Univerza *v Ljubljani*
Fakulteta *za elektrotehniko*

VGRADNI SISTEMI

UVOD V SISTEMSKO PROGRAMIRANJE NA PRIMERU LINUX

STANISLAV KOVAČIČ

JANEZ PERŠ IN ROK MANDELJC

Ljubljana, 16. maj 2017

Kazalo

Poglavje 1

Uvod v vgradne sisteme

To je knjiga o delovanju in uporabi operacijskih sistemov za vgradne sisteme. Kot primer operacijskega sistema smo izbrali Linux.

V prvem poglavju bomo nekaj malega povedali o vgradnih sistemih. Vprašali se bomo kaj so vgradni sistemi, od kje izhajajo in kako so zgrajeni. Navedli bomo znanja, ki so koristna pri načrtovanju in razvoju vgradnih sistemov.

Poglavja, ki bodo sledila, se bodo bolj in bolj osredotočala na funkcionalnosti, ki jih potrebujejo vgradni sistemi in rešitve, ki jim jih dajejo operacijski sistemi. Najprej se bomo seznanili z zgradbo in delovanjem operacijskih sistemov. Zatem bo večina snovi podrobno obravnavala vmesnik med uporabniškimi programi in operacijskim sistemom. Vse skozi pa nas bo spremljal Linux.

Namen tega poglavje je da obravnavano snov, to je poglobljeno obravnavo vmesnika med operacijskim sistemom in aplikacijami, umesti v področje vgradnih sistemov.

1.1 Kaj so vgradni sistemi

Vgradni sistemi so računalniki, ki so vgrajeni v naprave, ki niso računalniki. Denimo, dandanes je računalnik sestavni del pralnega stroja. Seveda nam

ne pride na misel, da bi pralnemu stroju zato rekli računalnik. Iz povedanega zaključimo:

> *Vgradni sistem (Angl. Embedded System) je sistem, ki deluje kot računalnik, a je vgrajen v sistem, ki navzven ni videti in se ne uporablja kot računalnik.*

Ali sistemu rečemo vgradni ali vgrajeni, ni najbolj pomembno. Lahko bi rekli, da je vgradni tedaj, ko je pripravljen za vgradnjo, in vgrajeni potem, ko je že vgrajen.

1.2 Kje jih najdemo

Vgrajeni računalniki so skoraj povsod. V isti napravi jih je lahko tudi več. V osebnem avtomobilu jih je zares veliko. Nadzirajo delovanje motorja, zavornega sistema, skrbijo za udobje, za varnost, za centralno zaklepanje in podobno.

Sodobna kuhinja ima cel kup takih naprav. Pomivalni stroj, ki sporoča, da je posoda oprana, ali kuhalna plošča, ki se pritožuje, da je nekaj nedovoljenega na grelni površini, spadata mednje.

Vgradni sistemi so vgrajeni v robote, videonadzorne kamere, merilne inštrumente. Na komunikacijsko stikalo lahko gledamo kot na vgrajeni sistem. Že skoraj vsaka mehanska in elektronska naprava deluje na osnovi računalniške strojne in programske opreme, ali pa se bo to kmalu zgodilo.

Vgradnih sistemov, ki delujejo kot računalniki, a se ne uporabljajo kot računalniki, je dosti več kot računalnikov za splošne namene. Tržišče računalniških komponent je po številu mikroprocesorjev na leto primerljivo s številom prebivalcev planeta Zemlja. Le približno vsak stoti procesor se znajde v računalniku, ostali se vgradijo v druge naprave. Današnje gospodinjstvo ima mogoče štiri osebne računalnike, a dosti več drugih naprav, v katerih je vgrajeno stokrat več procesorjev.

1.3 Na čem temeljijo

Vgradni sistemi temeljijo na elektronskih, računalniških in informacijskih tehnologijah.

Glavna komponenta večine današnih vgradnih sistemov je mikrokrmilnik (Angl. Microcontroller). Ko želimo še posebej poudariti, da vgradni sistem temelji na mikrokrmilniku, mu rečemo *mikrokrmilniški vgradni sistem*. Mikrokrmilnik je vezje, ki ima vsaj en procesor, več vrst pomnilnikov, primerno kombinacijo vhodnih in izhodnih vmesnikov ter krmilnikov, in vse to v enem samem čipu.

Nekateri vgradni sistemi temeljijo na namenskih mikroprocesorjih. Taki procesorji so tudi *digitalni signalni procesorji* ali s kratico DSP. Njihova moč je obdelava signalov. Zato jih najdemo v avdio in video opremi ter v komunikacijskih napravah.

Posebna podzvrst procesorjev za vgradne sisteme so *grafični procesorji* ali s kratico GPU. Spričo paralelne zasnove se odlikujejo po izredni procesni moči. Zato jih najdemo v napravah, kjer je bistvena pretočnost podatkov. Take so grafične, slikovne in video naprave. A ker so tako zmogljivi, se vse bolj uveljavljajo kot splošno namenski procesorji.

So tudi vgradni sistemi, ki ne temeljijo na mikroprocesorjih in mikrokrmilnikih. Nekateri temeljijo na *programljivih logičnih vezjih* FPGA (Angl. Field Programmable Gate Array). Spet drugi temeljijo na kombinaciji FPGA in mikrokrmilnikov. Teh bo verjetno v bodoče vse več.

Veliko je tudi takih vgradnih sistemov, ki temeljijo na splošnonamenskih računalnikih in so bili prilagojeni za vgradnjo. Seveda, vsak splošnonamenski računalnik z lahkoto postane namenska naprava. Pravzaprav je prav univerzalnost računalnikov dala vgradnim sistemom tak polet.

In vse več je takih naprav in sistemov, za katere ni povsem jasno, ali so 'zares pravi' vgrajeni sistemi. Tablice in prenosni telefoni so že tak primer.

In sčasoma bo poudarek na vgradnosti sistemov izpuhtel. A vgradni sistemi bodo ostali in končno postali komponente, kot so pred njimi že davno dioda,

tranzistor, ali operacijski ojačevalnik.

1.4 Od kje izvirajo

Vgradni sistemi sploh niso tako zelo novi. Tisto kar je novo, je njihova množična prisotnost ali *vsenavzočnost*.

Prvi mikroprocesorji so bili predvideni za vgradnjo v namenske naprave. Tedaj se od mikroprocesorjev ni pričakovalo, da bi nadomestili procesne enote splošnonamenskih računalnikov, čeprav se je prav to kmalu zgodilo. Danes ni računalnika brez mikroprocesorja. A ker je vsak procesor mikroprocesor, mu rečemo kar procesor.

Prvi komercialno uspešen mikroprocesor so izdelali pred skoraj pol stoletja v tedaj majhnem podjetju Intel. Mikroprocesor Intel 4004 se je na tržišču pojavil leta 1971. Poimenovali so ga mikroprogramljiv računalnik na čipu (Angl. Micro-programmable computer on a chip). Čip je bil zasnovan za naročnika, ki je izdeloval računske stroje, a ker so ga nameravali vgrajevali tudi v druge produkte, je bil zasnovan kot programljiv računalnik. To naj bi znižalo razvojne stroške.

Tedaj seveda ni bilo govora o vgradnih ali vgrajenih sistemih. Za množično prisotnost mikroprocesorjev in mikrokrmilnikov je bilo treba počakati še nekaj desetletij, dokler računalniških tehnologij niso začele sprejemati druge industrije. Najvplivnejše med temi so bile avtomobilska industrija, transportna industrija, komunikacijska industrija, zabavna industrija, avtomatizacija proizvodnih procesov in druge. Razvoj pa so narekovale potrebe na tržišču.

1.5 Zakaj vgradni sistemi

Vgradni sistemi so primerna tehnološka rešitev današnjega časa. Njihova bistvena lastnost je programljivost in posledično prilagodljivost.

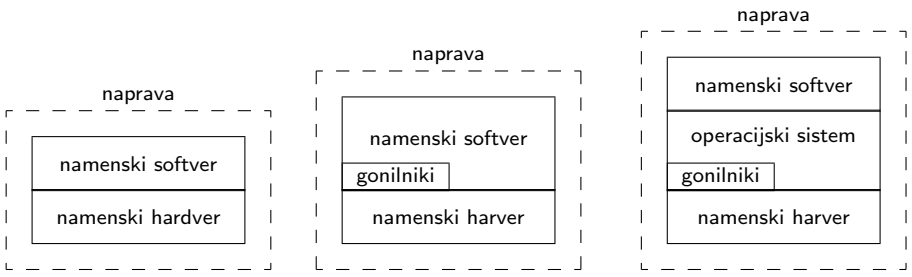
Ker so prilagodljivi, ponujajo proizvajalcem priložnost za znižaje razvojnih in proizvodnih stroškov. Vgradni računalniki omogoča cenejši in hitrejši razvoj novih produktov. Ne samo to. Ista mikrkrmilniška platforma je lahko podlaga raznovrstnim končnim produktom. To zniža tudi proizvodne stroške. Programljive naprave, kar vgradni sistemi so, ponujajo brezmejne možnosti za dodatno in naprednejšo funkcionalnost. Posledično to pomeni nove priložnosti na tržišču.

Uporabniki pričakujejo nižjo ceno, boljšo funkcionalnost in posledično večje zadovoljstvo. Brez te soodvisnosti vgradnih računalnikov ne bi bilo, ali pa jih ne bi bilo v takem obsegu, kot smo jim priča danes.

1.6 Kaj jih sestavlja

Vsak vgradni sistem temelji na ustrezni kombinaciji strojne in programske opreme, oboje pa je neločljivo povezano.

V oporo razlagi nam je slika 1.1. Ko je sistem enkrat vgrajen, služita tako strojna kot programska oprema dotičnemu specifičnemu namenu. Zato jima rečemo *namenski softver* in *namenski hardver*, čeprav bi, jasno, isti mikrokrmilnik lahko služil povsem drugemu namenu.



Slika 1.1: *Načelna zgradba vgradnih sistemov, od enostavnejših do kompleksnejših.*

Programska oprema majhnega vgradnega sistema poleg namenske pragramske opreme običajno vključuje zgolj minimalno dodatno programsko podporo za upravljanje strojne opreme, tipično nalagalni ali zagonski program

(Angl. Boot loader). Zagonski nalagalnik je res majhen program, ki vzpostavi začetne pogoje za popolni zagon sistema. Lahko je razširjen z osnovnimi funkcijami za preiskušanje ali diagnostiko.

Ker je programska oprema za upravljanje vhodno izhodnih krmilnikov, na primer ethernet krmilnika, CAN krmilnika, analogno digitalnega pretvornika ali časovnika, dokaj splošna oziroma neodvisna od konkretne aplikacije, so se razvile celovite programske rešitve imenovane gonilniki naprav (Angl. Device drivers).

Gonilnik je programski modul za upravljanje dotične strojne opreme. Gonilnik preko programskega vmesnika ustvari enotno abstrakcijo naprave, ki je neodvisna od konkretne izvedbe razpoložljivega hardvera ali konkretnega namena uporabe. Aplikacijska programska oprema je zato lažje *prenosljiva* ali lažje ponovno uporabljena v podobnem vgradnem sistemu ali na drugi platformi.

V kompleksnejših vgradnih sistemih se sočasno dogaja mnogo reči. Delovanje sistema koordinira *operacijski sistem* (OS). Tipične naloge operacijskega sistema so upravljanje vhodnih in izhodnih prenosov podatkov, razvrščanje opravil, sočasno napredovanje opravil ali *večopravilnost*, komunikacije med opravili, sinhronizacija opravil, obdelava prekinitev in izjem, in še kaj.

Operacijski sistem je programska oprema, ki nad strojno opremo nadgradi tisto funkcionalnost, ki je potrebna, a ni direktno vezana na en sam primer uporabe. Namenska programska oprema tudi ne dostopa direktno do strojne opreme. Za upravljanje s strojno opremo skrbi operacijski sistem.

V skoraj vseh vgradnih sistemih se od operaciskega sistema pričakuje, da bo zagotavljal delovanje v realnem času (Angl. Real-Time Operating System - RTOS). To pomeni, da bo poskrbel, da se bodo vse aktivnosti v sistemu seveda odvijale pravilno, predvidljivo in v skladu z vnaprej predpisanimi časovnimi zahtevami ali omejitvami.

1.7 Vgradni in splošnonamenski računalniki

Vgradni sistemi služijo specifičnemu namenu. Potem, ko se sistem vgradi, se programska oprema redko in le izjemoma spremeni, kar za splošnonamenske sisteme seveda ne drži. Računalniki za splošne namene se uporabljajo na veliko različnih načinov. Lahko se uporabljajo za raziskave in razvoj, za urejanje in oblikovanje besedil, za modeliranje in simulacije, za brskanje po spletu, branje elektronske pošte, in podobno. Programska oprema splošnonamenskih računalnikov se pogosto spremeni.

Programska oprema vgradnega sistema je neločljivo povezana s strojno opremo. Veliko vgradnih naprav deluje na podlagi programske opreme, ki je bila razvita v celoti za ta namen. To za splošnonamenske računalnike ne velja.

Programska oprema vgradnega sistema je največkrat razvita na razvojnem sistemu in se šele na koncu prenese na *ciljno platformo*. Za razvojni sistem služi splošnonamenski računalnik.

Strojna oprema vgradnih sistemov daje bistveno večji poudarek vhodno izhodnim prenosom in torej vmesnikom in krmilnikom za povezovanje z drugimi napravami. Vgradni sistem je denimo vezan s sistemom zavor, s črpalko, regulacijo obratov in tako dalje. Računalnik za splošno rabo ima nekaj tipiziranih zunanjih naprav, kot so diskovje, tiskalnik, zaslon, miška in podobno.

1.8 Katera znanja zahtevajo

Vgradni sistem je elektronska naprava, ki temelji na ustrezni kombinaciji strojne in programske opreme, obe pa sta tesno povezani.

Načrtovanje vgradnih sistemov zato zahteva znanja iz zgradbe in delovanja strojne in programske opreme. Koristna so znanja iz digitalnih vezij in sistemov, zgradbe in delovaja računalniških sistemov, algoritmov in podatkovnih struktur, načrtovanja in razvoja programov ter operacijskih

sistemov.

Vgradni sistem je vgrajen v napravo, ki služi končnemu namenu uporabe. Koristno, ali pa kar nujno, je poznavanje področja uporabe vgradnega sistema.

Glavni del vgradnega sistema je slej ko prej mikrokrmilnik. Za delo z mikrokrmilniki je nujno potrebno temeljito znanje o *arhitekturnih* lastnosti mikrokrmilniških sistemov. Spletne aplikacije ali podatkovne baze tega denimo ne zahtevajo.

In kaj je arhitektura? Pod arhitekturo razumemo zgradbo in delovanje sistema, ki sta neodvisna od same izvedbe. K arhitekturi mikrokrmilniškega sistema v splošnem prištevamo vse tisto kar je pomembno za načrtovalca programske opreme v enem od nižjenivojskih programskih jezikov, kot je na primer programski jezik C ali pač zbirni jezik. Med arhitekturne lastnosti spadajo predvsem:

- programski model procesorja (Angl. Programming Model). To je zgradba procesne enote kot jo vidi programer. Sem sodijo programsko dostopni registri procesorja in njigov pomen.
- Zbirka ukazov (Angl. Instruction Set), vključno z zgradbo oziroma obliko ukazov, načini naslavljanja (Angl. Addressing Modes) in cevovodi (Angl. Pipeline).
- Sistem prekinitev (Angl. Interrupts) in obdelava izjem (Angl. Exception Processing), torej kako se procesor odziva na zahteve od zunaj.
- Pomnilniška hierarhija in v njej predpomnilniki (Angl. Cache), pomnilniki, navidezni pomnilniki (Angl. Virtual Memory).
- povezovalne strukture ali vodila (Angl. Buses) in na primer širina vodil.

Področja zanimanja vgradnih sistemov torej so:

- arhitekture mikrokrmilniških s sitemov,
- načrtovanje in razvoj programov,

- programski jeziki, kot sta C/C++,
- sistemi realnega časa,
- operacijski sistemi za realni čas,
- področje uporabe.

1.9 Vgradni sistemi in ta knjiga

To je knjiga o zgradbi, delovanju in uporabi operacijskih sistemov za vgradne sisteme. Knjiga je namenjena študentom prvega letnika smeri Avtomatika in informatika magistrskega študijskega programa Elektrotehnika.

Glavnina snovi je oblikovana okrog sistemskih klicev in funkcij. Te tvorijo programski vmesnik med aplikacijami in operacijskih sistemom oziroma *sistemskim jedrom*. Ob spoznavanju programskega vmesnika bralec hkrati odkriva zgradbo in delovanje sistemskega jedra in pridobiva znanja, ki so potrebna pri razvoju aplikacij z neposredno uporabo jedra. Tem veščinam rečemo sistemsko programiranje.

Knjiga se torej osredotoča na sistemsko programiranje za vgradne sisteme. Govoriti o sistemskem programiranju, ne da bi se odločili za konkreten operacijski sistem, nima kakšne posebne koristi. V tej knjigi smo izbrali Linux.

Izbira Linuxa ni bila naključna. Linux je sodoben in hitro razvijajoči se operacijski sistem. Linux najdemo na majhnih in velikih vgadnih sistemih, na osebnih računalnikih in celo na večjedrnih strežnikih ter superračunalnikih. Uporaba Linuxa je svobodna. Vse to prispeva k izjemni priljubljenosti operacijskega sistema, njemu samemu pa zagotavlja trajnostni razvoj.

To pa ni knjiga, ki bi naslavljala arhitekture vgradnih sistemov ali načrtovanje in razvoj vgradnih sistemov. To ni knjiga o načrtovanju algoritmov ali o programskih jezikih. Iz te knjige se ne da učiti teorije operacijskih sistemov, niti ne administriranja ali dela z operacijskim sistemom. Taka knjiga bi morala imeti vsaj 2000 strani. Za učbenik za enosemestrski predmet

bi bilo to dosti preveč. So pa vsa ta znanja nujna za vse, ki vidijo svojo priložnost v načrtovanju in razvoju vgradnih sistemov, pa tudi širše.

1.10 Priporočeno nadaljnje čtivo

Za ta razdelek smo izbrali nekatere izmed številnih knjig, ki obravnavajo snov, ki je ne bomo obravnavali v okviru te knjige. Za razumevanje te knjige ta znanja niso nujna, a so koristna. So pa nujna za vse, ki bi se hoteli usposobiti za načrtovanje in razvoj vgradnih sistemov.

Najbolje bi bilo vzeti v roke eno od knjig, ki obravnavajo arhitekture računalniških sistemov na splošno, na primer [1], [2] ali kakšno od sorodnih knjig.

Priporočamo še kakšno knjigo, ki obravnava arhitekture, načrtovanje in razvoj vgradnih sistemov, kot sta knjigi [3] in [4]. In dobro bi bilo spoznati arhitekturo enega od sodobnih mikroprocesorjev/mikrokrmilnikov za vgradne sisteme. Takih knjig, učbenikov in priročnikov, je veliko.

In potem so tu algoritmi in podatkovne strukture ter načrtovanje in razvoj programov. Priporočljivo je solidno znanje vsaj enega programskega jezika. Programska jezika C in C++ sta najboljša izbira.

Nenazadnje so tu knjige iz operacijskih sistemov. A ker je to knjiga o operacijskih sistemih, bomo dodatno čtivo sproti navajali.

Literatura

[1] D. Kodek, *Arhitektura in organizacija računalniških sitemov*, Bi-tim, 2008.

[2] A. Tanenbaum, *Structured computer organization*, Pearson, 2010.

[3] Wayne Wolf, *Computers as components, 2-nd Ed.*, Morgan Kaufman, Elsevier, 2008.

[4] T. Noergaard, *Embedded systems architecture*, Elsevier, 2005.

[5] J. Valvano, *Embedded systems, Vol. 1-3*, CreateSpace, 2012-2014.

Poglavje 2

Uvod v operacijske sisteme

V tem poglavju se bomo posvetili operacijskim sistemom. Spoznali bomo osnovne naloge in načelno zgradbo operacijskih sistemov. Ozrli se bomo nazaj v preteklost ter poudarili glavne mejnike v razvoju sistemov UNIX®[1], kamor sodi tudi Linux®[2]. Na koncu bomo sistem Linux poskušali približali z uporabniškega stališča. A najprej se vprašajmo, kaj operacijski sistem sploh je.

2.1 Kaj je operacijski sistem

Računalniški sistem sestavljata strojna in programska oprema. Delu sistemske programske opreme, ki skrbi za to,

- da se vse dejavnosti v sistemu pravilno odvijajo,
- da so sredstva dobro izkoriščena,
- da je sistem zmožen interakcije z okoljem

ter da je spričo naštetega pripravljen za uporabo, rečemo *operacijski sistem.*

[1] UNIX je blagovna znamka The Open Group
[2] Linux je blagovna znamka Linusa Torvaldsa.

Glavne sestavine strojne opreme so procesor, pomnilnik ter vhodni in izhodni vmesniki ter krmilniki. Od operacijskega sistema se pričakuje, da z njimi upravlja učinkovito. Ko, denimo, uporabniški program potrebuje pomnilnik, mu ga dodeli operacijski sistem. Ali, ko uporabnik potrebuje dostop do diskovne naprave, mu to omogoči operacijski sistem. Operacijski sistem je *vmesnik* med strojno opremo in uporabniškimi programi. Pravo delo opravljajo uporabniški programi, medtem ko jim operacijski sistem pri tem pomaga.

Operacijski sistem ustvarja primerno okolje za izvrševanje programov. Z vidika delovanja sistema je bolj pomembno kaj se v sistemu dogaja kot pa na kakšen način, to je, kakšna je specifika procesorjev, pomnilnikov in povezav. Operacijski sistem nad strojno opremo ustvari za uporabo primernejšo *abstrakcijo* sistema. Denimo, uporabniški program na eni napravi komunicira s programom na drugi napravi. Operacijski sistem mu omogoča, da podatke pošlje ali sprejme. To je vse, kar program potrebuje in kar pričakuje. Pri tem ni važno kakšen je in kako deluje komunikacijski krmilnik ali kakšni so in čemu služijo registri krmilnika.

Ni vsa *sistemska programska oprema* operacijski sistem. In kaj v okviru sistemske programske opreme spada k operacijskemu sistemu? Urejavalniki besed in prevajalniki so del sistemske programske opreme, a ne spadajo v operacijski sistem.

V najožjem smislu je operacijski sistem *sistemsko jedro* (Angl. Kernel). To je tisti del programske opreme, ki je med delovanjem sistema stalno nameščen v pomnilniku. Jedro opravlja naloge, ki jih od njega zahtevajo uporabniškimi programi. Pri tem uporablja strojno opremo.

V nekoliko širšem smislu sestavljajo operacijski sistem sistemsko jedro in sistemske knjižnice. *Sistemske knjižnice* so zbirke različnih funkcij, brez katerih bi bilo delovanje in uporaba sistema okrnjeno ali celo nemogoče. Knjižnice približajo operacijski sistem uporabniškim programom.

V še širšem smislu prištevamo k operacijskemu sistemu program, ki mu rečemo *tolmač ukazne vrstice* (Angl. CLI - Command Line Interpreter). Ta program pomaga pri uporabi sistema. Od uporabnika sprejema ukaze,

jih tolmači in s pomočjo jedra poskrbi za njihovo izvršitev. V nekaterih operacijskih sistemih je ta program del sistemskega jedra. V sistemih UNIX/Linux je to samostojen program, ki se imenuje *školjka* (Angl. Shell). Za uporabnika je školjka bistven del sistema, a s stališča jedra se školjka praktično v ničemer ne razlikuje od ostalih uporabniških programov.

Včasih operacijski sistem razširimo še s programi, ki niso del jedra, a dodatno pomagajo pri uporabi sistema. Tem programom rečemo sistemska orodja ali sistemski *ukazi* (Angl. Utilities). To so ukazi za prepis, preimenovanje, brisanje datotek, listanje direktorijev in podobno. Denimo, ukaz za prepis datotek, v sistemih UNIX in Linux je to program z imenom `cp`, od operacijskega sistema preprosto zahteva branje in pisanje dokler datoteka ni v celoti prepisana. Sam pri tem opravi bore malo. Ti ukazi približajo operacijski sistem uporabniku. Od njih in školjke je odvisno, kako uporabniki doživljajo operacijski sistem.

V najširšem smislu se včasih k operacijskemu sistemu prišteva vse kar obsega *distribucija*, to je vse kar spada k sistemu, ki je pripravljen za končno uporabo. Denimo, poleg že navedenega jedra in ukazov, distribucija vsebuje poljubno in lahko tudi veliko programskih *paketov*, na primer: uporabniško grafično okolje, zbirko prevajalnikov, orodja za urejanje in oblikovanje besedil, razvojna programska orodja, in podobno. Vendarle je tako pojmovanje preveč posplošeno in ga odsvetujemo. V tem primeru je besedo 'operacijski' najbolje kar opustiti in reči preprosto *sistem* UNIX, sistem GNU/Linux, sistem Windows, itd.

V tem delu bomo izraz operacijski sistem vezali na funkcionalnost, ki jo daje sistemsko jedro. Funkcionalnost jedra bo dostopna bodisi direktno, bodisi posredno preko knjižnic in včasih tudi prek sistemskih ukazov.

2.2 Osnovne naloge operacijskega sistema

Operacijski sistem torej ustvarja primerno okolje za napredovanje programov. Programom v izvrševanju rečemo *opravila* (Angl. Task) ali *procesi*. Naloga operacijskega sistema je, da program namesti v pomnilnik oziroma

mu dodeli pomnilnik. Naloga operacijskega sistema je, da programu dodeli procesor ter mu s tem ustvari pogoje, da se izvrši. *Dodeljevanje procesorja* in *dodeljevanje pomnilnika* sta temeljni naloga operacijskega sistema.

V skoraj vsakem sistemu se v danem obdobju izvaja več dejavnosti. Teh sočasnih dejavnosti je lahko tudi zelo veliko. Naloga operacijskega sistema je, da v pravem trenutku omogoči napredovanje pravih opravil in s tem zagotovi, da bo sistem deloval v skladu z zahtevami in pričakovanji. *Upravljanje procesov* je poglavitna naloga operacijskega sistema. Operacijski sistem mora biti zato sposoben:

- ustvariti novo opravilo in ga zaključiti, ko ni več potrebno,
- odložiti ali obnoviti izvajanje opravila, če je to potrebno,
- časovno razvrščati opravila z namenom, da bo sistem deloval tako, kot to narekujejo potrebe.

Opravila lahko napredujejo neodvisno eno od drugega, ali pa sodelujejo. Če sodelujejo, morajo vsaj občasno med sabo komunicirati in se po potrebi časovno usklajevati. Podpora *medprocesnim komunikacijam* in *sinhronizaciji* spada med naloge operacijskega sistema.

Pomnilnik in procesor sta vitalnega pomena za napredovanje opravil. Opravilo lahko napreduje le tako, da se delno ali v celoti nahaja v pomnilniku. Če je aktivnih več opravil, mora vsako dobiti zadosten delež pomnilnika tedaj, ko ga potrebuje. *Upravljanje pomnilnika* je ena od bistvenih nalog operacijskega sistema, kar vključuje:

- dodeljevanje pomnilnika in s tem v zvezi odločanje kje v pomnilniku, na kakšen način, kdaj, katerim opravilom in za koliko časa jim dodeliti pomnilnik,
- zaščito pomnilnika, vključno z zaščito naslovnih prostorov procesov tako, da napredovanje enega procesa ne ogroža pravilnosti napredovanja drugega procesa,
- nadziranje pomnilnika, vodenje evidence kateri deli pomnilnika so dodeljeni, kateri so prosti, in podobno.

Večina sodobnih sistemov realizira koncept *navideznega pomnilnika*. Izvedba navideznega pomnilnika zahteva primerno strojno opremo, s katero upravlja programska oprema, ki je del operacijskega sistema.

Nadaljnja pomembna naloga operacijskega sistema je skrb za izmenjavo podatkov z zunanjimi napravami. To vključuje pravočasno ter predvidljivo odzivanje na izjemne dogodke in prekinitve od zunaj (Angl. Exceptions, Interrupts). V sistemih brez operacijskega sistema je obravnavanje zahtev za prekinitev naloga uporabniškega programa. V sistemih z operacijskim sistemom stoji med zahtevami za prekinitev in uporabniškim programom operacijski sistem.

Pod pojmom *prekinitev* razumemo električni signal, ki ga procesorju pošlje vmesnik ali krmilnik zunanje naprave, časovno števno vezje ali kakšna druga naprava in na ta način zahteva takojšnjo pozornost procesorja. Procesor spričo zahteve za prekinitev odloži izvrševanje tekočega zaporedja ukazov, izvrši prekinitveneno strežno rutino (Angl. ISR - Interrupt Service Routine) in s tem streže zahtevi za prekinitev. Potem obnovi izvrševanje prekinjenega zaporedja ukazov, kot da do prekinitve ne bi prišlo. Strežni rutini rečemo tudi *prekinitveni strežnik* (Angl. Interrupt handler) in je del operacijskega sistema.

Razlikujemo dva tipa prekinitev, sinhrone in asinhrone. Asinhrone zahteve se javljajo spontano, nenapovedano in torej *asinhrono* glede na izvrševanje tekočega programa. Na primer, pritisk tipke ima lahko za posledico tako zahtevo za prekinitev. Tem, asinhronim prekinitvam, rečemo tudi strojne prekinitve (Angl. Hardware interrupts) ali kar samo prekinitve.

Prekinitve sinhronega tipa se javljajo *sinhrono* z izvrševanjem programa in so ali direktna posledica izvršitve *prekinitvenega ukaza* ali stranski učinek izvršitve dotičnega ukaza. Tem prekinitvam običajno rečemo *izjeme* (Angl. Exceptions). Izjemi, ki je direktna in neizogibna posledica izvršitve ukaza, rečemo tudi *programska prekinitev* (Angl. Software interrupt) ali *past* (Angl. Trap). Tak primer je ukaz `INT n`, kjer je `n` številka pasti. Izjemi, ki nastopi pogojno kot stranski učinek spričo nepravilnosti izvrševanja ukaza, rečemo *napaka*. Primeri so deljenje z nič, neveljaven ukaz, napaka na vodilu in

napaka strani.

