ovu mogućnost, već sve fajlove smeštaju u isti direktorijum (tipično na „radnu površinu" korisničkog interfejsa operativnog sistema). Kada se sledeći put budete mučili da na radnoj površini pronađete nešto što Vam treba, setite se da ste mogli da fajlove razvrstate u direktorijume, što bi Vam verovatno olakšalo pretragu.

`/`). Sa druge strane, relativna putanja predstavlja putanju nekog fajla ili direktorijuma u odnosu na neki drugi direktorijum u stablu direktorijuma (tipično u odnosu na *tekući direktorijum*, tj. radni direktorijum koji je pridružen procesu koji pristupa sistemu datoteka). Na primer, ako je tekući direktorijum `/etc/`, tada putanja `apache/httpd.conf` predstavlja relativnu putanju fajla `httpd.conf` u odnosu na direktorijum `/etc/`.

Uobičajeno je da sistemi datoteka podržavaju neku vrstu zaštite podataka od neovlašćenog korišćenja, obično kroz sistem *prava pristupa*. U okviru sistema datoteka se, sa određenim stepenom granulacije, mogu definisati pojedinačna prava određenih korisnika i grupa korisnika nad datim fajlom ili direktorijumom (poput prava čitanja, modifikacije, kreiranja, brisanja, pokretanja programa i sl.). Operativni sistem je odgovoran da spreči svaki neovlašćen pristup fajlovima i direktorijumima u okviru sistema datoteka.

Komunikacija između procesa

Često postoji potreba da procesi međusobno komuniciraju, razmenjuju podatke i sinhronizuju se u cilju zajedničkog obavljanja nekog kompleksnog zadatka. U tu svrhu, operativni sistemi obično obezbeđuju različite mehanizme interprocesne komunikacije. Najjednostavniji mehanizmi podrazumevaju slanje jednostavnih poruka kojima se proces obaveštava o određenom događaju ili podstiče na neku akciju (poput *UNIX signala*). Složeniji mehanizmi komunikacije omogućavaju razmenu proizvoljnih podataka putem kreiranog kanala (primer takvih mehanizama su *cevi* (engl. *pipe*) i *priključci* (engl. *socket*) na *UNIX* sistemima). Ovi kanali mogu biti *jednosmerni* (engl. *half-duplex*) ili *dvosmerni* (engl. *full-duplex*). Ovi mehanizmi se tipično koriste za komunikaciju sa sistemskim servisima koje koriste različiti aplikativni programi (poput sistema za pristup štampačima), ali i za komunikaciju aplikacija sa korisničkim interfejsom. Najzad, kao naročito efikasan vid komunikacije između procesa može se koristiti *deljena memorija* (engl. *shared memory*). Ovaj način komunikacije podrazumeva da dva procesa dele neki zajednički deo virtuelnog adresnog prostora. Samim tim, ono što jedan proces upiše u tu deljenu memoriju automatski je vidljivo drugom procesu u okviru njegovog virtuelnog adresnog prostora. Na ovaj način procesi mogu brzo i efikasno razmenjivati velike količine podataka.

Na kraju, napomenimo da procesi koji komuniciraju ne moraju se obavezno izvršavati na istom računaru. Zahvaljujući umrežavanju, omogućeni su mehanizmi komunikacije između procesa koji se izvršavaju na različitim računarima koji su međusobno povezani. U tu svrhu se koriste *mrežni protokoli* koji definišu pravila komunikacije, a koji su nezavisni od konkretnog hardvera ili operativnog sistema.

Komunikacija sa korisnikom

Da bi korisnik mogao da koristi svoj računar, neophodno je da može da pokreće svoje aplikacije, upravlja njihovim radom, kao i da koristi resurse svog računara, poput podataka i ulazno-izlaznih uređaja. U tu svrhu, svaki operativni sistem sadrži komponentu koja se zove *korisnički interfejs* (engl. *user interface* (UI)). Korisnički interfejs predstavlja skup uslužnih programa koji su povezani sa standardnim ulazno-izlaznim uređajima poput tastature, miša i monitora, preko kojih mogu komunicirati sa korisnikom. Ovi programi se pokreću prilikom pokretanja operativnog sistema i prijavljivanja korisnika na sistem, čime se kreiraju odgovarajući procesi koji su aktivni dokle god je korisnik prijavljen. Korisnik može korisničkom interfejsu izdavati *komande* kojima od operativnog sistema zahteva neku akciju. Te akcije mogu podrazumevati pristup nekom fajlu u okviru fajl sistema, promenu tekućeg direktorijuma, kao i pokretanje programa. Kada se izda odgovarajuća komanda, tada proces korisničkog interfejsa odgovara na tu komandu tako što za korisnika izvršava odgovarajuću akciju. Na primer, ako je korisnik želeo da pokrene neki program, proces korisničkog interfejsa će kreirati novi proces i u okviru njega pokrenuti željeni program kome će predati kontrolu nad standardnim ulazom i izlazom, čime će nadalje biti omogućena komunikacija korisnika sa pokrenutim programom. Kada se program završi, kontrola će biti vraćena korisničkom interfejsu koji će spremno čekati novu komandu korisnika.

Tradicionalni korisnički interfejs je bio *tekstualni*, u smislu da su se komande zadavale u tekstualnoj formi, što je podrazumevalo da korisnik poznaje sintaksu komandnog interfejsa. Poruke koje je korisnički interfejs izdavao korisniku su takođe bile tekstualne. U novije vreme, tekstualni korisnički interfejsi su ustupili mesto *grafičkim korisničkim interfejsima* (engl. *graphical user interface* (GUI)). Grafički interfejs podrazumeva da korisnik izdaje akcije pomoću miša, pritiskom na različite grafičke elemente koji su prikazani na ekranu (poput dugmadi, menija, ikona i sl.). Korisnički intefejs, kao i aplikacije koje se u okviru njega pokreću, može korisniku takođe saopštavati poruke koristeći grafičke elemente (prozore, dijaloge, slike, animacije i sl.). Ovakav način korišćenja je znatno jednostavniji i udobniji, a naročito je prilagođen osobama skromnog tehničkog znanja. Ipak, naglasimo da tekstualni korisnički interfejsi i dalje nude mnoge mogućnosti koje grafički interfejsi nemaju. Otuda tekstualni korisnički interfejs i dalje ostaje prvi izbor naprednijih korisnika računara.

1.1.5 Hardverska podrška operativnim sistemima

Da bi operativni sistemi mogli da implementiraju svoje funkcionalnosti opisane u prethodnim odeljcima, neohodna je izvesna podrška od strane hardvera (pre svega od strane procesora). U ovom odeljku opisujemo neke osnovne hardverske funkcionalnosti neophodne za implementaciju operativnih sistema.

Nivoi privilegija procesora

Već je rečeno da korisnički programi ne bi trebalo da direktno pristupaju hardveru računara, već isključivo putem interfejsa operativnog sistema. Takođe, postoje registri procesora (poput registra PTA koji sadrži adresu tabele stranica) koji služe isključivo za implementaciju funkcija operativnog sistema, pa ne bi bilo dobro da korisnički programi mogu da im pristupaju. Kako bi se fizički sprečilo da korisnički programi direktno pristupaju resursima koji im nisu namenjeni, procesor obično podržava više različitih *nivoa privilegija* (engl. *privilege level*). U svakom trenutku, procesor se nalazi na jednom od mogućih nivoa privilegija. Svaki od nivoa podrazumeva određeni skup registara i instrukcija procesora koje se mogu koristiti na tom nivou. Ako program pokuša da pristupi registru ili izvrši instrukciju koja nije u tom skupu, dolazi do prekida rada programa i predaje kontrole odgovarajućoj funkciji operativnog sistema koja je za takvu situaciju predviđena (što obično dovodi do likvidacije procesa). Na primer, na arhitekturi *x86* (i *x86-64*) postoje četiri nivoa privilegija, označeni brojevima od 0 do 3. Nivo 0 je najviši nivo privilegija i u njemu su dostupne sve instrukcije i svi registri procesora. Svaki sledeći nivo podrazumeva manji skup instrukcija i registara koji su dostupni. Nivo 3 apsolutno sprečava svaki neovlašćen pristup hardveru računara kao i specijalnim registrima procesora predviđenim za implementaciju funkcija operativnog sistema. Po uključivanju računara, procesor se podrazumevano nalazi u najvišem nivou privilegija (nivo 0). Posebnom instrukcijom procesor se može prebaciti u neki od nižih nivoa po želji. Sa druge strane, kada se nalazimo na nižem nivou privilegije, ne postoji instrukcija koja bi nam omogućila da se vratimo na viši nivo. To znači da proces koji se izvršava sa nižim privilegijama (poput korisničkog procesa) neće moći da tek tako dobije više privilegije koje bi mu omogućile pristup osetljivim delovima sistema. Ono što procesor obično obezbeđuje jesu specijalni mehanizmi privremenog i kontrolisanog prelaska na viši nivo privilegija radi obavljanja sistemskih poslova (poput izvršenja sistemskih poziva operativnog sistema ili obrade prekida). Ovi mehanizmi su rezervisani isključivo za pozivanje kôda koje obezbeđuje jezgro operativnog sistema. Drugim rečima, ne postoji način da korisnički program izvrši proizvoljan kôd u privilegovanom režimu rada.

Za implementaciju operativnog sistema obično je dovoljno da procesor podržava dva nivoa privilegija: jedan koji obezbeđuje pristup svim resursima procesora (poput nivoa 0 na x86 arhitekturi) i jedan koji uskraćuje pristup svim sistemski osetljivim resursima procesora (poput nivoa 3 na x86 arhitekturi). Na ovom prvom se obično izvršava isključivo kôd jezgra operativnog sistema. Nazivamo ga i *nivo jezgra* (engl. *kernel level*). Na ovom drugom se izvršavaju svi korisnički programi (ne samo aplikativni programi, već i sistemski programi i delovi operativnog sistema koji ne zahtevaju privilegovani režim, poput korisničkog interfejsa). Nazivamo ga i *korisnički nivo* (engl. *user level*).

Podrška za upravljanje memorijom

Većina savremenih procesora podržava mehanizam straničenja za implementaciju virtuelne memorije. U tu svrhu, postoji memorijska jedinica koja vrši automatsko prevođenje virtuelnih u fizičke adrese, na osnovu tabele stranica čija se fizička adresa nalazi u posebnom registru (koji smo mi označavali kao PTA). Osnovni parametri koji definišu mehanizam straničenja jesu veličina virtuelnog i fizičkog adresnog prostora, kao i veličina stranica i okvira. Mnogi procesori podržavaju više različitih varijanti straničenja, kada su u pitanju vrednosti ovih parametara. Tako, na primer, x86 arhitektura podržava dve različite veličine stranica: 4KB i 4MB. Izbor ovih vrednosti parametara može se vršiti uključivanjem odgovarajućih bitova u specijalnom kontrolnom registru procesora, dostupnom samo iz privilegovanog nivoa. Ovo postavljanje parametara vrši operativni sistem prilikom svoje inicijalizacije.

Struktura i raspored bitova stavki tabele stranica je takođe hardverski definisana i operativni sistem se mora sa tim uskladiti. Pored rednog broja okvira fizičke memorije koji datu stranicu sadrži, stavka tabele stranica obično sadrži i druge bitove koji, između ostalog, određuju i nivo privilegija potreban za pristup toj stranici. Na primer, moguće je sprečiti pristup toj stranici od strane korisničkog programa, ako ona sadrži neke osetljive podatke operativnog sistema. Slično, moguće je podesiti da stranica bude nepromenljiva (engl. *read-only*), što je korisno za stranice koje sadrže programski kod. Na ovaj način hardver omogućava fino podešavanje prava pristupa memoriji od strane procesa. S obzirom da se i ulazno-izlazni kontroleri obično mapiraju u adresni prostor procesora, na ovaj način se može obezbediti i kontrola pristupa registrima ulazno-izlaznih kontrolera.

Sistem prekida

Sistem prekida predstavlja mehanizam koji omogućava zaustavljanje programa bez njegove volje u proizvoljnom trenutku i prenošenje kontrole operativnom sistemu. Ovo je vrlo važno u kontekstu operativnih sistema, kako bi on mogao da obavlja svoje funkcije. Bez sistema prekida ne bi bilo moguće zaustaviti program kada on jednom krene da se izvršava, osim da isključimo računar ili da sačekamo da on sam završi sa svojim radom. Potreba za prekidanjem programa javlja se u različitim situacijama.

