Procesi

1 Uvod

Program koji se izvršava naziva se *proces*. Da bi se izvršavao, procesu će biti potrebni određeni resursi - poput CPU-a, memorije, datoteka i U/I uređaja - da bi obavio svoj zadatak. Ovi resursi se dodeljuju procesu ili kada se proces kreira ili tokom njegovog izvršavanja.

Sistemi se sastoje od skupa procesa: procesi operativnog sistema izvršavaju sistemski kod, a korisnički procesi izvršavaju korisnički kod. Svi se ti procesi mogu istovremeno izvršavati.

Iako je, tradicionalno, proces sadržao samo jednu *nit* (thread) tokom svog izvršavanja, većina modernih operativnih sistema sada podržava procese koji imaju više niti izvršavanja. Niti su izuzetno značajne i njima ćemo posvetiti celo naredno predavanje.

Operativni sistem je odgovoran za nekoliko važnih aspekata upravljanje procesima i nitima:

- stvaranje i brisanje kako korisničkih, tako i sistemskih procesa;
- raspoređivanje (scheduling) procesa i
- obezbeđivanje mehanizama za sinhronizaciju, komunikaciju i razrešavanje mrtve petlje za procese.

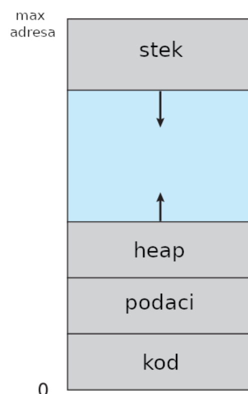
Prvobitni računari omogućavali su istovremeno izvršavanje samo jednog programa. Ovaj program je imao potpunu kontrolu nad sistemom i imao je pristup svim resursima sistema. Nasuprot tome, savremeni računarski sistemi omogućavaju učitavanje više programa u memoriju i istovremeno izvršavanje. Ova evolucija zahtevala je čvršću kontrolu i omogućavanje deljenja informacija između različitih programa; a ove potrebe rezultirale su pojmom *procesa* koji se odnosi na program koji se izvršava. Samim tim, proces je elementarna jedinica rada u savremenom računarskom sistemu deljenja vremena (time-sharing, multitasking).

Što je operativni sistem složeniji, više posla mora da se obavi za svakog pojedinačnog korisnika. Iako je glavna briga operativnog sistema izvršavanje korisničkih programa, on takođe treba da vodi računa i o raznim sistemskim zadacima koje je, iz više razloga, bolje izdvojiti iz samog kernela operativnog sistema.

Sistem se, dakle, sastoji od skupa procesa: procesi operativnog sistema koji izvršavaju sistemski kod i korisnički procesi koji izvršavaju korisnički kod. Potencijalno, svi ovi procesi se mogu istovremeno izvršavati, a CPU (ili više CPU-ova) se dele među njima. Deljenjem CPU-a između procesa, operativni sistem može da učini računar produktivnijim.

2 Koncept procesa

Pitanje koje se postavlja u svakoj diskusiji u vezi sa operativnim sistemima, odnosi se na to kako nazvati sve aktivnosti CPU-a. Paketni (batch) sistem izvršava poslove - *jobs* (Rani računarski sistemi predstavljeni od strane IBM-a



Slika 1: Proces u memoriji

60-tih godina 20. veka, predstavljali su mašine koje su izvršavale jedan ili više programa kroz tzv. skripte), dok sistem koji deli vreme (*time-sharing*) ima korisničke programe ili zadatke. U takvim sistemima, čak i u sistemu jednog korisnika, korisnik može biti u stanju da pokrene više programa odjednom: program za obradu teksta, web pretraživač i klijent za elektronsku poštu. Čak i ukoliko korisnik može da izvršava samo jedan program u datom trenutku, kao što je na embeded uređaju koji ne podržava multitasking i izvršavanje više zadataka odjednom, operativni sistem će, najverovatnije, trebati da podrži svoje interne aktivnosti, kao što je upravljanje memorijom. U mnogim aspektima sve ove aktivnosti su slične, pa ih sve nazivamo procesima. Pojmove zadatak i proces ćemo u nastavku ravnopravno koristiti. Iako preferiramo termin *proces*, veliki deo teorije i terminologije operativnog sistema je razvijen u vreme kada je glavna aktivnost operativnih sistema bila obrada zadataka. Stoga, bilo bi pogrešno izbegavati upotrebu opšteprihvaćenih izraza koji uključuju reč zadatak (kao što je raspoređivanje zadataka - task scheduling) samo zato što je termin proces vremenom zamenio zadatak.

Neformalno, kao što je ranije spomenuto, proces je program koji se izvršava. Međutim, proces je znatno više od samog programskog koda, koji je često poznat i kao *sekcija teksta* (*text section*). Proces, takođe, uključuje u sebe i trenutno stanje izvršavanja programa, predstavljenu vrednošću programskog brojača i sadržajem registara procesora. Proces uglavnom uključuje i procesni *stek* koji sadrži privremene podatke (kao što su parametri funkcija, povratne adrese i lokalne promenljive) i *sekciju sa podacima* (*data section*) koja sadrži globalne promenljive. Proces može takođe da uključi *heap* memoriju, koja predstavlja memoriju koja se dinamički dodeljuje tokom izvršavanja procesa. Struktura procesa u memoriji prikazana je na slici 1.

Imajući u vidu sve ovo, očigledno je da program sam po sebi *nije proces*. Program je pasivan entitet, kao što je datoteka koja sadrži spisak instrukcija sačuvanih na disku (često se, samim tim, naziva izvršna datoteka). Nasuprot

tome, *proces je aktivni entitet*, jer uključuje i programski brojač koji određuje narednu instrukciju za izvršenje i skup pridruženih resursa. Program *postaje* proces kada se izvršna datoteka učita u memoriju. Dve uobičajene tehnike za učitavanje izvršnih datoteka su:

- dvostruki klik na ikonicu koja predstavlja izvršnu datoteku i
- unošenje imena izvršne datoteke u komandnu liniju (kao u *prog.exe* ili *a.out*).

Iako dva (ili više) procesa mogu nastati iz istog programa, ipak se oni smatraju kao dve odvojene sekvence izvršavanja. Na primer, nekoliko korisnika može izvršavati različite kopije kompajliranog C programa, ili jedan isti korisnik može pozvati više kopija programa web pretraživača. Svaka od njih je poseban *proces*; i iako su *sekcije teksta* ekvivalentne, *sekcije podataka*, *heap* i *stek* se razlikuju.

Takođe je uobičajeno da proces može da pokreće druge procese dok se izvršava. O takvim situacijama ćemo detaljnije raspravljati kasnije.

Treba imati na umu da sam proces može obezbediti okruženje u kojem se izvršava drugi kod. Java programsko okruženje je dobar primer za ovo. U većini slučajeva izvršni Java program se izvršava u okviru Java virtualne mašine (JVM). JVM se izvršava kao proces koji interpretira učitani Java kod i izvodi akcije (korišćenjem instrukcija iz instrukcijskog seta za datu mašinu) u ime tog koda. Na primer, da bismo pokrenuli kompajlirani Java program *Student.class*, koristimo komandu

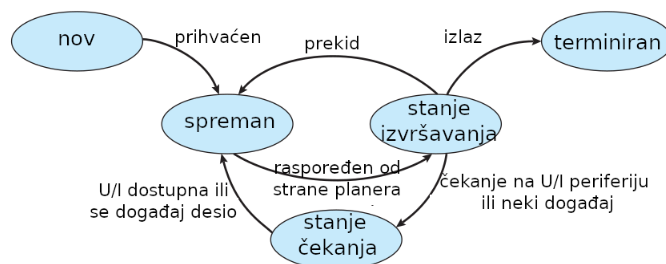
```
java Student
```

Komanda *java* pokreće JVM kao običan proces, koji zauzvrat izvršava Java program Student u virtualnoj mašini. Koncept je isti kao u slučaju emulacija, s tim što je kod, umesto da je napisan za drugi set instrukcija, napisan na Java programskom jeziku.

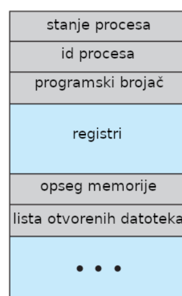
2.1 Stanja procesa

Tokom izvršavanja procesa, on menja svoje *stanje*. Stanje procesa je delom definisano trenutnom aktivnošću tog procesa, proces može biti u jednom od sledećih stanja:

- Nov (*New*): Proces je upravo kreiran;
- Izvršavanje (*Running*): Instrukcije se izvršavaju;
- Čekanje (*Waiting*): Proces čeka da se dogodi neki događaj (poput završetka U/I ili prijema signala);
- Spreman (*Ready*): Proces čeka da bude dodeljen procesoru;
- Terminiran (*Terminated*): Proces je terminiran.



Slika 2: Dijagram stanja procesa



Slika 3: Kontrolni blok procesa

Gornji nazivi su proizvoljni i razlikuju se od operativnog sistema do operativnog sistema. Međutim, stanja koja oni predstavljaju mogu se pronaći na svim sistemima.

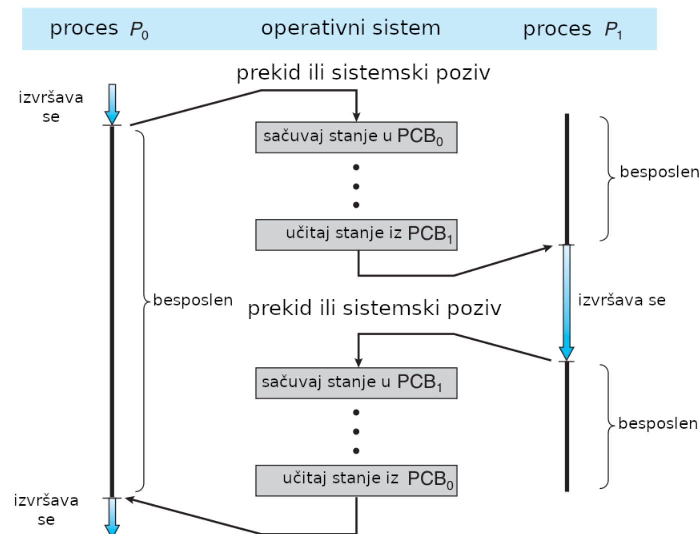
Određeni operativni sistemi, takođe, dodatno, fino razgraničavaju gornje navedena stanja procesa, pa svako od gornjih stanja ima svoja podstanja. Ipak, važno je shvatiti da samo jedan proces može biti u stanju ***izvršavanja*** na bilo kojem procesoru u bilo kojem trenutku. Mnogi drugi procesi su možda spremni i čekaju. Dijagram stanja koji odgovara iznad definisanim stanjima prikazan je na slici 2.

2.2 Kontrolni blok procesa (Process Control Block - PCB)

Svaki proces je u operativnom sistemu predstavljen *kontrolnim blokom procesa* (PCB), koji se takođe naziva i *kontrolnim blokom zadatka* (*task control block*). PCB je prikazan na slici 3.

PCB sadrži mnogo informacija koje se odnose na određeni proces, uključujući:

- Stanje procesa: Stanje je možda nov, spreman, u izvršavanju, čeka, terminiran



Slika 4: Zamena procesa koji se izvršava na jednom procesoru

- Programski brojač: pokazuje na adresu sledeće instrukcije koja će se izvršiti za u okviru ovog procesa
- Registri CPU-a: Sami registri se razlikuju u broju i vrsti, u zavisnosti od arhitekture računarskog sistema. Oni gotovo uvek uključuju akumulatore, indeksne registre, pokazivače steka i registre opšte namene, statusne registre. Zajedno sa programskim brojačem, ove informacije o stanju moraju se sačuvati kada dođe do prekida, kako bi se omogućilo da se proces pravilno nastavi kada dođe vreme za to (slika 4)
- Informacije o raspoređivanju CPU: Ove informacije uključuju prioritet procesa, pokazatelje na redove čekanja i bilo koje druge parametre.
- Informacije o upravljanju memorijom: Ove informacije mogu sadržavati stavke kao što su vrednost baznih registara (*base register*) i registara ograničenja (*bound register*) ili tabela stranica, zavisno od memorijskog sistema koji koristi operativni sistem
- Podaci za praćenje rada: Ove informacije uključuju odnos korišćenog CPU vremena i realnog vremena, vremenske rokove, identifikator procesa i tako dalje
- Informacije o U/I statusu: Ove informacije uključuju listu U/I uređaja dodeljenih procesu, spisak otvorenih datoteka i tako dalje.