Streženje zahtev za prekinitev je naloga operacijskega sistema. Brez primerne strojne podpore prekinitvenemu sistemu, s katero upravlja operacijski sistem, si sodobnega operacijskega sistema ne moremo niti zamisliti.

Končno, računalnik je naprava za obdelavo podatkov. Pomembna naloga operacijskega sistema je skrb za varnost, zanesljivost in celovitost podatkov. Za shranjevanje podatkov so potrebna pomnilniška sredstva in naprave. Tudi upravljanje naprav in primerno strukturiranje podatkov v *datotečni sistem* sta nalogi operacijskega sistema.

Če povzamemo, operacijski sistem v splošnem skrbi za:

- upravljanje procesov oziroma opravil (Angl. Process Management),
- upravljanje pomnilnika (Angl. Memory Management) in
- upravljanje vhodno/izhodnih prenosov ali naprav ter datotečni sistem (Angl. Input/Output, Storage/File System Management).

2.3 Načelna zgradba operacijskega sistema

Niso vsi operacijski sistemi grajeni po enakih načelih. Nekateri postavljajo v ospredje majhnost. Ta je pomembna v vgradnih sistemih. Drugi operacijski sistemi izpostavljajo splošnost. Ta je pomembnejša v splošno namenskih sistemih. Spet tretji poudarjajo prilagodljivost. V nekaterih operacijskih sistemih za vgradne sisteme je skoraj vse podrejeno realnemu času.

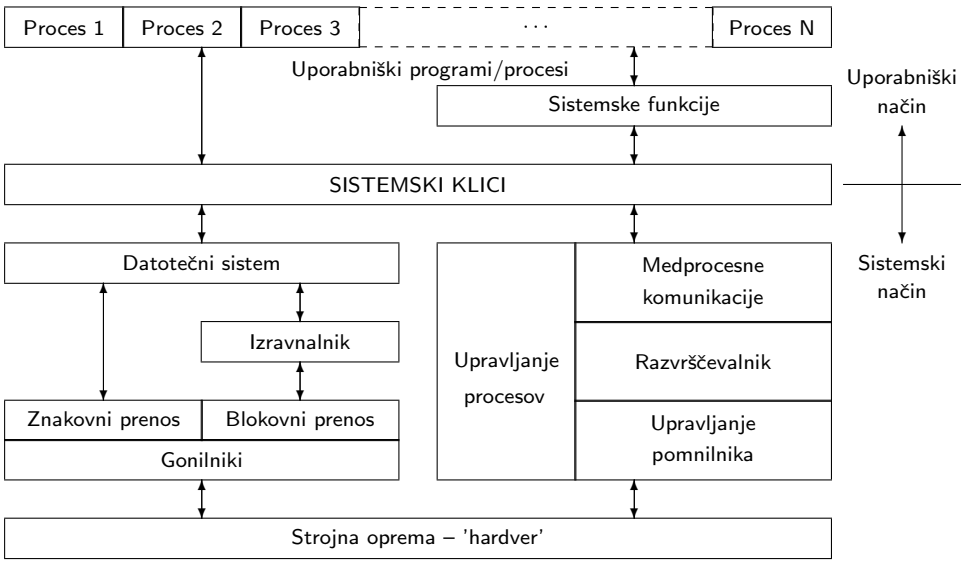
Zelo načelno zgradno operacijskega sistema prikazuje slika 2.1. Nad strojno opremo je sistemsko jedro, ki je stalno nameščeno v pomnilnik. Kadar sistem deluje oziroma napreduje v jedru, pravimo, da deluje v *sistemskem načinu* ali tudi v sistemskem *prostoru* (Angl. System mode, Kernel space). Jedro skrbi za upravljanje procesov, za razvrščanje procesov (Angl. Scheduling), za dodeljevanje procesne enote in menjavo procesov (Angl. Context switch), dodeljevanje oziroma upravljanje pomnilnika (Angl. Memory management – MM) ter medprocesno komunikacijo (Angl. Interprocess communication – IPC).

Del jedra so *gonilniki* ali *moduli* v podporo vhodnim in izhodnim napravam (Angl. Device drivers). V okviru le-teh se streže tudi zahtevam za prekinitev. Pretok podatkov poteka bodisi v obliki neprekinjenega zaporedja bajtov bodisi kot zaporedje večjih podatkovnih enot oziroma *blokov*. Ker deluje zunanja naprava v splošnem z drugačno hitrostjo kot računalnik, je potrebna izravnava hitrosti. Temu služijo medpomnilniki jedra ali *izravnalniki* (Angl. Buffers). Spričo tega poznamo *znakovne* in *blokovne* naprave oziroma znakovne in blokovne podatkovne prenose (angl. Character devices, Block devices). Sistemsko jedro skrbi tudi za datotečni sistem (Angl. File system) oziroma realizira operacije nad njim in s tem definira strukturo datotečnega sistema.

Operacijski sistem ustvarja primerno okolje, v katerem se odvijajo aplikacijski programi oziroma procesi. Teh je načeloma lahko poljubno in praviloma veliko. Procesi večino časa napredujejo v *uporabniškem* načinu oziroma prostoru (Angl. User mode, User space). V tem načinu proces denimo računa, ureja podatke, filtrira signale. Kadar proces potrebuje storitve sis-

temskega jedra, izvede *sistemski klic*. Primer sistemskega klica je denimo zahteva za branje z vhodne naprave. Tedaj se izvrševanje procesa prenese iz uporabniškega načina v *sistemski način* oziroma v jedro. Ko je zahtevi uporabniškega procesa zadoščeno, se izvajanje vrne iz sistemskega v uporabniški način. Prehod iz uporabniškega v sistemski način je realiziran prek *programskih prekinitev* oziroma *izjem* (Angl. Software interrupts, Exceptions). Vmesnik med jedrom in uporabniškimi procesi zaokrožajo sistemski klici (Angl. System calls). O tem bomo več spregovorili kasneje.

Funkcionalnost sistemskih klicev in s tem operacijskega sistema nadgrajujejo sistemske knjižnice. V okviru sistemskih knjižnic (Angl. System libraries) so realizirane sistemske funkcije (Angl. System functions). Sistemske funkcije se praviloma izvršujejo v uporabniškem načinu. Izvršitev določene sistemske funkcije, na primer C-jevske funkcije `printf` iz standardne vhodno izhodne knjižnice, pa ima lahko za posledico tudi sistemski klic `write` in spremembo delovanja iz uporabniškega v sistemski način. Tudi o tem bomo še govorili.



Slika 2.1: *Načelna zgradba operacijskega sistema.*

2.3.1 Monolitno jedro

Slika 2.1 najbolje ustreza zgradbi *monolitnega jedra*. Monolitno jedro pomeni programsko opremo jedra v enem nedeljivem kosu. Na sliki 2.1 spada vse pod vmesnikom sistemskih klicev k jedru. Če hočemo jedru kaj dodati ali mu kaj odvzeti, ga moramo nanovo *zgraditi*. To lahko vključuje prevajanje in povezovanje posameznih delov jedra v celoto.

Monolitno jedro tudi ni modularno in spričo tega najverjetneje večje kot bi bilo nujno potrebno za dotični primer uporabe. Tipično monolitno jedro je zato veliko in precej okorno. Značilnost monolitnega jedra je tudi ta, da se vse sistemske aktivnosti dogajajo v sistemskem načinu. Klasičen UNIX je tipičen predstavnik operacijskega sistema z monolitnim jedrom.

2.3.2 Mikrojedro

Nasprotje monolitnega jedra je *mikrojedro*. Kot nakazuje ime, je mikrojedro majhno jedro. Jedro je majhno tako po obsegu programske opreme kot po času, ko naprava teče v sistemskem načinu.

Po zamisli mikrojedra se operacijski sistem razcepi na manjše samostojno stoječe celote. To naj bi prispevalo k stabilnosti sistema. Na primer, napaka v godilniku monolitnega jedra zelo verjetno sesuje celoten sistem. V podobni sistuaciji v sistemu z mikrojedrom se sesuje gonilnik, kar pa ne moti delovanja drugih delov sistema.

Koncept mikrojedra naj bi zmanjšal število prehodov med sistemskim in uporabniškim načinom in s tem dosegel hitrejše napredovanje opravil. V okviru jedra se poskuša obdržati samo najbolj osnovne in bistvene naloge operacijskega sistema. Manj kot polovica jedra s slike 2.1 ostane v mikrojedru. Na primer, upravljanje datotečnega sistema se lahko v celoti prenese v uporabniški način. Medprocesne komunikacije se lahko v precejšnjem obsegu dogajajo v uporabniškem načinu. Tudi podpora omrežnim komunikacijam je lahko v uporabniškem načinu, medtem ko razvrščanje in menjava procesov ter upravljanje pomnilnika ostane v jedru. Spreminjanje jedra

spričo dodajanja ali odvzemanja gonilnikov naprav ni potrebna. Koncept mikrojedra naj bi bil torej naprednejši, bolj prilagodljiv in vsaj teoretično bolj učinkovit. Praksa kaže, da s stališča učinkovitosti temu ni nujno tako. Primeri sistemov z mikrojedrom so na primer Mach, MINIX in GNU Hurd.

2.3.3 Jedro Linux

Za sistem Linux bi lahko ugotovili, da se nahaja nekje vmes med monolitnim jedrom in mikrojedrom. Linux je sicer grajen po načelu monolitnega jedra, a je *modularen*. Modul jedra je samostojna in zaključena celota. Modul lahko jedru dodamo ali odvzamemo brez spreminjanja drugih delov jedra in tudi med delovanjem sistema. To je podlaga za prilagodljivost in posledično za ekonomičnost sistema.

2.3.4 Hipervizor

Zadnja in za današnje vgradne sisteme manj zanimiva rešitev je *hipervizor*. Zamisel hipervizorja je kompletna vitualizacija strojne opreme. Hipervizor je res *tanek* operacijski sistem, nad katerim gostujejo drugi, gostujoči operacijski sistemi. Spričo tega lahko na isti strojni opremi sobiva več enakih ali različnih operacijskih sistemov. Gostujoči operacijski sistemi tečejo kot uporabniški procesi in vsak zase ustvarjajo svoj abstrakten *virtualen računalnik*. So pa še drugi pristopi k virtualizaciji, a o tem ne bomo govorili.

2.4 Pogled nazaj

Vse kar ostane ima tudi svojo preteklost. Najprej je bil UNIX, prišel je GNU in sledil je Linux. Razvojno gledano so UNIX, Linux in GNU neločljivo povezani. Mi bomo obujali spomine v obratnem vrstnem redu kot je potekal razvoj.

2.4.1 Linux

Nastanek sistema Linux sega v leto 1991. Avtor sistemskega jedra Linux je Linus Torvalds, tedaj študent računalništva na univerzi v Helsinkih. Torvalds je sistemsko jedro zasnoval praktično iz nič. Navdih za razvoj lastnega operacijskega sistema je našel v tedaj zelo priljubljenih različicah sistemov UNIX. Pri razvoju jedra se je zgledoval po sistemu MINIX, ki ga je sredi osemdesetih let prejšnjega stoletja razvil Andrew Tanenbaum, avtor številnih knjig in univerzitetni profesor na Nizozemskem.

Tanenbaum je MINIX zasnoval po načelu *mikrojedra*, ki naj bi bilo naprednejše od tedaj že uveljavljenega koncepta monolitnega in spričo tega velikega jedra. Kot pravi avtor, je bil MINIX namenjen izključno za izobraževalne namene. Zato je bil zasnovan povsem splošno in neodvisno od ciljne računalniške arhitekture oziroma *platforme*. Učinkovitost naj ne bi bila tako pomembna. Po mnenju Torvaldsa je bil MINIX na tedaj vse bolj prevladujoči družini Intelovih procesorjev x86 premalo učinkovit, kar ga je dodatno spodbudilo k razvoju lastnega jedra.

Čeprav se je Torvalds odločil za lasten razvoj, pa je povsod, kjer je bilo mogoče, upošteval že uveljavljene rešitve. Do tedaj se jih je večina že katalizirala v specifikaciji POSIX[3]. To se je kasneje izkazalo kot dobra naložba v trajnostni napredek sistema.

Torvalds je sredi leta 1991 oznanil, da je delo na jedru pri koncu in jedro pripravljeno za uporabo. Jedro je ponudil v prosto uporabo ter povabil računalniško skupnost k nadaljnemu razvoju sistema. Odziv je bil izreden. Sledila so leta hitrega in stalnega razvoja, ki traja še danes. Verjetno so prav odprto kodni pristop pod licenčnimi pogoji GNU GPL (GNU General Public License), skladnost s POSIX ter skupnost izjemno nadarjenih programerjev, ki so prostovoljno prispevali nove rešitve, pripomogli k priljubljenosti in množični prisotnosti Linuxa.

[3]POSIX je skovanka iz Portable Operating System Interface for UNIX, a o tem kasneje.

2.4.2 GNU

Ko dandanes rečemo Linux, se običajno sklicujemo na celotno distribucijo Linuxa, čeprav bi bilo primerneje z Linux poimenovati zgolj sistemsko jedro. Namreč, večina programske opreme, ki je vključena v distribucijo sistema Linux, je začela nastajati že pred jedrom samim v okviru projekta GNU. Začetek projekta GNU sega v leto 1984, njegov idejni vodja pa je od samega začetka Richard Stallman. Stallman zagovarja stališče, da mora biti softver *svoboden za uporabno*, vključno z izvorno kodo. To sicer ne pomeni, da bi moral biti softver zastonj, čeprav posredno vendarle nakazuje, da ne bi smel biti posebej drag. Cilj projekta GNU je bil razvoj sistema tipa UNIX, ki bi bil svoboden za uporabo. Do leta 1990 je bilo večina kode gotove, kot denimo gcc (GNU compiler collection), manjkalo je le še jedro. Linux je tako prišel kot naročen in zapolnil 'še edini manjkajoči člen'. Po mnenju Stallmana bi se morala distribucija operacijskega sistema zato pravilno imenovati sistem GNU/Linux.

Večina programske opreme v distribuciji Linux-a je licencirana pod pogoji GNU GPL (GNU General Public License). Licenca GNU GPL je *odprto kodna* in distribuiranje je svobodno. To ne pomeni, da ta koda ne bi imela avtorstva ali lastništva, vendarle daje lastnik uporabniku pod temi licenčnimi pogoji pravico do uporabe in do vpogleda v izvorno kodo. Uporabnik sme kodo spreminjati, spremenjeno kodo uporabljati, a ga licenca zavezuje, da tako modificirano kodo daje naprej tudi drugim v uporabo, in sicer pod enakimi pogoji, to je pod pogoji GNU GPL. Licenca je prosto dostopna na spletnih straneh projekta GNU `http://www.gnu.org/licenses/`.

2.4.3 UNIX

Linux spada v družino operacijskih sistemov UNIX. Ko govorimo o nastanku sistema Linux, ne moremo mimo razvoja sistema UNIX in z njim neločljivo povezanega programskega jezika C. Operacijski sistem UNIX, kot tudi programski jezik C, predstavljata enega najvidnejših mejnikov v razvoju računalništva na sploh.

Pojav sistema UNIX sega v leto 1969, ko je majhna skupina računalniških zanesenjakov s Kenom Thompsonom na čelu v Bellovih Laboratorijih v New Jersyju zasnovala operacijski sistem, ki je sprva deloval na tedaj malo znanem mini računalniku PDP-7. Večino dela na operacijskem sistemu je opravil Ken Thompson sam, medtem ko je programski jezik C iznašel Denis Ritchie. Sistem UNIX je bil kmalu zatem prepisan v C. UNIX je bil prvi operacijski sistem, ki je bil skoraj v celoti narejen v višjem programskem programskem jeziku, kar mu je kasneje zagotovilo razmeroma gladko pot širitve na druge platforme.

Prva izdaja UNIX-a je bila objavljena leta 1971. Sprva je bil sistem v uporabi samo v Bellovih Laboratorijih, a priljubljensot mu je hitro naraščala. Bell Labs je bil v tem času del ameriškega telekomunikacijskega giganta AT&T, ki mu je bilo spričo monopolnega položaja na trgu telefonskih storitev prepovedano trženje programske opreme. Zato so se odločili za licenciranje uporabe programske opreme po simbolični ceni v višini stroškov distribucije. Mnoge univerze so se odločile za nakup. Z distribucijo so dobili vso dokumentacijo in kompletno izvorno kodo. Ta poteza AT&T je kmalu obrodila sadove. Mnogi študentje širom Amerike so se sprva učili sestavin operacijskih sistemov iz izvorne kode in kasneje bistveno prispevali k nadaljnemu razvoju sistema. Nekateri so mu ostali zvesti privrženci vse življenje.

Neizbrisno sled v razvoju UNIXa so pustili študentje računalništva na Kalifornijski univerzi v Berkelyju. K temu je dodatno prispeval Ken Thompson, ki je bil na Berkelyju v tem času gostujoči profesor. Rodila se je tako imenovana BSD (Berkely Software Distribution) veja UNIXa ali BSD UNIX. Posebej pomemben prispevek BSD veje UNIXa je bila podpora komunikacijskim omrežjem, vključno z vmesnikom komunikacijskih vtičnic (Angl. Sockets).

BSD UNIX je bil sprva še obremenjen z določeno licenčno kodo AT&T različice UNIX, dokler niso bili dokončno odstranjeni ali na novo narejeni tudi tisti kosi jedra. Prva licence prosta in hkrati zadnja verzija v razvojni verigi BSD, je bila 4.4BSD.

Večina komercialnih različic UNIXa so dejansko nasledniki bodisi zadnje AT&T različice UNIX SVR4 (System Five Release 4) bodisi BSD UNIX veje, ki se je zaključila v devetdesetih letih z različico 4.4BSD in končno z 4.4BSD-Lite.

2.5 Osnovno o Linuxu

V tem poglavju bomo zelo na kratko in brez posebne razlage povzeli lastnosti Linuxa, ki razkrivajo, da je Linux mlajši sorodnik UNIXa.

Za tiste, ki bi želeli pobližje spoznati Linux in delo z Linuxom, priporočamo dokumentacijo, ki je nastajala vzporedno z razvojem sistema Linux v okviru pobude TLDP (The Linux Documentation Project). Uporaba dokumentov je svobodna pod licenčnimi pogoji GNU FDL (GNU Free Documentation License). Primeren začetek je prosto dostopna knjiga *Introduction to Linux, a hands on guide*, `http://tille.garrels.be/training/tldp/` avtorice Machtelt Garrels. Najbolje kar ob delu z Linuxom.

2.5.1 Uporabniki

Linux je večuprabniški operacijski sistem. To pomeni, da lahko računalnik hkrati uporablja več uporabnikov. Med uporabo se vsakemu od uporabnikov zdi, da je računalnik dodeljen samo njemu.

Uporabnik lahko sedi neposredno za računalnikom, ali pa se nanj priključi prek omrežja. Sočasnih daljinskih dostopov je lahko več. Že od nekdaj je daljinski dotop šel prek storitve `telnet` ali pač sorodne storitve `rlogin`. Spričo ranljivosti `telnet` dostopa, prijavno geslo se skozi omrežje namreč prenaša povsem odkrito, se je ta način opustil. Dandanes omogoča varen dostop prek omrežja šifrirana storitev `ssh` (Angl. Secure shell).

Uporabnik je napram sistemu enoznačno določen s številko uporabnika. Številko uporabnika kratko značimo z UID (Angl. User IDentity). Običajno je tej številki prirejeno še *ime uporabnika*. S tem imenom se uporabnik

prijavi v sistem.

Zaradi preglednosti so uporabniki razvrščeni v skupine. Vsak uporabnik pripada vsaj eni skupini. Skupino uporabnikov označuje številka skupine ali kratko GID (Ang. Group IDentity)

Seznam uporabnikov je zbran v datoteki `/etc/passwd`. V tej datoteki je vsak uporabnik opisan z eno tekstovno vrstico. Vrstica vsebuje naslednja polja, ki so ločena z dvopičji:

- ime uporabnika, na primer `stanek`,
- prijavno geslo uporabnika v šifrirani obliki (Angl. User password)[4],
- številko uporabnika (UID),
- številko skupine uporabnika (GID),
- prava ime in priimek uporabnika,
- domači direktorij uporabnika, na primer `/home/stanek`,
- ime programa, ki se zažene ob prijavi. To je ime prijavne školjke, na primer `/bin/bash`.

pripada ena vrstica Skupini uporabnikov je prirejeno ime. Povezava med imenom skupine in številko skupine je zabeležena v datoteki `/etc/group`. V tej datoteki pripada vsaki skupini ena vrstica besedila. Vrstica poleg številke in imena skupine vsebuje seznam uporabnikov, ki tvorijo skupino.

2.5.2 Skrbnik

Eden od uporabnikov je uporabnik s posebnimi pooblastli (Angl. superuser). Njegova številka je nič in njegovo običajno ime je `root`. Uporabniku `root` je dovoljeno vse. Ker mu je dovoljeno vse, lahko naredi dosti dobrega, zato tudi ima vsa pooblastila. Namenoma ali nenamenoma pa lahko naredi še več škode.

Osebo, ki se prijavi z imenom `root`, imenujemo *sistemski administrator* ali

[4]Zaradi varnosti so se gesla v novejših sistemih prenesla v drugo datoteko.

skrbnik sistema. *Skrbnik* poskrbi za vzpostavitev in delovanje sistema. Delo skrbnika je spričo pooblatil, ki jih ima, odgovorno.

2.5.3 Školjka

Ob prijavi uporabnika v sistem (Angl. Login), se zažene *prijavna školjka* (Angl. Login shell). Školjka je poseben program, ki od uporabnika sprejema ukaze in poskrbi za njihovo izvršitev (Angl. Command interpreter). Odtipkanemu besedilu, ki ga školjka interpretira kot ukaz, rečemo ukazna vrstica (Angl. Command line).

Nekatere ukaze izvrši školjka kar sama. To so interni ukazi školjke. Tak ukaz je denimo `cd` (od Angl. change directory). Za izvršitev drugih ukazov, školjka od jedra zahteva izvršitev programa, ki realizira ukaz. Primer takega ukaza je denimo ukaz za prepis datotek `/bin/cp`. Na enak način školjka pomaga izvrševati programe, ki jih razvije uporabnik sam.

V nekaterih sistemih sledi takoj po prijavi uporabnika zagon uporabniškega grafičnega vmesnika (GUI), ali pa se vmesnik GUI vzpostavi že ob samem zagonu sistema. Podporo grafičnemu okolju daje sistem za upravljanje oken `X` (Angl. X windows). Uporabnik od tedaj naprej komunicira s sistemom precej več s pomikanjem in klikanjem miške. To sicer ne pomeni, da bi stik s školjko povsem izgubil. Uporabnikom sistema Linux je v navadi, da kmalu po vzpostavitvi vmesnika GUI odprejo več terminalskih oken. Za vsakim oknom deluje školjka.

Školjka je mnogo več kot tolmač ukazne vrstice. Školjka vzdržuje in osvežuje spremenljivke okolja (Angl. Environment variables), pomni zgodovino vnešenih ukazov, omogoča urejanje zgodovine ukazov in daje možnost za ponovno izvršitev ukaza. Školjka omogoča izvršitev zaporedja ukazov v *skriptni datoteki*. Skripta datoteka dovoljuje uporabo preprostih programskih sestavin, kot je ponovitveni stavek, pogojni stavek, spremenljivke in podobno. Skratka, školjka ima vgrajeno funkcionalnost interpreterskega programskega jezika.

Ker školjka ni integrirana v jedro, ampak je samostojno stoječi program,

jo je moč preprosto zamenjati. Morda najbolj priljubljena školjka sistema Linux je `/bin/bash` (Angl. Bourne-again shell). So pa še druge, `/bin/sh`, `/bin/csh`, `/bin/ksh`, `/bin/tcsh`. Uporabnik se lahko odloči za katerokoli od njih.

Odjava je enostavna. Z ukazom `exit` preprosto zapustimo školjko. Pravzaprav od nje zahtevamo, naj se zaključi, s čimer je konec prijavne seje.

2.5.4 Priročnik

Uporabniku koristna lastnost Linuxa so *strani priročika*. Do vsebine priročnika pridemo kadarkoli z ukazom `man`. Na primer,

```
man ls
```

izpiše namen, način uporabe ukaza `ls`, in še marsikaj. Priročnik je deljen na poglavja. Mogoče je, da so različne vsebine v različnih poglavjih dosegljive z enakim iskalnim geslom. Na primer,

```
man 2 read
```

izpiše razlago za sistemski klic `read` v poglavju 2.

2.5.5 Procesi

V sistemu Linux lahko sočasno napreduje večje število programov. Seveda, drugače tudi večuprabniški ne bi mogel biti. Sistemu, ki omogoča takšno obliko sočasnosti, rečemo večopravilni sistem. Programu, ki napreduje, rečemo *proces* ali opravilo.

V delujočem sistemu vedno obstaja precej veliko število procesov oziroma opravil. Vsak od prijavljenih uporabnikov potrebuje vsaj eno školjko in torej vsaj en proces. Potem so tu še številni pritajeni procesi ali *demoni*. Ti napredujejo nevidno *v ozadju*. Po večini delujejo kot *strežniki*, ki strežejo odjemalcem. Tak pritajeni proces je tudi `sshd` (Angl. Secure shell Daemon), ki uporabnikom omogoča daljinski dostop. In potem so tu še uporabniški procesi, ki opravljajo 'resnično' delo.

Seznam najbolj aktivnih procesov dobimo z ukazom `top`, seznam izbranih procesov pa z ukazom `ps` (Angl. Processes). Na primer,

```
ps -l -u stanek
```

izpiše seznam procesov uporabnika `stanek` v razširjeni obliki (zato -l kot long). Za vsak proces se na zaslon izpiše njegova številka, stanje procesa, prioriteta, velikost, čas, in še kaj.

2.5.6 Datoteke

UNIXa se je prijelo reklo, da kar ni proces, je datoteka. UNIX in tudi Linux obravnavata vse vhodne in izhodne prenose kot datoteke. Na tiskalnik tiskamo, kot bi pisali v datoteko. Iz tipkovnice beremo, kot bi brali iz datoteke. Za Linux je tudi kamera datoteka in datoteka je lahko senzor temperature. Celo disk je lahko obravnavan kot ena sama datoteka ali pa kot zbirka oziroma sistem datotek.

Vse datoteke so urejene v hierarhično drevesno strukturo, ki začne na vrhu z enim samim direktorijem (/) in se nadaljuje v globino. Večina datotek je regularnih oziroma *navadnega* tipa. Te vsebujejo podatke. Druge datoteke, ki jim rečemo *posebne datoteke*, pomagajo priti do podatkov. Direktoriji so podzvrst posebnih datotek, so pa še druge vrste posebnih datotek. Med temi so najbolj pomembne posebne datoteke naprav (Angl. Device special file). Seznam datotek v izbranem direktoriju izpišemo z ukazom `ls`.

Vsakemu uporabniku je dodeljen njegov direktorij, ki je za njega domač direktorij. Na primer, direktorij `/home/stanek` je domači direktorij uporabnika `stanek`. Kratek nadomestek za sklic na domači direktorij je ~.

Ko se uporabnik prijavi v sistem, se njegov domači direktorij obravnava kot delovni ali *tekoči* direktorij. To je tisti direktorij, na katerega se nanašajo vse operacije z datotekami, če ni eksplicitno izraženo drugače. Torej,

```
ls -l
```

izpiše vsebino tekočega direktorija, in opcija -l (kot long) zahteva izpis datodatnih določil datotek.

2.5.7 Direktorij /proc

V Linuxu je vse datoteka in kot rečeno, kar ni datoteka, je proces. Linux, kot nekateri drugi sistemi UNIX, preslikajo notranje strukture jedra v drevesno strukturo datotečnega sistema. Preko *datotečnega sistema*, ki je nameščen v direktorij `/proc`, je moč dostopati do notranjih struktur jedra, kot bi bile datoteke. Večina teh datotek je berljiva z ukazom `more`. Tu najdemo podatke o strojni opremi, o konfiguraciji sistema, o pomnilniku, o procesih, in še dosti več. Datoteke v direktoriju `/proc` niso prave datoteke in jih zastonj iščemo kjerkoli na disku tedaj, ko sistem ne deluje.

2.5.8 Datotečni sistem

virtualni datotecni sistem, tudji datotecni sistemi, ...

2.5.9 Terminalne naprave

kozola, kontrolni terminal, psevdo terminal.

2.5.10 Posli ter seje

2.5.11 Nekatere naloge skrbnika sistema

2.5.12 Linux in vgradni sistemi

2.6 Nadaljnje čtivo

Veliko koristnega o Linuxu je na spletnih straneh s prostim dostopom [1]. Tu sicer ne najdemo vedno najnovejših informacij o Linuxu, a viri so še aktualni. Linux se hitro razvija, in to tako hitro, da publikacije, ki gredo v podrobnosti jedra, kmalu zastarajo.

Za začetne uprabnike priporočamo [2]. Tiste, ki zanima administriranje sistema, naj sežejo po [4].

Koncepte operacijskih sistemov obravnavajo knjige [5, 6, 7] in druge knjige. To so dobri učbeniki za študij teorije operacijskih sistemov.

Med knjigami o načrtovanju in implementaciji operacijskih sistemov ne smemo spregledati UNIX klasike [8] in knjige o MINUXu [9]. Notranjost Linuxa, čeprav ne najnovejših verzij, najbolj celovito obravnavata knjigi [10] in [11].

O Linuxu in vgradnih sistemih pišejo na primer [12, 13] in drugi avtorji.

Literatura

[1] `http://tldp.org`

[2] Machtelt Garrels, *Introduction to Linux, A Hands on Guide*, CreateSpace Independent Publishing Platform, 2007. Prost dostop na `http://tille.garrels.be/training/tldp/`.

[3] Ellen Siever, Jessica P. Hackman, Stephen Spainhour, Stephen Figgins, *Linux in a Nutshell*, O'Reilly, 2009.

[4] Lars Wirzenius, Joanna Oja, Stephen Stafford, Alex Weeks, The Linux System Administrator's Guide, 2004.

[5] A. Silberschatz, P.B. Galvin, G. Gagne, *Operating System Concepts, 9th Ed.*, John Wiley & Sons, 2012.

[6] A. S. Tanenbaum, *Modern Operating Systems, 3rd Ed.*, Pearson Education, Prentice Hall, 2009.