Prvi slučaj je vezan za obradu asinhronih događaja izazvanih interakcijom sa ulazno-izlaznim uređajima računara. Na primer, takvi događaji nastaju pritiskom tastera na tastaturi, pomeranjem miša ili priključivanjem uređaja na USB podsistem. Ove događaje obično izaziva korisnik i mogu se desiti u bilo kom trenutku. Kada se dese, potrebno ih je obraditi u što kraćem roku od strane odgovarajućeg drajvera uređaja u okviru ulazno-izlaznog podsistema. Dva su razloga zbog čega je obrada ovakvih događaja hitna. Prvo, korisnik očekuje brzu reakciju sistema na svoje komande, tj. zelimo da sistem bude interaktivan i komunicira sa korisnikom u realnom vremenu. Drugo, memorijski prostor koji ulazno-izlazni kontroleri sadrže je obično ograničen i podaci pristigli od ulaznog uređaja ne mogu biti dugo sačuvani u registrima kontrolera, tj. postoji mogućnost njihovog gubitka. Zbog toga ih je potrebno što pre prebaciti u memoriju i smestiti u odgovarajuće strukture operativnog sistema koji će ih dalje obrađivati. Sistem prekida omogućava ulazno-izlaznim uređajima da privuku pažnju procesora kada se takav događaj desi, tako što aktiviraju poseban *signal prekida* koji se dovodi do procesora podsredstvom posebnog ulaznog priključka. Procesor nakon svake instrukcije proverava da li je ovaj signal aktiviran i, ako jeste, automatski zaustavlja dalje izvršavanje tekućeg procesa, pamti stanje registara u kontrolnom bloku procesa i zatim poziva unapred datu proceduru operativnog sistema koja je zadužena za *obradu prekida*. Prilikom izvršavanja ove procedure, procesor obično prelazi u privilegovani režim rada (ovo je često jedini način da se iz neprivilegovanog pređe u privilegovani režim rada, a dešava se automatski prilikom izazivanja prekida). Nakon obrade prekida, procesor se vraća u neprivilegovani režim rada i vraća kontrolu prekinutom procesu, tako što iz njegovog kontrolnog bloka kopira sačuvane vrednosti registara procesora (uključujući i programski brojač). Opisani mehanizam prekida spada u tzv. *hardverske prekide*, jer ih izaziva hardver, asinhrono, kao reakciju na spoljni događaj.

Drugi slučaj vezan je za implementaciju algoritma raspoređivanja procesa, u slučaju da algoritam podrazumeva prekidanje (za potrebe implementacije podele vremena). Tada je potrebno pustiti proces da se izvršava a zatim ga prekinuti kada istekne vremenski kvantum koji mu je dodeljen, kako bi drugi proces mogao da dobije priliku da se izvršava. Opisani proces je takođe omogućen zahvaljujući hardverskim prekidima. Obično u računarskom sistemu postoji poseban kontroler koji se naziva *programibilni tajmer* (engl. *programmable interval timer* (PIT)). Ovaj uređaj se može posebnim komandama programirati tako da generiše periodični signal za prekid u ravnomernim intervalima. Ovo obično obavi operativni sistem u fazi inicijalizacije. Zahvaljujući ovom periodičnom signalu koji pristiže u ravnomernim intervalima garantovano je da će programi biti prekidani, a kontrola biti vraćana operativnom sistemu nakon isteka programiranog vremenskog kvantuma. Operativni sistem obrađuje ovaj prekid tako što pokreće algoritam raspoređivanja i određuje sledeći proces koji treba da nastavi sa izvršavanjem.

Treći slučaj je mehanizam implementacije sistemskih poziva. Setimo se da se sistemskim pozivima zahteva od jezgra sistema da za potrebe i u ime pozivajućeg procesa obavi neku sistemsku funkciju ili obezbedi neki resurs procesu. Ove funkcionalnosti mora obavljati jezgro sistema i mora ga obavljati u privilegovanom režimu rada. Međutim, korisnički proces se izvršava u neprivilegovanom režimu, tako da je neophodno obezbediti kontrolisani prelazak u privilegovani režim prilikom poziva kôda jezgra operativnog sistema. Ovo se obavlja tako što se posebnom instrukcijom procesora namerno izazove prekid. Ovakvi prekidi se nazivaju *softverski prekidi*, jer ih izaziva sam programski kod procesa koji se trenutno izvršava. Dakle, proces sam sebe namerno prekida da bi prepustio kontrolu operativnom sistemu i omogućio prelazak u privilegovani režim rada. Operativni sistem tada izvršava sistemski poziv i nakon toga vraća kontrolu samoprekinutom procesu (vraćajući pritom procesor u neprivilegovani režim rada).

Četvrti slučaj predstavljaju situacije u kojima neka instrukcija procesa iz nekog razloga ne može uspešno da se izvrši. Razlog može da bude nedefinisana operacija (poput deljenja nulom), nedozvoljeni pristup nekom resursu (npr. pristup registru ili pozivanje instrukcije koja nije dostupna u trenutnom nivou privilegija), ili nedostupnost nekog resursa (poput ranije opisane greške straničenja). U takvim situacijama se automatski generiše prekid, a kontrola se predaje operativnom sistemu koji u privilegovanom režimu rada pokušava da obradi nastalu grešku. Ovakvi prekidi poznati su i pod nazivom *izuzetci* (engl. *exceptions*). Neki izuzetci se mogu uspešno razrešiti (poput greške straničenja), nakon čega se kontrola vraća prekinutom procesu. Neki drugi se obično ne mogu razrešiti (poput pristupa nedozvoljenom resursu), što za rezultat obično ima likvidaciju procesa.

Elektronska verzija (2024)

1.2 Operativni sistemi zasnovani na UNIX-u

Operativni sistem *UNIX* (čita se *juniks*) nastao je u Belovim laboratorijama davne 1969. godine. Originalni autor bio je Ken Tompson[4]. Prve verzije UNIX-a programirane su na asemblerskom jeziku računara PDP-11 i samo su se na tom računaru mogle prevoditi i izvršavati. Par godina kasnije, Tompsonov kolega iz Belovih laboratorija Denis Riči razvio je programski jezik C koji je, za razliku od drugih programskih jezika višeg nivoa koji su postojali u to vreme, bio pogodan za sistemsko programiranje, s ozbirom da je omogućavao pristup memorijskim adresama i hardveru na niskom nivou. Operativni sistem UNIX je tada ponovo isprogramiran na programskom jeziku C, što je omogućilo njegovo prevođenje i instalaciju na drugim hardverskim platformama. Tako je UNIX postao prvi operativni sistem koji je nadživeo računar za koji je inicijalno razvijen i zahvaljujući tome uspeo da u različitim oblicima preživi do današnjih dana (u trenutku pisanja ovih redova, to je punih 55 godina!).

Zahvaljujući fleksibilnim licencama pod kojima je inicijalno bio objavljen, UNIX se uspešno širio u okviru stručne i akademske zajednice. Različiti autori su ga dalje unapređivali, a nastajale su i potpuno nove verzije ovog sistema, među kojima je najpoznatija bila verzija nastala na univerzitetu u Berkliju, poznata kao *BSD UNIX* (engl. *Berkeley Software Distribution*). Neki derivati BSD sistema (poput sistema *FreeBSD*, *OpenBSD*, *NetBSD* i sl.) se i danas aktivno koriste na savremenim računarima.

Početkom 80tih godina XX veka UNIX licence postaju restriktivnije, te slobodni razvoj i upotreba ovog sistema od strane zajednice postaje otežana. Počinju da dominiraju komercijalne varijante sistema, poput HP-UX (kompanija HP), AIX (kompanija IBM), Solaris (Kompanija Sun) i Xenix (kompanija Majkrosoft). Kao protivteža ovakvom scenariju nastaje projekat *GNU* (engl. *GNU is Not Unix* (GNU)) sa ciljem razvoja operativnog sistema kompatibilnog sa UNIX-om koji će garantovano zauvek biti distribuiran pod slobodnom licencom, eliminišući time zavisnost od komercijalnih interesa kompanija koje su stajale iza drugih verzija UNIX-a. U okviru GNU projekta su najpre razvijeni programerski alati (editor, prevodilac, debager, linker, i sl.), kao i kompletno korisničko okruženje operativnog sistema (biblioteke, korisnički interfejs, osnovni alati za rad sa sistemom datoteka i sl.). Ono što je nedostajalo bio je najkomplikovaniji deo operativnog sistema – jezgro. Taj deo upotpunjen je pojavom *Linux* jezgra koji je početkom 90tih godina razvio finski programer Linus Torvalds. Ugradnjom Linux jezgra u GNU okruženje nastaje *GNU/Linux* operativni sistem[5]. Postoje različite *distribucije* GNU/Linux operativnog sistema koje se međusobno razlikuju pre svega po izboru softvera koji dolazi sa sistemom, kao i po inicijalnoj konfiguraciji. Neke od najpoznatijih su *Ubuntu*, *Mint*, *Debian*, *Fedora*, *Slackware* i sl.

1.3 Komandno okruženje operativnog sistema UNIX

Tradicionalni korisnički interfejs za UNIX-zasnovane operativne sisteme je *ljuska* (engl. *shell*). U pitanju je tekstualni korisnički interfejs koji zahvaljujući svojoj bogatoj sintaksi nudi ogromne mogućnosti korisniku. Ljuska omogućava jednostavno pokretanje komandi sa zadavanjem opcija komandne linije, ali i složenije načine pokretanja programa koji koriste preusmeravanje standardnog ulaza i izlaza, kao i nadovezivanje više programa koristeći *cevi* (engl. *pipe*). Ljuska takodje uključuje i sopstveni programski jezik koji omogućava korisniku da isprogramira složene postupke za rešavanje kompleksnijih zadataka, koji uključuju pokretanje većeg broja programa, rad sa fajlovima i sl. Zahvaljujući ljusci i njenim mogućnostima, korisnik može da kombinujući brojne, mahom jednostavne alate koji su mu dostupni obavlja veoma složene administrativne i druge poslove.

1.3.1 Pokretanje i prvi koraci

Pre pojave grafičkih korisničkih interfejsa, ljuska je bila podrazumevani korisnički interfejs na UNIX-zasnovanim sistemima. Otuda se ljuska pokretala automatski, nakon prijave korisnika na sistem. U današnje vreme, većina UNIX-zasnovanih sistema su podešeni tako da grafičko okruženje bude podrazumevano. Ipak, većina iskusnih UNIX korisnika će pre ili kasnije poželeti da se dokopa ljuske, kako bi mogli da obavljaju različite rutinske poslove. U tu svrhu se obično pokreće neka aplikacija sa grafičkim interfejsom koja predstavlja *emulator terminala* (često se zove *Terminal* ili *Console*).[6] Pokretanjem ove aplikacije otvara se prozor u okviru koga se pokreće i prikazuje ljuska. Prozor emulatora terminala zauzima deo ekrana koji kada je u fokusu preuzima vezu sa tastaturom i omogućava korisniku da komunicira sa ljuskom koja je u njemu pokrenuta, emulirajući tako fizički terminal.

[4] Pored UNIX-a, ime Kena Tompsona se vezuje i za UTF-8 kodiranje teksta, kao i za tzv. *Tompsonove automate* u teoriji formalnih jezika.

[5] Vrlo često ga nazivamo i samo *Linux*, iako je to pogrešno, s obzirom da je Linux samo jezgro koje je bez ostalih komponenti razvijenih u okviru GNU projekta potpuno neupotrebljivo.

[6] Pojam *terminal* tradicionalno označava tastaturu i monitor pomoću kojih jedan korisnik komunicira sa računarom. Računar može imati više terminala, što omogućava istovremeni rad više korisnika.

Postoji više različitih vrsta ljuski koje su međusobno veoma slične, a razlikuju se pre svega u nekim detaljima sintakse. Na sistemu može biti instalirano više ljuski, a korisnik može odabrati svoju podrazumevanu ljusku koja će se automatski pokretati. Neke od poznatijih ljuski su *sh* (i iz nje izvedena ljuska *bash*), *csh*, *tcsh* i sl. Kako je *bash* podrazumevana ljuska koji se koristi pod GNU/Linux sistemima, sve što u nastavku sledi odnosiće se upravo na *bash* ljusku.