Ukratko, PCB jednostavno služi kao spremište svih informacija koje mogu varirati od procesa do procesa.

2.3 Niti

Do sada je diskutovani model procesa podrazumevao da je proces program koji izvodi jednu nit (*thread*) izvršenja. Na primer, kada proces pokreće program za obradu teksta, izvršava se jedna nit instrukcija. Ova jednostruka nit izvršavanja omogućava procesu da obavlja samo jedan zadatak u datom trenutku. Korisnik tada ne bi mogao istovremeno tipkati tastere i pokrenuti proveru pravopisa, na primer, u okviru istog procesa. Većina modernih operativnih sistema proširila je koncept procesa kako bi omogućila da proces ima višestruko izvršavanje i na taj način izvršava više zadataka odjednom. Ova karakteristika je posebno korisna na višejezgarnim (*multicore*) sistemima, gde više niti može paralelno da se izvršava. Na sistemu koji podržava *niti*, PCB se proširuje kako bi uključio informacije za svaku pojedinačnu nit. Međutim, to nije dovoljno i dodatne promene u sistemu su takođe neophodne u cilju pružanja podrške za niti. Celo jedno predavanje ćemo posvetiti toj opširnoj temi.

2.4 Primer: Reprezentacija procesa u Linux-u

Kontrolni blok procesa u Linux operativnom sistemu predstavljen je C strukturom *task_struct*, koja se nalazi u <linux/sched.h> datoteci u direktorijumu izvornog koda kernela. Ova struktura sadrži sve potrebne informacije za predstavljanje procesa, uključujući stanje procesa, informacije o raspoređivanju i upravljanju memorijom, listu otvorenih datoteka i pokazivače na roditelja procesa, njegovu decu, kao i braću procese (*siblings*). Roditelj procesa je proces koji ga je stvorio, njegova deca su bilo koji procesi koje on stvara. Njegova braća su deca-procesi od istog roditelj procesa. Neka od ovih polja uključuju:

```
long state; /* state of the process */
struct sched_entity se; /* scheduling information */
struct task_struct *parent; /* parent of this process*/
struct list_head children; /* children of this process  */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```

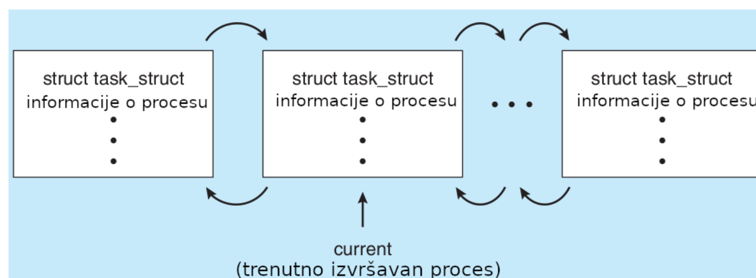
Na primer, stanje nekog procesa predstavljeno je poljem *state* u ovoj strukturi. Unutar Linux kernela svi aktivni procesi predstavljeni su korišćenjem dvostruko povezane liste *tast_struct* struktura. Kernel održava pokazivač - *current* - za proces koji se trenutno izvršava na sistemu, kao što je prikazano na slici ispod:

Kao ilustracija toga kako kernel manipuliše strukturama *task_struct* za dati proces, pretpostavimo da operativni sistem želi da promeni stanje procesa koji se trenutno izvršava u stanje *new_state*. Ako je *current* pokazivač procesa koji se trenutno izvršava, njegovo stanje se menja sa:

```
current->state = new state;
```

3 Raspoređivanje procesa

Cilj multi-programiranja je da se u svakom trenutku izvršava neki proces, kako bi se maksimizirala upotreba CPU-a. Sa druge strane, cilj deljenja vremena



Slika 5: Lista aktivnih procesa u Linux operativnom sistemu

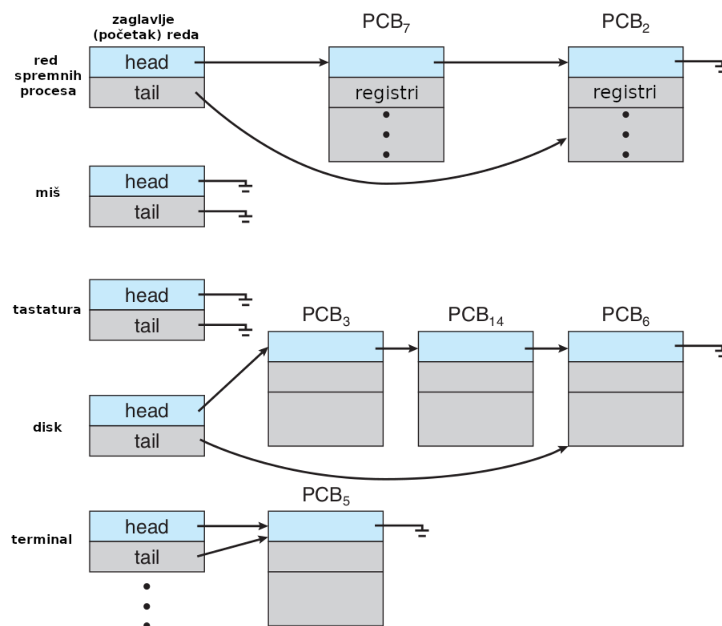
(*multitasking*-a) je raspoređivanje CPU-a između procesa na takav način i tako često da korisnici mogu komunicirati sa svakim programom dok se on izvršava. Da bi postigao ove ciljeve, planer procesa (*process scheduler*) odabira raspoloživi proces (moguće iz skupa nekoliko raspoloživih procesa) kako bi njemu dodelio CPU na izvršavanje. Za jedno-procesorski sistem to znači da će samo jedan od procesa moći da se izvršava u datom trenutku. Ako ima više procesa koji su spremni za izvršavanje, ostatak će morati da sačeka dok se taj jedini CPU ne oslobodi nakon čega tek može da bude dodeljen nekom drugom procesu.

3.1 Redovi raspoređivanja

Kako procesi dospevaju u sistem, postavljaju se u red procesa koji čine svi procesi u sistemu. Procesi koji ostaju u glavnoj memoriji, spremni su i čekaju da se izvrše, čuvaju se u listi koja se zove *red spremnih procesa* (eng *ready queue*). Ovaj *red spremnih procesa* se obično održava u formi *povezane liste*. Zaglavlje reda spremnih procesa sadrži pokazivače na prvi i poslednji PCB u listi. Sa druge strane, svaki PCB sadrži polje pokazivača koje pokazuje na PCB sledećeg procesa u redu čekanja.

Operativni sistem, takođe, održava i dodatne redove, osim reda spremnih procesa. Kada se nekom procesu dodeli CPU, on se izvršava određeni vremenski interval, nakon čega se zaustavlja, bilo usled prekida koji se desio, ili jer jednostavno mora da čeka na pojavu određenog događaja (kao što je, na primer, ispunjenje U/I zahteva). Pretpostavimo da proces zahteva U/I operaciju (kao što je čitanje ili upis) od strane deljenog uređaja (kao što je disk). Pošto, kao što smo rekli, postoji mnogo procesa u sistemu, disk će vrlo verovatno (da ne kažemo sigurno) biti zauzet obrađujući U/I zahtev nekog drugog procesa. Zbog toga će proces morati da sačeka da dođe na red i pristupi disku. Lista procesa koji čekaju određeni U/I uređaj naziva se *red uređaja* (*device queue*), pri čemu svaki uređaj ima svoj red uređaja (slika 6).

Uobičajeni prikaz raspoređivanja procesa je *dijagram čekanja* (eng *queueing diagram*), prikazan na slici 7. Svaki pravougaonik na slici predstavlja red čekanja. Postoje dve vrste reda čekanja: *red spremnih procesa* i skup *redova uređaja*. Krugovi predstavljaju resurse koji opslužuju redove, a strelice označavaju tok

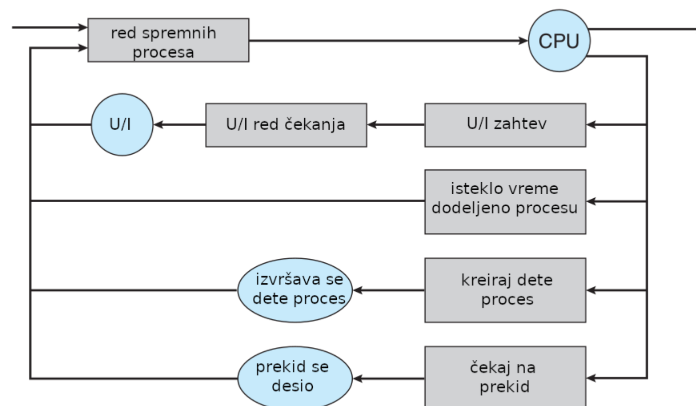


Slika 6: Red spremnih procesa i redovi različitih I/O uređaja

kretanja procesa u sistemu. Novi proces se, u početku, postavlja u red spremnih procesa. Tamo čeka dok nije izabran za izvršavanje, ili, kako se još kaže, *otpremljen* (eng *dispatched*). Jednom kada se procesu dodeli CPU i on krene da se izvršava, može se dogoditi jedan od nekoliko događaja:

- Proces bi mogao da generiše U/I zahtev i premesti se u red uređaja;
- Proces bi mogao da stvori novi proces (tzv. dete proces), nakon čega će da čeka da dete proces završi svoje izvršavanje;
- Procesu se može oduzeti CPU, kao posledica isteka vremena izvršavanja tog procesa, merenog od strane tajmera;
- Proces može čekati na prekid, a nakon što se prekid desi, proces će opet biti vraćen u red spremnih procesa.

U svim slučajevima osim u trećem, proces najpre prelazi u stanje čekanja, pa tek kasnije u stanje spreman kada će biti vraćen u red spremnih procesa. U trećem scenariju (kada proces izgubi kontrolu nad CPU kao posledica isteka dodeljenog mu vremena), on će se odmah premestiti u red spremnih procesa, obzirom na to da bi proces nastavio sa izvršavanjem da nije bio prekinut. Ovaj ciklus promena stanja procesa, opisan *dijagramom čekanja*, se nastavlja sve dok proces ne bude terminiran, kada će biti uklonjen iz svih redova čekanja, nakon čega će osloboditi svoj PCB i sve prethodno alocirane resurse.



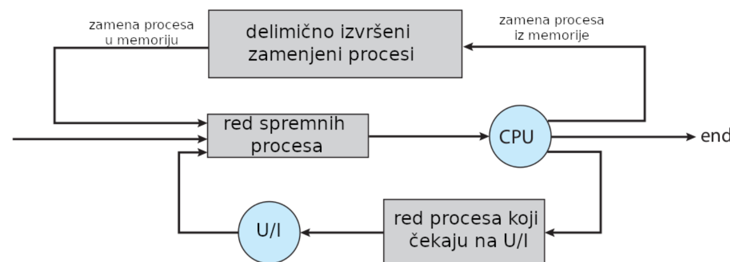
Slika 7: Dijagram čekanja

3.2 Planeri

Proces migrira među različitim redovima raspoređivanja tokom svog životnog veka. Operativni sistem mora, za potrebe raspoređivanja, po nekom pravilu odabirati procese iz tih redova. Postupak odabira procesa, u ovom smislu, sprovodi odgovarajući *planer*. Uglavnom se u sistemu pojavljuje više procesa nego što ih se može odmah izvršavati. Ovi procesi će biti prebačeni na uređaj za masovno skladištenje (obično disk), gde se čuvaju za kasnije izvršavanje. Dugoročni planer (eng *long-term scheduler*) ili *planer zadataka* bira procese iz ovog skupa i učitava ih u memoriju radi izvršavanja. Kratkoročni planer (eng *short-term scheduler*), ili CPU planer, bira između procesa koji se nalaze u memoriji spremni za izvršavanje i dodeljuje CPU jednom od njih.

Primarna razlika između ova dva planera je u učestanosti izvršavanja. *Kratkoročni planer* mora veoma često birati novi proces kojem će biti dodeljen CPU. Proces koji je odabran, tipično će se izvršavati samo nekoliko milisekundi pre nego što ne pošalje U/I zahtev ka određenoj periferiji. Često se kratkoročni planer izvršava najmanje jednom u svakih 100 milisekundi. Zbog kratkog vremena između izvršavanja, kratkoročni planer mora biti izuzetno brz. Ako je potrebno 10 milisekundi da se odabere proces koji će se izvršavati 100 milisekundi, tada će 9% (10 / (100 + 10)) vremena biti izgubljeno samo zbog raspoređivanja.

Dugoročni planer izvršava se mnogo ređe; minuta može proteći između stvaranja jednog novog procesa i sledećeg. Dugoročni planer kontroliše stepen multiprogramiranja (broj procesa u memoriji). Ako je stepen multi-programiranja stabilan, prosečna stopa stvaranja procesa mora biti jednaka prosečnoj stopi terminiranja procesa. Stoga će se pozivati dugoročni planer samo kada je proces terminiran i napušta sistem. Zbog dužeg intervala između sukcesivnih izvršenja, dugoročni planer može priuštiti sebi više vremena da odluči koji će proces biti izabran za izvršavanje. Važno je da dugoročni planer pažljivo odabira procese, kao što ćemo videti u nastavku.



Slika 8: Dodatak dijagramu čekanja za srednjeročni planer

Uopšteno, većina procesa može se svrstati u U/I kontrolisane ili CPU kontrolisane procese (eng *I/O bound* i *CPU bound*). U/I kontrolisan proces je onaj koji troši više svog vremena radeći sa ulazno/izlaznim uređajima, nego što troši na računanje sa podacima bez potrebe za U/I operacijama. Proces koji je CPU kontrolisan, nasuprot tome, generiše U/I zahteve retko, koristeći najveći deo svog svog vremena procesirajući podatke. Stoga, izuzetno je važno da dugoročni planer izabere dobru i izbalansiranu kombinaciju U/I kontrolisanih i CPU kontrolisanih procesa. Ako su svi procesi odabrani od strane dugoročnog planera U/I kontrolisani, red spremnih procesa će gotovo uvek biti prazan, a kratkoročni planer će, samim tim, uglavnom biti besposlen. Ako su svi procesi, sa druge strane, CPU kontrolisani, red čekanja uređaja gotovo uvek će biti prazan, uređaji će ostati neiskorišćeni, a sistem će ponovo biti neuravnotežen. Sistem sa najboljim performansama imaće, dakle, balansiranu kombinaciju procesa koji su podjednako CPU kontrolisani i U/I kontrolisani.

Na nekim sistemima dugoročni planer ne postoji ili je minimalno korišćen. Na primer, multitasking sistemi kao što su Unix i Microsoft Windows tipično nemaju dugoročni planer, već se svaki novi proces jednostavno dostavlja kratkoročnom planeru.

Sa druge strane, neki multitasking operativni sistemi mogu uvesti dodatni, srednji nivo raspoređivanja. Ovaj srednjeročni planer (*medium-term scheduler*) je prikazan na slici 8. Ključna ideja srednjeročnog planera je da ponekad može biti korisno uklanjanje procesa iz memorije, kako bi se na taj način smanjio stepen multi-programiranja. Kasnije se proces može ponovo vratiti u memoriju, a njegovo izvršavanje može se nastaviti tamo gde je prethodno stao. Ova šema se naziva *zamena* (eng. *swapping*). Proces se *zamenjuje iz memorije*, a kasnije ga *zamenjuje u memoriju* srednjeročni planer. Zamena se uglavnom obavlja kako bi se poboljšala kombinacija procesa (U/I i CPU kontrolisani) ili jednostavno zato što je došlo do promene u zahtevima za memorijskim resursima, pa se prepunila dostupna memorija, te je potrebno osloboditi je.

3.3 Zamena konteksta (Context switch)

Kao što je spomenuto ranije, prekidi uzrokuju da operativni sistem oduzme CPU procesu koji se trenutno izvršava i pokrene kernel rutinu. Ovakve operacije

se često dešavaju na sistemima opšte namene. Kada dođe do prekida, sistem treba da sačuva trenutni *kontekst* procesa koji se izvršava na CPU-u da bi mogao kasnije da rekonstruiše taj kontekst kada se izvrši obrada prekida, i nastavi prethodno obustavljeni proces. Kontekst se čuva u kontrolnom bloku procesa (PCB): on uključuje vrednost registara CPU-a, stanje procesa (vidi sliku 2) i informacije o upravljanju memorijom. Uopšteno, na ovaj način vršimo čuvanje trenutnog stanja CPU-a, bilo da se radi o kernel ili korisničkom modu rada, a zatim restauriramo prethodno stanje kako bismo nastavili prethodno prekinuto izvršavanje.

Slično kao u slučaju prekida, prebacivanje CPU-a drugom procesu takođe zahteva čuvanje stanja trenutnog procesa i rekonstrukciju stanja drugog procesa. Ovaj postupak je poznat kao *zamena konteksta*. Kada dođe do *zamene konteksta*, kernel čuva kontekst starog procesa u njegov PCB i učitava sačuvani kontekst novog procesa koji je potrebno pokrenuti.

Vreme zamene konteksta, iako je zamena konteksta neophodna, je zapravo uzaludno potrošeno vreme u sistemu, jer sistem ne može efikasno da se koristi tokom zamene. Brzina zamene varira od mašine do mašine, u zavisnosti od brzine rada memorije, broja registara koji se moraju kopirati i eventualnog postojanja posebnih instrukcija (kao što je na primer specifična instrukcija za učitavanje/čuvanje svih registara odjednom). Tipična brzina trajanja zamene konteksta je nekoliko milisekundi.

Vreme zamene konteksta u velikoj meri zavisi od hardverske podrške. Na primer, neki procesori (kao što je Sun Ultra SPARC) pružaju više skupova registara. Zamena konteksta kod njih jednostavno zahteva promenu pokazivača na trenutni skup registara. Naravno, ako postoji više aktivnih procesa od skupova registara, sistem ne može da izbegne kopiranje podataka iz registara u memoriju i iz nje, kao i ranije. Takođe, što je složeniji operativni sistem, veća je količina posla koji se mora obaviti tokom zamene konteksta. Kao što ćemo videti kasnije, napredne tehnike upravljanja memorijom mogu zahtevati prebacivanje dodatnih podataka sa svakim kontekstom. Na primer, adresni prostor tekućeg procesa mora biti sačuvan, jer je prostor sledećeg procesa pripremljen za upotrebu. Kako će se sačuvati adresni prostor i koja količina posla je potrebna da bi se on sačuvao, zavisi od načina upravljanja memorijom operativnog sistema.

Primer: Multitasking u operativnim sistemima iOS i Android

Zbog ograničenja koja su nametnuta mobilnim uređajima, rane verzije iOS-a nisu omogućavale multitasking za korisničke aplikacije. Samo jedna aplikacija se izvršava u prvom planu (*foreground application*), a sve ostale korisničke aplikacije su suspendovane. Zadaci operativnog sistema su mogli da koriste multitasking jer ih je razvio Apple, bili su dobro testirani i dobro su se ponašali. Međutim, počevši od iOS-a 4, Apple pruža ograničen oblik multitasking-a za korisničke aplikacije, omogućavajući tako da se jedna aplikacija u prvom planu pokreće istovremeno sa više pozadinskih aplikacija. (Na mobilnom uređaju aplikacija koja se izvršava u prednjem planu je aplikacija koja je trenutno otvorena i pojavljuje

se na ekranu. Pozadinska aplikacija - *background application* - ostaje u memoriji, ali ne zauzima ekran.) API za programiranje iOS 4 pruža podršku za obavljanje više zadataka istovremeno, omogućujući tako da se proces izvršava u pozadini, a da se ne suspenduje. Međutim, multitasking je i dalje ograničen i dostupan samo ograničenom broju tipova aplikacija, uključujući aplikacije koje

- izvršavaju jedan zadatak konačne dužine (kao što je preuzimanja sadržaja sa Interneta);
- primaju obaveštenja o događaju (kao što je nova e-mail poruka);
- su sa dugotrajnim pozadinskim zadacima (kao što je audio plejer.)

Apple verovatno ograničava multitasking zbog trajanja baterije i u cilju efikasnije upotrebe memorije. Sam CPU definitivno ima mogućnosti za podršku multitaskinga, ali Apple odlučuje da ne iskoristi neke od njih kako bi bolje upravljao upotrebom resursa.

Android, sa druge strane, ne postavlja takva ograničenja za vrste aplikacija koje se mogu izvršavati u pozadini. Ako aplikacija zahteva obradu dok je u pozadini, aplikacija mora koristiti *servise*, zasebne komponente aplikacije koje se pokreću u vlasništvu pozadinskog procesa. Na primeru aplikacije za *streaming* zvuka: ako se aplikacija premesti u pozadinu, *servis* nastavlja da šalje audio datoteke drajveru audio uređaja u ime pozadinske aplikacije. U stvari, servis će se nastaviti izvršavati čak i ako je pozadinska aplikacija suspendovana. Servisi nemaju korisnički interfejs i troše malo memorije, pa pružaju efikasnu tehniku za obavljanje više zadataka istovremeno u mobilnom okruženju.

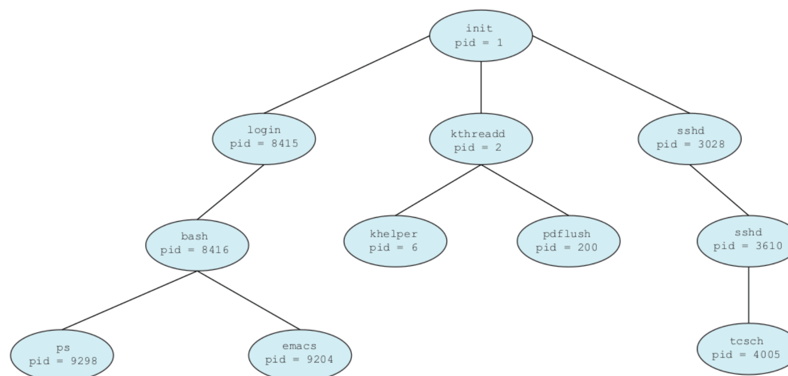
4 Operacije na procesima

Procesi u većini sistema, kao što smo rekli više puta do sada, se mogu istovremeno izvršavati, a mogu se kreirati i brisati dinamički. Stoga svi multitasking sistemi moraju da obezbede mehanizam za kreiranje i terminiranje procesa. U ovom odeljku prikazujemo mehanizme u vezi sa kreiranjem procesa i ilustrujemo nastanak procesa na Unix (Linux) sistemima.

4.1 Kreiranje procesa

Tokom izvršavanja proces može stvoriti mnoštvo novih procesa. Kao što je ranije pomenuto, proces koji kreira druge procese se zove proces roditelj, a novi procesi se nazivaju procesi deca (tog procesa). Svaki od ovih novih procesa može zauzvrat stvoriti druge procese, formirajući stablo procesa.

Većina operativnih sistema (uključujući Unix, Linux i Windows) identifikuje procese prema jedinstvenom identifikatoru procesa (ili pid-u, *process id*-u), koji je obično ceo broj. Pid pruža jedinstvenu vrednost za svaki proces u sistemu i može se koristiti kao indeks za pristup različitim atributima procesa unutar kernela.



Slika 9: Stablo procesa u Linux operativnom sistemu

Slika 9 prikazuje tipično stablo procesa za Linux operativni sistem, pokazujući naziv svakog procesa i njegov *pid*. *Init* proces (koji uvek ima pid 1) služi kao glavni roditeljski proces za sve korisničke procese. Nakon što se sistem pokrene, *init* proces takođe može da kreira mnogobrojne korisničke procese, kao što su web server ili print server, ssh server i slično. Na slici 9 vidimo troje dece procesa init procesa - *login*, *kthreadd* i *sshd*. Proces *kthreadd* je odgovoran za stvaranje dodatnih procesa koji izvršavaju zadatke u ime kernela (u ovom slučaju, *khelper* i *pdflush*). Proces *sshd* je odgovoran za upravljanje klijentima koji se povezuju na sistem sa udaljene lokacije korišćenjem *ssh* (što je skraćeno za *secure shell*). Proces *login* je odgovoran za upravljanje klijentima koji se direktno prijavljuju u sistem. U ovom primeru se klijent prijavio i koristi bash shell, kojem je dodeljen pid 8416. Koristeći bash interfejs komandne linije, ovaj korisnik je kreirao proces *ps* (proces pokrenut u cilju ispisa trenutno aktivnih procesa) kao i urеđivač teksta *emacs*. Na sistemima Unix i Linux, možemo dobiti spisak procesa pomoću komande *ps*. Na primer, komanda:

```
ps -el
```

će prikazati kompletne informacije za sve procese koji su trenutno aktivni u sistemu. Na osnovu dobijenih informacija, lako je konstruirati stablo procesa slično ovome prikazanom na slici 9 rekurzivnim praćenjem roditeljskih procesa sve do *init* procesa.

Uopšteno, kada proces kreira proces, procesu detetu će trebati određeni resursi (vreme CPU-a, memorija, datoteke, I / O uređaji) da bi izvršio svoj zadatak. Proces dete može dobiti svoje resurse direktno od strane operativnog sistema ili može u svom radu biti ograničen na podskup resursa dodeljenih roditeljskom procesu. Roditelj će, u tom drugom scenariju, morati da razdeli (particioniše) svoje resurse među svojom decom, ili će eventualno biti u stanju da deli resurse (poput memorije ili datoteka) sa njima. Ograničenje procesa deteta na podskup resursa roditelja sprečava bilo koji proces da preoptereti sistem stvarajući previše procesa.

Pored snabdevanja raznim fizičkim i logičkim resursima, roditeljski proces može proslediti inicijalizacione podatke (ulaz) ka procesu detetu. Na primer, razmotrite proces čija je funkcija prikazivanje sadržaja datoteke - recimo, image.jpg - na ekranu terminala. Kada se proces kreira, on će kao ulaz od procesa roditelja dobiti ime datoteke image.jpg. Koristeći to ime datoteke, otvoriće datoteku i prikazati sadržaj. Takođe može dobiti i ime izlaznog uređaja. Alternativno, neki operativni sistemi prenose resurse na dečije procese. U takvom sistemu, novi proces bi dobio dve otvorene datoteke, image.jpg i terminalni uređaj, te bi samo jednostavno preneo sadržaj između dva. Kada proces kreira novi proces, postoje dve mogućnosti za izvršenje:

1. Roditelj proces nastavlja da se izvršava u paraleli sa kreiranim procesom;
2. Roditelj proces čeka dok se neki ili sva deca procesi ne terminiraju.

Takođe, postoje dve mogućnosti u vezi sa adresnim prostorom kreiranog procesa:

1. Dete proces je duplikat roditeljskog procesa (koristi isti program i podatke kao i roditelj);
2. Dete proces ima novi program koji koristi i koji je učitan u njega.

Da bismo ilustrovali ove razlike, razmatramo Unix operativni sistem. U Unix-u je, kao što smo videli, svaki proces jednoznačno određen svojim identifikatorom procesa, koji je jedinstveni celi broj. Novi proces se stvara sistemskim pozivom **fork()**. Novi proces se sastoji od *kopije* adresnog prostora originalnog procesa. Ovaj mehanizam omogućava roditeljskom procesu da lako komunicira sa procesom detetom. Oba procesa (roditelj i dete) nastavljaju izvršavanje počevši sa instrukcijom koja sladi nakon *fork*(), s jednom razlikom: povratni kod sistemskog poziva fork() je nula za novi (dete) proces, dok je ne-nulta vrednost identifikatora procesa deteta vraćena roditelj-procesu.

Nakon sistemskog poziva fork(), jedan od dva procesa obično koristi sistemski poziv **exec()** radi zamene memorijskog prostora procesa novim programom, u slučaju kada dete proces treba da izvršava drugačiji program.

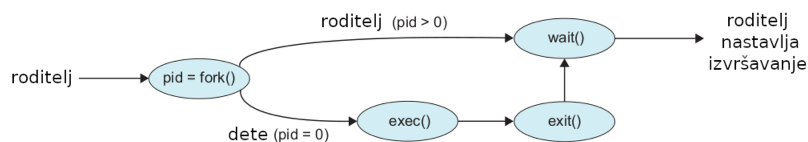
Sistemski poziv exec() učitava binarnu datoteku u memoriju (uništavajući memorijski sadržaj programa koji poziva sistemski poziv exec()) i započinje njegovo izvršavanje. Na ovaj način, dva procesa su u stanju da inicijalno komuniciraju, a zatim nastave svojim zasebnim putanjama izvršavanja. Roditelj, na ovaj način, može stvoriti više dece procesa. Nakon što to uradi, u slučaju da proces roditelj nema šta da radi dok se deca procesi izvršavaju, proces roditelj može da pozove sistemski poziv **wait()**, kako bi sam sebe uklonio iz reda spremnih procesa do terminiranja deteta procesa. Budući da poziv exec() prepisuje adresni prostor procesa novim programom, poziv exec() ne vraća kontrolu nazad, osim ako se ne dogodi greška.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h> //wait system call
```

```
int main() {
  pid_t pid;
  /* fork a child process */
  pid = fork();
  if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
  }
  else if (pid == 0) { /* child process */
    execlp("/bin/ls","ls",NULL);
        //execlp("/bin/ls","ls","-l",NULL); if arguments required
  }
  else { /* parent process */
    /* parent will wait for the child to complete */
    wait(NULL);
    printf("Child Complete\n");
  }
}
```

C program prikazan iznad ilustruje Unix sistemske pozive koje smo prethodno opisali. Sada imamo dva različita procesa koji pokreću kopije istog programa. Jedina razlika je u tome što je vrednost pid-a (identifikatora procesa) za dete-proces jednaka nuli, dok je za roditelj celobrojna vrednost veća od nule (u stvari je to stvarni pid dete-procesa). Dete proces nasleđuje privilegije i atribute raspoređivanja od roditelja, kao i određene resurse, poput otvorenih datoteka. Tada dete proces prepisuje svoj adresni prostor sa Unix naredbom /bin/ls (koja se koristi za dobijanje liste trenutnog direktorijuma) koristeći sistemski poziv **execlp()** (execlp() je verzija sistemskog poziva exec()). Roditelj čeka da se proces deteta završi sa wait() sistemskim pozivom. Kada se dete proces završi (bilo implicitno ili eksplicitnim pozivom sistemskog poziva exit()), roditelj proces nastavlja se nakon izlaska iz poziva wait(), i završava se korišćenjem sistemskog poziva exit(). Ovo ponašanje je takođe prikazano na slici 3.10.

Naravno, ništa ne sprečava dete proces da ne poziva exec() i umesto toga nastavi da se izvršava *kao kopija* roditeljskog procesa. U ovom scenariju, roditelj i dete su *konkurentni* procesi (*concurrent*) koji izvršavaju iste instrukcije. Pošto je dete kopija roditelja, svaki proces ima svoju kopiju bilo kojeg podatka koji se koristi. Važno je samo primetiti da se modifikacija neke od promenljivih od strane roditelj procesa, odnosno procesa deteta, neće videti na drugoj strani (kod deteta procesa, odnosno roditelj procesa). Razlog za to leži u činjenici da je kreiranjem novog procesa napravljena ***kopija*** podataka roditeljskog procesa, te kasnije izmene tih podataka neće biti vidljive u ostalim procesima.



Slika 10: Kreiranje procesa korišćenjem fork sistemskog poziva

4.2 Terminiranje procesa

Proces se završava kada završi izvršavanje svoje poslednje instrukcije i zatraži od operativnog sistema da ga izbriše korišćenjem *exit*() sistemskog poziva. U tom trenutku proces može vratiti statusnu vrednost (obično celi broj) roditelj procesu, koju će roditelj proces preuzeti putem sistemskog poziva wait(). Operativni sistem oslobađa (dealocira) sve resurse procesa - uključujući fizičku i virtualnu memoriju, otvorene datoteke i U/I bafere.

Terminiranje procesa se može dogoditi i u drugim okolnostima. Proces može izazvati terminiranje drugog procesa putem odgovarajućeg sistemskog poziva (na primer, *TerminateProcess*() u Windows-u, *kill* u Linux-u). Tipično, takav sistemski poziv može pozvati samo roditelj procesa koji treba terminirati. U suprotnom, korisnici bi mogli samovoljno terminirati procese jedni drugima. Ipak, treba imati na umu da roditelj mora znati identitet svoje dece ako ih želi terminirati. Ovo ne bi trebao biti problem jer, kao što smo već rekli, kada jedan proces kreira novi proces, identitet novostvorenog procesa se prenosi roditelju.

Roditelj može prekinuti izvršenje jednog od svoje dece iz više razloga:

- Dete proces je prekoračilo sa upotrebom nekih resursa koji su mu dodeljeni. (Da bi utvrdio da li se ovo dogodilo, roditelj mora imati mehanizam za inspekciju stanja svojih dece procesa);
- Zadatak dodeljen dete procesu više nije potreban da se izvršava;
- Roditelj proces se terminira, a operativni sistem ne dozvoljava dete procesu da nastavi ako je roditelj terminiran.

Neki operativni sistemi ne dozvoljavaju dete procesu da nastavi izvršavanje ako roditelj ne postoji u sistemu. U takvim sistemima, ako se proces završi, sva njegova deca procesi takođe moraju biti terminirani. Ovaj fenomen, nazvan *kaskadna terminacija*, pokreće se od strane operativnog sistema.

Da bismo ilustrovali izvršavanje i terminaciju procesa, posmatramo Linux i Unix sisteme, u kojima možemo okončati proces upotrebom sistemskog poziva exit(), obezbeđujući status izlaza kao parametar:

```
/* izlaz sa statusom 1 */
exit(1);
```

Zapravo, kod normalne terminacije, *exit*() se može pozvati bilo direktno (kao što je prikazano gore) ili indirektno (*return* u *main*()).

Roditeljski proces može sačekati na terminaciju procesa deteta upotrebom sistemskog poziva *wait*(). Sistemskom pozivu *wait*() prosleđuje se parametar koji roditelju omogućava da dobije izlazni status dete procesa. Ovaj sistemski poziv takođe vraća identifikator terminiranog dete procesa, kako bi roditelj mogao da zna koji je od dece procesa terminiran:

```
pid_t pid;
int status;

pid = wait(&status);
```

Kada se proces terminira, njegovi resursi se dealociraju i vraćaju operativnom sistemu. Međutim, odgovarajući unos u *tabeli procesa* mora ostati tamo dok roditeljski proces ne pozove wait(), jer tabela procesa sadrži izlazni status procesa.