[7] D. M. Dhamdhere, *Operating Systems, A Concept-Based Approach, 2-nd Ed.*, McGraw-Hill, 2006.

[8] M. J. Bach, *The Design of the UNIX Operating System*, Prentice Hall, 1986.

[9] A. S. Tanenbaum, A. S. Woodhull, *Operating Systems, The MINIX Book, 3rd Ed.*, Pearson Education, Prentice Hall, 2009.

[10] R. Love, *Linux Kernel Development, 3-rd ed.*, Pearson, Addison-Wesley, 2010.

[11] D. P. Bovet, M. Cesati, *Understanding the Linux Kernel, 3-rd ed.*, O'Reilly, 2006.

[12] D. Abbott, *Linux for Emmbedded and Real Time Applications, 2nd Ed.*, Elsevier, 2006.

[13] C. Hallinan, *Embedded Linux Primer*, Pearson, Prentice Hall, 2007.

Poglavje 3

Datotečni sistem

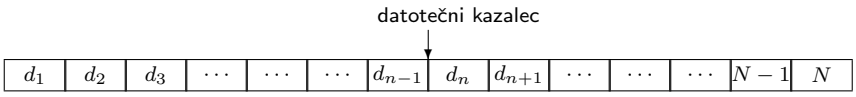
V tem poglavju si bomo ogledali osnovne značilnosti datotečnih sistemov. Na primeru Linux bomo razložili koncept datoteke, sistema direktorijev in posebnih datotek. Dotaknili se bomo tudi notranje stukture datotečnega sistema.

Namen tega poglavja je, da bi bralca opozorili na nekatere značilnosti datotečnih sistemov, vendar na čisto konceptualnem nivoju in na nepogobljen način.

3.1 Datoteka

Računalnik je stroj za obdelavo podatkov. Organizaciji podatkov, za katero skrbi operacijski sistem, rečemo *datotečni sistem* (Angl. File System). Datotečni sistem je najvidnejši del operacijskega sistema.

Osnovni objekt datotečnega sistema je *datoteka*. Datoteka je podatkovna struktura *sekvenčnega značaja*. Sekvenčnost datoteke ponazarja slika 3.1. Podatki v datoteki si sledijo drug za drugim, kot bi bili zabeleženi na traku. Navsezadnje tudi knjigo beremo sekvenčno. Razen prvega in zadnjega podatka v datoteki ima vsak podatek svojega predhodnika in svojega naslednika.



Slika 3.1: *Koncept sekvenčne datoteke. Datoteka vsebuje N podatkov. Da pridemo do n-tega podatka, moramo prej prebrati $n-1$ podatkov pred njim. Kje beremo ali pišemo beleži datotečni kazalec.*

Osnovni operaciji nad datoteko sta branje in pisanje. Predno lahko datoteko beremo ali pišemo, jo je potrebno ustvariti. Tedaj, ko datoteko ustvarimo, še nima podatkov in je *prazna*. Z vsako operacijo pisanja dodamo nove podatke v datoteko. Podatke dodajamo sekvenčno, drugega za drugim. Datoteka se s tem veča.

Tudi branje podatkov iz datoteke začne na začetku datoteke. Z branjem podatkov se pomikamo vzdolž datoteke. Denimo, če potrebujemo enajsti podatek, moramo pred tem prebrati deset podatkov pred njim, pa če jih rabimo ali ne. Prav ta lastnost dela datoteko sekvenčno. Z branjem se vsebina datoteke ne spremeni.

Položaj vzdolž datoteke, kjer beremo ali pišemo, beleži datotečni kazalec. Tej vrednosti datotečnega kazalca rečemo *datotečni odmik*. V datotečnih sistemih, kot sta na primer UNIX in Linux, je moč datotečni odmik neposredno spreminjati, ne da bi podatke brali ali pisali. Lahko se pomaknemo na začetek datoteke, na konec, ali kamorkoli drugam. Na ta način naredimo vsebino datoteke direktno dostopno. Denimo, najprej nastavimo vrednost odmika in nato beremo podatke. Čas za dostop do podatkov sicer ni enak za vse podatke v datoteki. Odvisen je od njihovega položaja v datoteki ter od zaporedja dostopov do datečne vsebine. Zato rečemo, da je dostop *direkten*, a ne tudi naključen. Da bi bil dostop naključen, bi moral biti dostopni čas vedno enak za vse podatke.

Nekateri sistemi omogočajo, da se vsebino datoteke preslika v naslovno področje programa. To omogoča tudi Linux. S tem postane vsebina datoteke naključno dostopna. A to nima ničesar s pojmovanjem datoteke ali pač

datotečnim sistemom.

Operacijski sistemi tipa UNIX/Linux vsebini podatkovne datoteke ne dajejo nobene pomenske strukture. Struktura in pomen vsebine datoteke sta stvar uporabe. A za operacijski sistem je vsebina datoteke golo zaporedje bajtov.

Datoteka ima poleg vsebine še datotečno ime ter dodatna določila ali atribute. Ime pomaga pri dostopu do datoteke. Določila pojasnjujejo kdo in kaj sme z datoteko početi, kdaj je datoteka nastala, kdaj je bila nazadnje spremenjena, kakšen je njen tip in podobno.

Niso vse datoteke podatkovne datoteke. Nekatere namreč sploh ne vsebujejo podatkov, ampak so v pomoč pri dostopu do podatkov. Spet druge datoteke sicer imajo vsebino, a je njihova vsebina internega značaja in zgolj pomaga pri delu z drugimi datotekami in z datotečnim sistemom. Vsem tem datotekam rečemo *posebne datoteke*. Med posebne datoteke spadajo tudi direktoriji. Direktorij je datoteka, katere vsebina so imena datotek. Datoteke, ki vsebujejo koristne podatke, so *navadne datoteke*, poimenovane tudi *regularne datoteke*.

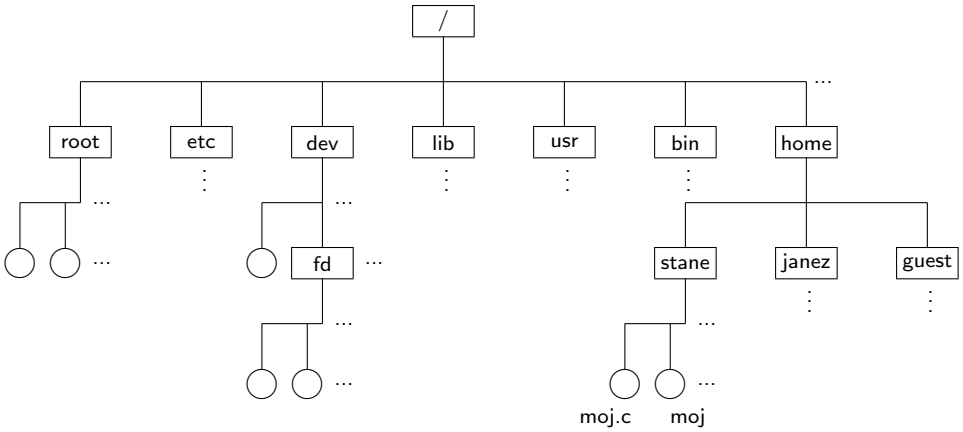
3.2 Direktoriji in hierarhični sistem datotek

Skupna lastnost sodobnih datotečnih sistemov je hierarhična urejenost datotek. Slika 3.2 prikazuje koncept datotečne hierarhije. Hierarhično urejenost datotečnega sistema realizira sistem direktorijev, imenovanih tudi imeniki, kazala ali mape (Angl. Directory, Folder). V sistemih UNIX in Linux začne hierarhija datotečnega sistema z enim samim direktorijem (Angl. Root directory). Ime glavnega direktorija je poševana črta (/). Glavni direktorij vsebuje imena datotek. Te datoteke so lahko navadne, lahko pa so direktoriji. To so direktoriji naslednjega nivoja. Ti direktoriji lahko spet vsebujejo imena direktorijev in navadnih datotek ter tako naprej. Na ta način se ustvari drevesna zgradba datotečnega sistema z enim samim korenom, ki je glavni direktorij.

S stališča datotečnega sistema in direktorijske strukture je čisto vseeno kje so dejansko shranjene datoteke. Datoteke so lahko hranjene na enem diskovnem pogonu, na več diskih, na USB 'ključkih', ali na diskovju, ki je dostopno prek omrežja. Tako napravo umestimo v drevesno direktorijsko strukturo z ukazom `mount`. S tem ukazom napravo pripnemo na primeren direktorij. Na primer

```
mount /dev/sda3 /home
```

pripne napravo `/dev/sda3` na direktorij `/home`. Od tedaj naprej je struktura datotečnega sistema na napravi dosegljiva kot vsebina direktorija, kamor je bila nameščena oziroma *montirana*.



Slika 3.2: *Hierarhična urejenost datotečnega sistema. Končna vozlišča drevesne strukture so (načeloma) datoteke, vmesna so direktoriji, ki so sicer tudi datoteke, vendar posebne vrste.*

Ne smemo pozabiti, da so tudi direktoriji datoteke, a za razliko od navadnih podatkovnih datotek, vsebujejo direktoriji imena datotek. Z vsebino direktorijev upravlja operacijski sistem. Njihovo strukturo in pomen torej določa operacijski sistem.

Datoteko v drevesni strukturi datotečnega sistema enoznačno določa *pot* (Angl. Path). Pot sestavlja zaporedje imen direktorijev ločenih s poševno črto (/). Sama poševnica ni del imena. Pot zaključi z imenom datoteke, ki je zabeleženo v zadnjem direktoriju poti.

Pot je bodisi absolutna bodisi relativna. Absolutna pot začne s poševno črto, torej v glavnem direktoriju. Poševna črta (/) je namreč ime glavnega direktorija. Primer take poti je:

```
/usr/include/stdio.h
```

Relativna pot ne začne s poševno črto, kar pa hkrati pomeni, da pot začenja v *tekočem* direktoriju. Ime tekočega direktorija dobite z ukazom `pwd` (Angl. print working directory), tekoči direktorij pa spremenite z `cd pot`. Primer relativne poti je:

```
include/stdio.h
```

Če je pot relativna, se pred njo pripne absolutna pot do tekočega direktorija in tako nastane polna pot.

Koncept tekočega direktorija je v bistvu vezan na program. V tem primeru na školjko. Školjka namreč beleži absolutno pot do izbranega direktorija v interni spremenljivki. Temu direktoriju potem rečemo, da je tekoči. Z ukazom `cd` pravzaprav vplivamo na to spremenljivko. Vedno, kadar bi bilo potrebno kje v ukazu navesti pot, pa navedbo opustimo, se upošteva pot do tekočega direktorija oziroma kot rečemo, tekoči direktorij.

Po dogovoru je ime direktorija zabeleženo v vsebini tega istega direktorija s piko (.). Dve piki (..) pa sta ime za direktorij nad njim. Denimo, če bi bil tekoči direktorij `stane` (Slika 3.2), bi ukaz `pwd` vrnil `/home/stane`. Ukaz za izpis vsebine direktorija je `ls pot`. Na primer,

```
ls .
```

izpiše vsebino tekočega direktorija, a ime lahko v tem primeru opustimo, ker ga privzame že sam ukaz `ls`. Nadalje,

```
ls ..
```

izpiše vsebino direktorija nad tekočim, ta je v našem primeru `home` in izpis bi izgledal takole

```
stane janez guest
```

Izpis je enak, kot bi bil v primeru ukaza:

```
ls /home
```

Imena datotek, ki začnejo s piko, so praviloma 'skrita' in se ne izpisujejo. Ukaz

```
ls -a pot
```

izpiše vsa, tudi skrita imena datotek in bi za naš primer `ls -a ..` verjetno izgledal takole:

```
.  .. stane janez guest
```

Pika (.) je nadomestno ime za `home`, dve piki (..) za direktorij nad njim, ki je v tem primeru kar glavni direktorij (/).

3.3 Notranja struktura datotečnega sistema

Čeprav se je notranja struktura datotečnega sistema z razvojem spreminjala, obstajajo pa tudi precejšnje razlike med datotečnimi sistemi različnih operacijskih sistemov, so datotečni sistemi vendarle zasnovani na skupnih načelih. Zelo načelno strukturo diska oziroma dela diska z datotečnim sistemom prikazuje slika 3.3. Delu diska rečemo *particija*. Particij je na disku običajno več, a spet ne tako zelo veliko. Na dejansko število particij največ vpliva velikost diska. Vsaka particija lahko vsebuje datotečni sistem, ki ga je moč vpeti v skupno drevesno strukturo datotečnega sistema. Ena od particij je glavna particija. Ta je na vrhu drevesne strukture.

Osnovnemu zapisu podatkov na disku rečemo sektor. Sektor je najmanjša podatkovna enota, ki jo je moč brati ali pisati. Velikost sektorja je odvisna od krmilnika diskovne naprave in naprave same ter povečini znaša 512 bajtov. S tehnološkim razvojem se kapaciteta pomnilniških naprav povečuje in s tem tendenca za večanje sektorja.

Datotekam se disk dodeljuje v blokih. Datoteka, tudi če vsebuje le en sam bajt, zavzame najmanj en blok prostora na disku. *Blok* je osnovna diskovna podatkovna enota operacijskega sistema. Diskovni blok obsega enega ali več sektorjev. Tipična velikost bloka je 4K ali 8K bajtov. V literaturi uporaba

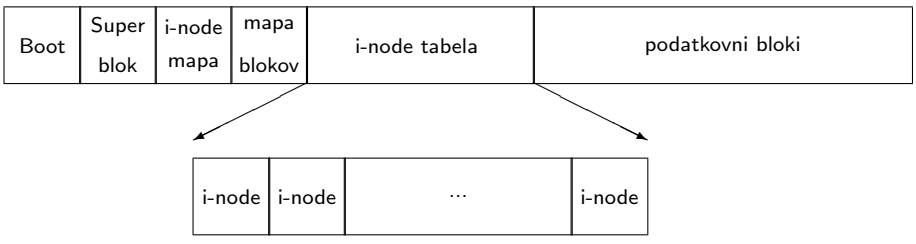
izrazov sektor in blok sicer ni prav dosledna, ali pa se na sektor preprosto pozabi.

Hierarhični sistem direktorijev je zgolj podlaga za preglednejšo organizacijo datotečnega sistema, ki lahko vsebuje več desettisoč datotek. Datoteke pa obstajajo in so dosegljive tudi brez direktorijev.

Datoteko v datotečnem sistemu enoznačno opredeljuje *indeksno vozlišče* (Angl. i-node). Indeksna vozlišča so vsa enake strukture in velikosti ter se nahajajo nekje na začetku diska, tako da tvorijo tabelo indeksnih vozlišč. Vsako indeksno vozlišče ima unikatno številko, ki je indeks v tabelo vozlišč. Če poznamo številko indeksnega vozlišča, že lahko preko indeksnega vozlišča dostopamo do vsebine datoteke.

`I-node` vsebuje določila datoteke, med drugim: tip datoteke, določilo lastnika (UID), določilo skupine iz katere je lastnik (GID), dovoljenja za branje, pisanje, izvrševanje lastnika, skupine ter vseh ostalih, število *povezav*, torej imen, čas zadnje spremembe in druga določila in nenazadje kazalce na podatkovne bloke na disku.

Slika 3.3 prikazuje notranjo strukturo datotečnega sistema. Prvi diskovni blok je blok z začetnim programom (Angl. Boot block). Naslednji blok je *super blok*. Ta vsebuje bistvene podatke datotečnega sistema. Za super blokom je bitni načrt indeksnih vozlišč. (Angl. i-node map), ki označuje katere indeksna vozlišča so dodeljena in katera so prosta. Potem pride bitni načrt podatkovnih blokov (Angl. Block bitmap), ki označuje zasedenost blokov na disku. Sledi tabela indeksnih vozlišč in nato podatkovni bloki datotek.



Slika 3.3: *Načelna shema organizacije diskovne particije.*

3.4 Notranja struktura direktorija

Pomen direktorijev smo že spoznali. Struktura in pomen vsebine direktorija sta sicer odvisna od sistema, a koncept je vedno isti. Direktorij je posebna datoteka, ki vsebuje imena datotek in vsakemu imenu je v direktoriju pridružen kazalec, ki kaže na vsebino datoteke. Če poznamo ime datoteke, lahko prek direktorija pridobimo kazalec na vsebino datoteke. V nadaljevanju se bomo osredotočili na strukturo direktorijev, ki jo poznamo iz sistemov UNIX.

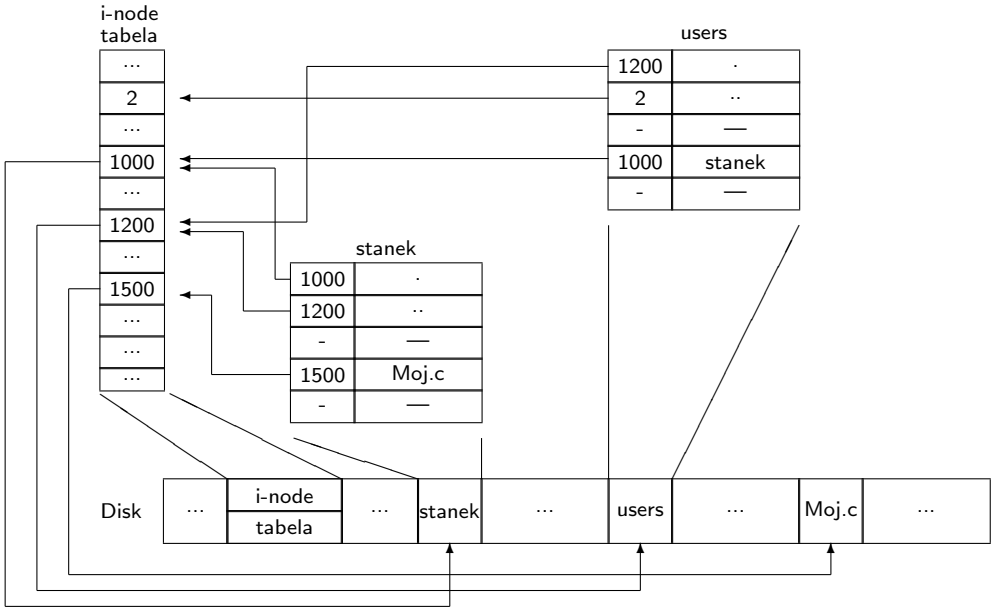
Kot rečeno lahko do vsebine datoteke pridemo preko indeksnega vozlišča. Zadostuje že, da poznamo njegovo številko. A lažje kot številke indeksnih vozlišč si zapomnimo imena datotek. Imena datotek se beležijo v direktorijih. V direktoriju je skupaj z imenom datoteke zapisana številka indeksnega vozlišča. Če poznamo ime datoteke, ali bolje rečeno, pot do nje, lahko iz direktorija pridobimo pripadajočo številko indeksnega vozlišča. Številka indeksnega vozlišča je indeks v tabelo indeksnih vozlišč, preko katerega je moč priti do vsebine datoteke. Datoteka ima lahko tudi več kot eno ime. Vsakemu paru:

`<številka indeksnega vozlišča> <ime datoteke>`

rečemo *povezava* (Angl. Link). Rečemo, da ima datoteka več povezav. V primeru dvoumnosti poudarimo, da ima datoteka več *trdih povezav* (Angl. Hard links). Datoteko ustvarimo, ji damo ime in s tem eno povezavo. Povezave dodajamo ali odvzemamo. Ko odvzamemo še zadnjo povezavo, datoteko zbrišemo. Tedaj dobi polje v direktoriju za številko indeksnega vozlišča vrednost nič, `ii-node` se sprosti, in sprostijo se podatkovni bloki.

Spomnimo se, da sta po dogovoru pika (.) in dve piki (..) imeni direktorijev. Pika je ime direktorija samega in dve piki (..) je ime za direktorij nad njim. Obe imeni sta zabeleženi v direktoriju. Bolj dosledno povedano, v direktoriju sta zabeleženi povezavi. Piki pripada številka `i-node` direktorija samega, dvema pikama pripada številka `i-node` nad njim. To omogoča elegantno pregledovanje drevesne strukture direktorijev.

Načelno shemo prikazuje slika 3.4. Iz slike ugotovimo, da je direktorij z imenom `stanek`, ki mu pripada `i-node` s številko 1000, zabeležen v direktoriju z imenom `users`, ki mu pripada `i-node` s številko 1200. Direktorij `users` je očitno zabeležen v direktoriju z `i-node` s številko 2. Vsebina tega direktorija ni skicirana. V direktoriju `stanek` je zabeležena navadna datoteka z imenom `Moj.c`. Njena vsebina je poljubna in na sliki ni ponazorjena.



Slika 3.4: *Načelna shema organizacije direktorijev.*

3.5 Posebne datoteke naprav

Kot smo ugotovili, ima vsaka datoteka svoj tip. Datoteka je lahko navadnega tipa. Tedaj vsebuje podatke. Druge datoteke so posebnega tipa. Eno vrsto posebnih datotek smo že spoznali. To so direktoriji. Med posebne datoteke pa prištevamo še datoteke naprav, cevi z imenom, komunikacijske vtičnice ter simbolične povezave. V tem razdelku si bomo ogledali posebne datoteke naprav (Angl. Device special file).

Zelo pomembna lastnost sistemov UNIX in Linux je enotno obravnavanje

vseh vhodnih in izhodnih naprav kot da bi bile datoteke. Denimo, branje zvočne kartice se v ničemer ne razlikuje od branja datoteke. Zvočna kartica je za uporabnika videti kot datoteka. Ali, izpis na tiskalnik je ekvivalenten pisanju v datoteko. Da je to mogoče, poskrbijo *posebne datoteke naprav*.

Posebna datoteka naprave nima vsebine, ima pa ime in določila. Torej ima tudi indeksno vozlišče. Namesto na vsebino datoteke, ki je ni, pa `i-node` kaže na gonilnik v sistemskem jedru, ki streže napravi. Po dogovoru se datoteke naprav nahajajo v direktoriju `/dev`[1]. Tudi shema za poimenovanje posebnih datotek naprav je priporočena. Na primer

```
/dev/rtc
```

je ime posebne datoteke naprave za uro realnega časa. Ime s stališča jedra in funkcionalnosti niti ni bistveno, seveda dokler je znano aplikacijam, ki jo uporabljajo.

Obstajata dva tipa posebnih datotek naprav, blokovne in znakovne.

- Posebna datoteka blokovne naprave (Angl. Block special file) ima med določili oznako 'b'. V tem primeru potekajo operacije z napravo (prenos podatkov) preko izravnalnika (Angl. Buffer) v obliki blokov predpisane velikosti (na primer 4KB). Tak tip naprave je primeren tedaj, kadar naprava omogoča direkten dostop. Disk je že taka naprava.

- Posebna datoteka znakovne naprave (Angl. Character special file) ima oznako 'c'. V tem primeru poteka prenos sekvenčno znak po znak brez izravnave v okviru sistemskega jedra. Tipkovnica terminala je tipična takšna naprava.

Posebno datoteko naprave ustvarimo z ukazom `mknod` (make node). Pri tem navedemo še veliko in malo številko gonilnika (Angl. major, minor number) ter tip, b ali c. Na primer,

```
mknod /dev/sd4 b 8 4
```

ustvari posebno datoteko blokovne naprave z imenom `/dev/sd4`, velika številka

[1] Pravzaprav bi morali reči imena datotek se nahajajo v direktoriju. A ker datoteke enačimo z imeni, je tak način sprejemljiv.

je 8 in mala je 4. Pomen številk je odvisen od implementacije, a je tako ali drugače povezan z gonilnikom in napravo. Običajno glavna številka določa gonilnik in mala številka določi eno od naprav, ki se jim streže prek gonilnika.

3.6 Datoteke FIFO in Socket

Naslednja posebna datoteka je datoteka tipa FIFO (First In First Out). Konceptualno je to datoteka z dvema koncema. Na enem koncu pišemo in na drugem beremo. Kar je prej zapisano bo prej prebrano. Datoteke FIFO niso namenjene za shranjevanje podatkov, temveč služijo za komunikacijo med procesi. Zato jim rečemo tudi *poimenovane cevi* (Angl. Named pipe). Datoteka FIFO nima vsebine, ima pa `i-node`, ki določa izravnalni medpomnilnik v okviru jedra, prek katerega teče komunikacija.

Tudi datoteka tipa Socket(komunikacijska vtičnica) predstavlja eno od možnih oblik komuniciranja v okviru iste naprave, ki je konceptualno identična s komukacijo med različnimi napravami v omrežju. Ta posebna datoteka je torej podobna datoteki FIFO, le da omogoča pretok podatkov v obeh smereh.

3.7 Simbolična povezava

Zadnji med obravnavanimi tipi posebnih datotek je *simbolična povezava* (Angl. Symbolic Link). Simbolična povezava je na videz sorodna trdi povezavi, od tu tudi ime. Sicer pa je simbolična povezava datoteka, ki kaže na datoteko s podatki ali direktorij. Simbolična povezava je torej datoteka, ki ima svoje ime, indeksni blok in vsebino, a njena vsebina je ime datoteke ali direktorija. Preko simbolične povezave je torej moč dostopati do druge datoteke z nadomestnim imenom, kar pride pogosto zelo prav. Če odstranimo (brišemo) simbolično povezavo, dejanska datoteka še vedno obstaja. Če pa brišemo dejansko datoteko s podatki, se njena vsebina seveda izgubi, a simbolična povezava ostane, le da kaže na datoteko, ki več

ne obstaja.

3.8 Delo z datotekami

Datoteke ustvarjamo, izpisujemo, izvršujemo, prepisujemo, preimenujemo, brišemo, datotekam spreminjamo določila, ipd. Vsebino tekstovne datoteke z imenom `moja.c` izpišemo na zaslon z:

```
more moja.c
```

Pomembno je vedeti, da je pika v imenu enakovredni sestavni del imena, isto velja za črko 'c', ki sledi piki. Vendar bi v takem primeru pričakovali, da je `moja.c` datoteka s C-jevskim programom. Vsebino datoteke bi ekvivalentno lahko izpisali tudi s `cat`:

```
cat moja.c
```

Datoteko `moja.c` bi lahko prepisali in dobili še eno datoteko z enako vsebino in različnim imenom:

```
cp moja.c Moja.c
```

Med majhnimi in velikimi črkami je razlika. Z ukazom `mv` (move) datoteko preimenujemo in z `rm` (remove) jo odstranimo:

```
rm moja.c
mv Moja.c moja.c
```

Nov direktorij ustvarimo z:

```
mkdir vaja1
```

Nekatere lastnosti oziroma določila datoteke dobimo z ukazom ls -l ime. Denimo:

```
ls -al /home/stane
```

bi lahko izgledal takole:

```
drwxr-xr-x  2 4096 .
```

```
drwxr-xr-x 12 4096 ..
-rw-r--r--  1  385 moja.c
-rwxr-xr-x  1 8112 moja
```

Prva črka v vrstici je `tip` datoteke: d: direktorij, -: navadna, c: znakovna naprava, b: blokovna naprava, s: socket, l: simbolična povezava. Torej sta pika (.) in dve piki (..) imeni direktorijev, kot smo pričakovali. Sledijo tri skupine: lastnik, skupina, ostali s po tremi zastavicami ali biti dovoljenj za pisanje (w), branje (r) in izvršitev (x). Torej

```
-rw-r--r--  1  385 moja.c
```

je navadna datoteka, ki jo lahko lastnik bere in spreminja, medtem ko jo vsi ostali lahko le berejo. Nadalje sledi število povezav. Število povezav je za večino datotek enako ena. Dodatno povezavo dodamo z ukazom `ln`,

```
ln moja.c tvoja.c
```

Tako dobimo eno datoteko z dvema trdima povezavama oziroma imeni,

```
-rw-r--r--  2  385 moja.c
-rw-r--r--  2  385 tvoja.c
```

Bite dovoljenj spremenimo s `chmod`, lastnika s `chown`. Mogoče najlažje razmišljamo o bitih dovoljenj kot o osmiški konstanti (3x3 biti). Na primer

```
chmod 0644 moja.c
```

ustreza stanju bitov v gornjem primeru.

Poglavje 4

Upravljanje pomnilnika

V tem poglavju bomo obravnavali upravljanje pomnilnika. Spoznali bomo razliko med logičnim in fizičnim naslovnim prostorom. Razložili bomo koncept navideznega pomnilnika. Ogledali si bomo izvedbo navideznega pomnilnika z ostranjenjem in segmentiranjem. Nazadnje si bomo ogledali ostranjenje pomnilnika v sistemu Linux.

4.1 Programi in pomnilnik

Upravljanje pomnilnika (Angl. Memory management) je ena temeljnih nalog operacijskega sistema. Upravljanje pomnilnika terja odgovore na vprašanja kje in v kakšnem obsegu, pa tudi kdaj in za koliko časa naj proces dobi pomnilnik. Ko se proces izvaja, procesna enota izvršuje njegove ukaze, ukaz za ukazom. Da je to mogoče, morajo biti ukazi v pomnilniku. Procesor za izvršitev ukaza najprej prebere ukaz iz pomnilnika. Času, ki je potreben za to, rečemo ukazno prevzemni cikel (Angl. Instruction fetch). Ko je ukaz prevzet v ukazni register procesorja, sledi dekodiranje ukaza ter njegova izvršitev. Času, ki je potreben za izvršitev ukaza, rečemo izvršilni cikel, celotnemu času za prevzem in izvršitev ukaza pa ukazni cikel. Izvršitev ukaza v splošnem zahteva prevzem operandov, to je branje operandov iz registrov ali pomnilnika in ko je operacija, ki jo zahteva ukaz, izvršena,

sledi shranjevanje rezultata v registre ali pomnilnik. Skratka, izvrševanje ukazov in napredovanje procesa je mogoče samo, če je proces nameščen v pomnilnik.

V manj zahtevnih vgradnih sistemih zadošča, če je programska oprema realizirana v okviru enega samega procesa oziroma opravila. To opravilo, denimo, periodično ali preko zahtev za prekinitev (Angl. Interrupt request) zajema signale iz senzorjev, obdeluje signale in na podlagi obdelave vhodnih signalov postavlja stanja izhodnih signalov. V enoopravilnih sistemih je *dodeljevanje pomnilnika* dokaj preprosto in mu spričo tega ne posvečamo posebne pozornosti. Pomnilnik je v celoti dodeljen enemu opravilu. Vgradni sistem z enim samim opravilom ne potrebuje podpore operacijskega sistema.