Odziv ljuske

Bez obzira na koji način je ljuska pokrenuta, korisniku se najpre prikazuje *odziv* (engl. *prompt*) ljuske, poput:

```
pera@desktop:programiranje$
```

Znak $ na kraju odziva naziva se *odzivni znak*. Tekst koji stoji ispred njega sadrži dodatne informacije i može se konfigurisati po želji korisnika. U našem primeru, ovaj tekst sadrži redom ime korisnika (`pera`), ime računara (`desktop`) i ime tekućeg direktorijuma (`programiranje`).

Jednostavno pokretanje programa. Putanja i promenljive okruženja

U nastavku odziva od korisnika se očekuje da unese komandu. Pritiskom na taster ENTER, komanda se analizira od strane ljuske, i ako ne postoje sintaksne greške, komanda se pokreće. Neke jednostavne komande su ugrađene u samu ljusku i izvršavaju se od strane samog procesa ljuske. Ipak, većina komandi podrazumeva pokretanje nekog drugog programa sa diska računara. U tom slučaju će proces ljuske kreirati novi proces i u okviru njega pokrenuti dati program. Uobičajena sintaksa pokretanja komande je:

```
naziv_programa arg1 arg2 ...
```

Pritom, `naziv_programa` predstavlja ime programa koji treba pokrenuti, dok su `arg1`, `arg2`, ... opcioni *argumenti komandne linije* koji se predaju programu. Ime programa predstavlja ime *izvršnog fajla* u kome se obično nalazi mašinski kôd programa, ili kôd napisan na nekom drugom programskom jeziku za koji postoji odgovarajući interpretator (to može biti programski jezik same ljuske ili neki drugi jezik poput jezika *Perl* ili *Python*). Po konvenciji, izvršni fajlovi na sistemu UNIX nemaju ekstenziju, ali korisnik mora imati dozvolu za pokretanje fajla. Kako bi ljuska mogla da na disku pronađe i pokrene program sa tim imenom, mora da zna putanju do direktorijuma u kome se izvršni fajl nalazi. U tu svrhu ljuska ima definisanu listu direktorijuma koje treba pretražiti, a koja se naziva *putanja*. Putanja se čuva kao vrednost *promenljive okruženja* (engl. *environment variable*) koja se zove `PATH`. Vrednost ove promenljive može se očitati na sledeći način:

```
echo $PATH
```

Komanda `echo` prosto ispisuje tekst koji joj se daje kao argument komandne linije. Sintaksa `$PATH` označava vrednost promenljive koja je navedena iza znaka `$` (kada bismo izostavili znak `$`, ljuska bi doslovno ispisala tekst `PATH`). Na primer, ispis gornje komande mogao bi da izgleda ovako:

```
/usr/local/bin:/usr/bin:/bin:/usr/games
```

Dakle, u pitanju su apsolutne putanje direktorijuma razdvojene dvotačkom. Kada pokrećemo komandu, ljuska redom pretražuje navedene direktorijume i pokreće prvi program sa tim imenom koji pronađe. Vrednost promenljive `PATH` se može podešavati, npr.:

```
PATH=$PATH:/usr/sbin
```

Na ovaj način se uzima tekuća vrednost promenljive `PATH` i na nju se dopisuje niska `:/usr/sbin`. Tako dobijena niska se postavlja za novu vrednost promenljive `PATH`. Sada bi `echo` komanda od malopre ispisala:

```
/usr/local/bin:/usr/bin:/bin:/usr/games:/usr/sbin
```

Dakle, sada imamo jedan dodatni direktorijum u kome ljuska može tražiti izvršne fajlove pri pokretanju komandi. Napomenimo da će ova promena biti aktivna samo tokom tekuće sesije. Kada zatvorimo ljusku i pokrenemo je ponovo, ponovo ćemo imati originalnu vrednost promenljive `PATH`. Naime, promenljiva `PATH` se, kao i druge promenljive okruženja, prilikom pokretanja ljuske postavlja na neku podrazumevanu vrednost koja je zadata u odgovarajućim konfiguracionim fajlovima. Samim tim, svaki put kada pokrenemo ljusku, promenljiva `PATH` će biti postavljena na istu početnu vrednost. Promene koje mi „ručno” vršimo u okviru ljuske biće aktivne samo

do kraja tekuće sesije. Ako želimo da trajno promenimo podrazumevanu vrednost PATH promenljive, moramo da promenimo odgovarajuće konfiguracione fajlove (što prevazilazi okvire ovog teksta).

Promenljiva PATH je samo jedna od promenljivih okruženja. Postoje i druge promenljive i svaka od njih ima neko posebno značenje za ljusku. Na primer, promenljiva PWD sadrži putanju do tekućeg direktorijuma ljuske, promenljiva HOME sadrži putanju do početnog direktorijuma prijavljenog korisnika, promenljiva USER sadrži ime prijavljenog korisnika, i td. Takođe, korisnik može definisati nove promenljive okruženja koje neće imati nikakvo predefinisano značenje za ljusku, ali će imati značenje za korisnika i programe koje će u nastavku pokretati. Na primer:

```
export BOJA=plava
```

Ovim se kreira nova promenljiva okruženja BOJA i dodeljuje joj se vrednost plava. Ključna reč export neophodna je prilikom inicijalnog kreiranja promenljive, kako bi ona postala deo okruženja ljuske (inače bi bila samo lokalna promenljiva ljuske). Vrednost ove promenljive se kasnije može menjati (sintaksom npr. BOJA=crvena) ili očitavati njena vrednost (sintaksom $BOJA). Spisak svih trenutno definisanih promenljivih okruženja može se dobiti komandom:

```
export
```

Važno je napomenuti da se promenljive okruženja automatski prenose procesima koje ljuska kreira, tako da će svi programi koje pokrenemo iz ljuske imati na raspolaganju sve promenljive koje su nasleđene iz ljuske. Operativni sistem obezbeđuje interfejs kojim se u okviru korisničkih programa mogu očitavati vrednosti promenljivih okruženja, a konkretna sintaksa zavisi od programskog jezika u kom je program napisan. Na ovaj način, promenljive okruženja se mogu koristiti kao mehanizam komunikacije između ljuske i programa koji se u okviru nje pokreću.

Ukoliko se program koji pokrećemo ne nalazi ni u jednom od direktorijuma navedenim u promenljivoj PATH, tada kažemo da program nije *u putanji*. Ako program nije u putanji, ljuska neće moći da ga pokrene i prikazaće grešku, npr.:

```
abcd
```

Pokretanjem nepostojećeg programa abcd dobijamo poruku:

```
bash: abcd: command not found
```

Ponekad se program koji želimo da pokrenemo nalazi u nekom drugom direktorijumu koji nije u putanji. Kako bi ljuska mogla da ga pronađe, možemo navesti apsolutnu ili relativnu putanju do izvršnog fajla:

```
/home/pera/programiranje/moj_program arg1 arg2
```

Dakle, umesto da navedemo samo ime programa moj_program, navodimo celu putanju. Ovo je znak za ljusku da ne treba da traži program u predefinisanim direktorijumima iz promenljive PATH, već mu je eksplicitno data putanja u kojoj se program nalazi. Ako je tekući direktorijum ljuske /home/pera, tada možemo navesti i relativnu putanju:

```
programiranje/moj_program arg1 arg2
```

Najzad, ako je tekući direktorijum ljuske /home/pera/programiranje, tada možemo pokrenuti komandu:

```
./moj_program arg1 arg2
```

Ovde je bitno primetiti da pre naziva programa stoji ./. Naime, tačka (.) predstavlja oznaku tekućeg direktorijuma. Samim tim, ./moj_program predstavlja relativnu putanju u odnosu na tekući direktorijum. Iako je i moj_program takođe ispravno zadata relativna putanja u odnosu na tekući direktorijum, komanda:

```
moj_program arg1 arg2
```

bi izazvala grešku, jer bi ljuska smatrala da se radi o programu koji je u putanji, pa bi pretraživala direktorijume iz promenljive PATH. Navođenjem ./ ispred naziva programa mi govorimo ljusci da program ne treba tražiti u putanji, već da ga treba tražiti u tekućem direktorijumu.

Elektronska verzija (2024)

Povratna vrednost programa

Od svakog programa se očekuje da ljusci vrati *povratnu vrednost*. U pitanju je ceo broj koji ljusci (ili korisniku) može pružiti informacije o tome kako je protekao rad programa. Po konvenciji, vrednost 0 označava da je sve prošlo u redu, dok vrednosti različite od nule predstavljaju različite kôdove grešaka. Tačno značenje kôda greške zavisi od programa i obično se može pronaći u pratećoj dokumentaciji.

Način na koji program vraća vrednost ljusci zavisi od programskog jezika u kome je program napisan. Na primer, programi pisani u programskim jezicima C ili C++ vraćaju povratnu vrednost svoje glavne (*main*) funkcije.

Da bi korisnik očitao povratnu vrednost programa čije je izvršavanje upravo završeno u okviru ljuske, može pokrenuti komandu:

```
echo $?
```

Argumenti komandne linije

Argumenti komandne linije predstavljaju dodatne informacije koje se prosleđuju pokrenutom programu. U okviru programa postoji mogućnost pristupa argumentima komandne linije, pri čemu konkretna sintaksa ponovo zavisi od samog programskog jezika u kome je program napisan. Značenje argumenata komandne linije zavisi od samog programa, dok ljuska argumente vidi prosto kao niske simbola i ne pridaje im nikakvo značenje. Ljuska podrazumevano smatra da su argumenti komandne linije međusobno razdvojeni razmacima. Na primer:

```
moj_program prvi drugi treci
```

U ovom primeru, ljuska će smatrati da postoje tri argumenta komandne linije i programu `moj_program` će prilikom pokretanja predati niz koji sadrži tri niske: `prvi`, `drugi` i `treci`. Ukoliko želimo da neki argument komandne linije sadrži razmak, tada je neohodno navesti argument pod apostrofima:

```
moj_program 'prvi argument' drugi treci
```

Sada ponovo imamo tri argumenta: `prvi argument`, `drugi` i `treci`. Dakle, prvi argument se sastoji iz dve reči. Da smo zaboravili da stavimo apostrofe:

```
moj_program prvi argument drugi treci
```

ljuska bi smatrala da imamo četiri argumenta komandne linije, redom `prvi`, `argument`, `drugi` i `treci`. Umesto apostrofa, moguće je koristiti i navodnike:

```
moj_program "prvi argument" drugi treci
```

U datom primeru, efekat će biti potpuno isti. Razlika između apostrofa i navodnika je u tome što navodnici dopuštaju *ekspanziju promenljivih*, tj. zamenu imena promenljive njenom vrednošću. Na primer:

```
echo "Moj tekuci direktorijum je $PWD"
```

U ovom primeru će najpre `$PWD` biti zamenjeno vrednošću promenljive `PWD` (npr. `/home/pera/programiranje`), a zatim će tako dobijeni tekst biti prosleđen kao (jedini) argument komandne linije programu `echo`. Ovo je isto kao da smo pozvali komandu:

```
echo "Moj tekuci direktorijum je /home/pera/programiranje"
```

što kao efekat ima ispis:

```
Moj tekuci direktorijum je /home/pera/programiranje
```

Sa druge strane, da smo napisali:

```
echo 'Moj tekuci direktorijum je $PWD'
```

ljuska ne bi vršila ekspanziju promenljive, pa bi tekst `$PWD` ostao nepromenjen i kao takav bi bio prosleđen programu `echo` kao deo argumenta komandne linije. Efekat ove komande bio bi ispis teksta:

```
Moj tekuci direktorijum je $PWD
```

Elektronska verzija (2024)

Na kraju, posmatrajmo još i sledeći primer:

```
moj_program "Moja boja: $BOJA"
```

U ovom slučaju će `$BOJA` biti zamenjeno vrednošću ove promenlive (npr. `plava`), a onda će ceo tekst `Moja boja: plava` biti prosleđen programu kao *jedan* argument komandne linije. Sa druge strane:

```
moj_program Moja boja: $BOJA
```

će takođe proizvesti ekspanziju promenljive, ali će sada tekst `Moja boja: plava` biti prosleđen programu kao niz od *tri* argumenta komandne linije: `Moja`, `boja:` i `plava`. Dakle, ekspanzija promenljivih se vrši i kada nema navodnika, ali navodnici omogućavaju da se više reči razdvojenih razmacima tretiraju kao jedan argument, umesto da ih ljuska posmatra kao posebne argumente komandne linije.