Proces koji je okončan, a čiji roditelj još nije zvao *wait*(), poznat je kao *zombi proces*. Svi procesi prelaze u ovo stanje kada su terminirani, ali generalno postoje kao zombi procesi samo kratko. Čim roditeljski proces pozove *wait*(), oslobađa se identifikator zombi procesa kao i njemu pripadajući unos u tabeli procesa.

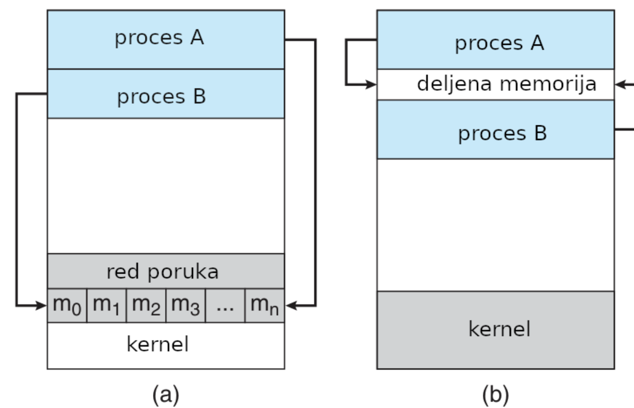
Ako roditelj ne pozove *wait*() već se umesto toga samo terminira, on tada ostavlja svoje dete proces kao *napušten proces* (*siroče, orphan*). Linux i Unix rešavaju ovakav scenario postavljanjem *init* procesa za novog roditelja ovakvih napuštenih procesa. Ovo je moguće, jer, kao što je prikazano na slici 9, *init* proces predstavlja koren u hijerarhiji procesa u sistemima Unix i Linux. Proces *init* periodično poziva wait(), omogućavajući tako da se prikupe izlazni statusi bilo kojeg napuštenog procesa, te da se oslobode identifikator napuštenog procesa i njegov unos u tabeli procesa.

5 Interprocesna komunikacija (Interprocess Communication, IPC)

Procesi koji se istovremeno izvršavaju u operativnom sistemu mogu biti ili *nezavisni* procesi ili *kooperativni* procesi (procesi koji međusobno sarađuju). Proces je nezavisan ako na njega ne mogu uticati ili on ne utiče na ostale procese koji se izvršavaju u sistemu. Svaki proces koji ne deli podatke sa bilo kojim drugim procesom je nezavisan.

Proces je *kooperativan* ako može uticati na druge procese koji se izvršavaju u sistemu ili oni mogu uticati na njega. Jasno je da je svaki proces koji deli podatke sa drugim procesima kooperativni proces. Postoji višestruka motivacija za postojanje okruženja koje omogućava kooperativne procese:

- Deljenje informacija: više korisnika je često zainteresovano za isti podatak (na primer, deljenu datoteku), pa se mora obezbediti okruženje koje će omogućiti istovremeni pristup takvim informacijama;
- Ubrzano računanje: ako želimo da se određeni zadatak brže izvrši, moramo ga razdvojiti na podzadatke, od kojih će se svaki izvršavati istovremeno sa ostalima. Naravno, takvo ubrzanje se može postići samo ako računar ima više jezgara koja mogu da se koriste za paralelnu obradu;
- Modularnost: često je efikasnije konstruisati sistem na modularan način, deleći sistemske funkcije na odvojene procese ili niti;
- Pogodnost: svaki pojedinačni korisnik može raditi na višestrukim zadacima, koji su na neki način funkcionalno povezani, u isto vreme.



Slika 11: Modeli komuniciranja: a)prenošenje poruka b)deljena memorija

Kooperativni programi zahtevaju mehanizam interprocesne (međuprocesne) komunikacije (IPC) koji će im omogućiti razmenu podataka i informacija. Postoje dva osnovna modela interprocesne komunikacije: *deljena memorija* (*shared memory*) i prenošenje poruka (*message passing*).

U modelu deljene memorije uspostavlja se region memorije koja se deli između kooperativnih procesa. Tada procesi mogu da razmenjuju informacije čitanjem i upisom podataka u taj deljeni prostor. U modelu prenošenja poruka, komunikacija se odvija putem poruka koje se razmenjuju između procesa koji sarađuju. Na slici 11 su predstavljena ova dva modela komunikacije.

Oba spomenuta modela su uobičajena u operativnim sistemima, i mnogi sistemi implementiraju i jedan i drugi. Prenošenje poruka korisno je za razmenu manjih količina podataka. Slanje poruka je takođe lakše implementirati u distribuiranim sistemima od deljene memorije. Deljena memorija može biti brža od prenošenja poruka, jer se sistemi za prenošenje poruka obično implementiraju pomoću sistemskih poziva i na taj način zahtevaju više vremena usled intervencije kernela. U sistemima sa deljenom memorijom sistemski pozivi su potrebni samo za uspostavljanje regiona deljene memorije. Jednom kada se on uspostavi, svi pristupi se tretiraju kao rutinski pristupi memoriji i nije potrebna dodatna pomoć kernela za to.

Nedavna istraživanja multi-jezgarnih sistema pokazuju da prenošenje poruka pruža bolje performanse od deljene memorije na takvim sistemima. U takvim sistemima, implementacija deljene memorije je problematična zbog koherencije keša (*cache coherence*) koji nastaje zato što se deljeni podaci prenose između višestrukih keš memorija. Kako se povećava broj jezgara na sistemima, očekivano je da prenošenje poruka postane preferirani mehanizam za IPC.

U ostatku ovog odeljka detaljnije istražujemo sisteme deljene memorije i prenošenja poruka.

5.1 Deljena memorija kao mehanizam za interpocesnu komunikaciju

Međuprocesna komunikacija pomoću deljene memorije zahteva, kao što smo rekli, uspostaljanje regiona zajedničke memorije od strane komunikacionih procesa. Tipično, oblast deljene memorije nalazi se u adresnom prostoru procesa koji kreira segment deljene memorije. Ostali procesi koji žele komunicirati korišćenjem ovog segmenta zajedničke memorije moraju ga pripojiti svom adresnom prostoru. Podsetimo da operativni sistem pokušava da spreči jedan proces da pristupi memoriji drugog procesa, te deljena memorija zahteva da se dva ili više procesa saglase, kako bi se uklonilo ovo ograničenje. Nakon što se uspostavi deljena memorija, procesi mogu razmenjivati informacije čitajući i upisujući podatke u deljenu memoriju. Struktura podataka, kao i lokacija određeni su ovim procesima i nisu pod kontrolom operativnog sistema. Procesi su takođe odgovorni za osiguravanje da ne upisuju na istu lokaciju istovremeno.

Da bismo ilustrovali koncept kooperativnih procesa, razmatramo problem proizvođač (podatka, *producer*) - potrošač (podatka - *consumer*), što je uobičajena paradigma kod kooperativnih procesa. Proces proizvođač proizvodi informacije koje konzumira proces potrošač. Na primer, kompajler može da proizvede asemblerski kod koji asembler koristi. Asembler, zauzvrat, može generisati objektne module koje koristi *loader* ili *linker*. Problem proizvođač-potrošač predstavlja metaforu za paradigmu klijent-server, gde je server uglavnom proizvođač, a klijent potrošač. Na primer, veb server proizvodi (tj. pruža) HTML datoteke i slike, koje troši (tj. čita ih) klijentski veb pretraživač koji zahteva resurse.

Jedno rešenje problema proizvođač-potrošač koristi deljenu memoriju. Da bismo omogućili da se procesi proizvođača i potrošača istovremeno izvršavaju, moramo imati dostupan bafer koji popunjava proizvođač, a troši potrošač. Ovaj bafer će se nalaziti u regionu memorije koji dele proizvođački i potrošački procesi. Proizvođač može proizvesti jedan podatak, dok potrošač koristi drugi. Proizvođač i potrošač moraju biti sinhronizovani, tako da potrošač ne pokušava da konzumira podatak koji još nije generisan.

Mogu se koristiti dve vrste bafera. Neograničeni bafer (*unbounded buffer*) ne definiše praktično ograničenje veličine bafera. Kod ovakvog bafera, potrošač će morati da sačeka nove podatke, ali proizvođač uvek može generistai nove podatke. Sa druge strane, ograničeni bafer (*bounded buffer*) predviđa fiksnu veličinu bafera. U ovom slučaju, potrošač mora sačekati ukoliko je bafer prazan, a proizvođač mora da sačeka da ukoliko je bafer pun. Pogledajmo detaljnije kako ograničeni bafer ilustruje međuprocesnu komunikaciju koristeći zajedničku memoriju. Sledeće promenljive se nalaze u regionu memorije koji dele proizvođački i potrošački proces:

```
#define BUFFER_SIZE 10
typedef struct {
        . . .
}item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Deljeni bafer je implementiran kao cirkularni niz sa dva logička pokazivača: *in* i *out*. Promenljiva in pokazuje na sledeću praznu poziciju u baferu, dok out pokazuje na prvu popunjenu lokaciju u baferu. Bafer je prazan ako je

```
in==out
```

a bafer je pun kada je

```
((in+1) % BUFFER_SIZE) == out
```

Kod za proces proizvođač je:

```
item next_produced;
while (true) {
  /* popuni podatak next_produced sa željenim informacijama */
  while (((in + 1) % BUFFER_SIZE ) == out)
    ; /* ne radi ništa, čekaj da se bafer isprazni */
  buffer[in] = next_produced;
  in = (in + 1) % BUFFER_SIZE ;
}
```

dok je kod za proces potrošač:

```
item next_consumed;
while (true) {
  while (in == out)
  ; /* bafer prazan, ne radi ništa */
  next_consumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;