V večopravilnem sistemu pa je v pomnilnik sočasno nameščenih več procesov. Vsak proces razpolaga z delom pomnilnika oziroma delom *naslovnega prostora* pomnilnika. Tem procesom razvrščevalnik procesov po določenem pravilu dodeljuje procesno enoto. Procesi napredujejo asinhrono, načeloma neodvisno oden od drugega in medtem naslavljajo pomnilnik. Naslovni prostori procesov so med sabo ločeni. Pomnilniška referenca izven naslovnega prostora procesa bi lahko ogrozila pravilnost napredovanja drugega procesa. Nameren ali nenameren poseg izven naslovnega prostora procesa je zato potrebno preprečiti.

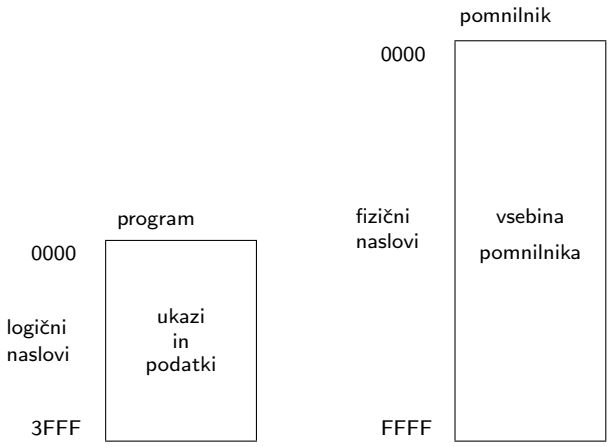
Dodeljevanje ter zaščita pomnilnika spadata v širši kontekst upravljanja pomnilnika. Za učinkovito izvedbo dodeljevanja in zaščite pomnilnika sta potrebni tako primerna strojna kot programska oprema. Večina sodobnih mikrokrmilniških sistemov ima vsaj osnovno strojno opremo, ki daje podlago za realizacijo zaščite pomnilnika, zmogljivejši mikrokrmilniki pa omogočajo realizacijo navideznega pomnilnika, bodisi z ostranjenjem (Angl. Paging) bodisi s segmentiranjem (Angl. Segmentation) ali s kombinacijo obeh. Tudi o tem bomo spregovorili v naslednjih poglavjih, a najprej si oglejmo razliko med logičnimi in fizičnimi naslovi.

4.2 Naslovni prostor

Za razumevanje mehanizmov za upravljanje pomnilnika v večopravilnih sistemih je bistvenega pomena, da razumemo razliko med logičnim in fizičnim naslovom, slika 4.1. *Fizični naslov* je naslov pomnilniške besede glavnega pomnilnika, to je pomnilniške besede tistega pomnilnika, do katerega ima procesna enota naključni dostop. Pomnilniška beseda ima naslov in vsebino. Fizični naslov je naslovna kombinacija signalov na naslovnem vodilu pri dostopu do njene vsebine. Do vsebine pomnilniške besede dostopa procesor z operacijo beri ali piši. Naslov dane pomnilniške besede je vedno enak in je določen z izvedbo računalnika, medtem ko se njena vsebina bere z operacijo *beri* in po potrebi spremeni z operacijo *piši*. Procesor postavi naslov pomnilniške besede na naslovno vodilo, postavi kontrolni signal beri ali piši in opravi prenos vsebine pomnilniške besede po podatkovnem vodilu. Rečemo tudi, da je pomnilniška beseda osnovna naslovljiva enota pomnilnika. Ker se naslovi pomnilniških besed interpretirajo kot cela števila, obravnavamo fizični pomnilnik kot linearno zaporedje pomnilniških besed. Velikost (obseg) fizičnega naslovnega področja je dana s *širino* naslovnega vodila oziroma s številom naslovnih signalov in posedično s številom bitov za naslov.

Za razliko od fizičnih naslovov, ki pripadajo pomnilniku, pa logični naslovi pripadajo programu oziroma procesu. *Logični naslov* je naslov znotraj programa, to je naslov ukaza ali podatka. Množica vseh logičnih naslovov procesa sestavlja logično naslovno področje. Logičnemu naslovnemu področju rečemo tudi logični naslovni prostor procesa. Velikost logičnega naslovnega prostora je dana z velikostjo procesa. Logični naslov je podlaga pomnilniški referenci, ki jo generira procesna enota med izvrševanjem procesa. Na primer, logični naslov določa vsebina programskega števca v ukazno prevzemni fazi ukaza, ali vsebina kakšnega od naslovnih registrov v fazi prevzema operandov ali shranjevanja rezultata ter posledično izvršitve ukaza.

V enostavnih sistemih brez upravljanja pomnilnika sta logični in fizični naslov po vrednosti enaka, zato razlikovanje med njima niti ni tako pomembno



Slika 4.1: *Logični naslovi sestavljajo naslovni prostor programa. To so naslovi ukazov in podatkov. Fizični naslovi sestavljajo naslovni prostor pomnilnika. To so naslovi pomnilniških besed.*

in bi bilo mogoče celo moteče. A da bi spoznali bistvo razlikovanja logičnih od fizičnih naslovov, si najprej poglejmo prav take sisteme.

Denimo, da načrtovalec načrtuje programsko opremo vgradnega sistema, za delovanje katerega bo zadoščal en sam program. Načrtovalec programske opreme dododbra pozna arhitekturo sistema. Pozna tudi organizacijo pomnilnika. Pri načrtovanju programske opreme upošteva kje je pomnilnik tipa RAM, kje je pomnilnik tipa Flash ter kje so registri vhodnih in izhodnih naprav. Iz poznavanja arhitekture tudi ve, kako je določen naslov prvega ukaza programa, torej ukaza, s katerim naj se začne izvrševati program. Vse to upošteva že ob načrtovanju izvorne kode: nameni prostor za ukaze v pomnilniku Flash, predvidi prostor za podatke in sklad v pomnilniku RAM, upošteva naslove registrov vmesnikov perifernih naprav. Ko prevajalnik program prevede, je preslikava med logičnimi in fizičnimi naslovi določena. Program je pripravljen, da se namesti v pomnilnik na predvidene naslove in izvrši.

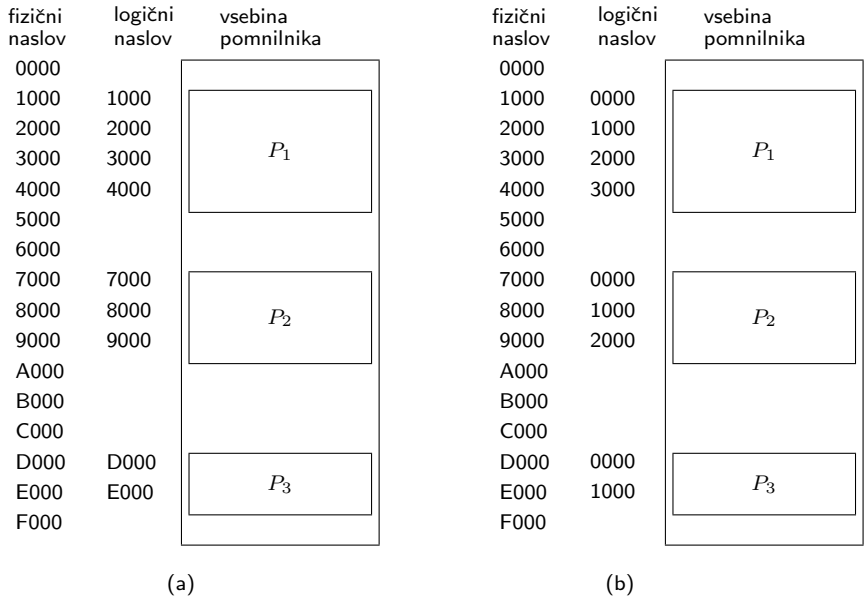
Alternativna možnost bi bila, da bi prirejanje fizičnih naslovov logičnim naslovom prepustili fazi povezovanja. Taka možnost je koristna tedaj, kadar je program sestavljen iz več različnih programskih modulov izvorne kode. Programski moduli bi bili lahko v splošnem napisani v različnih program-

skih jezikih. Programski modul se vsak zase najprej prevede v strojni jezik, nakar se prevedeni moduli povežejo skupaj v izvršljiv program. V fazi povezovanja se logičnim naslovom priredi fizične naslove pomnilnika. Program se nato namesti v pomnilnik na predvidene naslove in izvrši.

Nadaljnja možnost bi bila, da bi prirejanje fizičnih naslovov logičnim zadržali do faze nameščanja programa v pomnilnik. Ob nameščanju programa v pomnilnik bi logične naslove 'preračunali' v fizične naslove, program namestili v pomnilnik in nato izvršili. Tak način prirejanja naslovov je koristen tedaj, kadar programu izbiramo prostor za namestitev v pomnilnik tik pred izvršitvijo.

V vseh navedenih primerih so logični naslovi v času izvrševanja po vrednosti že enaki fizičnim, ne glede na to kdaj smo naredili prirejanje. V enoopravilnem sistemu je s tem naloga zaključena. V večopravilnem sistemu pa je v pomnilnik sočasno nameščenih večje število procesov. Vsakemu procesu je potrebno dodeliti del pomnilnika in to v obsegu, ki mu omogoča napredovanje. Različnim procesom morajo biti dodeljeni različni 'neprekrivajoči' se deli pomnilnika. Načrtovalec sistema je v tem primeru soočen z nalogo dodelitve pomnilnika večjemu številu procesov. To bi mu bilo moč realizirati v času prevajanja, povezovanja ali nameščanja, kot smo ravnokar opisali. Slika 4.2.(a) prikazuje mogočo rešitev. Vsakemu procesu je dodeljen del fizičnega pomnilnika. Procesu P_1 je dodeljen pomnilnik od naslova 1000 do naslova 4FFF, procesu P_2 je dodeljen pomnilnik od naslova 7000 do naslova 9FFF, medtem ko je proces P_3 nameščen na naslove od C000 do EFFF. Taki so tudi logični naslovi.

A boljša rešitev je celovit pristop k upravljanju pomnilnika in operacijski sistem. V večopravilnih sistemih z navideznim pomnilnikom logični in fizični naslov nista več enaka, slika 4.2.(b). V sistemih z navideznim pomnilnikom se logični naslov preslika v fizični naslov v času izvajanja. V našem primeru so logični naslovi procesa P_1 od 0000 do 3FFF in se v času izvajanja preslikajo v fizične naslove od 1000 do 4FFF, kjer je proces nameščen v pomnilnik. Enako velja za procesa P_2 in P_3. Posledično dvema po vrednosti enakima logičnima naslovoma, ki pripadata različnima



Slika 4.2: *Primer dodelitve pomnilnika trem procesom. (a) Fizični naslovi so enaki logičnim. (b) Fizični naslovi niso enaki logičnim naslovom. Logični naslovi se preslikajo v fizične naslove med izvrševanjem procesa, kar je značilnost navideznega pomnilnika.*

procesoma, ustrezata dve pomnilniški besedi z različnima fizičnima naslovoma. Z drugimi besedami, logični naslovi pripadajo procesu. Ko, spričo izvajanja procesa, procesna enota generira logični naslov, se le-ta preko 'skritih' registrov preslika v fizični naslov. Da se zaradi preslikave vsake pomnilniške reference bistveno ne upočasni izvrševanje programa/procesa, mora biti preslikava realizirana čimbolj hitro. Strojno opremo, ki daje podlago za kar se da učinkovito izvedbo navideznega pomnilnika, imenujemo enoto za upravljanje pomnilnika.

4.3 Navidezni pomnilnik

Preprosto povedano je navidezni pomnilnik (Angl. Virtual memory) mehanizem, ki daje uporabnikom vtis, da je glavnega pomnilnika več kot ga je v resnici. V sistemu z navideznim pomnilnikom se lahko izvršujejo večji programi in večje število programov kot bi sicer pričakovali. Navidezni

pomnilnik včasih daje uporabnikom celo vtis, da je pomnilnika v neomejenih količinih, torej vedno toliko kolikor ga ravno tedaj potrebujejo. Spričo tega pričakujejo, da lahko izvršujejo poljubno velike programe in poljubno število programov oziroma procesov. V realnosti to sicer ne drži, a vendar je učinek navideznega pomnilnika v glavnem zelo prepričljiv.

In na kakšen način se da doseči to navidezno povečanje pomnilnika?

Naj bo proces P velikosti $|P|$ in naj bo pomnilnik M velikosti $|M|$. V sistemu brez navideznega pomnilnika se mora proces tedaj, ko se izvršuje, v celoti nahajati v pomnilniku. Torej mora biti pomnilnik vsaj tolikšen kot je proces. V sistemu z navideznim pomnilnikom to ni potrebno. Navidezni pomnilnik omogoča izvršitev procesa tudi tedaj, ko ni v celoti nameščen v pomnilniku, kar pomeni, da je lahko večji od razpoložljive velikosti pomnilnika,

$$|P| \geq |M|.$$

Ker se proces kljub temu izvršuje, je navzven videti tako, kot bi bil v celoti cel čas v pomnilniku. Posledično zaključimo, da je pomnilnika več, kot ga je v resnici. Kako je izvrševaje procesa v takšninih pogoji sploh mogoče?

Izkaže se, da večino današnjih programov/procesov za izvršitev sploh ne potrebuje toliko pomnilnika, kolikor ga zahteva. Nekateri deli programa oziroma procesa se nikoli ali le redko izvršijo. Torej v pomnilniku sploh niso potrebni ali pa niso potrebni cel čas napredovanja procesa. Torej ni potrebe, da bi bili v pomnilnik nameščeni cel čas.

Nadalje, imejmo N procesov $P_i, i = 1, ..., N$ velikosti $|P_i|, i = 1, ..., N$. Navidezni pomnilnik omogoča sočasno namestitev večjega števila programov, ki so v skupnem obsegu večji od razpoložljive velikosti pomnilnika,

$$\sum_{i=1}^{N} |P_i| \geq |M|.$$

To se doseže z nameščanjem v pomnilnik samo tistih delov programov oziroma procesov, ki so za napredovanje procesa v danem obdobju resnično potrebni ali 'na zahtevo' (Angl. On demand). S tem povečamo stopnjo večopravilnosti in ponovno ustvarimo vtis, da je pomnilnika več, kot ga je

v resnici.

Za izvedbo in delovanje navideznega pomnilnika je bistvenega pomena lokalnost pomnilniških referenc. Za večino današnjih programov (procesov) velja empirično ugotovljena lastnost, da so pomnilniške reference (naslavljanje) lokalnega značaja. Reference se grupirajo krajevno in časovno:

- časovna lokalnost: referenca, ki je bila dana v nekem trenutku se bo z veliko verjetnostjo kmalu ponovila.
- Krajevna lokalnost: referenca v naslednjem trenutku se bo z veliko verjetnostjo malo razlikovala od predhodnjih referenc.

Zaporedje pomnilniških referenc torej ni povsem naključno. Na podlagi lastnosti lokalnosti je moč vzorec pomnilniških referenc, ki bodo sledile v naslednjem obdobju, dokaj dobro, ali vsaj do neke mere, napovedati. Če torej zagotovimo, da bodo v pomnilniku vedno prisotni vsaj tisti deli programov/procesov, ki bodo tedaj resnično potrebni, potem napredovanje procesa ne bo trpelo na hitrosti. V relanosti se prihodnosti napredovanja procesa ne da povsem zagotovo napovedati. A če z upoštevanjem lokalnosti pomnilniških referenc dovolj dobro uganemo prihodnost, potem napredovanje procesa ne bo bistveno prizadeto.

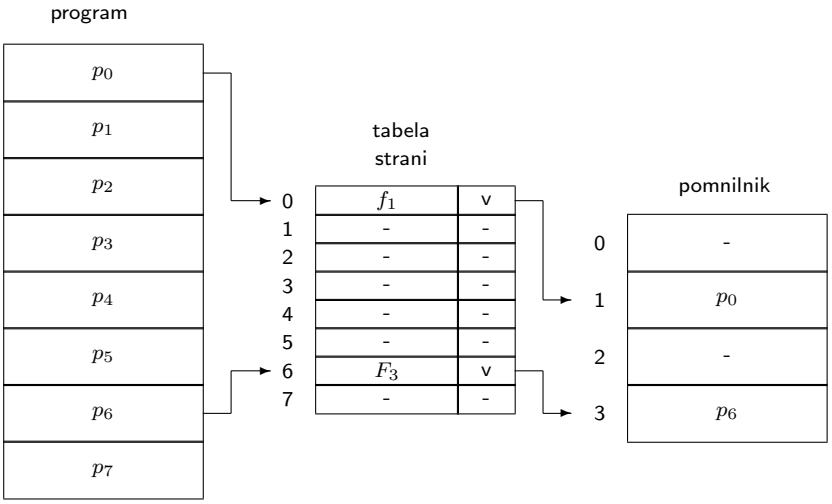
Dandanes ima navidezno povečanje pomnilnika na podlagi izvedbe navideznega pomnilnika dosti manjši pomen kot ga je imelo v preteklosti. To še toliko bolj velja za vgradne sisteme. Pomembno pa je, da navidezni pomnilnik realizira koncept ločenih logičnih naslovnih prostorov procesov, kar je podlaga za izvedbo sistema za zaščito pomnilnika. Spričo tega se izraz navidezni pomnilnik ali *virtuali naslovni prostor* pogosto uporablja tudi kot nadomestno ime za logični naslovni prostor procesa.

V nadaljevanju si bomo ogledali dva pristopa k izvedbi navideznega pomnilnika ter kombinacijo obeh.

4.4 Ostranjen navidezni pomnilnik

V ostranjenem navideznem pomnilniku delimo logično naslovno področje programa na enako dolge dele, ki jim rečemo strani (Angl. Page). Velikost strani, na primer 4096, je parameter sistema. Fizično naslovno področje glavnega pomnilnika delimo na enako dolge dele, ki jih imenujemo okvirji strani (Angl. Page Frames). Velikost strani je enaka velikosti okvirja strani. Katerokoli stran programa lahko namestimo v katerikoli okvir strani pomnilnika. Program ni zvezno nameščen v pomnilniku, zvezno je nameščena samo posamezna stran. Logični naslovi strani se med izvrševanjem programa preslikajo v fizične naslove okvirjev strani preko tabele strani (Angl. Page Table). V sistemih z navideznim pomnilnikom so v pomnilnik praviloma nameščene samo nekatere strani, to so tiste, ki so za napredovanje procesa tedaj potrebne, slika 4.3. Tem stranem rečemo tudi delovna množica stani. V primeru *zadetka*, to je reference na stran, ki je v pomnilniku, se izvrševanje procesa nadaljuje. V primeru *zgrešitve*, to je v primeru reference na stran, ki še ni v pomnilniku, se izvrševanje procesa prekine. Temu dogodku rečemo *napaka strani* (Angl. Page fault), čeprav ni s tako referenco nič narobe. Operacijski sistem namesti zahtevano stran v pomnilnik, osveži vsebino tabele strani in obnovi izvrševanje procesa. Pomnilnik se torej dodeljuje stranem *na zahtevo* (Angl. Demand paging). Dokler je verjetnost zadetka dovolj visoka, napredovanje procesa bistveno ne trpi na hitrosti.

Načelo preslikave logičnih naslovov v fizične prikazuje slika 4.4. Prek tabele stani se naslov strani preslika v okvir strani, spodnji del naslova pa je odmik zotraj strani in posledično okvirja strani. Velikost tabele strani je odvisna od števila strani. V tabeli strani so za vsako stran procesa poleg fizičnega naslova okvirja strani, kamor je stran nameščena, še dodatna določila: bit prisotnosti, bit veljavnosti, bit spremembe, biti pravic dostopa, in podobno. V primeru, da stran ni nameščena v pomnilnik, je bit prisotnosti nič in referenca v tabeli strani kaže na diskovni blok, ki hrani to stran. Diskovnega prostora za potrebe navideznega pomnilnika je običajno precej več kot samega glavnega pomnilnika in skoraj vedno več kot dovolj.

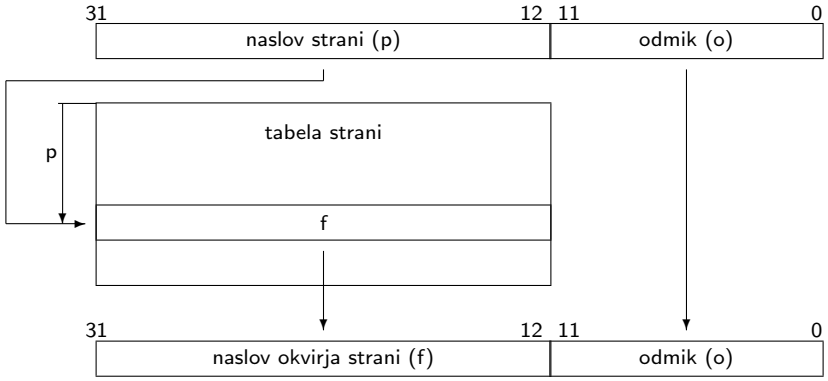


Slika 4.3: *Koncept ostranjenega navideznega pomnilnika. Strani programa se preko tabele strani preslikajo v okvirje strani. V konkretnem primeru sta v pomnilnik nameščeni samo dve od osmih strani programa. Stran 0 je nameščena v okvir 1 in stran 6 je nameščena v okvir 3.*

Tabel strani je v splošnem potrebnih toliko, kolikor je v pomnilniku sočasno nameščenih procesov. Teh je lahko tudi zelo veliko. Tabele strani se zato morajo nahajati v glavnem pomnilniku (slika 4.5). To pa hkrati pomeni, da je za vsako koristno pomnilniško referenco, potrebna še ena pomnilniška referenca za dostop do tabele strani. Da se napredovanje procesa zaradi dostopa do tabele strani ne upočasni, se aktivni del tabele drži v naslovnem predpomnilniku (Angl. Translation Lookaside Buffer – TLB), a o teh mehanizmih ne bomo govorili.

4.5 Segmentiran pomnilnik

Segmentiran pomnilnik temelji na izgledu naslovnega področja programa, kot ga vidi programer. Za programerja je program zbirka različno obsežnih kosov programa – segmentov. Segment je kos programa, ki ima s stališča programerja enoten pomen. Segment programa tvorijo na primer ukazi glavnega programa. To bi lahko poimenovali glavni ukazni segment. Na-



Slika 4.4: *Načelni potek preslikave logičnih naslovov v fizične naslove. Naslov strani (p) se čez tabelo stani preslika v naslov okvirja (f) in skupaj z odmikom znotraj strani tvori cel fizični naslov. Skicirani so 32 bitni logični in fizični naslovi za velikost strani 4096 oziroma 12 bitni odmik.*

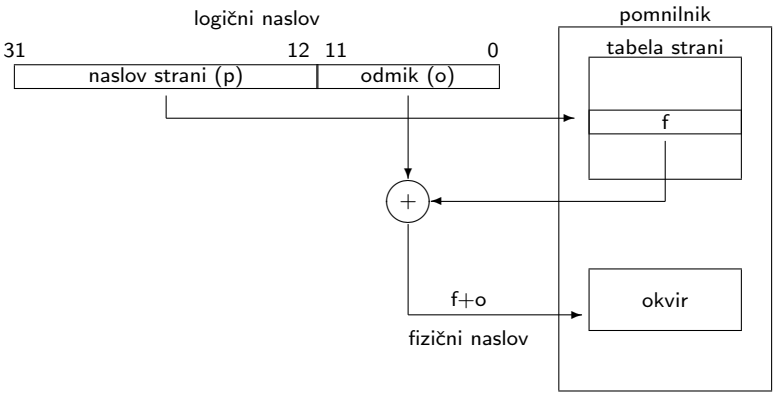
slednji segment bi lahko sestavljale funkcije. To bi lahko bil dodatni ukazni segment. Segment bi lahko tvorili podatki, zato bi ga poimenovali podatkovni segment. Podatkovnih segmentov bi bilo lahko po potrebi tudi več. Skladu bi namenili skladovni segment.

Logično naslovno področje programa torej sestavlja zbirka segmentov, kot to prikazuje slika 4.6. Da podamo logični naslov, potrebujemo dve določili: določilo segmenta (s) in odmik (o) znotraj segmenta. Segmenti so v splošnem različno veliki. Zato je potrebno za vsak segment podati še njegovo velikost oziroma *dolžino* (l).

V segmentiranem pomnilniku so posamezni segmenti programa nameščeni v pomnilnik v enem kosu, torej zvezno. V segmentiranem navideznem pomnilniku sicer ni treba, da bi bili v pomnilnik sočasno nameščeni vsi segmenti. A če je segment v pomnilniku, potem je v pomnilnik nameščen v celoti. V pomnilnik se nameščajo samo tisti sementi, ki so tedaj potrebni za napredovaje programa. V času izvajanja programa se logični naslovi preslikajo v fizične naslove preko segmentne tabele,

$$(s, o) \rightarrow f + o.$$

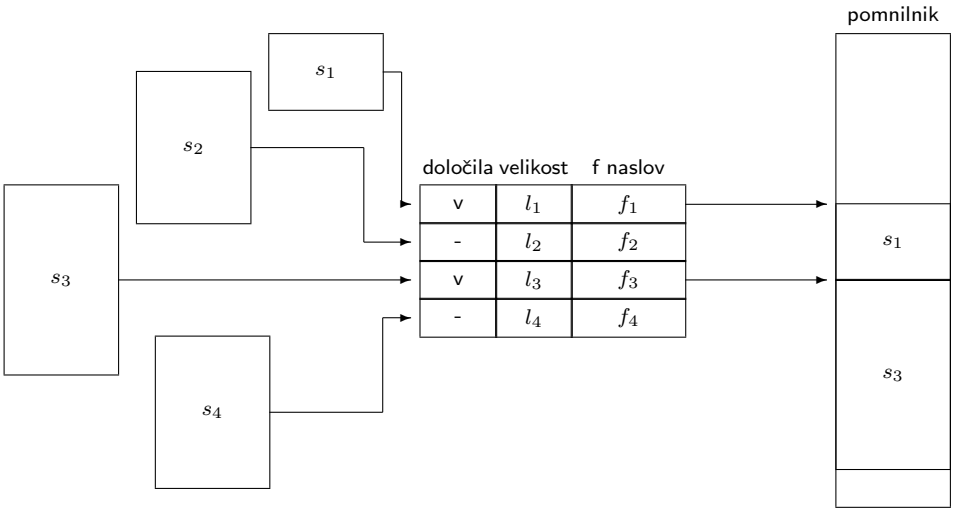
Koncept prikazuje slika 4.6. Segmentna tabela bi bila lahko tudi zelo ve-



Slika 4.5: *Potek preslikave naslovov v ostranjenem pomnilniku. Logični naslov se preko tabele strani preslika v fizični naslov. Tabela strani se nahaja v pomnilniku. Tabele strani so interne podatkovne strukture operacijskega sistema, tj. jedra. Vsaka koristna referenca v načelu zahteva dva dostopa do pomnilnika.*

lika, zato se nahaja v glavnem pomnilniku. V segmentni tabeli je za vsak segment programa zabeležen pripadajoči *segmentni deskriptor*. Deskriptor vsebuje preslikavo logičnega naslova v fizičnega, to je, določa začetni fizični naslov segmenta v pomnilniku. Poleg tega vsebuje deskriptor še velikost segmenta in dodatna določila, kot so tip segmenta, prisotnost v pomnilniku, pravice za branje in pisanje, stanje spremembe segmenta v pomnilniku, in podobno. V primeru reference na segment, ki ni v pomnilniku, strojna oprema povzroči prekinitev oziroma izjemo, izvajanje programa se začasno prekine, operacijski sistem streže izjemi in v pomnilnik namesti zahtevani segment, nakar obnovi izvrševanje in posledično napredovanje programa.

Tako kot v ostranjenem pomnilniku sta tudi v segmentiranem pomnilniku za vsako koristno pomnilniško referenco potrebna vsaj dva dostopa do pomnilnika, eden za dostop do segmentne tabele in drugi za dostop do ukaza ali operanda. Dvakratni dostop do pomnilnika za vsako pomnilniško referenco bi bistveno upočasnil napredovanje programa. Da se to ne zgodi, se v ostranjenem pomnilniku preslikavo pohitri z naslovnim preslikovalnim predpomnilnikom. V segmentiranem pomnilniku pa se preslikava pohitri z asociativnimi segmentnimi registri. Zadostuje že relativno malo segmentnih registrov. Na primer, ukazni segmenti register, skladovni segmentni

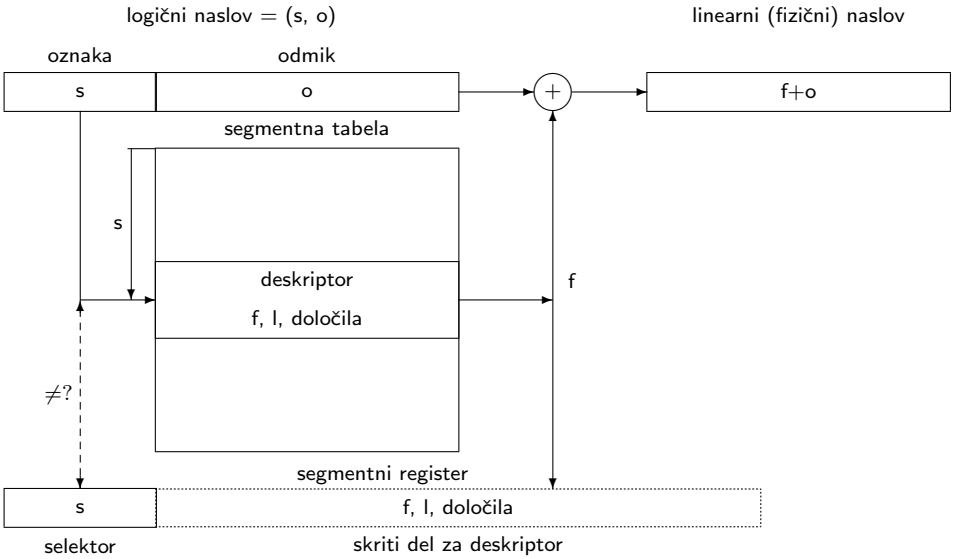


Slika 4.6: *Načelna preslikava naslovov v segmentiranem pomnilniku. 'Dvodimenzionalni' logični naslov, ki ga sestavljata oznaka segmenta in odmik, se preko tabele segmentov preslika v enodimenzionalni oziroma 'linearni' fizični naslov. V pomnilniku sta samo segmenta s_1 in s_3.*

register, in eden ali dva podatkovna segmentna registra.