Pored ekspanzije promenljivih, ljuska omogućava i druge vrste ekspanzija u okviru komandne linije. Na primer, simboli `*` i `?` omogućavaju zadavanje putanja do svih fajlova čiji se nazivi uklapaju u neki obrazac. Posmatrajmo komandu:

```
cat *.html
```

Program `cat` prihvata na komandnoj liniji jednu ili više putanja do fajlova, a zatim sadržaje svih tih fajlova redom izlistava na standardnom izlazu. U ovom primeru će se sadržaji svih fajlova sa ekstenzijom `.html` prikazati na standardnom izlazu. Naime, sintaksa `*.html` se od strane ljuske automatski razvija u listu svih fajlova čija se putanja uklapa u dati obrazac, razdvojenih razmacima. U našem primeru, to su svi fajlovi u tekućem direktorijumu čiji se naziv završava sa `.html`. Simbol `*`, dakle, predstavlja nula ili više proizvoljnih karaktera. Slično, mogli smo navesti:

```
cat /home/pera/*/index.html
```

Ovog puta se argument `/home/pera/*/index.html` zamenjuje listom svih putanja tog oblika. Ako nema ni jednog takvog fajla, tada navedeni argument ostaje nepromenjen, tj. nema ekspanzije. Pored simbola `*`, moguće je navesti i simbol `?`. Ovaj simbol predstavlja tačno jedan proizvoljan karakter. Sada se pri pozivu komande:

```
cat /home/pera/*/index?.html
```

navedeni argument zamenjuje listom svih postojećih putanja tog oblika, gde se `*` zamenjuje sa nula ili više proizvoljnih karaktera, a `?` tačno jednim proizvoljnim karakterom. Najzad, komanda:

```
cat /home/pera/{index,map}.html
```

će na standardnom izlazu prikazati sadržaje fajlova `index.html` i `map.html` iz direktorijuma `/home/pera`. Naime, sintaksa `/home/pera/{index,map}.html` se razvija u listu sastavljenu od dva argumenta `/home/pera/index.html` i `/home/pera/map.html` razdvojenih razmakom.

Navedene ekspanzije spadaju u tzv. *ekspanzije putanje* (engl. *path expansion*). Pored ovih ekspanzija, postoji i veliki broj drugih, kako ekspanzija, tako i drugih sintaksnih mogućnosti ljuske poput komandnih ekspanzija, evaluacije izraza, složenih naredbi i sl., čiji detaljni opis prevazilazi okvire ovog teksta. Zajedničko za sve ove sintaksne elemente ljuske je da podrazumevaju upotrebu simbola koji imaju specijalno značenje za ljusku. Takvi simboli su, na primer, `$`, `*`, `?`, `{`, `}`, `(`, `)`, `[`, `]`, `!`, `;`, `|`, `<`, `>` i sl. Kada ljuska u nazivu komande ili u okviru argumenata komandne linije uoči neki od ovih simbola, ona ih tumači u skladu sa njihovim specijalnim značenjem. Upotreba apostrofa sprečava specijalno tumačenje simbola, tj. primorava ljusku da sve ove simbole tretira kao obične karaktere, te da odgovarajuće argumente komandne linije prenese programu doslovno, bez ikakvih promena. Na primer, ako kao argument komandne linije navedemo `'$PWD'` ili `'/etc/*.conf'`, neće doći do odgovarajuće ekspanzije, zato što koristimo apostrofe, pa će simboli `$` i `*` biti tumačeni kao obični karakteri. Za razliku od apostrofa, unutar navodnika pojedini specijalni simboli (poput simbola `$`, ali ne i simbola `*` i `?`) zadržavaju specijalno značenje. Zato će se `"$PWD"` razviti u vrednost promenljive `PWD`, dok će `"/etc/*.conf"` ostati takav kakav je, bez ekspanzije. Navedimo još jedan zanimljiv primer:

```
grep '[a-zA-Z]*[0-9]{2}' ulaz.txt
```

Elektronska verzija (2024)

Program grep se koristi za pronalaženje uzoraka u tekstu. Prvi argument komandne linije opisuje uzorak koji se traži na jeziku tzv. *regularnih izraza* čiji detaljan opis ovde izostavljamo, zbog nedostatka prostora. U ovom primeru, tražimo sve uzorke koji se sastoje iz nula ili više malih ili velikih slova za kojima slede tačno dve cifre. Drugi argument je putanja do fajla koji se pretražuje. Program ispisuje sve linije fajla koje sadrže traženi obrazac. Primetimo da smo prvi argument programa smestili između apostrofa. Ovo je zato što bi inače različiti specijalni simboli u sintaksi obrazca (poput [, -,], {, }) bili tumačeni kao specijalni simboli ljuske. Mi to ne želimo, već želimo da se navedeni obrazac doslovno preda programu grep koji će ga dalje tumačiti u skladu sa svojim internim pravilima.

Tekući direktorijum

Tekući direktorijum ljuske je takođe važan parametar koji se prenosi na procese pokrenute u okviru ljuske. Sve relativne putanje se uvek odnose na ovaj direktorijum. Tekući direktorijum se prilikom pokretanja ljuske postavlja na početni korisnički direktorijum (dat promenljivom HOME). U toku rada, možemo ga promeniti komandom

```
cd <putanja>
```

gde je <putanja> apsolutna ili relativna putanja do direktorijuma za koji želimo da postane novi tekući direktorijum. Na primer, ako se nalazimo u direktorijumu /home/pera i želimo da promenimo tekući direktorijum na /home/pera/programiranje, možemo zadati komandu:

```
cd /home/pera/programiranje
```

tj. da zadamo apsolutnu putanju, ili:

```
cd programiranje
```

čime zadajemo relativnu putanju. U oba slučaja, efekat će biti isti. Ako pretpostavimo da je /home/pera početni direktorijum prijavljenog korisnika, tada je vrednost promenljive HOME upravo /home/pera. To znači da možemo napisati i:

```
cd $HOME/programiranje
```

i efekat će, nakon ekspanzije promenljive, biti isti. Najzad, pokretanje komande cd bez argumenata za efekat ima vraćanje na početni direktorijum korisnika. Dakle, komanda:

```
cd
```

je ekvivalentna komandi:

```
cd $HOME
```

Tekući direktorijum se uvek može očitati bilo prikazom vrednosti promenljive PWD:

```
echo $PWD
```

bilo pozivom komande:

```
pwd
```

Elektronska verzija (2024)

Preusmeravanje standardnog ulaza i izlaza

Svaki program može učitavati podatke sa *ulaza*, kao i saopštavati rezultate svog rada na *izlazu*. Ulaz i izlaz mogu biti fajlovi na disku koji su otvoreni za čitanje, odnosno pisanje od strane operativnog sistema. Specijalno, postoji *standardni ulaz* koji je podrazumevano povezan sa tastaturom, kao i *standardni izlaz* koji je povezan sa ekranom monitora. Kada program pokuša da čita sa standardnog ulaza, dolazi do prekida rada procesa (tj. proces odlazi u stanje *čeka*), a operativni sistem podsredstvom drajvera od kontrolera tastature zahteva da mu prosledi unos koji korisnik unosi. Kada korisnik unese tekst i pritisne ENTER, uneti tekst se prosleđuje procesu i on ponovo može da nastavi sa izvršavanjem (tj. vraća se u stanje *spreman*). Slično, kada program piše na standardni izlaz, odgovarajući tekst se prikazuje na ekranu.

Jedna od značajnih karakteristika ljuske je da ona omogućava *preusmeravanje standardnog ulaza/izlaza* (engl. *standard input/output redirection*). Ovim mehanizmom se standardni ulaz/izlaz može preusmeriti na neki fajl po želji korisnika. Na primer:

```
moj_program < ulaz.txt
```

Simbol `<` označava preusmeravanje standardnog ulaza, a iza njega se očekuje putanja (apsolutna ili relativna) do fajla na koji se preusmerava standardni ulaz. Ovo znači da će svaki pokušaj čitanja sa standardnog ulaza unutar programa za efekat imati čitanje iz fajla `ulaz.txt`, a ne sa tastature. Slično:

```
moj_program > izlaz.txt
```

preusmerava standardni izlaz na fajl koji je zadat putanjom koja se navodi nakon simbola `>`. Ovo znači da će svaki ispis na standardni izlaz unutar programa za efekat imati pisanje u fajl `izlaz.txt` umesto na ekran. Moguće je istovremeno preusmeriti i ulaz i izlaz:

```
moj_program < ulaz.txt > izlaz.txt
```

Kada je u pitanju preusmeravanje standardnog izlaza, podrazumevano ponašanje je da se navedeni fajl otvara za pisanje, a njegov prethodni sadržaj se briše. Ukoliko želimo da njegov prethodni sadržaj ostane nepromenjen, a da se novi sadržaj dopisuje na kraj fajla, možemo koristiti operator `>>` umesto `>`:

```
moj_program < ulaz.txt >> izlaz.txt
```

Pored standardnog izlaza, postoji i *standardni izlaz za greške*. Ovaj izlaz je takođe podrazumevano povezan sa ekranom i koristi se za saopštavanje poruka o greškama korisniku. Prilikom preusmeravanja, on se može posebno preusmeriti, tako da se poruke o greškama ispisuju u poseban fajl. Na primer:

```
moj_program < ulaz > izlaz.txt 2> greske.txt
```

Putanja do fajla na koji se preusmerava standardni izlaz za greške se navodi nakon operatora `2>` (ili `2>>`, u slučaju dopisivanja sadržaja). Primetimo da ako samo navedemo:

```
moj_program < ulaz.txt > izlaz.txt
```

tada će se samo standardni izlaz preusmeravati na `izlaz.txt`, dok će se poruke o greškama saopštene preko standardnog izlaza za greške i dalje prikazivati na ekranu. Dakle, standardni izlaz i standardni izlaz za greške se uvek posebno preusmeravaju.

U svim prethodnim primerima, fajlovi `ulaz.txt` i `izlaz.txt` su se nalazili u tekućem direktorijumu. Moguće je navoditi i složenije putanje koje vode ka fajlovima u drugim direktorijumima. Na primer:

```
moj_program < programiranje/ulaz.txt
```

preusmerava ulaz na fajl `ulaz.txt` u poddirektorijumu `programiranje` tekućeg direktorijuma. Slično:

```
moj_program > ../razno/izlaz.txt
```

preusmerava standardni izlaz na fajl `izlaz.txt` koji se nalazi u okviru direktorijuma `razno` koji se nalazi u roditeljskom direktorijumu tekućeg direktorijuma. Dakle, oznaka `..` u putanji uvek označava roditeljski direktorijum tekućeg direktorijuma (setimo se da je oznaka `.` označavala tekući direktorijum).

Elektronska verzija (2024)

Opcije programa

Kao što je ranije rečeno, argumente komandne linije ljuska vidi kao obične niske i ne pridaje im nikakvo predefinisano značenje, već tumačenje tih argumenata ostavlja samom programu. Ipak, većina programa pod UNIX-om prati neke uobičajene konvencije kada su u pitanju argumenti komandne linije. Jedan poseban tip argumenata komandne linije koji prepoznaje većina programa i daje im poseban značaj jesu *opcije*. Opcijama se bliže određuje način rada programa. Kako bi se sintaksno razlikovale od ostalih argumenata komandne linije (poput fajlova ili komandi), obično počinju simbolom `-`. Na primer, komanda:

```
wc moj_fajl.txt
```

podrazumevano broji linije, reči i bajtove u fajlu koji je dat kao argument komandne linije. Međutim, ako zadamo opciju `-l`:

```
wc -l moj_fajl.txt
```

program će promeniti svoje ponašanje i brojaće samo linije. Program `wc` će znati da argument `-l` nije naziv fajla (što je uobičajeni prvi argument ove komande), već opcija, s obzirom da počinje sa `-`, pa će je tumači na odgovarajući način. Slično, postoje opcije `-w` (samo reči) i `-c` (samo bajtove). Ako navedemo:

```
wc -l -c moj_fajl.txt
```

dobićemo broj linija i broj bajtova, dok se reči neće brojati.

Opcije su obiŋo kratke, sadrže jedno ili dva slova, što olakšava njihovo navođenje. Ipak, mnogi korisnici smatraju da je bolje da naziv opcije bude duži i opisniji, jer se tako lakše pamti, a komande postaju čitljivije. Ovakve, „dugačke" opcije se obično navode sa prefiksom `--`. Na primer, komanda `wc` ima i dugačke verzije opcija `-l`, `-w`, `-c`, a one glase `--lines`, `--words` i `--bytes`. Sada možemo navesti:

```
wc --lines --bytes moj_fajl.txt
```

Ova komanda će imati isti efekat kao prethodna, a izgleda jasnije i čitljivije (ali je i duža).