  /* iskoristi podatak dobijen kroz next_consumed */
}
```

Proces proizvođač ima lokalnu varijablu next_produced u koju se smešta novi podatak koji će se proizvesti. Proces potrošač ima lokalnu promenljivu next_consumed, u koju će, nakon čitanja iz daljene memorije, biti upisan podatak koji treba da se konzumira.

Ova šema omogućava najviše BUFFER SIZE - 1 elemenata u baferu istovremeno.

Problem koji se na ovom primeru ne uočava odnosi se na situaciju u kojoj i proizvođački proces i potrošački proces pokušavaju istovremeno pristupiti deljenom baferu. U jednom od kasnijih predavanja ćemo razgovarati o tome kako se sinhronizacija između kooperativnih procesa može efikasno implementirati u okruženju sa deljenom memorijom.

5.2 Prenošenje poruka kao mehanizam za interpocesnu komunikaciju

U prethodnom odeljku, pokazali smo kako kooperativni procesi mogu komunicirati u okruženju sa deljenom memorijom. Ovakav pristup nalaže da ovi procesi dele region memorije i da kod za pristup i manipulisanje zajedničkom memorijom bude eksplicitno napisan od strane programera. Drugi način da se postigne isti efekat je da operativni sistem obezbedi mehanizam kooperativnim procesima da komuniciraju između sebe putem infrastrukture za prenos poruka. Prenošenje poruka pruža mehanizam koji omogućava procesima da komuniciraju i da sinhronizuju svoje akcije bez deljenja istog adresnog prostora. Naročito je

koristan u distribuiranom okruženju gde komunicirajući procesi komuniciranja mogu biti locirani na različitim računarima povezanim mrežom. Infrastuktura za prenošenje poruka obezbeđuje najmanje dve operacije:

- pošalji - *send* (poruku)
- primi - *receive* (poruku)

Poruke koje proces šalje šalje mogu biti fiksne ili promenljive u veličini. Ako se mogu slati samo poruke fiksne veličine, implementacija na nivou sistema je jednostavna. Ovo ograničenje, međutim, otežava zadatak programerima. Suprotno tome, poruke promenljive veličine zahtevaju složeniju implementaciju na nivou sistema, ali zadatak programerima postaje jednostavniji. Ovo je uobičajena vrsta kompromisa koja se može videti svuda u dizajnu operativnog sistema.

Ako procesi P i K žele da komuniciraju, oni moraju slati poruke i primati poruke jedni od drugih: komunikaciona veza mora postojati između njih.

Ova veza se može realizovati na više načina. Ovde se nećemo zanimati fizičkom implementacijom veze (deljena memorija, hardverska magistrala ili mreža), već ćemo se fokusirati na njene logičke implementacije.

Evo nekoliko metoda za logičku implementaciju veze i operacije send()/receive():

- Direktna ili indirektna komunikacija
- Sinhrona ili asinhrona komunikacija
- Automatsko ili eksplicitno baferovanje

U nastavku analiziramo svaku od ovih implementacija.

5.2.1 Direktna/indirektna komunikacija

Procesi koji žele komunicirati moraju imati način da se obraćaju jedni drugima i oni mogu koristiti bilo direktnu ili indirektnu komunikaciju, u tu svrhu. Kod direktne komunikacije, svaki proces koji želi komunicirati mora izričito imenovati bilo primaoca, bilo pošiljaoca sa kojim planira da komunicira. U ovom modelu komunikacije, primitive za slanje i prijem su definisane kao:

- send(P, poruka) - Pošaljite poruku za proces P;
- receive(K, poruka) - primite poruku od procesa K.

Komunikaciona veza u ovoj šemi ima sledeća svojstva:

- Veza se uspostavlja automatski između svakog para procesa koji žele komunicirati;
- Procesi moraju da znaju samo identitet drugog da bi mogli komunicirati;
- Veza povezuje tačno dva procesa;
- Između svakog para procesa postoji tačno jedna veza.

Direktna komunikacija zahteva simetriju u adresiranju, odnosno, i proces pošiljaoc i proces prijemnik moraju imenovati onog drugog da bi mogli komunicirati. Varijanta ovog modela koristi asimetriju u adresiranju. Ovde samo pošiljalac imenuje primaoca, dok primalac nije dužan da imenuje pošiljaoca. Primitive za slanje i prijem su tada definisane na sledeći način:

- send(P, poruka) - Pošalji poruku procesu P;
- receive (id, poruka) - primite poruku od bilo kojeg procesa. Promenjiva *id* će biti postavljena na ime procesa od koga je poruka primljena.

Mana u oba data modela (simetričnom i asimetričnom) odnosi se identifikatore procesa. Promena identifikatora procesa obavezno zahteva ispitivanje referenci kod svih ostalih procesa. Moraju se naći sve reference na stari identifikator, kako bi se one mogle modifikovati u novi identifikator. Generalno, sve *hard-coded* tehnike, gde se identifikatori moraju izričito navesti, manje su poželjni od tehnika koje uključuju indirektnost, koju ćemo opisati u nastavku.

Pomoću indirektne komunikacije, poruke se šalju i primaju iz poštanskih sandučića (*mailbox*). Poštansko sanduče može abstraktno da se posmatra kao objekat u koji procesi mogu upisivati poruke i iz kojih se poruke mogu uklanjati. Svako poštansko sanduče ima jedinstvenu identifikaciju na nivou sistema. Na primer, *redovi poruka* (message queues) u Linux operativnom sistemu koriste celobrojnu vrednost za identifikaciju poštanskog sandučeta. Proces može da komunicira sa drugim procesom preko više različitih poštanskih sandučića, ali dva procesa mogu komunicirati samo ako imaju deljeno poštansko sanduče. Primitive za slanje i prijem, ovde su definisane na sledeći način:

- send(A, poruka) - Pošaljite poruku u poštansko sanduče A;
- receive(A, poruka) - primite poruku iz poštanskog sandučeta A.

U ovoj šemi, komunikaciona veza ima sledeća svojstva:

- Uspostavlja se veza između para procesa samo ako oba člana para imaju zajedničko poštansko sanduče;
- Veza može povezivati više od dva procesa;
- Između svakog para procesa koji komuniciraju može postojati više različitih veza, pri čemu svaka veza odgovara jednom poštanskom sandučiću.

Sada pretpostavimo da procesi P1, P2 i P3 svi dele poštansko sanduče A. Proces P1 šalje poruku u A, dok P2 i P3 primaju poruku iz A. Koji će proces primiti poruku koju je poslao P1? Odgovor zavisi od toga koju od sledećih metoda izaberemo:

- Dozvoliti da najviše dva procesa koriste jednu vezu;
- Dopustiti najviše jednom procesu da, u datom trenutku, izvrši operacija receive();

- Dopustiti sistemu da proizvoljno odabere koji će proces primiti poruku (tj. P2 ili P3 će dobiti poruku, ali ne i jedan i drugi). Sistem tada može definisati algoritam za izbor procesa koji će primiti poruku (na primer, *round-robin*, gde procesi naizmenično primaju poruke). Takođe, sistem može da identifikuje primaoca pošiljaocu.

Poštansko sanduče može biti u vlasništvu bilo procesa bilo operativnog sistema. Ako je poštanski sandučić u vlasništvu procesa (to jest, poštanski sandučić je deo adresnog prostora procesa), tada razlikujemo vlasnika (koji može samo primati poruke putem ovog poštanskog sandučeta) i korisnika(koji može samo slati poruke do poštanskog sandučeta). Pošto svaki poštanski sandučić ima jedinstvenog vlasnika, ne može doći do nesporazuma u vezi sa procesom koji treba da primi poruku poslatu u ovaj poštanski sandučić. Kada se proces koji je vlasnik poštanskog sandučeta terminira, poštansko sanduče nije više dostupno i svaki proces koji naknadno pošalje poruku ovom poštanskom sandučetu mora biti obavešten o tome.

Suprotno tome, poštansko sanduče koje je u vlasništvu operativnog sistema je nezavisno i nije vezano ni za jedan određeni proces. Operativni sistem tada mora da obezbedi mehanizam koji procesu omogućava sledeće:

- Kreiranje novog poštanskog sandučeta;
- Slanje i primanje poruka preko poštanskog sandučeta;
- Brisanje poštanskog sandučeta.

Proces koji kreira novo poštansko sanduče je podrazumevano vlasnik tog poštanskog sandučeta. Inicijalno je vlasnik jedini proces koji može primati poruke putem ovog poštanskog sandučeta. Međutim, privilegija vlasništva i primanja poruka može se preneti na druge procese putem odgovarajućih sistemskih poziva. Naravno, ovakav prenos bi mogao rezultovati sa više prijemnika za svako poštansko sanduče.

5.2.2 Sinhronizacija

Komunikacija između procesa odvija se korišćenjem primitiva send() i receive(). Postoje različite mogućnosti dizajniranja i implementacije svake od primitiva.

Slanje poruka može biti ili *blokirajuće* ili *neblokirajuće* - takođe poznato kao *sinhrono* i *asinhrono*:

- Blokirajuće slanje: Proces pošiljalac je blokiran dok poruka ne bude primljena od strane prijemnom procesa prijema ili poštanskog sandučeta;
- Neblokirajuće slanje: Proces pošiljalac šalje poruku i nastavlja sa radom;
- Blokirajući prijem: Proces primalac blokira dok nije dostupna poruka;
- Neblokirajući prijem: Proces primalac preuzima pristiglu poruku ili dobija informaciju da ona nije pristigla.

Moguće su različite kombinacije slanja i prijema. Kad su i slanje i prijem blokirajući, imamo scenario koji je poznat kao *sastanak* (*randevouz*) između pošiljaoca i primaoca. Rešenje problema proizvođač-potrošač od malopre postaje trivijalno kada koristimo blokirajuće slanje i primanje. Proizvođač samo poziva blokirajuće slanje i čeka dok se poruka ne isporuči primaocu ili poštanskom sandučetu:

```
message next_produced;
while (true) {
        /* produce an item in next_produced */
        send(next_produced);
}
```

Isto tako, potrošač poziva prijem i blokira se dok nije dostupna poruka:

```
message next_consumed;
while (true) {
        receive(next_consumed);
        /* consume the item in next consumed */
}
```

5.2.3 Baferovanje

Bez obzira da li je komunikacija direktna ili indirektna, poruke koje se razmenjuju između komunicirajućih procesa nalaze se u *redu čekanja*. U osnovi, takvi redovi se mogu implementirati na tri načina:

1. Nulti kapacitet: Red čekanja ima maksimalnu dužinu nula - na taj način, veza ne može da čuva nijednu poruku u okviru sebe. U ovom slučaju, pošiljalac se mora blokirati dok primalac ne primi poruku;

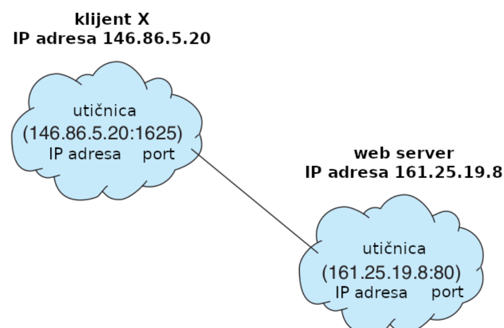
2. Ograničeni kapacitet: Red ima konačnu dužinu n; na taj način, najviše n poruka može da boravi u njemu. Ako red poruka nije ispunjen prilikom slanja nove poruke, poruka se postavlja u red (ili je poruka kopirana ili se kopira pokazivač na poruku), a pošiljalac može nastaviti sa svojim izvršavanjem bez čekanja. Međutim, ako je red poruka ispunjen, pošiljalac se mora blokirati dok u redu čekanja nije dostupan prostor;

3. Neograničeni kapacitet: Dužina reda poruka je potencijalno beskonačna - na taj način, bilo koji broj poruka može da čeka u njemu. Pošiljalac se nikad ne blokira.

Implementacija sa redom poruka nultog kapaciteta se ponekad naziva sistem poruka bez baferovanja. Ostale implementacije se nazivaju sistem sa automatskim baferovanjem.

5.3 Komunikacija u klijent-server sistemima

U prethodnom poglavlju videli smo kako procesi mogu da komuniciraju korišćenjem deljene memorije i prosleđivanja poruka. Ove tehnike mogu biti korišćene i u cilju komunikacije u server/klijent sistemima, takođe. U ovom odeljku ćemo pomenuti i druge strategije koje se koriste u cilju klijent/server komunikacije: utičnice (*sockets*), poziv udaljenih procedura (*Remote Procedure Call* - RPC) i cevi (*pipes*).



Slika 12: Povezivanje procesa putem utičnica

5.3.1 Utičnice

Utičnica je definisana kao krajnja tačka za komunikaciju. Par procesa koji komuniciraju preko mreže koristi par utičnica - po jedan za svaki proces. Utičnica se identifikuje IP adresom povezanom sa brojem porta. Uopšteno, utičnice koriste arhitekturu klijent-server. Server čeka dolazne zahteve klijenta osluškivanjem na određenom portu. Nakon što primi zahtev, server prihvata vezu od klijentske utičnice da bi dovršio povezivanje. Serveri koji implementiraju određene usluge (kao što su telnet, FTP i HTTP) slušaju unapred definisane i poznate portove (telnet server sluša port 23; FTP server sluša port 21; veb ili HTTP, server sluša port 80). Svi portovi ispod 1024 smatraju se poznatim i standardnim - možemo ih koristiti za implementaciju standardnih usluga.

Kada klijentski proces pokrene zahtev za vezu, njegov računar (*host*) dodeljuje mu port. Ovaj port ima neki proizvoljni broj veći od 1024. Na primer, ako klijent na mašini X sa IP adresom 146.86.5.20 želi da uspostavi vezu sa veb serverom (koji sluša na porta 80) na adresi 161.25.19.8, X-u se može se dodeliti port 1625. Veza će se sastojati od para utičnica: (146.86.5.20:1625) na hostu X i (161.25.19.8:80) na veb serveru. Ova situacija je prikazana na slici 12.

Paketi koji putuju između računara isporučuju se odgovarajućem procesu na osnovu broja odredišnog porta. To znači da više procesa može da egzistira na istoj mašini, da svaki koristi svoju utičnicu kako bi komunicirao sa udaljenom mašinom.

Sve veze moraju biti jedinstvene. Stoga, ako je drugi proces takođe na mašini X hteo uspostaviti drugu vezu s istim veb serverom, to je moguće tako što mu se dodeljuje broj porta veći od 1024 koji nije jednak 1625. To osigurava da se sve veze sastoje od jedinstvenog para utičnica.

Iako većina primera programskog koda koristi C programski jezik, ilustrovaćemo primer sa utičnicama na Java programskom jeziku, jer Java pruža mnogo lakši interfejs za utičnice i ima bogatu biblioteku za rad sa mrežama.

Java nudi tri različite vrste utičnica. Utičnice orijentisane konekcijom (*connection oriented*) (TCP) implementirane su klasom *Socket*. Utičnice bez konek-

cije (*Connectionless sockets*) (UDP) koriste klasu *DatagramSocket*. Konačno, klasa MulticastSocket je potklasa klase DatagramSocket i ona omogućava slanje podataka ka više primalaca.

Naš primer opisuje server datuma koji koristi TCP utičnice orijentisane konekcijom. Operacija omogućava klijentima da zatraže trenutni datum i vreme od servera. Server sluša na portu 6013, iako bi sam port mogao imati bilo koji proizvoljni broj veći od 1024. Kada se veza uspostavi (klijent se javio), server vraća datum i vreme klijentu. Kod servera datuma izgleda ovako:

```
import java.net.*;
import java.io.*;
public class DateServer
{
  public static void main(String[] args)
  {
    try
      {
        ServerSocket sock = new ServerSocket(6013);
        /* now listen for connections */
        while (true)
        {
          Socket client = sock.accept();
          PrintWriter pout = new PrintWriter(client.getOutputStream(), true);
          /* write the Date to the socket */
          pout.println(new java.util.Date().toString());
          /* close the socket and resume */
          /* listening for connections */
          client.close();
        }
    }
    catch (IOException ioe)
    {
      System.err.println(ioe);
    }
  }
}
```

Server kreira ServerSocket objekat koji definiše da će slušati na portu 6013. Server zatim počinje osluškivanje porta metodom *accept*(). Server se blokira pozivajući metodu *accept*() čekajući da klijent zatraži vezu sa serverom. Kada se primi zahtev za povezivanje, *accept*() vraća utičnicu koju server može da koristi za komunikaciju sa tim klijentom. Detalji kako server komunicira koristeći utičnicu su sledeći: Server prvo kreira PrintWriter objekat koji će koristiti za komunikaciju sa klijentom. Objekat PrintWriter omogućava serveru da piše u socket koristeći *print*() i *println*() metode za ispis (isto kao što bi se upisivalo u datoteku). Server proces zatim šalje datum klijentu, pozivajući metodu *println*(). Kada upiše datum koristeći utičnicu, server zatvara utičnicu za klijenta i nastavlja sa osluškivanjem u cilju reagovanja na sledeće zahteve od istog ili drugih klijenata. Sa druge strane, klijent komunicira sa serverom kreirajući utičnicu i povezujući se sa portom na kome server sluša. Implementacija klijenta u programskom jeziku Java je data ispod:

```
import java.net.*;
import java.io.*;
public class DateClient
{
  public static void main(String[] args)
  {
    try
```

```
    {
      /* make connection to server socket */
      Socket sock = new Socket("127.0.0.1",6013);
      InputStream in = sock.getInputStream();
      BufferedReader bin = new BufferedReader(new InputStreamReader(in));
      /* read the date from the socket */
      String line;
      while ( (line = bin.readLine()) != null)
        System.out.println(line);
      /* close the socket connection*/
      sock.close();
    }
    catch (IOException ioe)
    {
      System.err.println(ioe);
    }
  }
}
```

Klijent kreira *Socket* objekat i zahteva vezu sa serverom na IP adresi 127.0.0.1 na portu 6013. Kada je veza uspostavljena, klijent može da čita iz utičnice koristeći normalne I/O tokove u Javi (isto kao što bi čitao podatke iz tekstualne datoteke). Nakon što primi datum sa servera, klijent zatvara utičnicu i terminira se. IP adresa 127.0.0.1 je posebna IP adresa poznata kao povratna adresa (*loopback*). Kada se računar referencira na IP adresu 127.0.0.1, referencira se zapravo na sebe. Ovaj mehanizam omogućava klijentu i serveru da budu locirani na istoj mašini, a da ipak komuniciraju koristeći TCP/IP protokol. IP adresa 127.0.0.1 može se zameniti IP adresom druge mašine na kojoj se eventualno pokrene serverski proces. Pored IP adrese, može se koristiti i stvarno ime *host* mašine, kao što je www.elektronika.ftn.uns.ac.rs.

Komunikacija pomoću utičnica - iako uobičajena i efikasna - smatra se oblikom komunikacije između distribuiranih procesa na niskom nivou. Jedan od razloga je taj što utičnice omogućavaju razmenu samo nestruktuiranog toka bajtova između komunicirajućih procesa. Odgovornost klijentske ili serverske aplikacije je da nametne strukturu podataka koji će se slati.

U naredna dva odeljka posmatraćemo dva načina komunikacije distribuiranih procesa višeg nivoa: *poziv udaljinih procedura* (RPC) i *cevi* (*pipes*).

5.3.2 Poziv udaljenih procedura (*Remote Procedure Call* - RPC)

Jedan od najčešćih oblika izvršavanja udaljenih servisa je paradigma RPC. RPC je zamišljen kao način da se abstrakuje mehanizam za pozivanje procedura koji bi se koristio između sistema sa mrežnim konekcijama. U mnogim je aspektima sličan IPC mehanizmu opisanom ranije, a obično je IPC polazna tačka za izgradnju RPC. Kod RPC, međutim, obzirom na to da se bavimo okruženjem u kojem se procesi izvršavaju na zasebnim sistemima, moramo koristiti komunikaciju zasnovanu na porukama u cilju pružanje daljinske usluge. Za razliku od IPC poruka, poruke koje se razmenjuju u RPC komunikaciji su dobro struktuirane i samim tim poruke više nisu samo paketi podataka. Svaka poruka je upućena RPC demonu (deamon - proces koji se izvršava u pozadini, kao servis) koji na udaljenom sistemu osluškuje određeni port, a svaka poruka sadrži identifikator koji određuje funkciju koju treba izvršiti i parametre koji se prosleđuju toj funk-

ciji. Funkcija se zatim izvršava prema zahtevu, a dobijeni izlaz se vraća procesu koji je zahtevao RPC, u zasebnoj poruci.

Port je jednostavno broj koji je dodat na početku paketa poruke. Dok sistem obično ima jednu mrežnu adresu, on može imati mnogo portova vezanih za tu adresu da bi razlikovao brojne mrežne usluge koje podržava (slično, zapravo isto kao kod utičnica). Ako je udaljenom procesu potrebna usluga, on adresira odgovarajući port. Na primer, ako sistem želi omogućiti drugim (udaljenim) sistemima da mogu izlistati njegove trenutne korisnike, on bi kreirao RPC demon koji osluškuje na portu, recimo, 3027. Bilo koji udaljeni sistem mogao bi tada dobiti potrebne informacije (tj. listu trenutnih korisnika na udaljenom sistemu) slanjem RPC poruke na port 3027 na serveru. Podaci će biti primljeni u odgovoru, za svaki primljeni zahtev odvojeno.

Semantika RPC-a omogućava klijentu da poziva RPC, kao što bi pozvao proceduru lokalno. RPC sistem skriva detalje koji omogućavaju komunikaciju tako što obezbeđuje *posrednika (*eng. *stub*) na strani klijenta. I RPC klijent i RPC server imaju svog *posrednika*: klijent proces misli da poziva server dok zapravo poziva klijent posrednika, server proces misli da je pozvan od strane klijenta, a zapravo je pozvan od strane server posrednikaa, dok dva posrednika zapravo šalju poruke jedan drugome kako bi omogućili RPC.

Obično postoji zaseban *posrednik* za svaku udaljenu proceduru. Kada klijent poziva udaljenu proceduru, RPC sistem poziva odgovarajućeg *posrednika* na klijentskoj strani i prosleđuje mu parametre prosleđene prilikom poziva udaljene procedure. Ovaj *posrednik* locira port na serveru i *organizuje* (eng. marshal) parametre. Organizovanje parametara uključuje pakovanje parametara u oblik koji se može preneti preko mreže. Klijentski posrednik zatim šalje poruku serveru koristeći sistem za slanje poruka. Sličan *posrednik* na strani servera prima ovu poruku i poziva proceduru na serveru. Ako je potrebno, povratne vrednosti se vraćaju klijentu na isti način.

Detalj na koji se mora obratiti pažnja odnosi se na razlike u reprezentaciji podataka na klijentu i serveru. Kao primer, razmotrimo reprezentaciju 32-bitnih celih brojeva. Neki sistemi (poznati kao *big-endian*) prvo u memoriju skladište najznačajniji bajt, dok drugi sistemi (poznati kao *little-endian*) prvo skladište bajt najmanje značajnosti. Nijedan redosled sam po sebi nije „bolji“, već je jednostavno stvar izbora unutar određene računarske arhitekture. Da bi rešili razlike poput ove, mnogi RPC sistemi definišu mašinski-nezavisno (*machine-independent*) predstavljanje podataka. Jedna takva reprezentacija poznata je kao reprezentacija eksternih podataka (XDR - eXternal Data Representation). Na strani klijenta, organizacija parametara uključuje pretvaranje (mašinski-zavisnih) podataka u XDR pre nego što se pošalju serveru. Na strani servera, XDR podaci se inverzno organizuju i pretvaraju u reprezentaciju koja odgovara serverskoj mašini.

Drugo važno pitanje na koje se mora skrenuti pažnja uključuje semantiku poziva. Dok pozivi lokalnih procedura ne uspevaju samo u ekstremnim okolnostima, RPC može da ne uspe, ali i da se duplicira i izvrši više od jednom, kao rezultat uobičajenih grešaka u umreženim sistemima. Jedan od načina da se reši ovaj problem je da operativni sistem osigura da se pozivi izvrše „tačno jednom“

(*exactly once*), umesto „najviše jednom“ (*at most once*). Većina poziva lokalnih procedura ima "tačno jednom" funkcionalnost, ali je nju teže implementirati kod RPC.

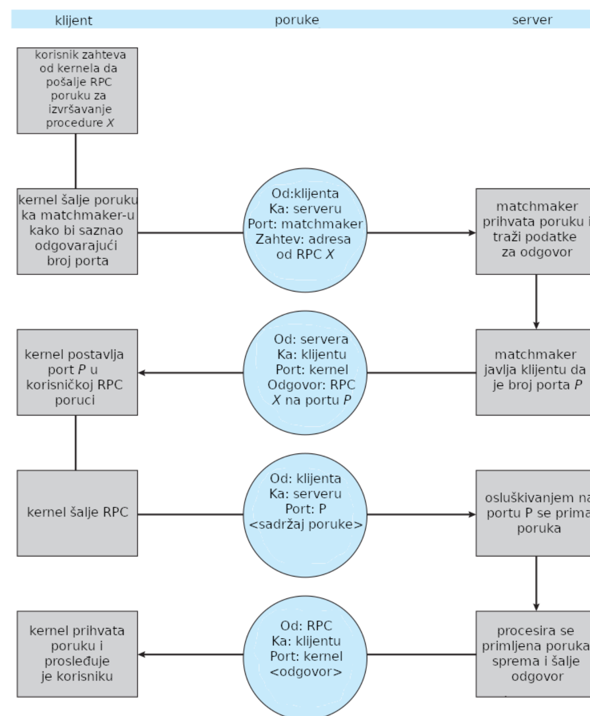
Prvo, razmatramo "najviše jednom". Ova semantika se može implementirati dodavanjem vremenske oznake svakoj poruci - tako što se na samu poruku doda vreme slanja poruke. Server tada mora čuvati istoriju svih vremenskih oznaka poruka koje je obradio ili mora imati istoriju dovoljno veliku da osigura da se detektuju ponovljene poruke. Dolazne poruke koje imaju vremensku oznaku koja se već nalazi u istoriji se ignorišu. Klijent tada može poslati poruku jednom ili više puta i biti uveren da će ona biti izvršena samo jednom.

Za „tačno jednom“ moramo ukloniti rizik da server nikada neće primiti zahtev. Da bi to postigao, server mora implementirati gore opisani protokol „najviše jednom“, ali takođe mora potvrditi klijentu da je RPC poziv primljen i izvršen. Ove tzv. ACK poruke (od engleske reči *Acknowledge*) su česte u svim mrežama. Klijent mora periodično ponovo slati svaki RPC poziv, sve dok za taj poziv ne primi ACK.

Još jedno važno pitanje odnosi se na komunikaciju između servera i klijenta. Kod standardnog poziva procedure, ime procedure se prilikom poziva zamenjuje memorijskom adresom na kojoj počinje procedura. RPC sistem zahteva slično vezivanje klijenta i servera, ali se postavlja pitanje kako klijent da zna broj porta na serveru? Nijedan sistem nema kompletne informacije o drugom, jer oni ne dele zajedničku memoriju.

Dva pristupa su uobičajena. Prvo, ove informacije mogu biti unapred određene kroz fiksnu adresu porta. U trenutku kompajliranja, RPC poziv dobija fiksni broj porta. Jednom kada se program kompajlira, server ne može promeniti broj porta za traženu uslugu. Drugo, povezivanje se može obaviti dinamički pomoću takozvanog *randevouz* mehanizma. Tipično, operativni sistem obezbeđuje randevouz demon (naziva se i *matchmaker*) na fiksnom RPC portu. Klijent zatim šalje poruku koja sadrži ime RPC-a ka randevouz demonu, zahtevajući broj porta za RPC-a koji želi da izvrši. Broj porta se vraća, nakon čega se RPC pozivi mogu slati na taj port dok se proces ne terminira (ili se server sruši). Ova metoda zahteva dodatnu komunikaciju usled prvobitnog zahteva, ali je fleksibilnija od prvog pristupa. Na slici 13 prikazan je primer razmene poruka tokom obrade RPC zahteva.

RPC mehanizam je koristan u implementaciji distribuiranog sistema datoteka (*distributed file system* - DFS). Takav sistem se može implementirati kao skup RPC demona i klijenata. Poruke su upućene ka portu distribuiranog sistema datoteka na serveru na kojem će se operacije nad datotekama izvršavati. Poruka sadrži operaciju koju treba izvesti na datoteci. Operacije mogu biti čitanje, upis, preimenovanje, brisanje ili status, što odgovara uobičajenim sistemskim pozivima koji se odnose na datoteke. Povratna poruka sadrži sve podatke koji su rezultat tog poziva, izvršenog od strane DFS demona. Na primer, poruka može sadržati zahtev za prenos cele datoteke klijentu ili jednostavno biti ograničena na prenos jednog bloka. Ako je ovo drugo slučaj, biće potrebno nekoliko zahteva za prenos cele datoteke.



Slika 13: Primer razmene poruka prilikom obrade RPC zahteva

5.3.3 Cevi (Pipes)

Cev je komunikacioni kanal koji omogućava komunikaciju između dva procesa. Cevi su bile jedan od prvih IPC mehanizama u ranim Unix sistemima. Pružaju jedan od jednostavnijih načina za komunikaciju, iako imaju i određena ograničenja. U implementaciji cevi moraju se uzeti u obzir četiri pitanja:

1. Da li cev omogućava dvosmernu komunikaciju ili je komunikacija jednosmerna?
2. Ako je dvosmerna komunikacija dozvoljena, da li je *half-duplex* (podaci se mogu slati samo u jednom smeru u datom trenutku) ili full-duplex(podaci mogu da se šalju u oba smera istovremeno)?
3. Mora li postojati odnos (poput roditelja i deteta) između komunikacionih procesa?
4. Mogu li cevi komunicirati preko mreže ili moraju komunikacioni procesi da se nalaze na istoj mašini?

U nastavku istražujemo dve uobičajene vrste cevi koje se koriste kako na Unix, tako i na Windows sistemima: neimenovane (obične) cevi i imenovane cevi.

Neimenovane (obične - *ordinary*) cevi

Obične cevi omogućavaju komunikaciju dva procesa na standardni proizvođač - potrošač način: proizvođač podatka upisuje na jedan kraj cevi (kraj za „upis"), a potrošač podatka čita s drugog kraja (kraj za „čitanje"). Kao rezultat toga, obične cevi su jednosmerne, omogućavajući samo jednosmernu komunikaciju. Ako je potrebna dvosmerna komunikacija, moraju se koristiti dve cevi, pri čemu svaka cev šalje podatke u različitom smeru. Ilustrovaćemo konstrukciju obične cevi na Unix sistemu. U datom primer jedan proces piše poruku „Pozdrav" u cev, dok drugi proces čita tu poruku iz cevi.

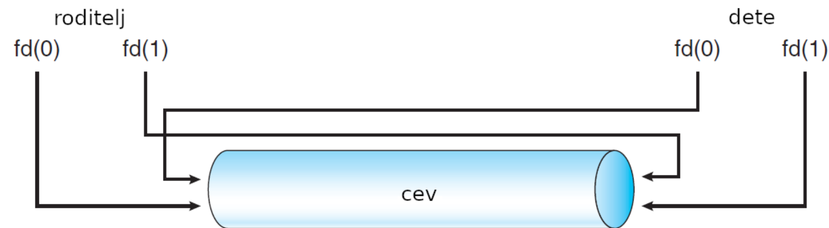
Na Unix sistemima obične cevi se kreiraju pomoću funkcije