Segmentni registri so namenski naslovni registri procesorja. Segmentni registri se polnijo s segmentnim deskriptorjem iz segmentne tabele ob prvi referenci na aktivni segment. Za vse naslednje reference znotraj aktivnega segmenta dostop do segmentne tabele ni več potreben. Hkrati je torej lahko aktivnih največ toliko segmentov kot je segmentnih registrov. Načelno preslikavo ter polnjenje segmentnih registrov prikazuje slika 4.7.

Logični naslov sestavljata dve določili: oznaka segmenta in odmik znotraj segmenta. Segmentni register je sestavljen iz dveh delov, tako imenovanega vidnega oziroma selektorskega dela in skritega dela. Preslikava dvodimenzionalnega logičnega naslova (oznaka segmenta, odmik znotraj segmenta) v enodimenzionalni (linearni) fizični naslov poteka nekako takole. Oznaka segmenta, ki je del logičnega naslova, se primerja z vidnim delom segmentnega registra. V primeru ujemanja oziroma *zadetka* se iz *skritega* dela segmentnega registra pridobi fizični naslov segmenta. Ta naslov skupaj z odmikom znotraj segmenta tvori dejanski fizični naslov. V primeru *zgrešitve*



Slika 4.7: *Preslikava logičnih naslovov v fizične preko segmentne tabele in asociativnih segmentnih registrov. Selektorski del registra se polni z oznako segmenta. V primeru zgrešitve, se skriti del registra polni s selektorjem iz segmentne tabele. Sicer se logični naslov (s,o) preslika v linearni fizični naslov preko skritega dela segmentnega registra.*

se selektorski del registra polni z oznako segmenta, sledi dostop do segmentne tabele in polnjenje skritega dela segmentnega registra. Pri tem vidni del segmentnega registra služi kot indeks v segmentno tabelo za dostop do pripadajočega deskriptorja.

4.6 Segmentiranje z ostranjenjem

V segmentiranem in ostranjenem pomnilniku se logično naslovno področje programa deli najprej na različno obsežne dele – segmente. Vsak segment se nadalje deli na enako velike dele – strani. Povedano drugače, segmente se ostrani. Dvodimenzionalni logični naslovi se preslikajo v enodimenzionalne (linearne) naslove preko segmentne tabele. Segmentna tabela se nahaja v glavnem pomnilniku. Aktivni deskriptorji segmentne tabele se nahajajo v hitrih (asociativnih) segmentnih registrih. V pomnilniku se nahajajo samo

nekateri segmenti. Ker so segmenti ostranjeni, linearni naslov še ni fizični naslov. Linearni naslovi se preko tabele strani preslikajo v fizične naslove. Tabela strani se nahaja v glavnem pomnilniku. Aktivni deli tabele strani se nahajajo v naslovnem preslikovalnem predpomnilniku. V pomnilniku se nahajajo samo nekatere strani segmenta.

Teoretično obstajajo različne možnosti za izvedbo navideznega pomnilnika z segmentiranjem in ostranjenjem. Dodeljevanje pomnilnika na zahtevo, kar je navsezadnje bistvo navideznega pomnilnika, se lahko izvede na nivoju segmentov. V pomnilnik so nameščeni le nekateri segmenti. Če so segmenti ostranjeni, se v pomnilniku nahajajo vse strani segmenta. Pravzaprav potem niti ne bi bilo potrebno ostranjenje, a omogoči nezvezno nameščanje segmentov v pomnilnik.

Druga možnost je, da se navidezni pomnilnik realizira na nivolju ostranjenih segmentov. Pomnilnik je sicer segmentiran, a so v pomnilniku vsi segmenti. Ni nujno, da bi bili segmenti nameščeni v celoti, v pomnilnik se namestijo samo nekatere strani segmentov.

Najbolj splošna izvedba bi bila kombinacija obeh, to pomeni, da bi bile v pomnilniku samo nekatere strani nekaterih segmentov. Vendar se takšen način spričo prevelike kompleksnosti ne uporablja. V preteklosti je bil glavni motiv za razvijanje segmentiranega pomnilnika razširitev naslovnega področja. Danes potreba po tem, spričo vse bolj navzočih 64-bitnih arhitektur, počasi izginja.

Ostranjenje ima kar nekaj prednosti pred segmentiranjem. Prvič, koncept ostranjenja je za uporabnika povsem transparenten. Drugič, strani so manjše od segmentov, zato je nameščanje strani v pomnilnik ter umikanje strani iz pomnilnika hitrejše kot v primeru segmentov. Tretjič, strani so enako velike, zato je dodelitev pomnilnika lažja kot v primeru različno dolgih in tipično precej velikih segmentov.

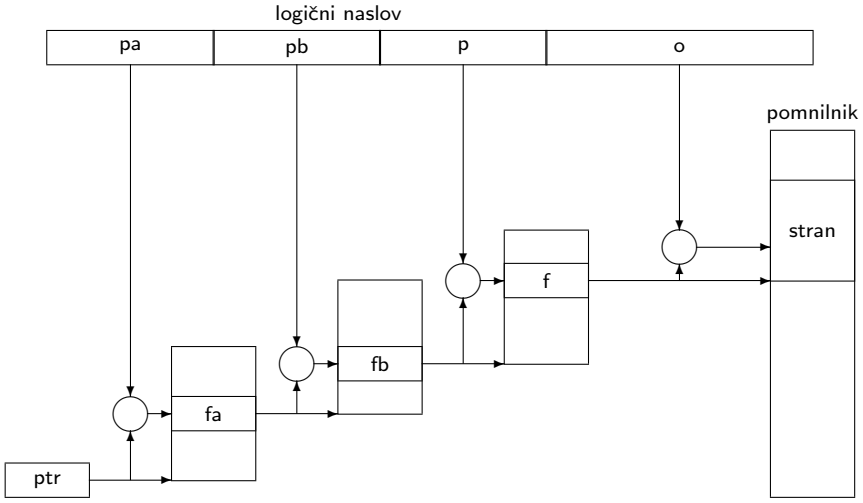
4.7 Upravljanje pomnilnika in Linux

Kot rečeno je za realizacijo navideznega pomnilnika potrebna ustrezna programska oprema in ustrezna podpora strojne opreme. Izhodišče pri razvoju sistema za upravljanje pomnilnika v sistemu Linux sta bila učinkovitost in prenosljivost. Oboje gre v prid ostranjenemu navideznemu pomnilniku. Večina današnjih arhitektur daje podlago ostranjenemu pomnilniku, medtem ko je segmentiran pomnilnik značilnost predvsem Intelovih arhitektur.

Večina današnjih arhitektur daje podlago večnivojskemu sistemu tabel. To pomeni, da preslikava logičnega naslova v fizični naslov ne poteka preko ene same, posledično zelo velike, tabele strani, temveč prek hierarhično urejene zbirke tabel. Za primer si zamislimo 32 bitno arhitekturo ter ostranjenje s stranmi velikosti 4096. Pri tej velikosti strani je odmik 12 biten, zato ostane dvajset bitov za indeks v tabelo strani, kar znese 1M naslovov. Če vsak element tabele strani zavzame 4 bajte, rabimo za tabelo 4MB pomnilnika. Če to množimo s številom procesov, denimo 100, dobimo 400MB, kar je preveč.

Linux realizira večnivojski sistem tabel. V uporabi so trije nivoji tabel, ne glede na to, koliko nivojev direktno podpira dotična arhitektura procesorja. Potek preslikave prikazuje slika 4.8. Register procesorja (ptr) kaže na začetek strani zgornjega nivoja. Zgornji del logičega naslova je indeks v tabelo zgornjega nivoja, ki izbere eno od tabel srednjega nivoja. Srednji del logičnega naslova je indeks v tabelo srednjega nivoja ter izbere eno od tabel strani. Iz te tabele se izbere eno izmed strani oziroma začetni naslov okvirja strani, ki skupaj z odmikom tvori fizični naslov.

Posebej močan mehanizem dodeljevanja pomnilnika v sistemu Linux je *prepis ob vpisu* (Angl. Copy-on-Write). To daje podlago za učinkovito izvedbo upravljanja procesov in večopravilnosti. Mehanizem prepis ob vpisu se nanaša na stran. Stran, ki je nameščena v pomnilnik in se ne spreminja, je lahko skupna večim procesom. Strani z ukazi so že tak primer. Tudi strani s podatki, ki jih vsi procesi samo berejo, so tak primer. Če pa se vsebina strani tudi spreminja, si jo lahko deli več procesov samo do prvega vpisa.



Slika 4.8: *Preslikava logičnega naslova v fizični naslov prek sistema tabel strani.*

Proces, ki bi skušal izvesti operacijo vpisa na stan, ki jo rabijo tudi drugi procesi, bo prekinjen. Sledi izjema, ki ji streže jedro. Jedro procesu dodeli novo stran, prepiše vsebino stare strani, osveži tabelo strani in obnovi napredovanje procesa.

Poglavje 5

Upravljanje procesov

V tem poglavju bomo govorili o programih, procesih, stanjih procesa, prehajanju med stanji procesa, menjavi konteksta in menjavi načina napredovanja procesa. Spregovorili bomo o nadzornem bloku procesa oziroma deskriptorju procesa in navedli nekaj lastnosti Linuxa s tem v zvezi.

5.1 Programi in procesi

Osnovni objekt, s katerim se bomo od sedaj naprej ukvarjali večino časa, je proces. Računalniški proces, ali krajše kar proces, je program v izvrševanju. Takšna raba izraza proces je nastala ter se obdržala predvsem v sistemih UNIX ter njegovih naslednikih, kot je Linux. Namesto proces je v rabi še izraz opravilo (Angl. Task). Oba izraza sta v sistemih UNIX povsem enakovredna.

V nadaljevanju bomo večinoma uporabljali izraz proces, le tu pa tam bomo procese poimenovali opravila. Denimo, raje bomo rekli večopravilnost in večopravilni sistemi kot večprocesni sistemi. S tem se bomo izognili morebitni dvoumnosti med večprocesorskimi in večprocesnimi sistemi. Večprocesorski sistemi so namreč sistemi z več procesorji, večprocesni sistemi pa so sistemi z več sočasnimi procesi oziroma opravili.

V nekaterih večnitnih sistemih, kot je tudi Linux, se izraz opravilo včasih uporablja tako za procese kot za dele procesov, to je niti. A o nitih bomo govorili v naslednjem poglavju.

5.2 Program

Kot rečeno je proces program v izvrševanju. Torej mora biti program tak, da ga je moč izvršiti. In kaj razumemo pod pojmom program?

Izraz *program* uporabljamo v različnih besednih in pomenskih zvezah. Program je na primer izvorni program, C-jevski program, strojni program, objektni program, in podobno. No, pa spregovorimo par besed tudi o tem.

Izvorni program je programsko besedilo v izbranem programskem jeziku. Na primer, izvorni program je besedilo v programskem jeziku C. Programskemu besedilu rečemo tudi *izvorna koda*. Izvorni program se nahaja v tekstovni datoteki. Povsem samoumevno je, da se take datoteke ne da izvršiti.

Strojni program je program v strojnem jeziku. Program v strojnem jeziku dobimo tako, da izvorni program s prevajalnikom prevedemo v strojno obliko. Programu ali delu programa po prevajanju rečemo tudi objektni program ali *objektna koda*, datoteki pa objektna datoteka. Objektni program še ni pripravljen za izvršitev. Objektni datoteki je potrebno dodati še od sistema odvisne objektne datoteke. K tem sistemskim objektnim datotekam spadajo ponavadi zelo kratek začetni kos programa (Angl. start-up routine), s katerim se izvrševanje programa začne in nekatere funkcije, ki so potrebne med izvrševanjem programa. Te funkcije se nahajajo v sistemskih knjižnicah. Primer sistemske knjižnice je denimo `libc.so`. Sestavljanju izvršljivega programa iz večjega števila objektnih kosov rečemo *povezovanje* (Angl. Linking). Povezovanje opravi povezovalnik. S povezovanjem končno dobimo izvršljiv program.

V sistemu GNU/Linux opravi vse naštete korake drugega za drugim ukaz `gcc`. Najprej in še pred prevajanjem se izvede predobdelava programskega

besedila. To nalogo opravi makro predprocesor. Pravimo, da razreši ali razširi makro definicije. Če je potrebno, se v tej fazi združi več izvornih datotek v skupno programsko besedilo. Zatem sledi prevajanje. Prevajalnik analizira izvorno programsko besedilo in ga prevede v zbirni jezik. Programsko besedilo v zbirniškem jeziku se potem prevede v strojni jezik. Po enakem postopku se lahko zaporedoma obdela več izvornih datotek. V eni od teh datotek bi bil denimo glavni program oziroma funkcija `main()`, v drugih datotekah bi bile lahko definicije različnih uporabniških funkcij. Končni rezultat prevajanja je v splošnem ena ali več tako imenovanih objektnih datotek. Nazadnje se objektne datoteke povežejo skupaj. K njim se po potrebi dodajo še funkcije iz programskih knjižnic. Rezultat je izvršljiv program.

Ko rečemo program, običajno posebej ne poudarjamo za kakšno obliko programa dejansko gre. Ali mislimo na izvorno kodo ali na strojno kodo je moč razbrati iz konteksta. Izvorni program in iz njega narejeni strojni program sta v bistvu semantično ekvivalentna. Včasih celo rečemo, da bomo program izvršili, ko imamo pred sabo izvorno C-jevsko kodo. Seveda nam ne pride na misel, da bi izvorno kodo programa tudi zares poskusili izvršiti.

Izvršljiv program se nahaja v *izvršljivi* datoteki (Angl. Executable file). Datoteka mora biti označena kot izvršljiva in mora tudi zares vsebovati strojni program, ki ga je moč naložiti v pomnilnik ter izvršiti. Nalaganje v pomnilnik, ali kot rečemo *dodelitev pomnilnika*, opravi nalagalnik (Angl. Loader), ki je del sistemskega jedra. Struktura izvršljive datoteke je predpisana in znana nalagalnemu programu. Večina današnjih sistemov UNIX / Linux uporablja tako imenovano ELF (Executable and Linking Format) strukturo datoteke. Struktura ELF datoteke je precej kompleksna in je tu ne bomo obravnavali.

Na področju vgradnih sistemih često razvijamo programsko opremo na razvojnem računalniškem sistemu, ki je praviloma bistveno zmogljivejši od vgradne platforme, na kateri bo delovala razvita programska oprema. Tedaj, ko *ciljna platforma* ali arhitektura ni hkrati *razvojna platforma*, go-

vorimo o križnem prevajanju. Ko rečemo prevajanje, imamo dejansko v mislih vse korake, ki so potrebni, da nastane izvršljiv program.

5.3 Proces

Izvršljivi program je torej moč naložiti v pomnilnik ter izvršiti. Tako nastane proces. Proces v splošnem sestavlja več delov ali *segmentov*: segment za ukaze, segment za podatke oziroma operande ter sklad. Segmentom včasih pravimo tudi *sekcije*. To pa zato, da bi se izognili dvoumnosti med segmeti programa in segmentiranim navideznim pomnilnikom. Ti segmenti namreč nimajo direktno nič skupnega s segmentiranjem pomnilnika. Segmente programa imamo namreč ravno tako tudi v ostranjenem pomnilniku.

Strukturo procesa prikazuje slika 5.1. Delu procesa z ukazi, torej ukaznemu segmentu, četudi nekoliko zavajajoče, pravimo tekst. Takoj *nad* tekstom je podatkovni segment *inicializiranih* podatkov (data segment). To so tisti podatki, ki imajo definirane vrednosti že z izvorno kodo in se *inicializirajo* ob nalaganju programa v pomnilnik. V C-jevskem programu se definicije teh podatkov nahajajo pred funkcijo `main()`. Na primer, z navedbo:

```
int  Stevec = 128;
char *Niz = "Znakovni niz";
```

pred funkcijo `main()` oziroma izven katerekoli funkcije dosežemo, da bodo ti podatki v segmentu inicializiranih podatkov in da bodo imeli s programskim besedilom določene vrednosti. Rečemu tudi, da so ti podatki *statični*. Razen statičnih podatkov imamo v programu še *dinamičo* alocirane podatke. Dinamični podatki se nahajajo na skladu. Dosegljivi so samo znotraj dotične funkcije in veljavni tedaj, kadar se funkcija izvršuje. Na primer, navedba

```
int  Indeks;
```

znotraj funkcije `main()` bo za spremenljivko `Indeks` priskrbela prostor na skladu.

Nad inicializiranimi podatki so neinicializirani podatki. To so podatki, ki

jim je pomnilnik sicer dodeljen statično, a sprva, ko se program naloži v pomnilnik, še nimajo definiranih vrednosti. V sistemu Linux jim nalagalnik dodeli vrednost nič, a se na to ne gre zanašati. Na primer,

```
int Polje[128];
```

pred funkcijo `main()`, bo poskrbel za pomnilnik v zahtevanem obsegu. Ta podatkovni segment v sistemu UNIX/Linux iz zgodovinskih razlogov označujejo z `bss` (Block Starting with Symbol) segment. BSS je bilo nekoč namreč ime enega od zbirniških ukazov (mnemonik ukaza) za rezervacijo pomnilnika za podatke. Nad bss segmentom je prostor za *kup* (Angl. Heap) oziroma kopico. Tu se pomnilnik zaseže dinamično, tako kot narekujejo potrebe med izvrševanjem programa. Za dinamično alokacijo pomnilnika služita vsem dobro znani funkciji `malloc()` in `calloc()` ter mogoče manj znani funkciji `brk()` in `sbrk()`. S slednjima direktno vplivamo na najvišji naslov ali 'prelom' kopice. Na najvišjih naslovih programa je sklad, ki se polni *navzdol* proti nižjim naslovom. Vsak proces ima svoj uporabniški sklad. V naslovnem področju pod skladom in nad kupom je prostor za deljene knjižnice (Angl. Shared Libraries), deljen pomnilnik (Angl. Shared memory), a o tem kasneje.

Velikosti segmentov programa se da dobiti z Linux ukazom `size`. Za ilustracijo smo programček `moj.c` zapisali na dva načina ter obe različici prevedli.

```
/* moj1.c */

int  x = 1;
int  y;

int main( void )
{
  return 0;
}
```

```
/* moj2.c */

int main( void )
{
  int  x = 1;
  int  y;

  return 0;
}
```

Ukazu:

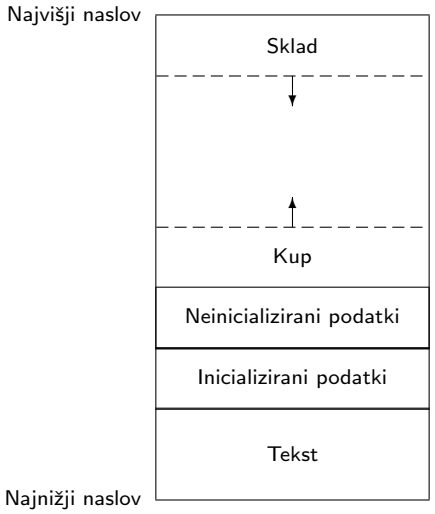
```
size moj1 moj2
```

je sledil izpis:

```
   text    data     bss     dec     hex   filename
```

```
1129      544       8     1681      691    moj1
1129      540       4     1673      689    moj2
```

Iz prvega stolpca izpisa razberemo, da sta ukazna segmenta `text` obeh programov enako velika in obsegata 1129 bajtov. Iz prve vrstice izpisa razberemo, da je podatkovni segment programa `moj1` z imenom `data` velik 544 bajtov. V njem je spremenljivka `x` zasedla 4 bajte. Segment `bss` obsega 8 bajtov, v njem je spremenljivka `y` zasedla 4 bajte. Spremenljivki `x` in `y` sta statični. V drugem programu bo za obe spremenljivki uporabljen sklad. Spremenljivki `x` in `y` sta alocirani dinamično. Od tu tudi razlika v velikosti segmetov `data` in `bss`. Iz druge vrstice razberemo, da sta oba segmenta v programu `moj2` manjša za 4 bajte, kar je toliko, kolikor pomnilnika porabi vsaka od spremenljivk `x` in `y`, ki sta tipa `int`. Prvi program je velik 1681 bajtov (0x691 bajtov) in drugi 8 bajtov manj, to je 1673 bajtov (0x689).

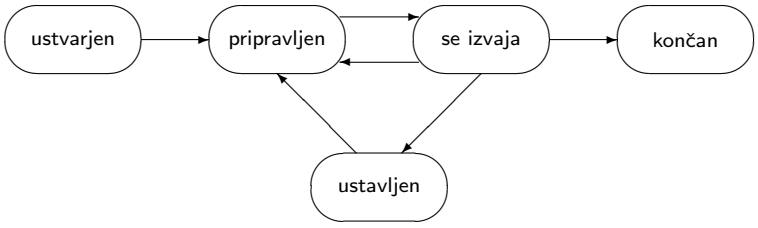


Slika 5.1: *Struktura procesa: ukazi (tekst), podatki z začetno vrednostjo, neinicializirani podatki, kopica in sklad. Ukazi in začetni podatki so s klicem* `exec` *naloženi (inicializirani) iz programa, neinicializirani del podatkov postavi na nič jedro kot posledica izvršitve sistemskega klica* `exec`.

5.4 Stanja procesa

Proces je torej program v izvrševanju. Proces nastane iz programa, ki se običajno nahaja v datoteki na disku. Ko se to zgodi, pravimo, da je proces ustvarjen. V splošnem lahko iz enega programa nastane oziroma je inicializiranih več procesov. Proces med napredovanjem spreminja stanje. Pravimo, da je proces aktiven, kadar ima zagotovljene pogoje za napredovanje. Sicer je ustavljen. Ustavljen je tedaj, ko nima vseh pogojev za napredovanje. To je na primer tedaj, ko čaka na zaključek vhodne ali izhodne operacije, denimo na branje ali pisanje na disk, odčitavanje stanja stikal, postavljanje izhodnih signalov in podobno. Tedaj proces ne potrebuje procesne enote.

V aktivnem stanju se proces poteguje za procesno enoto ali pa mu je le-ta že dodeljena (Angl. Runnable). A proces zares napreduje samo tedaj, ko ima procesno enoto. Pravimo, da se izvaja (Angl. Running). V nasprotnem primeru je proces pripravljen (Angl. Ready-to-run). Pripravljen je torej tisti proces, ki ima zagotovljene vse pogoje, da bi se izvajal, če bi mu bila dodeljena procesna enota. Med svojim obstojem proces spreminja stanje, kot to prikazuje diagram prehajanja stanj na sliki 5.2, dokler nazadnje ne konča.



Slika 5.2: *Stanja procesa in prehajanja med stanji procesa.*

V stanje *se izvaja* gre proces lahko samo iz stanja *pripravljen*. Tedaj, ko proces potrebuje vhodno/izhodni prenos, gre iz stanja *se izvaja* v stanje *ustavljen* (Angl. Blocked ali Stopped). Ko je prenos opravljen, postane proces ponovno *pripravljen*. V stanje *pripravljen* gre proces lahko tudi iz stanja *se izvaja*. To se zgodi tedaj, ko mu poteče dodeljeni čas ali pa mu pro-

cesno enoto prevzame kakšen od procesov z višjo prioriteto. Glede na to, ali procesor sproščajo procesi sami ali pa mu je le-ta prevzeta s strani drugega procesa, razlikujemo razvrščanje procesov s prevzemanjem in brez prevzemanja. Razvrščanje s prevzemanjem procesne enote je bistvenega pomena za sisteme, ki morajo delovati v stvarnem času. Lastnost prevzemanja procesne enote s strani drugega procesa imenujemo tudi *predopravilnost* (Angl. Preemptive scheduling).

S tega vidika je še posebej pomembno vprašanje, kako se sistem odzove na zahtevo procesa višje prioritete, ko se proces z nižjo prioriteto izvaja v sistemskem jedru ali povedano drugače, ko sistemsko jedro streže procesu z nižjo prioriteto. Klasične izvedbe sistemov UNIX/Linux niso podpirale predopravilost znotraj sistemskega jedra. Jedro je bilo *neprekinljivo* (Angl. Non-preemptive kernel). Zato tako Linux kot UNIX v osnovi nista bila najbolj primerna za časovno kritične sisteme oziroma sisteme realnega časa. To funkcionalnost so zagotovile šele kasnejše razširitve oziroma popravki jedra.

V večopravilnem operacijskem sistemu obstaja v določenem obdobju več procesov, od katerih je pač vsak na določeni stopnji napredovanja. V enoprocesorskem sistemu je v stanju izvajanja v določenem trenutku samo en proces. V večprocesorskem sistemu je takih procesov največ toliko, kolikor je procesnih enot. V stanju pripravljen ali ustavljen je praktično poljubno število procesov, a iz praktičnih razlogov je število sočasnih procesov navzgor omejeno.

Del operacijskega sistema, ki določa, kateri proces se bo izvajal naslednji oziroma kdaj in za koliko časa, imenujemo razvrščevalnik procesov (Angl. Scheduler). Del operacijskega sistema, ki procesu dodeli procesor oziroma opravi menjavo procesov, imenujemo tudi dispečer in je del razvrščevalnika. Menjavo procesov imenujemo *kontekstni (pomenski) preklop* (Angl. Context switch). Kadar je procesu dodeljena procesna enota rečemo tudi, da procesor deluje v kontekstu danega procesa.

Za razumevanje delovanja in uporabe operacijskih sistemov in računalniških sistemov na sploh, je razlikovanje med procesom, kot abstraktno tvorbo in

procesorjem, kot delom strojne opreme, ključnega pomena. V računalniškem sistemu lahko v določenem obdobju sočasno obstaja večprocesov. V enoprocesorskem sistemu pač v določenem obdobju napreduje (se izvaja) samo en proces, v večprocesorskem sistemu pa tudi več. Bistvo večopravilnosti je torej v dodeljevanju procesne enote procesom tedaj, ko jo potrebujejo. Tedaj, ko proces čaka na vhodno izhodni prenos, ne potrebuje procesorja, torej se procesor dodeli procesu, ki ga potrebuje.

5.5 Sistemski in uporabniški način

Proces torej ob določenem času nastane, nato napreduje, dokler enkrat ne konča. Trenutek nastanka procesa je lahko pomemben podatek, kakor tudi trenutek, ko proces konča. Intervalu od nastanka do zaključka procesa običajno rečemo resnični čas procesa. Proces med napredovanjem uporablja procesno enoto. Skupnemu času uporabe procesorja rečemo procesorski ali kar CPE čas. Večino procesorskega časa naj bi proces porabil v *uporabniškem načinu* (Angl. User running). Ko pa izvede sistemski klic, se izvajanje prenese v sistemsko jedro. Procesor dela za proces v *sistemskem načinu* (Angl. System running). Rečemo, da jedro teče v kontekstu procesa. Nekoliko podrobnejši diagram prehajanja stanj procesa bi izgledal takole, slika 5.3. Kot bomo videli v naslednjih poglavjih proces nastane s klicem `fork` in konča s klicem `exit`. Medtem spreminja stanje in način. Ko na primer izvede klic `read`, se izvajanje prenese v jedro in ko se klic vrne, se izvajanje povrne v uporabniški način.

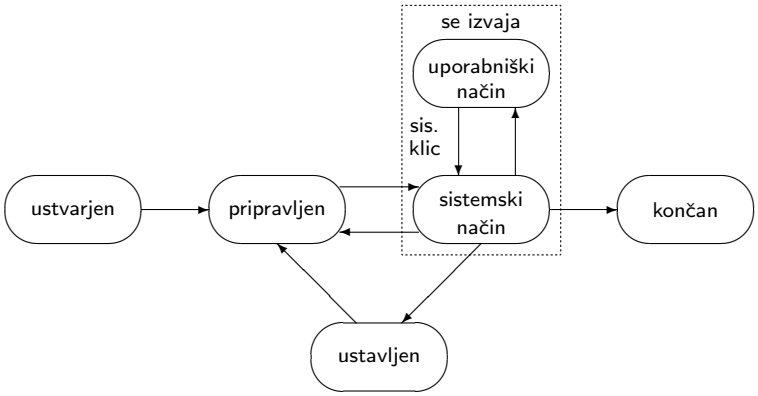
Čase, ki jih proces prebije v enem in drugem načinu lahko dobimo z ukazom `time`. Na primer, program `mcp` izvršimo takole:

```
time mcp p pp 16
```

Ko se program `mcp` konča, `time` izpiše

```
real    0m6.581s
user    0m0.229s
sys     0m6.337s
```

Resnični čas procesa je bil v tem primeru 6,581 sekund. Procesorski čas v



Slika 5.3: *Stanja procesa in prehajanja med stanji procesa.*

uporabniškem načinu je znašal 229 milisekund in 6,337 sekund se je proces izvajal v sistemskem načinu. Še več informacij o procesu oziroma procesih lahko dobimo z ukazom `ps`. Možnosti je veliko. Ukaz

```
ps -o pid,pri,pcpu,size,user,tty,cmd
```

na primer oblikuje naslednji izpis:

```
 PID  PRI %CPU SIZE  USER    TT     CMD
12313 19 0.0   1440  stanek  pts/2  xterm
15761 19 0.7 305088  stanek  pts/2  gedit s.c
16116 19 0.0    936  stanek  pts/2  ps -o pid,pri,pcpu,size,user,tty,cmd
28235 19 0.0   2316  stanek  pts/2  /bin/bash
```

Vsaka vrstica vsebuje podatke za enega od procesov. V prvem stolpcu je unikatna številka procesa (PID), v drugem stolpcu je prioriteta procesa, sledijo odstotek obremenitve procesorja, velikost procesa v pomnilniku (v KB), ime uporabnika, kontrolni terminal in izgled ukazne vrstice. In od kje ukazu `time` in ukazu `ps` informacije o procesih?

Odgovor leži v podatkovnih strukturah sistemskega jedra. Sistemsko jedro beleži za vsak proces podatke o procesu v podatkovni strukturi jedra. Tej podatkovni strukturi na splošno rečemo nadzorni blok procesa (Angl. Process Control Block).