Napomenimo na kraju da iako se opisane konvencije u najvećoj meri poštuju od većine standardnih UNIX programa, ipak treba imati u vidu da je tumačenje argumenata komandne linije u potpunosti u domenu programa, tako da mogu postojati programi koji od ovih konvencija odstupaju. Kako biste se upoznali sa opcijama koje konkretan program prepoznaje i njihovim značenjem, najbolje je konsultovati dokumentaciju koja je isporučena uz program. Ova dokumentacija je obično dostupna u formi *stranica sa uputstvom za korišćenje* (engl. *manual pages*). Ove stranice se otvaraju pomoću programa `man`:

```
man wc
```

Program `man` izlistava uputstvo programa koji je naveden kao argument komandne linije. Uputstvo se može listati pomoću strelica za *gore*/*dole* (ili *PageUp*/*PageDown* tastera, u slučaju dugačkih uputstava). Izlazak iz `man` programa se postiže pritiskom na taster `Q`.

Inače, `man` stranice nisu dostupne samo za programe. Postoje `man` stranice i za funkcije standardne biblioteke programskog jezika C, kao i za sistemske pozive UNIX sistema. Na primer:

```
man strlen
```

otvara stranicu posvećenu funkciji `strlen()` standardne C biblioteke, dok komanda:

```
man fork
```

otvara stranicu posvećenu sistemskom pozivu `fork()`.

Više detalja o samom programu `man` možete pročitati iz njegove `man` strane:

```
man man
```

Elektronska verzija (2024)

Nadovezivanje programa

Nadovezivanje programa podrazumeva da se pokreće istovremeno više programa, pri čemu se standardni izlaz prvog preusmerava na standardni ulaz drugog, standardni izlaz drugog na standardni ulaz trećeg i td. Ovo se postiže mehanizmom *cevi* (engl. *pipe*) koji predstavlja jedan od najjednostavnijih mehanizama interprocesne komunikacije pod UNIX-om. Ljuska omogućava jednostavno nadovezivanje programa na sledeći način:

```
echo "Uvod u informatiku" | wc -w
```

Dakle, operatorom `|` možemo postići da se standardni izlaz komande `echo` preusmeri na standardni ulaz programa `wc`. Program `wc`, u odsustvu fajla navedenog na komandnoj liniji, svoj ulaz čita sa standardnog ulaza. Samim tim, izlaz programa `echo` (tj. tekst `Uvod u informatiku`) neće biti prikazan na ekranu, već će biti prosleđen programu `wc` koji će prebrojati reči u tom tekstu i dobijeni broj (3) prikazati na svom standardnom izlazu (koji nije preusmeren, pa će biti prikazan na ekranu). Pogledajmo još jedan primer:

```
cat primer.txt | head -7 | tail -5
```

Program `cat` ispisuje na standardnom izlazu sadržaj fajla koji je naveden kao argument komandne linije (u ovom primeru `primer.txt`). Međutim, njegov izlaz se preusmerava na standardni ulaz programa `head` koji ispisuje prvih 7 linija sa ulaza (opcija `-7`). Međutim, njegov izlaz je dalje preusmeren na standardni ulaz programa `tail`, koji ispisuje poslednjih 5 linija sa svog ulaza (opcija `-5`). Krajnji rezultat je da će deo sadržaja fajla `primer.txt` od treće do sedme linije biti prikazane na standardnom izlazu.

Evo još jednog primera:

```
cat primer.txt | sort | uniq
```

Ovog puta se sadržaj fajla `primer.txt` preusmerava na standardni ulaz programa `sort`, koji sortira linije fajla u rastućem leksikografskom poretku i ispisuje ih na standardni izlaz. Međutim, njegov izlaz je preusmeren na standardni ulaz programa `uniq` koji sa svog ulaza eliminiše susedne linije koje su identične. Na ovaj način smo iz fajla `primer.txt` izbacili duplikate linija, jer će nakon sortiranja duplikati svakako biti susedni, pa će ih `uniq` eliminisati. Dobijeni rezultat se ispisuje na standadnom izlazu.

Prikazani primeri pokazuju kako nadovezivanjem programa koji vrše jednostavne operacije nad ulazom možemo obavljati veoma složene poslove. UNIX okruženje uključuje veliki broj ovakvih jednostavnih alata i njihovo dobro poznavanje je preduslov za uspešno korišćenje nadovezivanja u cilju obavljanja rutinskih poslova u ljusci.

1.3.2 Upravljanje korisnicima UNIX sistema

Operativni sistem UNIX je višekorisnički sistem. Svaki korisnik ima sopstveno korisničko ime, identifikacioni broj (engl. *user ID* (UID)), lozinku, svoj osnovni korisnički direktorijum, kao i svoju podrazumevanu ljusku. UNIX sve korisnike deli u dve kategorije: *administrator* (sa korisničkim imenom `root`) i *obični korisnici*. Administrator ima ID jednak 0 i kao takav ima poseban tretman od strane UNIX sistema. On ima prava pristupa svim fajlovima i direktorijumima, kao i pravo da pokreće sve programe i izvršava sve operacije nad sistemom. Obični korisnici imaju ograničen pristup i mnoge operacije (poput nekih sistemskih poziva) im nisu dostupni. Takođe, u okviru sistema datoteka se mogu postaviti ograničenja pristupa fajlovima i direktorijumima za obične korisnike.

Pored korisnika, postoje i *grupe korisnika*. Svaki korisnik mora pripadati bar jednoj grupi (to je njegova *podrazumevana grupa*), i to se određuje prilikom kreiranja korisnika. Korisnik se naknadno može dodavati i u druge grupe po želji. Smisao grupa je u davanju dodatnih prava pristupa nad nekim fajlovima i drugim resursima za određeni skup korisnika. Grupa takođe ima svoje ime i identifikacioni broj (engl. *group ID* (GID)).

Nova grupa se može dodati komandom `groupadd`:

```
groupadd moja_grupa
```

Novi korisnici se mogu dodavati komandom `useradd`:

```
useradd pera -d /home/pera -m -g users -s /bin/bash -c "Pera Peric"
```

Ovom komandom se kreira korisnik čije je korisničko ime `pera`, UID je jednak prvom sledećem dostupnom broju, korisnički direktorijum je `/home/pera` (koji će biti kreiran ako već ne postoji, zahvaljujući opciji `-m`), podrazumevana grupa mu je `users` (koja bi trebalo da već postoji), podrazumevana ljuska mu je `/bin/bash`, a puno ime mu je `Pera Peric`.

Nakon kreiranja korisnika, potrebno je postaviti lozinku za korisnika. Ovo se može uraditi komandom:

Elektronska verzija (2024)

```
passwd pera
```

nakon čega je potrebno dva puta uneti željenu lozinku. Prilikom unosa lozinke ne prikazuje se ništa na ekranu, iz bezbednosnih razloga. Nakon toga, korisnik se može prijaviti na sistem koristeći tu lozinku. Nakon prijave (ili kada pokrene emulator terminala, u slučaju grafičkog korisničkog interfejsa), otvoriće mu se njegova podrazumevana ljuska, a tekući direktorijum biće njegov korisnički direktorijum.

Ako želimo da korisnik bude član i nekih dodatnih grupa, možemo prilikom kreiranja korisnika dodati i opciju `-G`:

```
useradd pera -d /home/pera -m -g users -G moja_grupa,wheel,audio -s /bin/bash
```

Ova komanda će korisnika dodatno učlaniti i u grupe `moja_grupa`, `wheel` i `audio`. Ove grupe nazivaćemo *suplementarne grupe* tog korisnika. Suplementarne grupe možemo menjati i naknadno:

```
usermod -a -G games
```

Ovom komandom ćemo dodatno korisnika učlaniti u grupu `games`. Listu svih suplementarnih grupa korisnika uvek možemo videti komandom:

```
groups
```

Neke od grupa imaju neko predefinisano značenje za sistem i bitno je dodati korisnika u te grupe da bi dobio neke određene privilegije. Na primer, grupa `wheel` obično omogućava korisniku da pod određenim uslovima izvršava komande kao administrator. Grupa `cdrom` omogućava korisniku da koristi CD i DVD diskove, dok grupa `plugdev` omogućava rad sa uređajima poput fleš memorija. Na ovaj način, administrator može fino kontrolisati šta ko od korisnika može da radi na sistemu.

Korisnik se može ukoloniti komandom:

```
userdel pera
```

Slično, grupa se može ukloniti komandom:

```
groupdel moja_grupa
```

Sve ove komande može pokretati isključivo `root` korisnik. Posledica je da običan korisnik ne može sam sebe dodavati u grupe, kreirati nove korisnike i davati im pristup i sl. Ovo je bitno iz bezbednosnih razloga. Izuzetak je komanda `passwd`, nju može pokretati i bilo koji korisnik, bez argumenata. Tom prilikom pokrenuće se procedura za promenu lozinke tog korisnika, pri čemu će se prvo očekivati od korisnika da unese svoju trenutnu lozinku.

Na računarski sistem može biti priključeno više terminala, a zahvaljujući umrežavanju, postoji i mogućnost udaljenog pristupa sistemu. To znači da u svakom trenutku možemo imati više korisnika koji su prijavljeni na sistem. Da bismo videli sve korisnike koji su trenutno prijavljeni, možemo pokrenuti komandu:

```
finger
```

Ista komanda se može koristiti i za dobijanje više informacija o korisniku, bez obzira da li je trenutno prijavljen ili ne:

```
finger pera
```

Najzad, spisak korisnika koji su se prijavljivali na sistem u prethodnom periodu može se videti komandom:

```
last
```

Elektronska verzija (2024)

Korisnici i procesi

Svaki proces ima svog *vlasnika*. Vlasništvo nad procesima se nasleđuje, tj. kada jedan proces pokrene drugi, dete proces će imati istog vlasnika kao i roditeljski proces. Prvi procesi koji se kreiraju od strane jezgra prilikom pokretanja sistema su u vlasništvu `root` korisnika. Kada se korisnik uspešno prijavi na sistem, njegov korisnički interfejs (ljuska ili grafičko okruženje) pokreće se u vlasništvu tog korisnika. Na dalje, svi procesi koje korisnik bude kreirao podsredstvom korisničkog interfejsa biće u njegovom vlasništvu. Vlasništvo nad procesom određuje prava koja proces ima, a koja odgovaraju pravima i privilegijama korisnika koji je vlasnik procesa.

Pored vlasnika, svaki proces ima i korisničku grupu kojoj pripada, kao i listu suplementarnih grupa. Ove informacije se takođe postavljaju prilikom prijavljivanja korisnika na sistem (na osnovu podrazumevane korisničke grupe korisnika i njegove liste suplementarnih grupa), a na dalje se nasleđuju od strane svih procesa koje korisnik pokrene putem korisničkog interfejsa. Informacije o grupama kojima proces pripada su bitne za operativni sistem kada odlučuje o tome da li proces ima prava da izvrši određenu operaciju. Ovo se pre svega odnosi na pristup fajlovima i direktorijumima, o čemu će biti više reči kasnije.

1.3.3 Rad sa sistemom datoteka

Svi sistemi datoteka se pod UNIX-om vide kao jedno jedinstveno stablo direktorijuma. Koreni direktorijum je označen kosom crtom (`/`). Svi sistemi datoteka *montiraju* se na neke od čvorova ovog stabla koje nazivamo *tačke montiranja*. Particija diska na kojoj je instaliran sam operativni sistem se montira u tački `/`, pa će koreni direktorijum sistema datoteka te particije odgovarati putanji `/`. U okviru ovog sistema datoteka možemo kreirati bilo koji prazan direktorijum, a zatim njega iskoristiti kao tačku montiranja za neki drugi sistem datoteka (na drugoj particiji ili drugom uređaju). U tu svrhu se koristi komanda `mount`:

```
mount -t ext4 /dev/sda2 /mnt/tacka
```

U ovom primeru, opcijom `-t ext4` zadat je *tip* sistema datoteka koji se montira (*ext4* je standardni tip sistema datoteka koji se koristi na GNU/Linux sistemima). Pretpostavka je da je sistem datoteka koji montiramo ispravno formatiran i da je tog tipa, inače montiranje neće biti uspešno. Argument `/dev/sda2` označava uređaj na kome se sistem datoteka nalazi. U ovom primeru, u pitanju je druga particija prvog hard diska. Treći argument `/mnt/tacka` je apsolutna putanja koja označava tačku montiranja. Ova putanja mora ukazivati na direktorijum koji postoji u okviru trenutnog stabla i trebalo bi da bude prazan. Nakon izvršavanja ove komande direktorijum `/mnt/tacka` više neće biti prazan, veće se u njemu videti sve ono što postoji u korenom direktorijumu sistema datoteka uređaja koji je montiran u toj tački. Drugim rečima, stablo direktorijuma montiranog sistema datoteka se ugrađuje u jedinstveno stablo UNIX sistema datoteka kao podstablo sa korenom u tački montiranja. Sistem datoteka se može kasnije i *demontirati*, tj. ukloniti iz jedinstvenog stabla, komandom `umount`:

```
umount /mnt/tacka
```

Napomenimo da je za montiranje i demontiranje obično neophodno biti administrator sistema, osim ako nije drugačije podešeno. Za pregled svih trenutno montiranih sistema datoteka, moguće je pokrenuti komandu:

```
mount
```

Napomenimo da se sistemi datoteka koji postoje na ugrađenim diskovima montiraju automatski prilikom podizanja sistema (ovo se postiže odgovarajućom konfiguracijom sistema). Sa druge strane, sistemi datoteka na prenosivim uređajima poput CD/DVD diskova ili fleš memorija se montiraju kada se priključe (ranije se to radilo ručno, `mount` komandom, dok se u moderno vreme to obavlja prilično automatski, zahvaljujući grafičkom korisničkom interfejsu).