```
pipe(int fd[])
```

Ova funkcija stvara cev kojoj se pristupa kroz int fd[] deskriptore datoteke: fd[0] je kraj za čitanje iz cevi, a fd[1] je kraj za upis.

Unix cev tretira kao posebnu vrstu datoteke. Stoga se cevima može pristupiti pomoću uobičajenih sistemskih poziva *read*() i *write*(). Običnoj cevi ne može se pristupiti izvan procesa koji ju je stvorio. Obično roditeljski proces kreira cev i koristi je za komunikaciju sa procesom detetom kojeg stvara koristeći sistemski poziv *fork*(). Pominjali smo ranije da proces-dete nasleđuje otvorene datoteke od svog roditelja. Pošto je cev posebna vrsta datoteke, dete nasleđuje cev od svog roditeljskog procesa. Na slici 14 prikazan je odnos deskriptora datoteka fd prema roditeljskom procesu i dete procesu.

U Unix programu prikazanom ispod, roditeljski proces stvara cev i zatim poziva *fork*() kreirajući proces dete. Šta će se dogoditi nakon poziva *fork*() zavisi od načina prenosa podataka kroz cev. U ovom slučaju roditelj upisuje u



Slika 14: Desktiptor datoteke cevi i roditeljski/dete proces

cev, a dete čita iz nje. Važno je primetiti da i roditeljski i dete proces inicijalno zatvaraju kraj cevi koji neće koristiti. Iako program prikazan u primeru ne zahteva ovu akciju, važan je korak kako bi se osiguralo da proces koji čita iz cevi može detektovati kraj datoteke (*read*() vraća 0) kada je proces koji upisuje u cev zatvorio svoj kraj cevi.

```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
  char write_msg[BUFFER_SIZE] = "Pozdrav";
  char read_msg[BUFFER_SIZE];
  int fd[2];
  pid_t pid;
  /* create the pipe */
  if (pipe(fd) == -1)
  {
    fprintf(stderr,"Pipe failed");
    return 1;
  }
  /* fork a child process */
  pid = fork();
  if (pid < 0)
  {
    /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
  }
  if (pid > 0)
  {
    /* parent process */
    /* close the unused end of the pipe */
    close(fd[READ_END]);

    /* write to the pipe */
    write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

    /* close the write end of the pipe */
    close(fd[WRITE_END]);
  }
  else
```

```
  {
    /* child process */
    /* close the unused end of the pipe */
    close(fd[WRITE_END]);

    /* read from the pipe */
    read(fd[READ_END], read_msg, BUFFER_SIZE);
    printf("read %s",read_msg);

    /* close the write end of the pipe */
    close(fd[READ_END]);
  }
  return 0;
}
```

Treba imati na umu da obične cevi zahtevaju odnos roditelj-dete za komunikacione procese kako na Unix, tako i na Windows sistemima. To znači da se ove cevi mogu koristiti samo za komunikaciju između procesa na istoj fizičkoj mašini.

Imenovane cevi

Obične (neimenovane) cevi pružaju jednostavan mehanizam koji omogućava komunikaciju između para procesa. Međutim, obične cevi postoje samo dok procesi komuniciraju jedni sa drugima. Na sistemima Unix i Windows, kada procesi završe sa komunikacijom i terminiraju se, obična cev prestaje da postoji. Imenovane cevi pružaju mnogo moćniji mehanizam komunikacije. Komunikacija može biti dvosmerna i nije potreban odnos roditelj - dete, kao kod neimenovenih cevi. Nakon što se kreira imenovana cev, više procesa može je koristiti za komunikaciju. Zapravo, u tipičnom scenariju, imenovana cev ima nekoliko procesa koji upisuju u nju. Pored toga, imenovane cevi postoje i nakon terminiranja procesa koji komuniciraju. Unix i Windows sistemi podržavaju imenovane cevi, mada se detalji implementacije znatno razlikuju.

Umenovane cevi se u Unix sistemima nazivaju i FIFO. Jednom kada su kreirane, pojavljuju se kao tipične datoteke u fajl sistemu. FIFO se stvara sistemskim pozivom

```
mkfifo()
```

a njime se manipuliše uobičajenim sistemskim pozivima *open*(), *read*(), *write*() i *close*(). Postojaće u sistemu sve dok eksplicitno ne bude izbrisan iz fajl-sistema. Iako FIFO koncept omogućava dvosmernu komunikaciju, dozvoljen je samo polu-dupleks prenos. Ako podaci moraju biti prenošeni u oba smera, obično se koriste dva FIFO-a. Uz to, procesi koji komuniciraju moraju da se nalaze na istoj mašini. Ako je potrebna komunikacija preko mreže, moraju se koristiti *utičnice*.

Imenovane cevi na Windows sistemima predstavljaju znatno bogatiji komunikacioni mehanizam u poređenju sa Unix-om. Kod Windows-a je dozvoljena je dupleks (*full-duplex*) komunikacija, a procesi komunikacije mogu se nalaziti na istim ili različitim mašinama. Pored toga, putem Unix FIFO-a mogu se prenositi samo podaci kao nestruktuirani i bajt-orijentisani, dok Windows sistemi omogućavaju prenos podataka koji su ili bajt-orijentisani ili sadrže struktuirane poruke. Imenovane cevi kreiraju se funkcijom *CreateNamedPipe*(), a klijent

se može povezati s imenovanom cevi pomoću *ConnectNamedPipe*(). Komunikacija putem imenovane cevi može se ostvariti pomoću *ReadFile*() i *WriteFile*() funkcija.