5.6 Nadzorni blok procesa

Nadzorni blok procesa je temeljna podatkovna struktura sistemskega jedra. Nahaja se v naslovnem prostoru jedra in do nje uporabnik nima neposrednega dostopa. Nadzornemu bloku procesa v sistemu Linux rečemo *procesni deskriptor* (Angl. Process descriptor). V procesnem deskriptorju so zabeleženi vsi podatki, ki jih jedro potrebuje za upravljanje procesa. Teh je kar precej, tako da procesni deskriptor zasede skoraj 2 KB pomnilniškega prostora.

In kateri so ti podatki? Nekatere smo že spoznali. Vsak proces ima svojo številko. Številka procesa je celo pozitivno število. Največja dovoljena vrednost za številko procesa je parameter sistema in posredno določa največje možno število sočasnih procesov. V sistemih Linux so številke procesov v območju 16 bitnih predznačnih števil, tako da je največja dovoljena vrednost 32767, s čimer je posredno določeno maksimalno število sočasnih procesov. To je sicer moč razširiti na 32 bitov s spremembo parametrov sistema. Dejansko vrednost preverimo v

```
/proc/sys/kernel/pid_max
```

Procesu je dodeljen pomnilnik v obsegu, ki mu omogoča napredovanje. Kje in v kakšnem obsegu je proces nameščen v pomnilnik, je pomemben podatek o procesu. Sem spada tudi tabela strani.

Nadalje so procesu dodeljene vhodne in izhodne naprave, če jih potrebuje. Proces ima do njih dostop prek deskriptorjev odprtih datotek. Tabela deskriptorjev spada v deskriptor procesa.

Procesu je dodeljena procesna enota, kadar se oziroma zato, da se izvaja. Vsebine registrov procesorja (podatkovnih registrov, naslovnih registov, akumulatorjev, registra stanja procesorja, programskega števca) v času, ko je procesna enota dodeljena izbranemu procesu, imenujemo tudi *kontekst* procesne enote. Rečemo tudi, da procesor teče v kontekstu procesa. Kadar se zgodi *pomenski preklop* oziroma menjava procesa, ko se torej procesor dodeli drugemu procesu, se aktualni kontekst procesorja shrani na varen prostor, registri procesorja pa se polnijo s shranjenim kontekstom procesa, ki se bo izvajal naslednji. Tudi kontekst procesorja spada v procesni de-

skriptor.

Nadzorni bloki vseh procesov so na primeren način zbrani v urejenem seznamu procesov. Ta je običajno realiziran kot povezana, dvojno povezana, ali dvojno krožno povezana vrsta procesov. V naslednjem seznamu so navedeni tipični elementi nadzornega bloka procesa:

- unikatna številka procesa (Angl. Process ID – PID),
- lastništvo procesa (Angl. User ID – UID),
- stanje procesa: ustvarjen, pripravljen, ustavljen, v izvajanju, končan,
- prioriteta procesa, druga določila relevantna za razvrščanje procesov,
- stanje procesorja oziroma kontekst procesne enote: programski števec, register stanja procesorja, skladovni kazalec, drugi registri procesorja,
- stanje pomnilnika: nameščenost v pomnilniku, kje, v kakšnem obsegu, druga določila za upravljanje pomnilnika,
- stanje vhodnih in izhodnih prenosov, deskriptorji odprtih datotek, druga določila medprocesnih komunikacij.

Nadzorni bloki aktivnih procesov (pripravljenih in tistega v izvajanju) so običajno zabeleženi v vrsti pripravljenih procesov (Angl. Ready Queue). Na opisni blok procesa, ki se izvaja, kaže kazalec procesa v izvajanju. Opisni bloki ustavljenih procesov, ki čakajo na konec v/i operacije, so zabeleženi v čakalnih vrstah v/i naprav, na primer diskovne naprave. Ko proces pridobi pogoje za napredovanje, zapusti čakalno vrsto v/i naprave in gre na seznam pripravljenih procesov.

5.7 Procesi, stanja procesov in Linux

Sistem Linux je v pogledu stanja procesa nekoliko svojski. V evidenci procesov namreč ne razlikuje med procesom, ki se izvaja in procesi, ki bi se lahko izvajali, to je pripravljenimi procesi. Vsi procesi so izvršljivi (Angl. Runnable). Diagram prehajanja stanj je sicer ekvivalenten. Po

drugi strani pa razčleni ustavljene procese na tiste, ki so prekinljivi (Angl. Interruptable sleep) in druge, ki niso prekinljivi (Angl. Uninterruptable sleep). Neprekinljivo stanje traja običajno kratek čas in v njem so procesi, ki jih ni moč obuditi s *signalom*. Poleg teh dveh tipov procesov pa so še tisti ustavljeni procesi, ki so bili ustavljeni s *signalom*.

Tudi med zaključnenimi procesi sta dve vrsti procesov. Tisti, ki so dokončno končani in tisti, ki so sicer končani, a opuščeni. A o tem več kasneje. Mogoča stanja z dogovorjenimi oznakami stanj prikazuje naslednji seznam:

- R: pripravljen ali se izvaja (Angl. Runnable),
- D: neprekinljivo ustavljen (Angl. Unintteruptable sleep),
- S: prekinljivo ustavljen (Angl. Interruptable sleep),
- T: ustavljen s signalom (Angl. Stopped),
- X: končan,
- Z: opuščen (Angl. Defunct ali zombie).

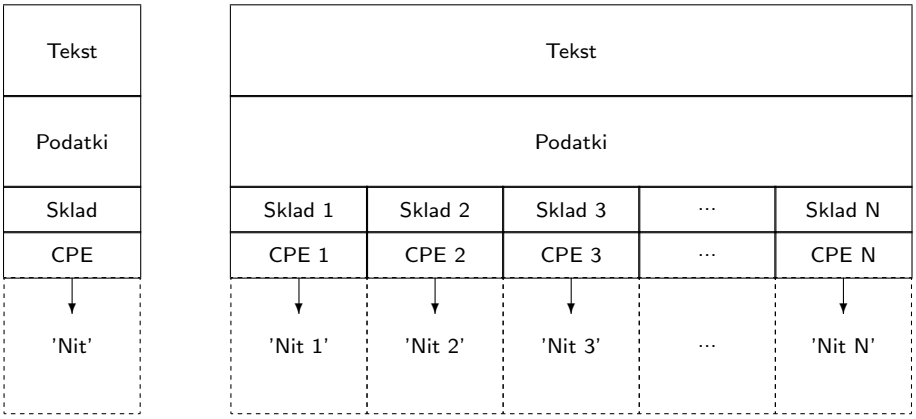
Poglavje 6

Niti

V predhodnih poglavjih smo se osredotočili na koncept sekvenčnega procesa in na večopravilnost (Angl. Multitasking). Spoznali smo upravljanje procesov. Ogledali smo si bistvene koncepte in mehanizme za komunikacijo med sekvenčnimi procesi, vključno s sinhronizacijo asinhronih sočasnih procesov. Večopravilnost realizira obliko sočasnosti v okviru množice sočasnih procesov. V tem poglavju se bomo srečali z obliko sočasnosti v okviru posameznega procesa, poimenovano večnitnost (Angl. Multithreading). Približali si bomo upravljanje niti ter sinhronizacijo niti, kadar je sinhronizacija potrebna.

6.1 Koncept večnitnega procesa

V večopravilnem sistemu obstaja v danem obdobju večje število procesov. Ti procesi napredujejo na videz sočasno ali zares sočasno, ali kot rečemo v slednjem primeru, vzporedno. S konceptualnega stališča sicer ni najbolj pomembno ali procesi napredujejejo resnično sočasno ali pa je zgolj videti tako. To je odvisno od pravila razvrščanja, od števila razpoložljivih procesnih enot in od tega, ali procesi sploh imajo pogoje za napredovanje. Dejstvo je, da se procesi vsak zase izvajajo *sekvenčno* in da obstajajo v istem obdobju.



Slika 6.1: *Struktura enonitnega (levo) in večnitnega procesa (desno). V skupnem naslovnem prostoru večnitnega procesa hkrati napreduje več dejavnosti – niti, od katerih vsaki pripada lastna procesna enota ter lastni sklad, medtem ko so ukazi in podatki skupni.*

V okviru danega procesa se ukazi izvajajo sekvenčno, po eden na kmalu, drug za drugim, ali pa je vsaj videti tako[1]. Ukazi puščajo sled. Zaporedje ukazov, ki jih procesna enota izvršuje drugega za drugim, imenujemo *nit*. Klasični sekvenčni računalniški program oziroma proces napreduje v eni niti. Za razliko od koncepta sočasnih procesov, ki v večopravilnem sistemu realizira sočasnost na nivoju sistema, pa niti s podporo jedra realizirajo sočasnost v okviru istega procesa. V okviru istega procesa hkrati napreduje več dejavnosti, za vsako od sočasnih dejavnosti obstaja ena nit. Konceptualno torej vsaki niti v času napredovanja pripada procesna enota oziroma bolje rečeno kontekst procesne enote in sklad (slika 6.1). V večprocesorskih sistemih ali v sistemih z več jedrnimi procesorji lahko v resnici hkrati napreduje več niti in največ toliko kolikor je procesorjev oziroma jeder[2].

A tudi v sistemu z enim samim procesorjem lahko v določenem obdobju obstaja več niti in videti je, kot da bi imeli več procesorjev, čeprav v resnici napreduje le ena nit, tista, ki ji je dodeljen procesor. Za dodeljevanje pro-

[1] Današje procesne enote namreč realizirajo različne oblike ukaznih in podatkovnh paralelizmov, na primer cevovodnost in superskalarnost, česar uporabnik ne opazi.

[2] Večnitnosti ne smemo mešati s hipernitnostjo (Angl. Hyperthreading), ki je arhitekturna lastnost nekaterih Intelovih procesnih enot.

cesorja skrbi sistemsko jedro. Ko nit, ki ima procesor, nima več pogojev za napredovanje ali pa bi morala napredovati druga nit, jedro opravi menjavo niti. To pomeni, da shrani kontekst procesne enote tekoče niti na varno mesto, obnovi kontekst procesne enote naslednje niti in s tem je menjava opravljena.

Večnitnost je torej koncept in hkrati primeren nivo abstrakcije, ki ni direktno vezan na število razpoložljivih procesorjev. Število niti je odvisno od strukture programa oziroma procesa. Denimo, z eno nitjo bi proces lahko stregel vhodni napravi, z drugo nitjo bi na primer stregel izhodni napravi, medtem ko bi v okviru tretje niti mogoče urejal podatke. Tako bi proces v okviru tretje niti lahko napredoval, tudi če nobena od zunanjih naprav ne bi omogočala napredovanja. V primeru enonitnega procesa bi proces morebiti obstal, to je čakal na vhodni ali izhodni prenos in v tem času ne bi napredoval, tudi če bi bila na razpolago prosta procesna enota.

6.2 Niti in naslovni prostor

Vsakemu procesu pripada naslovni prostor, ki mu je dodeljen del pomnilnika, do katerega ima proces izključni dostop. Če želijo procesi med sabo komunicirati prek deljenega pomnilnika, mora biti isti del fizičnega pomnilnika preslikan v naslovne prostore teh procesov.

Za razliko od procesov oziroma opravil, od katerih ima vsak svoj in od drugih procesov ločen naslovni prostor, pa niti obstajajo oziroma si delijo skupen naslovni prostor procesa, kot ponazarja slika 6.1. Vse niti lahko dostopajo do istega segmenta inicializiranih podatkov, do segmenta neinicializiranih podatkov, do dinamično dodeljenega pomnilnika s kupa in seveda do skupnega ukaznega segmenta. Komunikacija med nitmi prek deljenega pomnilnika je torej sama po sebi zagotovljena. Denimo, ena nit piše v pomnilnik, druga nit iz njega bere. Ena nit postavi vrednost spremeljivke in druga nit vrednost te spremenljivke bere. Po drugi strani pa obstaja nevarnost, da napredovanje ene niti moti pravilnost napredovanja drugih niti.

Niti napredujejo asinhrono, tako kot to narekuje programska logika in pravila razvrščanja. V primeru, da je potrebna sinhronizacija, je za usklajeno napredovanje potrebno posebej poskrbeti. Vprašanje kritičnih področij in v splošnem vprašanje sinhronizacije pride pri večnitnosti direktno do izraza. Če povzamemo:

- Različni procesi imajo različne oziroma ločene naslovne prostore, niti si delijo skupni naslovni prostor procesa.
- V okviru enonitnega procesa napredujejo dejavnosti sekvenčno, tudi če so med seboj neodvisne in bi lahko napredovale sočasno ali vzporedno. V okviru večnitnega procesa lahko sočasno napreduje toliko niti, kolikor je neodvisnih dejavnosti.
- V primeru enonitnega procesa bo proces čakal, če tekoča dejavnost nima pogojev za napredovanje (na primer v/i prenos). V primeru večnitnega procesa bo proces napredoval v okviru druge oziroma drugih niti.

6.3 Niti in Linux

Linux je večnitni operacijski sistem. Pristop k nitnosti v sistemu Linux je na nek način unikaten. Dejansko je nit osnovna razvrstljiva enota, ki se poteguje za procesno enoto. Napreduje tedaj, ko ji razvrščevalnik sistemskega jedra dodeli procesno enoto. Večnitni proces sestavlja *skupina niti*, ki imajo med sabo veliko skupnega, najpomembneje pa je, da si delijo skupen naslovni prostor. Včasih zato slišimo, da tvori skupino niti množica *lahkih procesov*.

Tudi jedro Linuxa je večnitno. To so niti jedra. Skupna značilnost teh niti je, da se v celoti izvajajo v sistemskem načinu in torej nimajo uporabniškega naslovnega prostora. Te niti opravljajo dejavnosti jedra.

Poglavje 7

Komunikacije med procesi

V večopravilnem sistemu je v pomnilnik sočasno nameščenih več procesov. Vsakemu od teh procesov je dodeljen del pomnilnika, do katerega ima proces izključni dostop. Videti je, kot da bi imeli toliko resničnih pomnilnikov kolikor je procesov, od katerih ima vsak toliko pomnilnika kolikor ga rabi. Povedano še drugače, procesi so v pomnilnik nameščeni tako, da se med seboj ne prekrivajo. Ena in ista pomnilniška beseda je dodeljena enemu samemu procesu.

Procesi napredujejo v svojem naslovnem področju, ki smo ga poimenovali logični naslovni prostor procesa. Poseg (referenca) izven naslovnega področja procesa ni dovoljena. Nedovoljeno oziroma *neveljavno* referenco presteže strojna oprema in razreši sistemsko jedro. Referenca izven naslovnega področja procesa namreč povzroči izjemo (Angl. Exception), ki se ji streže v okviru jedra. Jedro v takem primeru običajno predčasno zaključi proces.

Naslovni prostori procesov so torej popolnoma ločeni. Takšna rešitev je idealna za procese, ki so drug od drugega povsem neodvisni. Denimo, za urejavalnik besedila in spletni brskalnik, za video predvajalnik in C-jevski prevajalnik, ipd. A vendarle se slej ko prej pokaže potreba po sodelovanju procesov med seboj in torej za komunikaciji med procesi. Na primer, spletni brskalnik potrebuje video predvajalnik za predvajanje spletnih vse-

bin, ali C-jevski prevajalnik potrebuje urejavalnik za urejanje programskega besedila. Tedaj je potrebno sodelovanje oziroma komunkacija med procesi.

Razlikujemo dve glavni obliki komunikacij med procesi, ki jih večkrat označujemo kar s kratico IPC (Angl. Interprocess Communications - IPC):

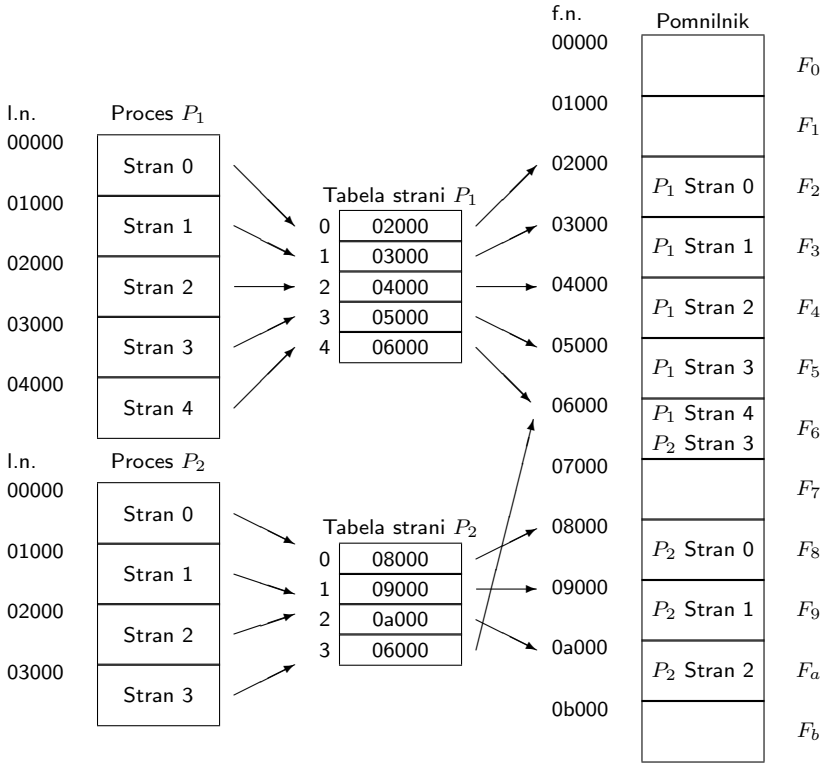
- komunikacija s pomočjo deljenega pomnilnika (Angl. Shared Memory). Komunikacija poteka po načelu beri/piši vsebino pomnilniške besede.
- Komunikacija s pomočjo sporočil (Angl. Message Passing). V tem primeru poteka komunikacija po komunikacijskem kanalu po načelu pošlji/sprejmi sporočilo.

K tem dvem osnovnim oblikam moramo pridružiti še *sinhronizacijo*. Potreba po sinhronizaciji, to je potreba po časovnem usklajevanju, spremlja tako komunikacijo prek deljenega pomnilnika kot komunikacijo na osnovi sporočil. Dejansko bi lahko na sinhronizacijo gledali tudi kot na posebno obliko komunikacije.

7.1 Deljen pomnilnik

V sistemu z deljenim pomnilnikom poteka komunikacija prek pomnilnika oziroma dela pomnilnika, do katerega ima dostop večje število procesov. Namesto deljeni bi lahko rekli tudi skupni pomnilnik. Sistemsko jedro namreč več procesom dodeli isti del fizičnega pomnilnika. Ko je pomnilnik dodeljen, poteka komunikacija med procesi direktno brez posredovanja jedra.

Deljen pomnilnik predstavlja najhitrejšo obliko medprocesnega komuniciranja, saj do prenašanja podatkov v pravem pomenu besede sploh ne pride. Eden ali več procesov v pomnilnik enostavno piše in drugi procesi to vsebino pomnilnika po potrebi berejo. Za koordinacijo, torej sinhronizacijo, pri sočasnem dostopu do pomnilnika, če je ta potrebna, in skoraj vedno je potrebna, morajo poskrbeti procesi sami.



Slika 7.1: *Koncept skupnega pomnilnika. Procesa imata vsak svoje logično naslovno področje in sta nameščena v pomnilnik tako, da je stran 4 procesa P_1 nameščena v isti okvir (F_6) pomnilnika (naslovi pomnilnika 06000 do 07000) kot stran 3 procesa P_2. Logična referenca 04000 procesa P_1 ima za posledico isti pomnilniški naslov kot logična referenca 03000 procesa P_2. Logični naslovi so po vrednosti različni.*

Deljen pomnilnik (Angl. Shared Memory) je torej del fizičnega pomnilnika, ki je hkrati dodeljen več kot enemu procesu. Povedano drugače, isti fizični pomnilnik je hkrati preslikan v logična naslovna področja večjega števila sočasnih procesov. Slika 7.1 daje osnovo za lažjo razlago deljenega pomnilnika. Procesa P_1 in P_2 sta nameščena v pomnilnik tako, da si delita del pomnilnika, v narisanem primeru je to okvir strani F_6. Vsak proces ima svoj logični naslovni prostor, ki je razdeljen na strani. Stran 4 procesa P_1 je nameščena v isti del pomnilnika kot stran 3 procesa P_2. Obema stranema je dodeljen isti del fizičnega pomnilnika. To konkretno pomeni, da logičnemu naslovu 04000 procesa P_1 pripada pomnilniška beseda z na-

slovom 06000, a tej isti pomnilniški besedi pripada logični naslov 03000 procesa P_2. Logična naslova procesov sta različna, a imata za posledico isti fizični naslov pomnilniške besede pomnilnika. Ko proces P_1 piše na naslov 04000, se spremeni vsebina pomnilniške besede 06000, ki jo proces P_2 lahko bere na logičnem naslovu 03000.

7.2 Sporočila

V primeru sporočilnega sistema se med procesi prenašajo sporočila. Koncept prikazuje slika 7.2. Sporočilo je zaporedje podatkovnih enot, denimo bajtov. Sporočilo lahko obsega en sam bajt ali pa je pač poljubno daljše. En proces sporočilo ali več sporočil odda in drugi proces sporočila, delno ali v celoti, sprejme. Še prej je med procesi potrebno *vzpostaviti* oziroma napeljati komunikacijski kanal. To je naloga sistemskega jedra. Tudi prenašanje podatkov med procesi, torej *vzdrževanje* komunikacijskega kanala, je naloga sistemskega jedra. Navsezadnje je naloga jedra tudi *sproščanje* kanala, ko ni več potreben.

Slika 7.2: *Koncept sporočil. Kar en proces pošlje, drugi proces v enakem zaporedju sprejme. Komunikacijski kanal zagotovi jedro.*

Komunikacijo med procesi po načelu sporočil bi sicer lahko realizirali na različne načine. Denimo, komunikacije bi lahko potekale prek ustreznih povezovalnih struktur, ali prek komunikacijskih krmilnikov, prek diskovja, ali pač tudi prek skupnega pomnilnika, s katerim upravlja sistemsko jedro. A kadar govorimo o medprocesnih komunikacijah, nas običajno ne zanima, kako so procesi, procesorji ali pomnilniki v resnici povezani med seboj.

Sporočilni sistem upelje drugačen nivo abstrakcije. Glavna vprašanja v povezavi s sporočilnimi sistemi so:

- kako vzpostaviti ali *napejati* komunikacijski kanal med za komunikacijo zainteresirane procese. Konkretno, kako zagotoviti, da bo en konec komunikacijskega kanala dostopen enemu procesu in drugi konec tega istega kanala dosegljiv drugemu procesu.
- Ali je komunikacijski kanal enosmeren (prenos podatkov samo v eno smer) ali dvosmeren (prenos podatkov v obe smeri).
- Ali je vse, kar je oddano, zagotovo dostavljeno. Torej kako je z zanesljivostjo.
- Kakšen je odnos med v komunikacijo udeleženima procesoma. Denimo,
  - ali je odnos med procesoma simetričen oziroma ali sta oba procesa v pogledu komuniciranja enakovredna (Angl. peer-to-peer),
  - ali je odnos med procesoma asimetričen in ali gre za koncept odjemalec-strežnik (angl. Client-Server).
- Kakšne komunikacijske storitve zagotavlja dani komunikacijski kanal. Denimo,
  - ali ohranja meje v zaporedju oddanih sporočil tako, da jih sprejemni proces lahko prepozna,
  - ali sam po sebi ohranja *strukturo* sporočil.
- Ali komunikacijski kanal sam skrbi za sinhronizacijo med sprejemno in oddajno stranjo. Denimo,
  - kaj se zgodi, če skuša sprejemni proces sprejeti sporočilo še predno ga oddajni proces odda.
  - Ali, kako sprejemni proces prepozna, če sploh, da je oddajna stran zaključila z oddajo.

Vsa ta in še marsikatera druga vprašanja pridejo pri komunikaciji na pod-

lagi sporočil bolj ali manj do izraza. Katera vprašanja so ključnega pomena, je odvisno od končnega namena uporabe. Zato daje večina operacijskih sistemov komunikacijam po načelu sporočil le elementarno podlago, medtem ko sodi njihova nadgradnja že v domeno aplikacij in komunikacijskih protokolov.

Naše zdajšnje področje zanimanja pa lahko opredelimo na naslednji način. V sistemu obstaja več procesov. Ti procesi v splošnem napredujejo vsak s svojo in vnaprej nepredvidljivo hitrostjo, torej asinhrono. Nekateri med njimi napredujejo samostojno in neodvisno drug od drugega. Taki procesi nas posebej ne zanimajo. Zanimajo nas taki procesi, ki na nek način sodelujejo pri reševanju skupnega problema. Zato morajo vsaj občasno med seboj komunicirati. Imamo torej množico asinhronih sočasnih procesov, ki med sabo komunicirajo ter se občasno časovno usklajujejo ali sinhronizirajo.

7.3 Sinhronizacija procesov

V tem razdelku bomo najprej spregovorili o problemih in rešitvah problemov, ki se pojavijo tedaj, kadar večje število procesov dostopa do sredstva, ki ne dovoljuje sočasnega dostopa. Primeri takih sredstev so na primer skupen pomnilnik ali del pomnilnika, skupna datoteka, tiskalnik, in podobno. Z drugimi besedami, operacije, ki so definirane nad takimi sredstvi, se med seboj izključujejo. Če dovolimo izvrševanje ene od operacij, moramo preprečiti začetek izvrševanja ostalih, dokler ni operacija, ki je v teku, končana. Pravimo, da moramo zagotoviti medsebojno izključevanje izvrševanja nasprotujočih si operacij. Dostop do skupnega sredstva moramo časovno uskladiti ali *sinhronizirati*. Ko govorimo o medsebojnem izključevanju, mislimo v bistvu na mehanizme oziroma algoritme, ki naj preprečijo sočasen dostop. Lahko bi rekli, da iščemo algoritme, ki se izvajajo v več procesih ali kar porazdeljene algoritme oziroma protokole za preprečitev konflikta.

Na koncu razdelka si bomo ogledali še nekaj drugih primerov sinhronizacije, ki je denimo potrebna zaradi pomanjkanja sredstev, na primer pomnilniškega prostora, nerazpoložljivosti podatkov, in podobno.

7.3.1 Predstavitev problema

Imamo večje število sočasnih procesov. Procesi so sočasni, ker obstajajo ob istem času, v istem časovnem obdobju. Ti procesi napredujejo vzporedno ali na videz vzporedno v smislu večopravilnosti, dejavnosti znotraj posameznega procesa pa potekajo zaporedno (sekvenčno).

Sočasni procesi so lahko med sabo popolnoma neodvisni, torej ne komunicirajo. Taki procesi nas ne zanimajo. Zanimajo nas tisti sočasni procesi, ki so med seboj nekako povezani, njihovo izvajanje je v nekem smislu soodvisno. Predpostavljamo, da ti procesi napredujejo vsak s svojo in vnaprej neznano (nepredvidljivo) hitrostjo. Zato pravimo, da so procesi asinhroni. Zanimajo nas torej asinhroni sočasni procesi, ki med sabo sodelujejo (rešujejo isti problem) in se morajo zato občasno sinhronizirati. Medprocesna komunikacija

se torej javlja kot problem sinhronizacije procesov.

Primer: Dostop do skupne spremenljivke

Imamo procesa P_1 in P_2, ki uporabljata skupno spremenljivko x, kot prikazuje spodnja shema. Procesa naj napredujeta asinhrono, neodvisno in z neznano hitrostjo. S spremenljivko x bi morda šteli število oseb v prostoru. Proces P_1 ob vsakem prihodu osebe skozi vhodna vrata poveča x za ena. Proces P_2 ob vsakem odhodu osebe skozi izhodna vrata zmanjša x za ena. Osebe prihajajo in odhajajo naključno in tako se povečuje ali zmanjšuje x. Naj bo vrednost x-a v danem trenutku 10.

P_1	P_2
`int x = 10;`	`int x; /* skupna spremenljivka */`
...	...
...	...
`x++`	`x--`
...	...
...	...

Sedaj si zamislimo, da v danem trenutku ena oseba pride in druga oseba v istem trenutku odide. Proces P_1 bo povečal x za 1, medtem ko mora proces P_2 spremenljivko x zmanjšati za 1. Pričakujemo, da bo vrednost spremenljivke x ostala še naprej 10.

Izmed več možnih zaporedij operacij si oglejmo naslednje zaporedje operacij, R je eden od podatkovnih registrov procesorja:

	Proces P_1	R (P_1)	Proces P_2	R (P_2)	x
1.	P_1 bere x v R	10	--	--	10
2.	--	10	P_2 bere x v R	10	10
3.	--	10	P_2 zmanjša R	9	10
4.	P_1 poveča R	11	--	9	10
5.	P_1 shrani R v x	11	--	9	11
6.	--	--	P_2 shrani R v x	9	9

Po izvršitvi tega zaporedja operacij bi spremenljivka x zavzela vrednost 9. Zapisano zaporedje operacij se, sicer upravičeno, zdi malo verjetno, a ni čisto nemogoče. Mogoči in tudi bolj verjetni so še drugačni scenariji. Ob drugačnem razpletu dogodkov bi x lahko dobil vrednost 11 ali 10. Spre-

menljivka x lahko torej zavzame eno od vrednosti {9, 10, 11 }, od katerih je le ena pravilna. Potek izvršitve je nepredvidljiv in vrednost spremenljivke x je odvisna od naključja, kar je nedopustno.

Očitno je, da bi bi bil rezultat pravilen samo tedaj, ko bi se operaciji povečanje in zmanjšanje vrednosti x-a ne prepletali. S tem mislimo na scenarija, ko se zgodi najprej zvečanje in nato zmanjšanje, ali obratno. V obeh slučajih se operaciji zvrstita sekvenčno ena za drugo, medtem ko vrstni red ni pomemben. Na koncu bo rezultat x-a vedno 10, kar je tudi pravilno.

Da bi se operaciji zvečanja in zmanjšanja zvrstili sekvenčno v vsakih okoliščinah, bi se morali izvršiti časovno nedeljivo ali *atomično*. Z drugimi besedami, ko operacija enkrat začne, se mora izvršiti do konca brez prekinitve.