Tipovi fajlova na sistemu UNIX

UNIX sistem razlikuje nekoliko tipova fajlova. Osnovni tip fajlova su *regularni fajlovi*, koji predstavljaju sve ono što uobičajeno podrazumevamo pod fajlom, poput tekstualnih fajlova, izvršnih fajlova, bibloteka, slika, video i audio zapisa i sl. Za razliku od nekih drugih operativnih sistema, UNIX ne pravi razliku među regularnim fajlovima na osnovu ekstenzije u nazivu fajla, već ostavlja programu koji koristi fajl da utvrdi konkretan tip fajla na osnovu sadržaja.[7]

[7] Ekstenzije u nazivima mogu postojati, ali one su tu pre svega da korisniku omoguće da lakše raspoznaje fajlove po tipovima. Takođe, grafički korisnički interfejs može fajlovima na osnovu ekstenzije pridruživati aplikaciju koja će automatski otvarati fajl kada se klikne na njega.

Elektronska verzija (2024)

Drugi tip fajlova koji UNIX razlikuje jesu *direktorijumi*. UNIX direktorijume vidi kao fajlove čiji je sadržaj spisak imena fajlova i direktorijuma koji se u njemu nalaze, uz reference na strukture podataka (tzv. *i-čvorove*, engl. *i-node*) u sistemu datoteka koji sadrže bliže informacije o tim fajlovima i direktorijumima. Svaki direktorijum uvek sadrži dve posebne stavke u sebi: jedna koja ukazuje na njega samog (označena sa `.`) i jedna koja ukazuje na roditeljski direktorijum (označena sa `..`). Ovo nam omogućava da koristimo putanje poput `./fajl` ili `../../fajl`.

Posebni tipovi fajlova se koriste za predstavljanje uređaja u okviru računarskog sistema. U pitanju su fajlovi koji nemaju sadržaj, već se pisanjem u te fajlove i čitanjem iz njih direktno komunicira sa drajverom za odgovarajući ulazno-izlazni uređaj. Ovi fajlovi se tradicionalno nalaze u direktorijumu `/dev/`, a kreira ih jezgro operativnog sistema automatski, prilikom inicijalizacije ulazno-izlaznih uređaja. Na primer, fajlovi poput `/dev/sda`, `/dev/sdb` predstavljaju hard (ili SSD) diskove ili druge memorijske uređaje poput fleš memorija. Ukoliko odgovarajući uređaj na sebi ima formirane particije, za njih se kreiraju posebni fajlovi `/dev/sda1`, `/dev/sda2` i sl. Fajlovi poput `/dev/cdrom`, `/dev/dvd` mogu ukazivati na uređaje za čitanje CD ili DVD diskova. Fajl `/dev/mouse` obično ukazuje na miša.

Simbolički linkovi (engl. *symbolic link*) predstavljaju poseban tip fajlova koji predstavljaju reference na druge fajlove u okviru sistema datoteka.

Pored navedenih tipova fajlova, UNIX raspoznaje još i posebne tipove fajlova koji se koriste u interprocesnoj komunikaciji, tj. pisanjem u ove fajlove i čitanjem iz njih vrši se komunikacija sa procesima koji su te fajlove kreirali. U ove tipove fajlova spadaju FIFO fajlovi i UNIX priključci (engl *UNIX socket*).

Kao što smo videli, pored regularnih fajlova, direktorijumi, uređaji kao i kanali interprocesne komunikacije se predstavljaju fajlovima. Ovo je jedna od najvažnijih karakteristika UNIX sistema, a ogleda se u krilatici „Sve je fajl" (engl. *Everything is a file*). Ovo znači da se u okviru UNIX sistema većina resursa predstavlja fajlom, što omogućava unifikovani pristup tip resursima.

Struktura UNIX sistema datoteka

Prilikom instalacije UNIX sistema, u okviru korenog direktorijuma kreiraju se neki standardni direktorijumi koji imaju svoju unapred određenu ulogu u sistemu. Neki od njih su:

- `/bin`, `/sbin` sadrže osnovne UNIX programe (izvršne fajlove) i po pravilu se nalaze u putanji ljuske
- `/etc` sadrži konfiguracione fajlove, obično u tekstualnom formatu
- `/dev` sadrži fajlove koji predstavljaju interfejs ka uređajima
- `/boot` sadrži fajlove jezgra sistema i punioca koji se koriste prilikom uključivanja računara i podizanja sistema
- `/lib`, `/lib64` sadrže osnovne sistemske biblioteke
- `/home` sadrži korisničke direktorijume
- `/usr` sadrži instalirani korisnički softver
- `/tmp` sadrži privremene fajlove koje procesi kreiraju

Ostavljamo čitaocu da dalje istraži ove direktorijume i bolje se upozna sa njihovim sadržajem.

Prava pristupa

Da bi rad sa sistemom datoteka bio bezbedan, neophodno je ograničiti prava pristupa korisnicima kada su u pitanju fajlovi i direktorijumi. Kao što je ranije rečeno, `root` korisnik (tačnije, procesi u njegovom vlasništvu) ima neograničena prava i može pristupati svim fajlovima i direktorijumima i vršiti sve moguće radnje nad njima. Ostalim korisnicima se nameću restrikcije prema ulozi koju oni imaju kada je u pitanju konkretan fajl ili direktorijum. Svaki fajl ili direktorijum ima svog *vlasnika* kao i *grupu vlasnika*. Kada se fajl ili direktorijum kreira, njegov vlasnik i grupa vlasnik će inicijalno biti redom vlasnik i grupa vlasnik procesa koji ga je kreirao. To znači da će svi fajlovi ili direktorijumi koje korisnik kreira podsredstvom korisničkog interfejsa biti inicijalno u njegovom vlasništvu. Vlasnik fajla se naknadno može promeniti, ali samo od strane `root` korisnika. Drugim rečima, obični korisnici ne mogu da „otuđe" svoje fajlove i direktorijume (ovo je takođe bitno iz bezbednosnih razloga). Sa druge strane, običan korisnik može promeniti grupu vlasnika nekog svog fajla ili direktorijuma, ali pritom nova grupa vlasnik može biti samo neka od grupa kojoj korisnik pripada.

Promena vlasnika fajla (od strane `root` korisnika), obavlja se komandom `chown`:

Elektronska verzija (2024)

```
chown pera /var/file
```

Ovim vlasnik fajla `/var/file` postaje korisnik `pera`. Ako želimo da istovremeno promenimo i grupu vlasnika, možemo pokrenuti sledeću komandu:

```
chown pera:users /var/file
```

Sada je novi vlasnik fajla `/var/file` korisnik `pera`, a nova grupa vlasnik je `users`. Ako želimo da promenimo samo grupu vlasnika, možemo pokrenuti komandu:

```
chgrp users /var/file
```

Prilikom utvrđivanja da li neki proces (koji nije u vlasništvu `root` korisnika) ima prava da obavi određenu operaciju sa fajlom ili direktorijumom, operativni sistem razvrstava korisnike u tri kategorije. Prvu kategoriju čini vlasnik fajla ili direktorijuma, drugu čine svi korisnici koji pripadaju grupi koja je vlasnik fajla ili direktorijuma, a treću čine svi ostali korisnici. Za svaku od ove tri kategorije u sistemu datoteka se čuva informacija o pravima koje korisnici iz tih kategorija imaju nad tim fajlom ili direktorijumom. Ta prava mogu biti:

- pravo čitanja (označeno sa `r`): u slučaju fajla, to znači da možemo otvoriti fajl za čitanje i samim tim videti njegov sadržaj; za direktorijum ovo pravo omogućava izlistavanje sadržaja direktorijuma.
- pravo pisanja (označeno sa `w`): u slučaju fajla, ovo pravo omogućava procesu da otvori fajl za pisanje i, samim tim, da modifikuje sadržaj fajla. Za direktorijum, ovo pravo omogućava kreiranje novih ili brisanje postojećih fajlova ili direktorijuma u okviru tog direktorijuma.
- pravo izvršavanja (označeno sa `x`): u slučaju fajla, ovo pravo označava da korisnik može pokretati taj fajl kao izvršni program; u slučaju direktorijuma, ovo pravo označava da proces može „ući" u taj direktorijum, tj. postaviti ga za svoj tekući direktorijum ili pristupati fajlovima i direktorijumima u okviru tog direktorijuma.

Napomenimo da ovo poslednje znači da ako želimo da pristupimo nekom fajlu pomoću putanje (apsolutne ili relativne), tada za svaki od direktorijuma koji se pominju u toj putanji moramo imati pravo `x`, a za sam fajl kome pristupamo moramo imati ono pravo koje odgovara operaciji koju želimo da izvršimo (`r` za čitanje, `w` za modifikaciju, `x` za pokretanje).

Prava pristupa fajlu obično prikazujemo u obliku niske koja se sastoji iz devet karaktera: prva tri označavaju prava pristupa vlasnika (redom `r`, `w` i `x`), druga tri označavaju prava pristupa pripadnika grupa, a treća tri označavaju prava pristupa ostalih korisnika. Na primer, niska:

```
rwxr-xr-x
```

označava da vlasnik ima sva tri prava, dok pripadnici grupa i ostali korisnici imaju prava `r` i `x`, dok nemaju pravo `w` (ovo je označeno simbolom `-` umesto `w`).

Prilikom utvrđivanja prava pristupa, operativni sistem prvo utvrđuje da li je vlasnik procesa `root` korisnik. Ako jeste, dodeljuje mu pravo pristupa. U suprotnom, ako je vlasnik procesa vlasnik fajla, prava pristupa određena su pravima koja ima vlasnik. U suprotnom, ako se grupa vlasnik procesa ili neka od suplementarnih grupa procesa poklapa sa grupom vlasnikom fajla, tada su prava pristupa određena pravima koja imaju pripadnici grupe vlasnika fajla. U suprotnom, prava su određena pravima koja nad tim fajlom imaju ostali korisnici.

Prava pristupa se mogu menjati komandom `chmod`:

```
chmod u+x,go-rwx /var/file
```

Ovom komandom se vlasniku fajla `/var/file` (označenom sa `u`) dodaje (`+`) pravo `x`, dok se pripadnicima grupe (`g`) i ostalim korisnicima (`o`) oduzimaju (`-`) sva prava (`r`, `w` i `x`). Promenu prava pristupa može vršiti samo vlasnik fajla (ili `root` korisnik).