Obe operaciji trajata malo časa, zato bi bila zahteva po atomičnosti operacij sprejemljiva rešitev. Večino časa bi procesa napredovala neodvisno in ne bi drug drugemu motila napredovanja. Občasno bi proces P_1 spremenljivko x povečeval in proces P_2 bi jo občasno in ob kakšnih drugih trenutkih zmanjševal. Le v redkih primerih, ko bi prišlo do sočasnega dostopa do spremenljivke x, bi se obe zahtevi razvrstili sekvenčno.

Na podlagi povedanega pa zaključimo, da se v splošnem obe operaciji medsebojno časovno izključujeta: če dovolimo eno, moramo prepovedati drugo, dokler se prva ne konča.

V zgornjem primeru se zdi zahteva po atomičnosti operacije spreminjanja x-a smiselna, ni pa univerzalna ali splošna. Zamislimo si, da si procesa ne delita ene spremenljivke navadnega tipa, ampak obsežno sestavljeno podatkovno stukturo ali polje podatkov. Dostop do take strukture podatkov sicer ne traja zelo dolgo, pa vendar mogoče predolgo, da bi si lahko privoščili njeno izvršitev v enem, časovno nedeljivem, kosu.

Primer: Dostop do datoteke, ki si jo delita dva procesa.

Imamo dva procesa, P_1 in P_2, ki dostopata do skupne datoteke D. Datoteka D bi mogoče vsebovala podatkovno zbirko o razmerah na cestah. Ni izključeno

naslednje neželeno zaporedje operacij:

1. `P1` delno spremeni datoteko `D`,
2. `P2` bere delno spremenjeno datoteko `D`,
3. `P1` dokončno spremeni datoteko in njena vsebina je veljavna.
4. ...

Potencialne težave so očitne. V času spreminjanja datoteke je njena vsebina nekonzistentna. Če bi se spreminjanje datoteke reliziralo od začetka do konca brez prekinitve, torej atomično, do težav ne bi moglo priti.

A spreminjanje datoteke lahko traja kar nekaj časa, morda celo sekundo. Zahteva po atomičnosti operacije je prehuda. Zadostuje, da ustavimo izvajanje procesa `P2` in to le v primeru, da bi zahteval dostop do delno spremenjene datoteke. Sicer pa proces `P2` lahko neovirano napreduje.

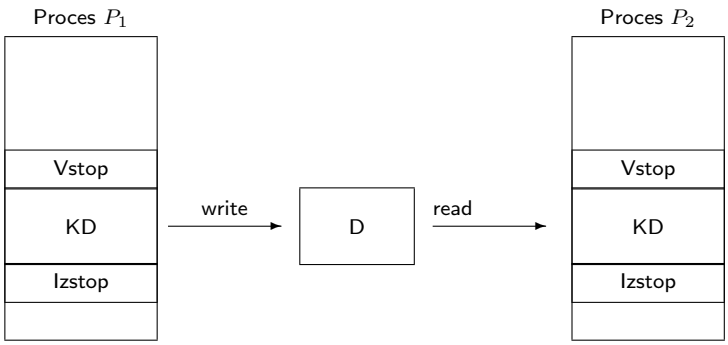
Z drugimi besedami, ko se en proces izvaja v tistem delu, ki posega po datoteki, se drug proces ne sme izvajati v tistem njegovom delu, ki posega po tej isti datoteki, sicer pa se lahko izvaja brez omejitve. Delu programa oziroma procesa, ki dostopa do skupnega sredstva, pravimo kritični del ali '*kritično področje* (Angl. Critical Section, Critical Region). Tedaj, ko proces izvaja zaporedje ukazov, ki posega po skupnem sredstvu pravimo, da se izvaja v kritičnem delu, slika 7.3.

Osnovna zahteva je, da se izvajanje procesov znotraj kritičnega dela med seboj časovno izključuje. Ko se eden izvaja v kritičnem delu, se drugi ne smejo izvajati v svojem kritičnem delu.

7.3.2 Kritično področje

Splošna rešitev problema kritičnega področja (ang. Critical Section Problem) mora zadostiti zahtevi po medsebojnem izključevanju in še nekaterim drugim zahtevam:

1. medsebojno izključevanje v vsakih okoliščinah (Angl. Mutual Exclusion),



Slika 7.3: *Problem dostopa dveh procesov do skupnega sredstva (datoteke). Tedaj, ko proces P_1 dostopa do datoteke in se torej izvaja v kritičnem delu (KD), moramo preprečiti dostop do datoteke procesu P_2 ali povedano drugače, preprečiti mu moramo vstop v kritični del dokler preces P_1 ne zapusti svojega kritičnega dela.*

2. Ne sme priti do zastoja (zastoj je posebno stanje procesa, iz katerega ni regularnega izhoda. Proces je v stanju zastoj (Angl. Deadlock), kadar mu prehod iz stanja zastoj omogoči samo proces, ki je tudi v stanju zastoj).

3. Ne sme priti do nepredvidljivo dolgega odlaganja (vstop v kritični del se določenemu procesu odlaga) (Angl. Indefinite postponement, Starvation).

4. Izvajanje enega procesa znotraj kritičnega dela ne sme ovirati napredovanja drugih procesov izven kritičnega dela.

Rešitev ne sme temeljiti na kakršnikoli predpostavki glede hitrosti napredovanja procesa. Procesi napredujejo z različno in nepredvidljivo hitrostjo. Predpostavljamo le, da se posamezna operacija (ukaz procesorja) izvrši atomično (časovno nedeljivo – ko enkrat začne, se izvrši do konca brez prekinitve).

Pri načrtovanju splošne rešitve bomo za model vzeli naslednjo zgradbo procesa:

P_i /* Proces i */

```
while ( 1 ){      /* Neskončna zanka */
```

```
    Splošni (nekritični) del
    Začetni del - vstop v K.D.
    KRITIČNI DEL (K.D.)
    Končni del - izstop iz K.D.
    Splošni (nekritični) del
}
```

Imejmo dva procesa P_0 in P_1 (P_i, $i = 0, 1$), ki v K.D. dostopata do skupnega sredstva (npr. datoteke). Definiramo skupno spremenljivko t, ki ima naslednji pomen:

- t = 0; v K.D. sme P_0,
- t = 1; v K.D. sme proces P_1.

(V primeru večjega števila procesov, bi imeli krožno dodeljevanje). Naj bo začetna vrednost spremenljivke t enaka nič.

P_i /* Proces i */

```
int t = 0;              /* skupna spremenljivka */
while ( 1 ){      /* Neskončna zanka */
    Splošni (nekritični) del
    while( t != i );      /* Čakaj na dovoljenje za vstop */
    KRITIČNI DEL
    t = j;          /* Dovoli vstop drugemu, j != i */
    Splošni (nekritični) del
}
```

S tem smo zagotovili, da se izvajanje v kritičnem delu izključuje. Rešitev je primerna, kadar se strogo zahteva izmenično izvajanje v K.D. (nejprej prvi, nato drugi, potem spet prvi, i.t.d.), sicer pa to ni splošna rešitev. V primeru, da bi proces P_0 hotel ponovno v K.D., predno gre v kritični del proces P_1, bi to ne bi bilo mogoče.

Prvo splošno rešitev problema K.D. je sicer predlagal Dekker sredi šestdesetih let, bolj pregledno (elegantno) rešitev pa je našel Peterson (1981).

Peterson je vpeljal pomožno tretjo skupno spremenljivko (t), ki v primeru sočasnega vstopa obeh procesov v K.D. dovoli vstop tistemu, ki jo zadnji spremeni. Naslednja zanimiva posebnost rešitve je v tem, da ima vsak proces eno spremenljivko (f), ki jo lahko spremeni samo on, medtem ko jo lahko bereta oba. Do konflikta pri dostopu do spremenljivk f zato ne more priti.

P_i /* Proces i (Petersonova rešitev)*/

```
int f[2] = { 0, 0 }        /* skupni spremenljivki, najprej 0, 0 */
int t = 0;                 /* skupna spremenljivka, 0 ali 1 */
while ( 1 ){               /* Neskončna zanka */
  Splošni (nekritični) del
  f[i] = 1;                /* Prihajam v K.D. */
  t = j;                   /* Dajem prednost drugemu */
  while((f[j]==1) && (t == j));   /* Čakam na pogoj za vstop */
  KRITIČNI DEL
  f[i] = 0;                /* Zapuščam K.D. */
  Splošni (nekritični) del
}
```

Petersonova rešitev deluje za dva procesa, v eno ali večprocesorskem sistemu.

Podpora sinhronizaciji – ukaz TST

Praktično vsak sodoben procesor daje sinhronizaciji procesov osnovno podlago, bodisi z zaklepanjem vodila bodisi z ukazom nalašč za te namene, ki se vedno izvrši časovno nedeljivo. Tak ukaz se običajno poimenuje TST

(TestAnd-Set) ali kako drugače.

Posebnost ukaza TST je v tem, da realizira dostop do pomnilnika tipa R-M-W (Read-Modify-Write) vedno atomočno, to je od začetka do konca brez prekinitve. V večprocesorskem sistemu je v tem času pomnilniško vodilo zaklenjeno. Simbolično bi ukaz TST opisali takole:

```
int TestAndSet( int *Test )
{   int Temp;
    Temp = *Test;
    *Test = 1;
    return Temp;;
}
```

Vrednost spremenljivke (pomnilniške besede) se prebere, testira in nato postavi na vrednost različno od nič (t.j. ena). Uporaba tega ukaza je na primer naslednja:

P_i /* Proces i */

```
int t = 0;              /* skupna spremenljivka */
  while ( 1 ){          /* Neskončna zanka */
    Splošni (nekritični) del
    while( TestAndSet(& t ) != 0 );     /* Čakaj na dovoljenje za vstop */
    KRITIČNI DEL
    t = 0;              /* Dovoli vstop drugemu */
    Splošni (nekritični) del
  }
```

Pri vstopu v K.D. se nedeljivo preveri in postavi vrednost skupne spremenljivke (t). V primeru, da je vrednost t-ja že bila različna od nič, en proces čaka v zanki, da jo drug proces postavi na nič tedaj, ko zapusti K.D. Če pa je vrednost t-ja nič, proces takoj vstopi v K.D., ob tem pa postavi vrednost t-ja na 1.

Podrobna analiza pokaže, da takšna rešitev ne preprečuje nepredvidljivo

dolgega čakanja, zadostuje pa za večino praktičnih primerov.

7.3.3 Semafor

Semafor je pripomoček, ki se uporablja za sinhronizacijo procesov, tipično pri dostopanju dveh ali več procesov do skupnega sredstva, ki ne dopušča sočasnega dostopa. Uporabo semaforja je predlagal Dijkstra sredi šestdesetih let prejšnjega stoletja. Semafor je dejansko celoštevilčna spremenljivka, na kateri sta poleg inicializacije možni dve (časovno nedeljivi – atomični) operaciji: *čakaj* in *javi* (tudi zakleni in odkleni, postavi in spusti, preveri in postavi). Operacijo čakaj (Angl. Wait) bomo označili s P, operacijo javi (Angl. Signal) pa z V. Obe operaciji sta običajno realizirani na nivoju operacijskega sistema (dosegljivi sta torej preko sistemskega klica), trajata malo časa v primerjavi s samim kritičnim delom, ki lahko traja tudi relativno dolgo in se ne izvrši v 'enem kosu'.

V uporabi sta dva tipa semaforjev: binarni in števni. Binarni semafor lahko zavzame samo dve vrednosti, tipično 0 in 1, kar na primer pomeni: skupno sredstvo je/ni prosto. Pravila pri dostopanju procesa do takega sredstva so naslednja. Kadar proces potrebuje sredstvo (na primer datoteko), preveri stanje semaforja z operacijo čakaj. V primeru, da je njegova vrednost večja od nič, zaseže sredstvo ter zmanjša vrednost semaforja na nič. S tem drugim procesom začasno prepreči dostop do sredstva, dokler ga ima v uporabi. Če pa je vrednost semaforja nič, pomeni to, da sredstvo koristi že kakšen drug proces, zato se proces sam postavi v stanje 'čakanja na semaforju'. Kadar proces sredstva več ne potrebuje, ga sprosti oziroma z operacijo *javi* zveča vrednost semaforja za ena. Tako javi, da je sredstvo sprostil in dovoljuje koriščenje sredstva drugim procesom. Števni semafor lahko zavzame vsako nenegativno vrednost. Primer uporabe števnega semaforja bi bil lahko naslednji. Vrednost semaforja pomeni število razpoložljivih sredstev. Dokler je vsaj eno od sredstev na voljo, je procesu, ki ga potrebuje, dostop zagotovljen takoj. Ko vrednost semaforja pade na nič (sredstva so pošla), bo prvi proces, ki zahteva dostop, moral čakati 'na semaforju', dokler se eno od sredstev ne sprosti. Operacijski sistemi, ki realizirajo sistem semaforjev,

včasih dopuščajo 'razširjen' nabor operacij, kot je na primer preverjanje stanja semaforja brez spreminjanja vrednosti.

Operacija P

Za naše potrebe definirajmo nov podatkovni tip `Sem`, ki ustreza definiciji semaforja. Operacija P omogoča ekskluziven vstop v kritični del in bi v C-ju podobnem zapisu lahko izgledala takole (To nikakor ne pomeni, da se jo tako tudi realizira. Sistem samaforjev je skoraj vedno realiziran na nivoju operacijskega sistema. Kako pa bi vi realizirali semafor?).

```
P( Sem *Semafor )
{
   while( *Semafor <= 0);    /* Čakaj, da se Semafor postavi */
   (*Semafor)- -;
}
```

Operacija V

Operacija V javlja, da proces zapušča kritični del.

```
V( Sem *Semafor )
{
   (*Semafor)++;
}
```

Nekaj primerov

Rešitev problema K.D. s semaforjem je enostavna in pregledna:

P_i /* Proces i */

```
Sem Semafor = 1;      /* Semafor - inicializacija */
  while ( 1 ){      /* Neskončna zanka */
    Splošni (nekritični) del
    P( & Semafor);      /* Čakaj na dovoljenje za vstop */
```

```
    KRITIČNI DEL
    V( & Semafor);        /* Dovoli vstop drugemu */
    Splošni (nekritični) del
}
```

Primeri uporabe semaforja so številni. Denimo, da se mora operacija O_α v procesu P_a realizirati v vsakem primeru šele potem, ko se relalizira operacija O_β v procesu P_b. Možna je naslednja rešitev (semafor S je inicializiran na 0):

P_a	Proces P_b
...	...
...	...
P(& S)	...
O_α	O_β
...	V(& S)
...	...
...	...

V primeru, da P_a 'prehiteva', bo moral čakati na semaforju, dokler se v procesu P_b ne izvrši operacija O_β in to javi z V(& S). V nasprotnem primeru, ko P_a 'zaostaja', se bo operacija O_α izvršila takoj, brez čakanja.

Za izmeničen dostop procesov P_0 in P_1 do datoteke D sta potrebna dva semaforja, rešitev s semaforji pa je enostavna, pregledna in izgleda takole:

```
Proces P0                    Proces P1

  while(1){                    while( 1 ){
    P( & S1 )                    P( & S0 )
    K.D.                         K.D.
    V( & S0 )                    V( & S1 )
  }                            }
```

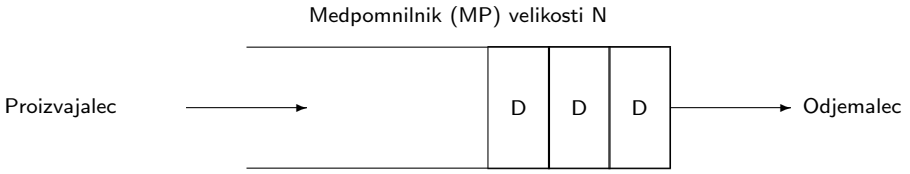
Za vajo pa v zgoščeni obliki zapišite krožen dostop do skupnega sredstva s pomočjo semaforjev za N procesov.

7.3.4 Klasični primeri medprocesne sinhronizacije

Problem končnega medpomnilnika

Problem medpomnilnika končne velikosti (ang. Bounded Buffer Problem) je poznan tudi kot problem proizvajalca in odjemalca ali porabnika (ang. Producer Consumer Problem). Eden od procesov, t.i. proizvajalec, proizvaja podatke in drug proces oziroma odjemalec jih uporabi. V pogledu medpomnilnika to pomeni, da eden od procesov pripravi podatek in ga da v pomnilnik, drug proces (pre)vzame podatek iz pomnilnika in s tem sprosti pomnilnik. V primeru, da bi bil medpomnilnik dovolj (neskončno) velik in bi proizvajalec pripravljal hitreje kot jih odjamalec lahko prevzema, bi oba procesa vedno napredovala neodvisno eden od drugega. Končna velikost medpomnilnika in nepredvidljiva hitrost obeh procesov zahteva občasno sinhronizacijo,

- če je medpomnilnik poln, mora proizvajalec počakati, da odjemalec prevzame (vsaj en) podatek,
- če je medpomnilnik prazen, potem čaka odjemalec, dokler proizvajalec ne pripravi (vsaj en) podatek.



Slika 7.4: *Ponazoritev problema končne velikosti medpomnilnika.*

Uvedemo dva semaforja in pomožnega tretjega:

- semafor Poln inicializiramo na N. Ko pade na nič ustavimo proizvajalca, ker to pomeni, da je medpomnilnik poln.
- Semafor Prazen inicializiramo na nič. Dokler in kadar je njegova vrednost nič, ustavimo odjemalca, ker je medpomnilnik prazen.
- Kadar proizvajalec piše v medpomnilnik, moramo odjemalcu pre-

prečiti dostop. Zato uvedemo tretji semafor; naj bo ta semafor označen z Mux.

Proizvajalec

```
Sem Poln = N;      /* Semafor - inicializacija na velikost MP */
Sem Prazen = 0;    /* Semafor - v začetku MP prazen */
while ( 1 ){       /* Neskončna zanka */
  Pripravi podatek D
  P( & Poln );     /* MP poln ? */
  P( & Mux );
  Vstavi podatek D v medpomnilnik
  V( & Mux );
  V( & Prazen);    /* V MP je en podatek več*/
  Splošni (nekritični) del
}
```

Odjemalec

```
Sem Poln = N;      /* Semafor - inicializacija na velikost MP */
Sem Prazen = 0;    /* Semafor - v začetku MP prazen */
while ( 1 ){       /* Neskončna zanka */
  Pripravi podatek D
  P( & Prazen );   /* MP prazen ? */
  P( & Mux );
  Vzemi podatek D iz medpomnilnika
  V( & Mux );
  V( & Poln);      /* V MP je en podatek manj */
  Splošni (nekritični) del
}
```

Problem branja in pisanja

Problem branja/pisanja (ang. Readers/Writers Problem) je zastavljen takole:

- procesi (P_{W_i}, $i = 0, 1, \ldots, I$) pišejo v skupen pomnilnik (ali datoteko),
- procesi (P_{R_j}, $j = 0, 1, ..., J$) iz pomnilnika berejo.

Nekateri ali vsi procesi lahko nastopajo v obeh vlogah. Ko eden od procesov P_{W_i} piše, je prepovedano pisanje in branje za vse ostale. Nasprotno, ko eden od procesov P_{R_j} bere, lahko sočasno bere poljubno število drugih bralnih procesov, pisanje procesom P_{W_j} pa je tedaj prepovedano.

Možna bi bila naslednja rešitev:

P_{W_i}

```
Sem Write;      /* Semafor, ki prepreči konflikt pri pisanju */
while ( 1 ){    /* Neskončna zanka */
  Pripravi podatek D
  P( & Write );     /* Še kdo piše ? */
  Vpiši podatek D
  V( & Write );       /* Dovoli vstop drugim */
  Splošni (nekritični) del
}
```

P_{R_j}

```
Sem Write;      /* Semafor, ki prepreči konflikt pri pisanju */
Sem Mux;      /* Pomožni semafor za koordinacijo bralnih procesov */
int r = 0;      /* Števec aktivnih bralnih procesov */
while ( 1 ){    /* Neskončna zanka */
  P( & Mux );
  r++;      /* Štejemo zahteve po branju */
```

```
    if( r == 1) P( & Write );     /* Če sem prvi bralec, preverim pisalce */
    V( & Mux ); /* Dovolim branje drugim bralnim procesom */
    Preberem podatek D
    P( &Mux );    /* To je potrebno, ker spreminjam r */
    r- -;      /* Odštevamo zahteve po branju */
    if( r == 0) V( & Write );     /* Sem zadnji bralec */
    V( &Mux );
    Splošni (nekritični) del
}
```

Problem petih mislecev

Slika 7.5: *Filozofi pri kosilu.*

Problem petih filozofov pri kosilu (Angl. Dinning Philosophers Problem) si je izmislil Dijkstra, da bi služil kot referenčni primer za preizkušanje algoritmov za medprocesno sinhronizacijo. Definicija problema je naslednja. Za mizo sedi pet filozofov ('procesov'). Vsak filozof ima svoj krožnik, med krožniki so položene vilice. Na mizi je torej pet krožnikov in pet vilic – sosednja filozofa si delita vilico.

Filozofi neodvisno eden od drugega nekaj časa razmišljajo in potem nekaj časa jedo. Ko filozov jé, potrebuje dve vilici, po eno za vsako roko. Zato seže po vilicah. Če se uspe polastiti obeh vilic, začne jesti. Njegova neposredna soseda ta čas ne moreta jesti. Sočasno lahko potem jesta največ dva filozofa. V primeru, da bi skušali hkrati jesti vsi filozofi, bi lahko prišlo do zastoja. Na primer, vsak od filozofov vzame desno ležeče vilico, nato se skuša polastiti leve, kar mu ne uspe. Zato čaka, da sosed sprosti vilice, vendar do tega ne pride, ker tudi on čaka in zastoj je tu. Potrebna je sinhronizacija.

Naša naloga je poiskati splošno rešitev problema, ki bi ne peljala v zastoj, pa da filozofi ne bi stradali. Predlagajte in komentirajte vsaj dve rešitvi

problema.

Poglavje 8

Realni čas in razvrščanje opravil

V tem razdelku se bomo na kratko seznanili s sistemi realnega časa. Spoznali bomo, kakšne vrste sistemov realnega časa poznamo in kje jih najdemo. Navedli bomo bistvene lastnosti računalniških sistemov za realni čas. Največ pa bomo govorili o razvrščanju opravil in o vprašanjih s tem v zvezi.

8.1 Realni čas

Realni čas je preprosto čas, v katerem živimo. Ko govorimo o sistemih realnega časa, imamo v mislih računalniške sisteme z določenimi lastnostmi, ki jih vsi računalniški sistemi nimajo. Računalniški sistem realnega časa spoznamo po tem, da je njegovo delovanje neposredno povezano z realnim svetom. Tak sistem zaznava dogajanja v okolju in se nanje sproti odziva.

Delovanje v realnem času je lahko pogojeno z veliko količino podatkov, ki jih je treba obdelati *pravočasno*. Takšni sistemi so na primer multimedijski sistemi. Še večkrat je delovanje v realnem času vezano na redke kratkotrajne dogodke, na katere se mora sistem odzivati pravočasno in *predvi-*

dljivo. Predvidljivo pomeni, da je odzivni čas zajamčen in znan vnaprej. Takšne primere najdemo recimo v avtomobilski industriji. So pa še sistemi, ki so nekje vmes in v sebi združujejo oboje, velike količine podatkov in redke pomembne dogodke. Takšne sisteme najdemo med umetnimi inteligentnimi sistemi.

Za sisteme realnega časa je pomembno, da delujejo pravilno, predvidljivo ter v okviru vnaprej predpisanih časovnih zahtev. Rezultat obdelave, ki ni na razpolago ob pravem času, ni uporaben, tudi če bi bil pravilen. Na primer, varnostna vreča v avtomobilu je potrebna le redkokdaj, a če je potrebna, mora biti aktivirana ob pravem času, niti ne prej niti ne kasneje.

Sisteme realnega časa razlikujemo po tem, kako striktne so zahteve glede predvidljivosti. Na primer, v transakcijskih sistemih, kot je denimo elektronsko bančništvo, ni bistven takojšen odziv. Važen je pravilen odziv v doglednem času. Odzivnost sistema se lahko s časom spreminja in občasno sme celo preseči predpisani skrajni čas. Zakasnjeni odziv sicer vpliva na zadovoljstvo uporabnika, vendar občasno kršenje odzivnega časa nima usodnih posledic. Enako bi lahko rekli za upravljanje avtomobilskega okna. Sisteme, v katerih smejo biti nekatere časovne zahteve občasno kršene, imenujemo sistemi realnega časa z mehkimi oziroma *ohlapnimi* omejitvami (Angl. Soft real-time systems). Tudi multimedijske sisteme bi lahko uvrstili med te sisteme. Občasna slabša kakovost avdio ali video vsebin nima katastrofalnih posledic, čeprav je za uporabnika lahko to neprijetna izkušnja.

Časovno kritični sistemi, v katerih nobena časovna zahteva ne sme biti nikoli kršena, imenujemo sistemi realnega časa s trdimi oziroma *strogimi* omejitvami (Angl. Hard real-time systems). Tipični časovno kritični sistemi so sistemi za vodenje in nazor kemijskih ali fizikalnih procesov, proizvodnih sistemov, transportnih sistemov, sistemov distribucije, vojaških sistemov, itd. Ni vseeno, kakšen je odzivni čas sistemov za upravljanje avtomobilskih zavor, varnostnih vreč, ali kemijskega reaktorja.

Ni treba, da bi bili sistemi realnega časa zelo hitri, čeprav se jih običajno ima za zelo hitre. A za sistem realnega časa je bolj pomembna odzivnost kot prepustnost. Hitrost sistema se torej kaže predvsem v kratkih odzivnih

časih in manj v številu operacij ali količini podatkov, ki jih je zmožen obdelati.

Večina sistemov ali podsistemov realnega časa je namenskih sistemov in večina jih spada med vgradne sisteme. Veliko vgradnih sistemov za realni čas ne potrebuje operacijskega sistema. V sistemu, ki delujejo na podlagi operacijskega sistema, pa mora podporo za delovanje v realnem času zagotoviti operacijski sistem (RTOS - Real Time Operating System).

Operacijski sistemi za splošne namene praviloma niso sistemi realnega časa. Vzroke za to najdemo v tehnološkem razvoju. Operacijski sistemi so se pač začeli razvijati tedaj, ko so bili računalniki še zelo dragi. Izkoriščenost sistema je bila posledično bistveno važnejša kot njegova odzivnost ali predvidljivost. Tudi sicer so se računalniki sprva uporabljali za *paketno* (Angl. Batch) in *posredno* (angl. Off-line) obdelavo, kar že v samem načelu ni bilo neposredno vezano na realni čas. Potrebe po odzivnosti sistema so se stopnjevale z razvojem tako imenovanih *interaktivnih* sistemov in s konceptom neposredne (Angl. On-line) obdelave. Resnične potrebe po operacijskih sistemih za realni čas pa so se pojavile z uporabo računalnikov v laboratorijske namene ter z vključitvijo računalnika direktno v vodeni proces.

Podporo delovanju v realnem času dajejo operacijski sistemi, ki so bili razviti nalašč za ta namen, ali pa so bili prilagojeni oziroma nadgrajeni s funkcionalnostjo, ki jo zahteva delovanje v realnem času. Primer zelo priljubljenega sistema za realni čas je VxWorks. Splošno znani sistemi tipa RTOS so še FreeRTOS, eCos, LynxOS, uOS, OSEK ter drugi. Sistem UNIX ni bil zasnovan za delovanje v realnem času. Tudi sistem Linux sprva ni bil načrtovan za časovno kritične procese. Podpora sistemov UNIX/Linux za realni čas je prišla kasneje. Nadgradnji sistema Linux za realni čas sta na primer RTLinux in RTAI. Novejše verzije Linux jedra dajejo podporo aplikacijam v realnemu času.

Tipični poudarki operacijskih sistemov za delo v realnem času so:

- večopravilnost, prioriteno razvrščanje opravil (Angl. Multitasking, Priority-based scheduling),

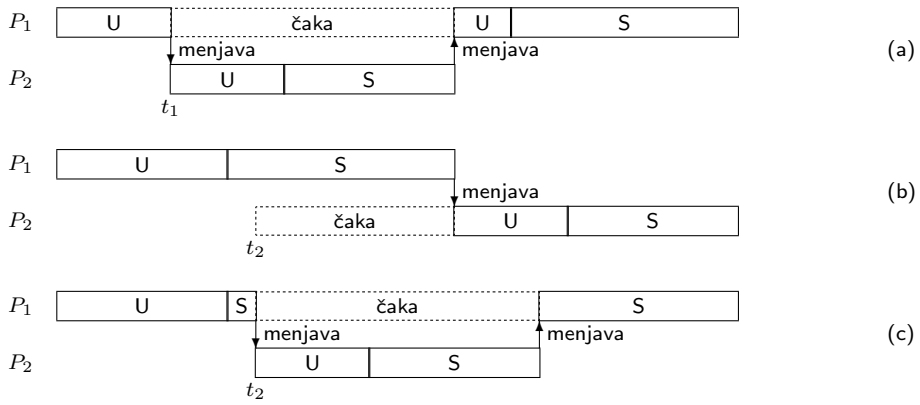
- večnitnost in prioritetno razvrščanje niti (Angl. Multithreading),
- prednostno razvrščanje ali *predopravilnost* (Angl. Preemptive scheduling) ter zlasti predopravilnost v okviru sistemskega jedra (Angl. Preemptive kernel),
- primerno fina delitev časa in primerno fino časovno razvrščanje opravil. To vključuje:
  - enkratno izvršitev opravila v točno določenem trenutku (absolutni čas),
  - enkratno izvršitev opravila po natanko določenem intervalu (relativno na dani trenutek),
  - periodično izvrševanje opravil, z nastavljivim intervalom do prve izvršitve, točno določene zakasnitve.
- podpora medprocesnim komunikacijam, deljenemu pomnilniku, in sinhronizaciji opravil,
- asinhrono odzivanje na zunanje dogodke z vnaprej znanimi kasnitvami,
- zaklepanje procesov v pomnilniku (Angl. Memory locking).

Operacijski sistem, ki ima zgoraj navedene lastnosti, daje podlago aplikacijam realnega časa. Kot rečeno je poglavitna lastnost sistemov realnega časa pravočasen in predvidljiv odziv. To nalogo si enakovredno delita aplikacija in jedro, a brez ustrezne podpore jedra bi bila aplikacija nemočna.