Elektronska verzija (2024)

Navigacija kroz sistem datoteka

Već od ranije znamo da u svakom trenutku možemo videti u kom se direktorijumu trenutno nalazimo pozivanjem komande `pwd`. Da bismo izlistali sadržaj tekućeg direktorijuma (pod pretpostavkom da na to imamo pravo, tj. da imamo pravo `r` nad tim direktorijumom), koristimo komandu:

```
ls
```

U slučaju da želimo da izlistamo neki drugi direktorijum, a ne tekući, možemo navesti putanju do direktorijuma kao argument komandne linije:

```
ls /usr/
```

Izlaz ove komande bi mogao da izgleda ovako:

```
bin/  etc/  games/  include/  info/  lib/  lib64/  man/  sbin/  share/  src/
```

Ukoliko želimo da imamo bogatiji ispis koji uključuje neke korisne informacije o fajlovima, poput prava pristupa, veličine fajla, datuma poslednje modifikacije, vlasnika i grupe vlasnika i sl., možemo zadati opciju `-l`:

```
ls -l /var/www/htdocs/
```

Izlaz ove komande može izgledati ovako:

```
drwxr-xr-x  2 root root 4,0K Feb 13  2021 htdig/
-rw-r--r--  1 root root   45 Jun 11  2007 index.html
-rw-r--r--  1 root root   45 Jun 11  2007 index.html.bak.13676
-rw-r--r--  1 root root   45 Jun 11  2007 index.html.bak.19999
-rw-r--r--  1 root root   45 Jun 11  2007 index.html.bak.23618
-rw-r--r--  1 root root   45 Jun 11  2007 index.html.bak.27667
-rw-r--r--  1 root root   45 Jun 11  2007 index.html.bak.32338
drwxr-xr-x 14 root root 4,0K Oct 19  2023 manual/
```

Svaka linija ovog ispisa daje informacije o jednom od fajlova ili direktorijuma koji se nalaze u okviru izlistanog direktorijuma. Ukoliko je u pitanju direktorijum, tada linija počinje simbolom `d`, u suprotnom počinje simbolom `-`. Nakon toga slede informacije o pravima pristupa, vlasniku, grupi vlasniku, veličini fajla i datumu i vremenu poslednje modifikacije. Na kraju linije navedeno je samo ime fajla (u slučaju direktorijuma, završava se simbolom `/`).

Pod UNIX-om postoji i koncept *skrivenih fajlova*. U pitanju su fajlovi koji se podrazumevano ne prikazuju prilikom izlistavanja direktorijuma. Ovi fajlovi se prepoznaju tako što njihov naziv počinje tačkom. Na primer, ako kreiramo fajl `.moj_fajl`, ovaj fajl će biti skriven, tj. neće se prikazivati na izlazu komande `ls`. Ako ipak želimo da nam se prikažuju i skriveni fajlovi, možemo zadati opciju `-a`:

```
ls -l -a /var/www/htdocs/
```

Izlaz ove komande bi mogao izgledati ovako:

```
drwxr-xr-x  4 root root 4,0K Sep  7 13:46 ./
drwxr-xr-x  7 root root 4,0K Oct 19  2023 ../
-rw-r--r--  1 root root    0 Sep  7 13:46 .hidden.html
drwxr-xr-x  2 root root 4,0K Feb 13  2021 htdig/
-rw-r--r--  1 root root   45 Jun 11  2007 index.html
-rw-r--r--  1 root root   45 Jun 11  2007 index.html.bak.13676
-rw-r--r--  1 root root   45 Jun 11  2007 index.html.bak.19999
-rw-r--r--  1 root root   45 Jun 11  2007 index.html.bak.23618
-rw-r--r--  1 root root   45 Jun 11  2007 index.html.bak.27667
-rw-r--r--  1 root root   45 Jun 11  2007 index.html.bak.32338
drwxr-xr-x 14 root root 4,0K Oct 19  2023 manual/
```

Ono što primećujemo je fajl `.hidden.html` koji ranije nije prikazivan, jer je skriven. Takođe, primećujemo još dva direktorijuma: `./` i `../`. Kao što od ranije znamo, `.` i `..` označavaju redom tekući direktorijum i prethodni (roditeljski) direktorijum. Dakle, svaki direktorijum sadrži referencu na samog sebe, kao i na svog roditelja. Kako ove reference počinju tačkom, skrivene su i neće biti podrazumevano prikazivane pri pozivu komande `ls`.

Elektronska verzija (2024)

Kreiranje i brisanje fajlova i direktorijuma

Da bismo kreirali novi direktorijum, potrebno je pokrenuti komandu `mkdir`:

```
mkdir novi_dir
```

Ovim se kreira direktorijum `novi_dir` u tekućem direktorijumu. Ako želimo da kreiramo novi direktorijum u nekom drugom direktorijumu, a ne u tekućem, dovoljno je navesti putanju (apsolutnu ili relativnu) do novog direktorijuma:

```
mkdir /home/pera/novi_dir
```

Ovim će biti kreiran novi direktorijum `novi_dir` u okviru direktorijuma `/home/pera`.

Fajlovi se unutar direktorijuma mogu kreirati podsredstvom programa koje pokrećemo. Na primer, ako pokrenemo editor teksta:

```
emacs novi_fajl.txt
```

otvoriće nam se editor teksta `emacs` u kome ćemo moći da uređujemo fajl. Nakon što sačuvamo fajl i izađemo iz editora, fajl `novi_fajl.txt` biće kreiran u tekućem direktorijumu.

Ako želimo da kreiramo prazan fajl, to možemo najjednostavnije uraditi pomoću komande `touch`:

```
touch novi_fajl.txt
```

Da bismo obrisali fajl, možemo pokrenuti komandu `rm`:

```
rm novi_fajl.txt
```

Sa druge strane, za brisanje direktorijuma potrebno je pokrenuti komandu:

```
rmdir /home/pera/novi_dir
```

Da bi ova komanda uspela, potrebno je da direktorijum koji brišemo bude prazan. U slučaju da nije prazan, možemo najpre obrisati njegov sadržaj, pa onda obrisati i njega. Alternativno, možemo zahtevati rekurzivno brisanje svih fajlova i poddirektorijuma iz datog direktorijuma:

```
rm -R /home/pera/novi_dir
```

Opcija `-R` zahteva od komande `rm` da rekurzivno obriše sve fajlove i direktorijume iz direktorijuma `novi_dir`, nakon čega će biti uklonjen i on sâm.

Podsetimo se da je za kreiranje i brisanje fajlova i direktorijuma unutar nekog direktorijuma *dir* potrebno imati pravo `w` nad tim direktorijumom *dir*. Sa druge strane, nije neophodno imati pravo `w` nad samim fajlom ili direktorijumom koji brišemo. Ipak, `rm` program može prijaviti upozorenje u slučaju brisanja fajla za koji nemamo pravo modifikacije. Da bismo izbegli to upozorenje, dovoljno je zadati opciju `-f`:

```
rm -f -R /home/pera/novi_dir/
```

Ovim će biti obrisan kompletan sadržaj direktorijuma `novi_dir`, kao i sam taj direktorijum, čak i da u njemu postoje fajlovi za koje nemamo pravo modifikacije (ili čak fajlovi u tuđem vlasništvu).

Kopiranje i premeštanje fajlova i direktorijuma

Fajl se može jednostavno kopirati komandom `cp`:

```
cp fajl.txt kopija.txt
```

Ovim se u tekućem direktorijumu kreira fajl `kopija.txt` sa identičnim sadržajem kao i `fajl.txt`. Kopija se ne mora kreirati u tekućem direktorijumu, već se može navesti putanja do bilo koje druge lokacije:

```
cp fajl.txt /home/pera/kopija.txt
```

Naravno, potrebno je da imamo pravo `w` nad direktorijumom u kom kreiramo kopiju, kao i pravo `x` za sve direktorijume koji se pominju u putanji. Kao drugi argument komande `cp` možemo navesti i samo putanju do direktorijuma u kom treba napraviti kopiju, bez navođenja naziva kopije:

Elektronska verzija (2024)

```
cp fajl.txt /home/pera/
```

U tom slučaju će u datom direktorijumu biti kreirana kopija sa istim imenom kao i originalni fajl. Ako fajl sa tim imenom već postoji, biće prebrisan, tj. njegov sadržaj će biti zamenjen sadržajem fajla koji se kopira (za ovo je neophodno dodatno imati i pravo `w` nad odredišnim fajlom).

Ako želimo da kopiramo ceo direktorijum, neophodno je da se rekurzivno kopiraju svi njegovi poddirektorijumi i fajlovi. Ovo se može postići opcijom `-R`:

```
cp -R novi_dir/ /home/pera/
```

Ovom komandom se u okviru direktorijuma `/home/pera/` kreira kopija direktorijuma `novi_dir/` sa njegovim celokupnim sadržajem.

Ukoliko želimo da premestimo fajl na drugu lokaciju, umesto da kreiramo kopiju, koristićemo komandu `mv`:

```
mv fajl.txt /home/pera/
```

Ovom komandom se fajl `fajl.txt` premešta u direktorijum `/home/pera/`. Za ovu operaciju, potrebno je imati pravo `w` i nad direktorijumu u kome se fajl trenutno nalazi (kod nas je to tekući direktorijum) i nad direktorijumom u koje se vrši premeštanje (`/home/pera/`), s obzirom da se sadržaj oba direktorijuma menja. Slično, moguće je premeštati čitave direktorijume:

```
mv /home/pera/novi_dir/  .
```

Ovom komandom vrši se premeštanje direktorijuma `novi_dir` iz direktorijuma `/home/pera` u tekući direktorijum (označen sa `.`, ko i obično).

U prethodnim primerima, kao odredište smo navodili putanju do direktorijuma u koji se fajl ili direktorijum premešta. Pritom, fajl ili direktorijum koji se premešta će na svojoj novoj destinaciji imati isto ime koje je imao i do sada. Ako želimo da mu prilikom premeštanja promenimo ime, tj. da se na novom odredištu zove drugačije, dovoljno je samo u nastavku odredišne putanje navesti i novo ime:

```
mv fajl.txt /home/pera/novi_fajl.txt
```

Sada će fajl biti premešten u direktorijum `/home/pera/`, ali će tamo biti sačuvan pod novim imenom `novi_fajl.txt`. Ova mogućnost komande `mv` se koristi i kada želimo samo da preimenujemo fajl, bez premeštanja:

```
mv fajl.txt novi_fajl.txt
```

Ovim se fajl „premešta" iz tekućeg direktorijuma u tekući direktorijum, tj. ne premešta se uopšte. Jedini efekat ove komande je preimenovanje fajla iz `fajl.txt` u `novi_fajl.txt`.

Napomenimo da je premeštanje fajla daleko efikasnija operacija od kopiranja. Naime, prilikom kopiranja, neophodno je kreirati novi fajl i u njega kopirati celokupan sadržaj originalnog fajla. Ovo može zahtevati vreme koje je proporcionalno veličini fajla koji se kopira. Naročito drastičan slučaj može biti kopiranje celih direktorijuma, što podrazumeva kopiranje svih fajlova i direktorijuma koji se u njemu nalaze. Sa druge strane, premeštanje fajla ne podrazumeva fizičko premeštanje njegovog sadržaja na disku, već samo premeštanje reference na fajl iz jednog direktorijuma u drugi. Ova operacija se obavlja vrlo efikasno, bez obzira na veličinu fajla ili direktorijuma koji se premešta.

Ponekad je potrebno obezbediti da referenca na isti fajl ili direktorijum bude dostupna na više različitih lokacija. Da bi se ovo postiglo, pod UNIX-om možemo koristiti *simboličke linkove* (engl. *symbolic link*). Simbolički link predstavlja specijalnu vrstu fajla čija je jedina uloga da referiše na drugi postojeći fajl. Možemo ga kreirati komandom `ln`:

```
ln -s /home/pera/fajl.txt lfajl.txt
```

Ovom komandom se u tekućem direktorijumu kreira fajl `lfajl.txt` koji predstavlja simbolički link koji ukazuje na fajl `/home/pera/fajl.txt`. U nastavku, svaka operacija nad simboličkim linkom (izuzev brisanja i premeštanja) se ne odnosi na link, već na fajl na koji link ukazuje. Na primer:

```
cp lfajl.txt /tmp/kopija.txt
```

Elektronska verzija (2024)

Ovim se sadržaj fajla `/home/pera/fajl.txt` kopira u fajl `/tmp/kopija.txt`. Simbolički link zauzima veoma malo prostora na disku (taman koliko je dovoljno da se sačuva putanja do fajla na koji se ukazuje). Može ukazivati na fajl ili direktorijum u bilo kom sistemu datoteka koji je montiran. Simbolički linkovi sami po sebi nemaju prava pristupa, već se prilikom utvrđivanja prava pristupa koriste prava fajla na koji link ukazuje. Takođe, komanda `chmod` menja prava pristupa fajla na koji link ukazuje:

```
chmod go-rwx lfajl.txt
```

Ovom komandom se članovima grupe i ostalim korisnicima oduzimaju sva prava nad fajlom `/home/pera/fajl.txt`.

Pretraga fajlova

Često nam je potrebno da u okviru sistema datoteka pronađemo fajl ili fajlove sa određenim karakteristikama. Te karakteristike se mogu odnositi na naziv fajla, tip fajla, veličinu fajla, vreme poslednje modifikacije, ili na prava pristupa. Pod UNIX-om, u ovu svrhu se može koristiti komanda `find`:

```
find /home/pera/ -name '*.html'
```

Ova komanda će pronaći i izlistati sve fajlove u okviru direktorijuma `/home/pera` (i njegovih poddirektorijuma) čije ime (opcija `-name`) je oblika `*.html`, gde `*` označava bilo koji niz karaktera. Apostrofi oko obrasca `*.html` su neophodni, da bi se sprečila ekspanzija simbola `*` u okviru same ljuske. Slično, ako pozovemo komandu:

```
find /home/pera/ -size +5M
```

dobićemo spisak svih fajlova iz direktorijuma `/home/pera` čija je veličina 5Mb ili više (umesto simbola `+` mogao je stajati simbol `-`, što bi nam dalo sve fajlove veličine ne veće od 5Mb). Ako pozovemo komandu:

```
find /home/pera/ -perm -u=w,g=w
```

dobićemo spisak svih fajlova u direktorijumu `/home/pera` kod kojih i vlasnik (`u`) i članovi grupe vlasnika (`g`) imaju pravo `w`. Ako umesto toga pozovemo komandu:

```
find /home/pera/ -perm /u=w,g=w
```

dobićemo spisak svih fajlova kod kojih *ili* vlasnik *ili* članovi grupe vlasnika imaju pravo `w`. Dakle, prefiks `-` označava da sva navedena prava moraju da postoje, dok prefiks `/` označava da bar jedno od navedenih prava mora da postoji. Možemo vršiti pretragu i prema vremenu poslednje modifikacije. Na primer:

```
find /home/pera -mtime 0
```

Ova komanda će izlistati sve fajlove iz direktorijuma `/home/pera` koji su modifikovana u poslednja 24 časa. Naime, argument iza opcije `-mtime` predstavlja broj *celih* dana koji su protekli od poslednje modifikacije fajla (eventualni ostatak vremena se odbacuje). Vrednost `0` stoga označava da je prošlo nula celih dana, tj. manje od 24 sata. Vrednost `+1` bi označavala bar jedan ceo dan, tj. više od 24 sata, dok bi vrednost `-5` označavala najviše 5 celih dana. Ako želimo da pronađemo sve obične fajlove (tj. da izostavimo direktorijume), imamo sledeću komandu:

```
find /home/pera -type f
```

Tip `f` označava obične fajlove, dok tip `d` označava direktorijume. Tip `l` označava simboličke linkove. Možemo pretraživati sve fajlove koji su u vlašnistvu nekog korisnika:

```
find /home/pera -user mika
```

Ova komanda izlistava sve korisnike u direktorijumu `/home/pera` čiji je vlasnik `mika`. Moguće je navoditi i više različitih uslova:

```
find /home/pera -name '*.html' -user mika -perm -u=rw
```

Ova komanda pronalazi sve fajlove u direktorijumu `/home/pera` sa imenom oblika `*.html`, u vlasništvu korisnika `mika`, pri čemu vlasnik poseduje prava `r` i `w`. Slično, komanda:

```
find /home/pera -name '*.html' '!' -user pera
```

Elektronska verzija (2024)

Pronalazi sve fajlove u direktorijumu `/home/pera` sa imenom oblika `*.html` koji *nisu* u vlasništvu korisnika `pera`. Negacija uslova postiže se navođenjem operatora `!` ispred odgovarajućeg uslova (apostrofi oko `!` su neophodni da bi se izbeglo specijalno značenje koje operator `!` ima u ljusci).

Komanda `find` podrazumevano izlistava pronađene fajlove. Ako želimo da se nad pronađenim fajlovima izvrši neka druga akcija, možemo je navesti u nastavku komande:

```
find . -name '*.html' -exec cat '{}'
```

Ovom komandom se na sve pronađene fajlove redom primenjuje komanda `cat`. Ova komanda prosto izlistava *sadržaj* fajla na standardnom izlazu. Oznaka `{}` označava putanju do pronađenog fajla (apostrofi su ponovo neophodni, kako bi se izbeglo specijalno značenje koje simboli `{` i `}` imaju u ljusci). Slično komanda:

```
find . -name '*~' -exec rm '{}'
```

briše sve fajlove iz tekućeg direktorijuma čije se ime završava sa `~`. Ekvivalentni efekat bi imala komanda:

```
find . -name '*~' -delete
```

1.3.4 Upravljanje procesima

U prethodnim odeljcima videli smo na koji načini se mogu pokretati procesi u okviru komandne linije. U ovom odeljku detaljnije razmatramo upravljanje procesima.

Prednji i pozadinski procesi

Za proces pokrenut u komandnoj liniji (ili grupu procesa, u slučaju nadovezivanja programa) kažemo da je *prednji* (engl. *foreground*), ako su mu standardni ulaz i izlaz povezani sa tastaturom i monitorom. Ovo je podrazumevano ponašanje prilikom pokretanja procesa u komandnoj liniji – komandni interfejs predaje pokrenutom procesu kontrolu nad tastaturom i monitorom i nadalje korisnik komunicira sa tim procesom umesto sa ljuskom. Kada se proces završi, kontrola nad tastaturom i monitorom se vraća ljusci, čime je omogućeno pokretanje novih komandi.

Sa druge strane, proces koji je pokrenut može da ne želi da komunicira sa korisnikom putem tastature i monitora. Postoje procesi koji se pokreću da bi obavili neki drugi zadatak koji ne podrazumeva komunikaciju sa korisnikom. Ako taj proces, pritom, radi dugo, korisniku će ljuska biti blokirana dok se proces ne završi. Umesto toga, postoji mogućnost da se proces pokrene *u pozadini*. Ovakve procese nazivamo *pozadinski procesi* (engl. *background*). Sintaksa pozivanja procesa u pozadini podrazumeva dodavanje simbola `&` na kraj komande:

```
/usr/sbin/sshd &
```

Ovom komandom se pokreće program `/usr/sbin/sshd` u pozadini. To znači da ljuska neće tom procesu predati kontrolu nad tastaturom i monitorom, već će tu kontrolu zadržati za sebe, što će omogućiti pokretanje drugih komandi dok prethodno pokrenuti proces radi u pozadini.

Mehanizam pokretanja procesa u pozadini se obično koristi za pokretanje procesa koji rade jako dugo, bez ikakve interakcije sa korisnikom. Ovakve procese nazivamo *demonski procesi* (engl. *deamon process*). Ovi procesi, iako mogu biti pokrenuti iz ljuske, nisu vezani za ljusku i mogu nastaviti svoj rad čak i kada se ljuska isključi, pa čak i kada se korisnik odjavi sa sistema. Najčešća upotreba demonskih procesa je za implementaciju različitih *servera* – procesa čija je uloga da pružaju različite usluge klijentskim programima (poput servera štampe, veb servera i sl.). Serveri su obično aktivni tokom čitavog rada računara i operativnog sistema.

Druga upotreba pokretanja procesa u pozadini je na sistemima sa grafičkim korisničkim interfejsom, kada iz ljuske (pokrenute u okviru emulatora terminala) želimo da pokrenemo neku aplikaciju sa grafičkim korisničkim interfejsom (poput veb pregledača, programa za tabelarna izračunavanja i sl.). Ovi programi se mogu pokrenuti iz ljuske, ali oni neće koristiti standardni ulaz i izlaz, već će kreirati poseban prozor na ekranu i komuniciraće sa korisnikom putem svog grafičkog interfejsa. Sa druge strane, ljuska će biti bespotrebno blokirana i neće biti moguće pokretati druge programe. Da bi se to sprečilo, možemo pokrenuti aplikaciju u pozadini:

```
firefox &
```

Aplikacija pokrenuta u pozadini se može vratiti u status prednjeg procesa komandom:

```
fg
```

Elektronska verzija (2024)

Ukoliko u nekom trenutku imamo više procesa pokrenutih u pozadini, tada svaki od njih ima dodeljen broj (ovaj broj se ispisuje nakon pokretanja procesa u pozadini). Ako neki od tih procesa želimo da prebacimo u status prednjeg procesa, tada komandi `fg` moramo navesti broj tog pozadinskog procesa:

```
fg 1
```

Takođe, proces koji je pokrenut u pozadini može i sam preći u status prednjeg procesa, ako pokuša da vrši ispis na standardni izlaz.

Sa druge strane, ako je neki proces greškom pokrenut kao prednji proces, možemo ga naknadno prebaciti u status pozadinskog procesa. To radimo tako što najpre na tastaturi pritisnemo kombinaciju tastera `Ctrl-Z`, a zatim u komandnoj liniji zadamo komandu:

```
bg
```

Pregled pokrenutih procesa

Pregled pokrenutih procesa može se obaviti komandom:

```
ps
```

Ova komanda izlistava sve aktivne procese pokrenute u okviru ljuske iz koje se komanda pokreće. Ovo obično uključuje samu ljusku kao i proces `ps`. Uz to, prikazuju se i pozadinski procesi pokrenuti iz te ljuske. Ukoliko želimo da prikažemo sve procese u vlasništvu nekog određenog korisnika, možemo pokrenuti komandu:

```
ps -u pera
```

Podrazumevano, za svaki od procesa prikazuje se PID, terminal u okviru koga je pokrenut, vreme izvršavanja procesa, kao i naziv programa. Po želji, različitim opcijama se mogu zahtevati i dodatne informacije o procesima. Na primer:

```
ps -u pera -o user,pid,ppid,state,time,command
```

Opcijom `-o` i argumentima koji (razdvojeni zarezima) slede iza nje zadaju se parametri procesa koje želimo da komanda ispiše. U našem primeru, želimo da za svaki proces ispišemo njegovog vlasnika (`user`), njegov PID (`pid`), kao i PID njegovog roditelja (`ppid`), zatim stanje procesa (`state`), vreme izvršavanja (`time`) i punu komandu (`command`) kojom je proces pokrenut, sa sve putanjom do programa i argumentima komandne linije. Stanje procesa se označava slovima poput `R` (*radi* ili *spreman*), `S` (*čeka*), i sl. Neki uobičajeni ispis koji uključuje najznačajnije parametre procesa (poput ovih koje smo u prethodnoj komandi eksplicitno navodili) se može postići i zadavanjem opcije `-F`:

```
ps -u pera -F
```

Ako želimo da prikažemo sve procese svih korisnika, možemo zadati komandu:

```
ps -e -F
```

Pored programa `ps`, postoji i program `top` koji omogućava interaktivni i dinamički prikaz procesa koji se izvršavaju na sistemu. Program se pokreće jednostavno:

```
top
```

Nakon pokretanja ovog programa, u okviru ljuske se prikazuje tabela sa procesima i njihovim najznačajnijim parametrima. Ta tabela se podrazumevano osvežava na svake tri sekunde, tako da se mogu kontinuirano pratiti resursi koje procesi zauzimaju (poput fizičke i virtuelne memorije i procesorskog vremena). Takođe, u gornjem delu ekrana prikazuje se ukupno zauzeće resursa od strane svih procesa.

Elektronska verzija (2024)

Prekidanje rada procesa

Ako želimo da neki proces nasilno prekinemo, možemo mu iz ljuske poslati signal za prekid. Ovo se radi komandom `kill`:

```
kill 2456
```

Ovim se procesu sa PID-om 2456 šalje signal `SIGTERM` za koji je podrazumevana akcija prekid rada procesa. Pojedini procesi mogu ignorisati ovaj signal ili biti programirani da obavljaju neku drugu akciju prilikom pristizanja tog signala. Alternativno, možemo pokrenuti komandu:

```
kill -s SIGKILL 2456
```

Ovim se procesu sa PID-om 2456 šalje signal `SIGKILL` koji se ne može ignorisati, niti je moguće redefinisati njegovo značenje, tako da garantovano ubija proces.

PID procesa koji želimo da likvidiramo uvek možemo da vidimo na izlazu komande `ps`. Ipak, ako nam je zgodnije da navedemo ime programa koji želimo da likvidiramo, možemo pokrenuti komandu:

```
killall -s SIGKILL firefox
```

Da bismo mogli da pošaljemo signal procesu, taj proces mora biti u vlasniku korisnika koji pokreće `kill` komandu, osim u slučaju `root` korisnika koji ima prava da pošalje signal bilo kom procesu.

Elektronska verzija (2024)