Slika 8.1: *Primer dveh procesov. P_1 se izvaja 60 časovnih enot v uporabniškem načinu (U) in 80 enot v jedru, to je v sistemskem načinu (S). P_2 se izvaja 40 časovnih enot v uporabniškem načinu (U) in 60 enot v jedru (S).*

8.1.1 Predopravilnost in jedro

Eden glavnih poudarkov v operacijskih sistemih za realni čas je vprašanje odzivosti zlasti tedaj, ko napredovanje procesa poteka v sistemskem jedru. Za splošnonamenske operacijske sisteme je značilno, da se procesa tedaj, ko napreduje v sistemskem jedru, ne da prekiniti. Prevzem procesne enote s strani drugega procesa ni mogoč, četudi bi imel višjo prioriteto od trenutno izvajanega procesa. Jedro ne obravnava prednostno zahtev z višjo prioriteto, jedro ni 'predopravilno'. Posledično lahko napredovanje procesa z nižjo prioriteto v sistemskem jedru traja zelo dolgo. Še slabše pa je, da lahko odlaganje procesa z višjo prioriteto traja nepredvidljivo dolgo. Zato, da je operacijski sistem primeren za delovanje v realnem času, mora predopravilnost omogočati tudi sistemsko jedro. Odzivnost in menjava procesov v uporabniškem načinu s stališča operacijskega sistema je manj kritična ali ni kritična. Učinek predopravilnosti v uporabniškem načinu in nepredopravilnosti v sistemskem načinu prikazujeta za primer procesov s slike 8.1 sliki 8.2.a in 8.2.b. Predopravilnost jedra ponazarja slika 8.2.c.



Slika 8.2: *(a) Predopravilnost v uporabniškem načinu. Proces P_2, ki ima višjo prioriteto od P_1, pride v trenutku t_1 in prevzame procesno enoto. (b) Nepredopravilno jedro. Proces P_2 pride v trenutku t_2, ko P_1 teče v jedru. P_2 čaka, kljub višji prioriteti. (c) Predopravilno jedro. Proces P_2 pride v trenutku t_2 in takoj prevzame procesno enoto.*

Ključna parametra operacijskega sistema za realni čas sta odzivni čas na zahtevo za prekinitev (Angl. Interrupt latency) in čas za menjavo niti in/ali

procesov (Angl. Thread switching latency, Context switching latency). Da je operacijski sistem predvidljiv, mora biti za vsak sistemski klic vnaprej znano, koliko časa traja njegova izvršitev.

8.2 Razvrščanje opravil

Glavna komponenta operacijskega sistema za delo v realnem času je razvrščevalnik procesov oziroma opravil in razvrščevalnik niti (Angl. Scheduler). Razvrščevalnik je zadolžen za večino nalog, ki smo jih navedli v prejšnjem poglavju.

Razvrščevalnik sestavlja časovni razpored opravil in dodeljuje procesno enoto. Ko razvrščevalnik odzvame procesor enemu procesu ter ga dodeli drugemu procesu pravimo, da opravi menjavo procesov oziroma *pomenski preklop* (Angl. Context Switch). Pomenski preklop mora biti kar se da hiter.

Razvrščevalnik je del sistemskega jedra in zato, da opravlja delo, potrebuje procesno enoto. Sestavljanje razporeda predstavlja dodatno breme sistema, za katerega želimo, da bi bilo čim manjše. Zato implementacija razvrščevalnika praviloma realizira enostavna pravila in algoritme razvrščanja.

Vprašajmo se še, kdaj je najbolj primerno ali potrebno, da nastopi razvrščevalnik? Ti trenutki ali dogodki nastopijo ob prehodu med stanji procesa in so navedeni spodaj.

- Nastanek procesa. Tedaj, ko nastane nov proces, postane pripravljen in se uvrsti na seznam pripravljenih procesov. Razvrščevalnik v seznamu pripravljenih procesov poišče proces z najvišjo prioriteto in mu dodeli procesor.
- Konec procesa. Ker procesa ni več, se procesor dodeli drugemu pripravljenemu procesu. Tistemu z najvišjo prioriteto.
- Blokada procesa zaradi v/i prenosa, semaforja ali drugega vzroka. Proces nima pogojev za napredovanje, zato se procesor dodeli nasle-

dnjemu pripravljenemu procesu.

- Odblokada procesa, ker je v/i prenos končan, semafor odklenjen ali spirčo kakšnega drugega vzroka. Proces se uvrsti na seznam pripravljenih procesov in glede na prioriteto se procesor dodeli naslednjemu pripravljenemu procesu.
- Zahteva za prekinitev s strani v/i naprave. Spričo tega lahko postane pripravljen proces, ki je bil dotlej v stanju ustavljen in glede na prioriteto se mu lahko dodeli procesno enoto, ali pa se le-ta dodeli drugemu procesu z višjo prioriteto.
- Zahteva za prekinitev iz časovnika, ker je potekel dodeljeni ali dovoljeni čas. To je eksplicitna zahteva za menjavo procesov.

Od razvrščevalnika opravil za realni čas se pričakuje predvsem obravnavanje opravil v skladu z njihovo pomembnostjo oziroma prioriteto ter predopravilnost. V primeru enakovrednih opravil se od razvrščevalnika pričakuje poštenost. Seveda ostaja odprto vprašanje kako dodeljevati prioritete. To vprašanje pa spada v domeno aplikacije.

V nadaljevanju si bomo ogledali nekatere najpogostejše pristope k razvrščanju, med drugim:

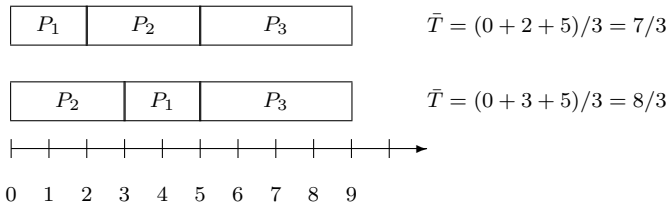
- prvi pride prvi strežen (Angl. First come first served (FCFS) scheduling, ali FIFO),
- krožno razvrščanje (Angl. Round-robin (RR) time sharing scheduling),
- krajše opravilo prej (Angl. Shortest job first (SJF)),
- pogostejše opravilo prej (Angl. Rate monotonic (RM) scheduling),
- najkrajši ali najzgodnejši skrajni rok najprej (Angl. Shortest deadline first - SDF, Earliest Deadline First - EDF).

8.2.1 Krožno razvrščanje opravil

V sistemu s krožnim razvrščanjem dobi vsak proces na razpologo enak in vnaprej predpisan časovni interval, 'kvant' oziroma rezino (Angl. Time slice), na primer v trajanju nekaj deset milisekund. V primeru, da proces porebuje manj časa, se procesor takoj dodeli naslednjemu procesu. Če pa proces, ki se izvaja, po preteku dodeljenega časa še ni končan, se mu procesna enota odvzame, možnost za napredovanje dobi naslednji proces in tako dalje v krožnem zaporedju. Prekinjeni proces se uvrsti na konec (krožnega) seznama, torej dobi možnost za napredovanje šele potem, ko se pred njim zvrstijo vsi ostali procesi. Uspešnost razvrščanja je bistveno odvisna od dolžine časovnega intervala ali 'kvanta'. Če je interval dolg, se RR razvrščanje obnaša podobno kot FCFS. Če pa je kvant razmeroma kratek, se sistem z N procesi obnaša podobno kot bi imeli N počasnejših procesorjev s taktom 1/N plus dodatno breme za menjavo procesov oziroma pomenski preklop (Angl. Context switch). Čas, ki je potreben za menjavo (tipično nekaj deset mikrosekund), zmanjšuje pretočnost sistema, zato mora biti v primerjavi s časovnim kvantom majhen ali kar zanemarljiv. To omejuje dolžino intervala navzdol. Po drugi strani daljšanje časovnega kvanta zmanjšuje odzivnost. Praktično pravilo v splošno namenskih sistemih pravi, naj bo časovni kvant tak, da bo približno 80% zahtev končanih v enem kvantu ali prej. Razvrščanje RR je deterministično. Vsak od N procesov čaka največ $N - 1$ časovnih intervalov da pride na vrsto. V primeru enakovrednih procesov je tak način razvrščanja smiseln. Na primer, imamo 24 procesov, od katerih vsak rabi eno uro da se konča. Če vsi procesi začnejo opolnoči ter jim dodeljujemo procesor krožno po načelu delitve časa, bodo na koncu dneva v zadnjih desetinkah sekunde zaključeni vsi procesi. Kaj pa, če bi moral biti eden od procesov opravljen do 12 ure? V tem primeru bi morali temu procesu dodeliti večprocesorskega časa na račun drugih procesov, dvigniti bi mu morali 'prioriteto'.

8.2.2 Krajše opravilo prej

Pri razvrščanju po pravilu 'krajše opravilo prej'(SJF) dobi procesno enoto najprej najkrajši proces, nato naslednji najkrajši ter nazadnje najdaljši proces. Dejansko gre za obliko prioritetnega razvrščanja, pri čemer je prioriteta procesa obratno sorazmerna njegovemu trajanju. Seveda ni vedno koristno, da bi proces s krajšim trajanjem imel tudi višjo prioriteto. Poglavitna lastnost razvrščanja SJF je, da procesom v povprečju zagotavlja najkrajši čakalni čas. Grafično ponazoritev te lastnosti ponazarja slika 8.3. Neželen stranski učinek razvrščanja SJF je 'stradanje' dalgotrajnih procesov. SJF v kombinaciji s krožnim razvrščanjem nastane razvrščaje po pravilu 'najkrajši preostanek procesa prej'(Angl. Shortest remaining time next).



Slika 8.3: *Razvrščanje po pravilu 'krajše opravilo prej' in povprečni čakalni čas. Razvrščanje SJF zagotavlja najkrajši povprečni čakalni čas.*

8.2.3 Priotitetno razvrščanje

V sistemih s prioritetnim razvrščanjem imamo končno število prioritetnih nivojev (na primer 32 ali več) in vsakemu procesu oziroma opravilu je prirejena prioriteta. Procesi si delijo procesno enoto na osnovi dodeljenih prioritet. V sistemu s predopravilnostjo je zagotovljeno, da se bo v vsakem trenutku izvajal proces z najvišjo prioriteto. Če se prioriteta procesom med napredovanjem ne spreminja, govorimo o razvrščanju s fiksnimi prioritetami. Denimo, da imamo dva procesa, P_1 in P_2, ki naj se periodično ponavljata, prvi s periodo $T_1 = 20$ časovnih enot in drugi s periodo $T_2 = 50$ enot. Oba procesa sta sproščena (Angl. Released) oziroma pripravljena (Angl. Ready-to-run ali Runable) v istem trenutku. Zahtevamo, da se

vsak od obeh procesov izvrši do konca pred naslednjo ponovitvijo, skrajna roka sta kar enaka periodama, $D_1 = T_1 = 20$ in $D_2 = T_2 = 50$. Prvi proces naj za izvršitev potrebuje $C_1 = 10$ enot in enako drugi proces, $C_2 = 10$. Prvi proces obremenjuje procesor petdeset odstotno, $U_1 = C_1/T_1 = 0,5$, medtem ko je delež bremena drugega procesa $U_2 = C_2/T_2 = 0,2$. Skupna stopnja bremena je

$$U = U_1 + U_2 = 0,5 + 0,2 = 0,7,$$

kar je manj od ena oziroma manj kot 100%. Skupna stopnja bremena je v resnici merilo za delež časa v ponovitvenem ciklu, ko je procesor obremenjen. Za naš primer znaša dolžina ponovitvenega cikla 100 enot, kar je najmanjši skupni mnogokratnik vseh (obeh) period. V naslednjem ponovitvenem ciklu se vzorec opravil ponovi. Torej za naš primer,

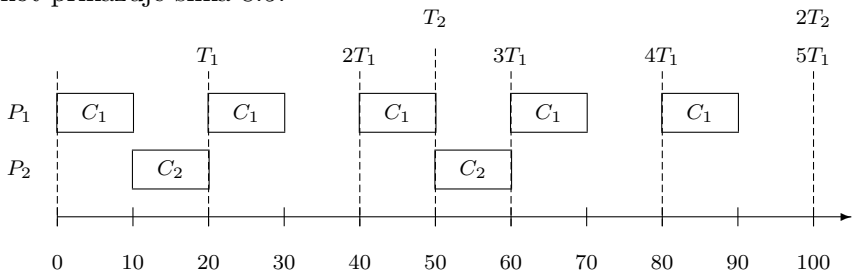
$$U = \frac{5.10 + 2.10}{100} = 0,7.$$

Pri maksimalni stopnji bremena je procesor v rabi cel čas. Iz tega sledi, da pri stopnji bremena $U > 1$ vseh procesov ni moč izvršiti, ne glede na pravilo razvrščanja.

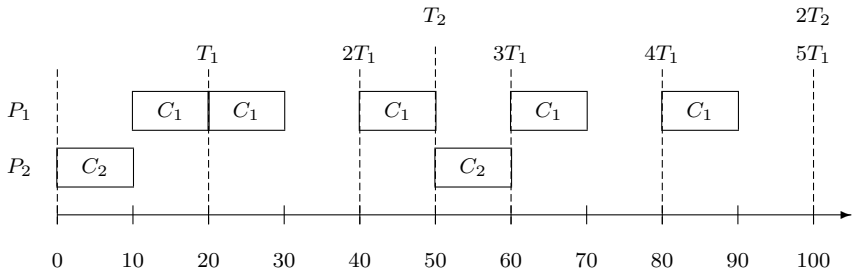
V našem primeru se zdi nekako samoumevno, da bo moč izvrševati oba procesa, saj je skupno breme manjše od 100% in brez časovnih omejitev bi dejansko tudi bilo tako, a ne smemo pozabiti na skrajne roke porcesov. Dodatno breme zaradi razvrščanja in menjave procesov zanemarimo.

Dajmo procesu P_1 višjo prioriteto kot P_2, kar mu zagotavlja, da dobi procesor prej. Razmere prikazuje slika 8.4. Po času 10 enot in torej pred rokom ($T_1 = 20$) je prvi proces obdelan, procesor se dodeli drugemu procesu za 10 enot in torej tudi proces P_2 konča pred rokom 50 enot. Prvi proces ponovno prevzame procesor za naslednjih 10 enot ter spet konča pred rokom, nakar je procesna enota prosta. Nato ponovno pride proces P_1 in se izvrši. Zaključimo, da sta skrajna roka za oba procesa spoštovana, procesa 'razvrstljiva' ter delovanje v realnem času zagotovljeno.

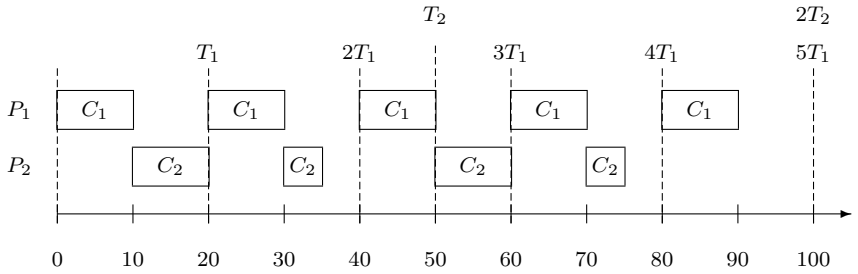
Sedaj pa dodelimo višjo prioriteto procesu P_2. V teh razmerah se začne prvi izvrševati proces P_2, ki konča po 10 časovnih enotah, torej pred rokom ($T_2 = 50$). Proces P_2 po 10 enotah sprosti procesor, ki ga zaseže proces P_1. Proces P_1 sedaj napreduje naslednjih 10 enot ter konča ravno v trenutku, ko mu poteče skrajni rok. Skrajna roka nista kršena, procesa sta razvrstljiva, kot prikazuje slika 8.5.



Slika 8.4: *Razvrstitev dveh opravil,* $T_1 = 20$, $T_2 = 50$, $C_1 = 10$, $C_2 = 10$, $U = 0,7$, *prioriteta*(P_1) > *prioriteta*(P_2).



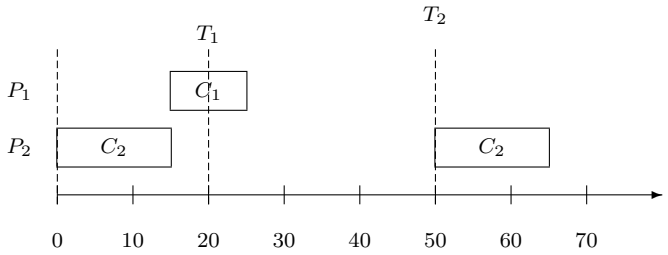
Slika 8.5: *Razvrstitev dveh opravil,* $T_1 = 20$, $T_2 = 50$, $C_1 = 10$, $C_2 = 10$, $U = 0,7$, *prioriteta*(P_2) > *prioriteta*(P_1).



Slika 8.6: *Razvrstitev dveh opravil,* $T_1 = 20$, $T_2 = 50$, $C_1 = 10$, $C_2 = 15$, $U = 0,8$, *prioriteta*(P_1) > *prioriteta*(P_2).

Procesa bi bila po pravilu prioriteta P_1 > prioriteta P_2 razvrstljiva tudi, če bi kateri od procesov potreboval za izvršitev več časa, na primer $C_2 = 15$

enot, kot prikazuje slika 8.6. V tem slučaju bi bilo skupno breme 80% . Vendar, če prioriteti obrnemo, prioriteta P_2 > prioriteta P_1, bi bil skrajni rok procesa P_1 kršen. Razmere prikazuje slika 8.7).



Slika 8.7: *Razvrstitev dveh opravil,* $T_1 = 20$, $T_2 = 50$, $C_1 = 10$, $C_2 = 15$, $U = 0,8$, *prioriteta*(P_2) > *prioriteta*(P_1).

Očitno ni vseeno, kako procesom dodelimo prioritete. V splošnem bi bil skrajni rok procesa P_1 kršen, če bi bil čas za obdelavo obeh procesov skupaj daljši od skrajnega roka procesa z nižjo prioriteto, v slednjem primeru T_1. Da bi bila procesa ob višji prioriteti procesa z daljšo periodo razvrstljiva, bi potem moralo veljati

$$C_1 + C_2 \leq T_1. \tag{8.1}$$

V primeru pa, da ima pogostejši proces (P_1) višjo prioriteto kot P_2 (sliki 8.4,8.6), pride v času T_2 proces P_2 enkrat na vrsto, medtem ko se P_1 v tem času v celoti zvrsti $\lfloor T_2/T_1 \rfloor$-krat. Zatorej mora za razvrstljivost veljati (potreben pogoj):

$$\lfloor T_2/T_1 \rfloor C_1 + C_2 \leq T_2. \tag{8.2}$$

Gotovo velja tudi:

$$\lfloor T_2/T_1 \rfloor T_1 \leq T_2. \tag{8.3}$$

Iz neenakosti (8.1) ter (8.3) sledi:

$$\lfloor T_2/T_1 \rfloor (C_1 + C_2) \leq \lfloor T_2/T_1 \rfloor T_1 \leq T_2. \tag{8.4}$$

Ker je $\lfloor T_2/T_1 \rfloor \geq 1$, velja:

$$\lfloor T_2/T_1 \rfloor C_1 + C_2 \leq \lfloor T_2/T_1 \rfloor C_1 + \lfloor T_2/T_1 \rfloor C_2 \leq \lfloor T_2/T_1 \rfloor (C_1 + C_2) \leq \lfloor T_2/T_1 \rfloor T_1 \leq T_2.$$

Iz povedanega sledi: če so opravila razvrstljiva na drugačen način kot po padajočih pogostostih, bodo gotovo razvrstljiva tudi po padajočih pogostostih. Obratno pa ne drži. Povejmo še, da je obravnavani primer posebne

narave. V splošnem (za poljubno kombinacijo bremen in period procesov) razvrstitev pri 100% bremenu namreč ne obstaja. Izkaže se, da je za razvrstljivost procesov s fiksnimi prioritetami zgornja meja skupnega bremena precej nižja.

8.2.4 Pogostejše opravilo prej

Razvrščanje po pravilu 'pogostejše opravilo prej'(Angl. Rate monotonic – RM) je pomembno za sisteme v realnem času. V sistemih realnega časa je velika večina opravil periodičnih, ali pa se da njihovo pogostost vsaj oceniti. Vzemimo torej, da imamo N periodičnih opravil,

$$P_i, (i = 1, ..., N),$$

s ponovljivostmi

$$T_1 < T_2 < \ldots < T_i < ... < T_N$$

ter potrebnimi časi procesorja

$$C_1, C_2, ..., C_i, ..., C_N.$$

Delež bremena opravila P_i je torej $U_i = C_i/T_i$ (Angl. Processor utilization). Kadar perioda ni hkrati skrajni rok za izvršitev opravila, imamo za vsak proces podan še skrajni rok,

$$D_i, (i = 1, ..., N).$$

Velja,

$$0 < C_i < D_i \leq T_i.$$

Že v prejšnem razdelku smo se na primeru prepričali, da sta procesa 'razvrstljiva', če pripišemo procesu s krajšo periodo višjo prioriteto, v nasprotnem primeru pa ne. Razvrščanje po pravilu 'pogostejše opravilo prej' priredi vsakemu opravilu fiksno (statično) prioriteto, ki je obratna s trajanjem periode – ponovljivostjo. Razvrščanje in razvrstljivost po pravilu RM je pomembno, saj velja naslednje pravilo: če opravila niso RM razvrstljiva,

potem niso razvrstljiva z nobenim algoritmom na osnovi statičnih prioritet. Intuitivno bi sklepali, da mora za ravrstljivost veljati pogoj

$$U = \sum_{i=1}^{N} U_i \leq 1.$$

Liu in Layland [1] pa sta pokazala, da za razvrstljivost v sistemu fiksnih prioritet razvrstitev vedno obstaja, če velja:

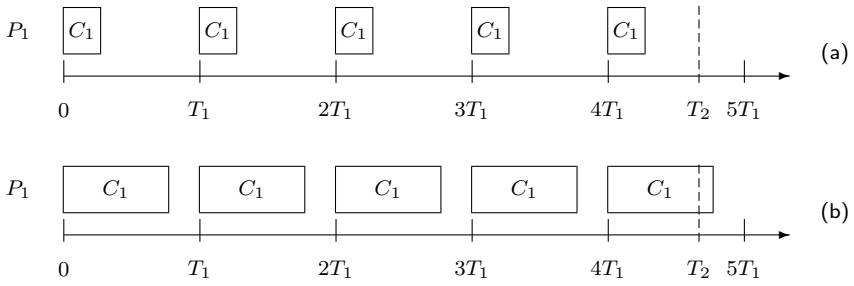
$$U = \sum_{i=1}^{N} U_i = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N(2^{1/N} - 1). \tag{8.5}$$

Z N maksimalna mogoča stopnja bremena U, ki še zagotavlja razvrstljivost, monotono pada in gre za velik N proti $ln(2)$ oziroma $\approx 70\%$. Pri nižji stopnji bremena bo množica N opravil razvrstljiva, ne glede na dejanske vrednosti bremen C_i in ponovljivosti T_i. To je pomembna ugotovitev ne le s teoretičnega, temveč tudi ali predvsem iz praktičnega vidika. Namreč, za dano množico opravil preverimo pogoj (8.5) in če je izpolnjen, bodo opravila gotovo razvrstljiva. S tem je delovanje sistema v realnem času zagotovoljeno. Procesor z opravili realnega časa ni polno obremenjen, preostanek časa je na voljo za druga, neperiodična opravila brez strogih časovnih zahtev. Neenačba (8.5) postavlja zadosten pogoj, ki pa ni tudi potreben, a razvrstljivost pri višji stopnji bremena ni zagotovljena in je mogoča le za izbrane primere opravil oziroma ponovitvenih časov ter bremen opravil. Enega njih smo spoznali v predhodnem razdelku.

Sedaj pa se prepričajmo v relacijo (8.5) za dve opravili, $N = 2$. Za $N = 2$ je meja $\approx 83\%$. Imejmo opravili P_1 in P_2 z bremenoma C_1 in C_2 ter periodama T_1 in T_2, $T_1 < T_2$. Opravilo P_1 ima po pravilu razvrščanja RM višjo prioriteto. T_2 v splošnem ni mnogokratnik T_1, $nT_1 \neq T_2$, slika 8.8. Opravilo P_1 v času T_2 postavi $\lceil T_2/T_1 \rceil$ zahtev po procesorju. Mogoča sta dva primera:

1. Čas C_1 je dovolj majhen, tako da so vse zahteve opravila P_1, vključno z zadnjo, strežene znotraj intervala T_2. Torej velja

$$C_1 \leq T_2 - T_1 \lfloor T_2/T_1 \rfloor.$$



Slika 8.8: *Opravili P_1 in P_2 imata ponovljivosti T_1 in T_2, breme opravila P_1 je C_1. (a) C_1 je manj kot $T_2 - T_1\lfloor T_2/T_1 \rfloor = T_2 - 4T_1$. (b) C_1 je več kot $T_2 - T_1\lfloor T_2/T_1 \rfloor = T_2 - 4T_1$.*

Neenačba pravi, da mora biti C_1 manjši od zadnjega 'necelega' intervala, ki je v splošnem krajši od T_1. Največja mogoča vrednost za C_2 je preostanek prostega časa procesorja v intervalu T_2,

$$C_2 = T_2 - C_1\lceil T_2/T_1 \rceil.$$

Stopnja bremena, s katerim procesa P_1in P_2 obremenita procesor, je

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{C_1}{T_1} + \frac{T_2 - C_1\lceil T_2/T_1 \rceil}{T_2} = 1 + C_1\left(\frac{1}{T_1} - \frac{\lceil T_2/T_1 \rceil}{T_2}\right).$$

Člen v okroglem oklepaju je negativen in stopnja bremena monotono pada s C_1.

2. Čas C_1 ni dovolj majhen in streženje zadnji zahtevi opravila P_1 pade 'čez' interval T_2,

$$C_1 \geq T_2 - T_1\lfloor T_2/T_1 \rfloor.$$

Največja vrednost za C_2 potem je:

$$C_2 = T_1\lfloor T_2/T_1 \rfloor - C_1\lfloor T_2/T_1 \rfloor$$

in stopnja bremena procesorja je:

$$U = \frac{T_1}{T_2}\lfloor T_2/T_1 \rfloor + C_1\left(\frac{1}{T_1} - \frac{1}{T_2}\lfloor T_2/T_1 \rfloor\right).$$

Člen v okroglem oklepaju je pozitiven in U monotono narašča s C_1.

Najmanjša stopnja bremena U zato leži na meji med obema primeroma, torej za

$$C_1 = T_2 - T_1\lfloor T_2/T_1 \rfloor, \;\; C_2 = T_2 - C_1\lceil T_2/T_1 \rceil.$$

Za boljšo preglednost vpeljimo oznaki $n = \lfloor T_2/T_1 \rfloor$ in $n+1 = \lceil T_2/T_1 \rceil$. Stopnja bremena je:

$$\begin{aligned} U &= \frac{C_1}{T_1} + \frac{C_2}{T_2} \\ &= \frac{T_2 - nT_1}{T_1} + \frac{T_2 - (n+1)(T_2 - nT_1)}{T_2} \\ &= (\frac{T_2}{T_1} - n) + (1 - n + n^2\frac{T_1}{T_2} - 1 + n\frac{T_1}{T_2}) \end{aligned}$$

Vpeljimo še oznako $q = T_2/T_1$,

$$U = q - 2n + n(n+1)\frac{1}{q}.$$

Parameter n je celo število, $(n = 1, 2, 3, \ldots)$. Stopnja bremena U monotono narašča z n, kar pomeni, da ima najmanjšo vrednost za $n = 1$,

$$U = q + \frac{2}{q} - 2.$$

Odvajajmo U na q ter odvod izenačimo z nič, da določimo še najmanjši U v odvisnosti od q,

$$\frac{dU}{dq} = 1 - \frac{2}{q^2} = 0; \Rightarrow q = \sqrt{2}.$$

Najmanjša vredost za U pa je,

$$U = \sqrt{2} + \frac{2}{\sqrt{2}} - 2 = 2(\sqrt{2} - 1) = 2(2^{1/2} - 1),$$

kar ustreza izrazu (8.5) za $N = 2$. Do splošnega izraza (8.5) pridemo z matematično indukcijo.

8.2.5 Najkrajši skrajni rok najprej

Razvrščanje po pravilu 'najkrajši skrajni rok najprej' (Angl. Earliest deadline first) spada med algoritme razvrščanja na osnovi dinamičnih prioritet.

Po tem pravilu se prioritete opravil spreminjajo s časom v odvisnosti od skrajnih rokov opravil. Razvrščevalnik v danem trenutku dodeli procesor tistemu opravilu, ki ima najkrajši skrajni rok. Procesi ni treba da so periodični, morajo pa imeti ocenjene skrajne roke. Algoritem je optimalen v smislu, da bo zagotovil razvrstljivost za vse procese in 100% izkoristek procesorja. Drugače povedano, v primeru periodičnih opravil je pogoj razvrstljivosti enostavno:

$$\sum_{i=1}^{N} U_i \leq 1.$$

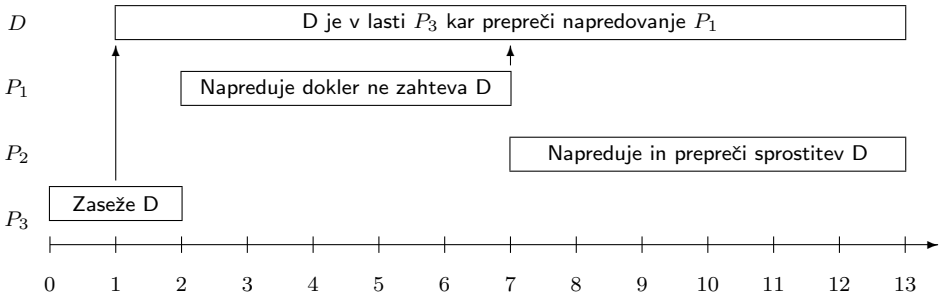
V primeru preobremenjenega sistema je kršenje skrajnih rokov nepredvidljivo. Povejmo še, da ugotovitve in trditve v tem in v predhodnih razdelkih v celoti držijo samo za neodvisne procese. V primeru soodvisnosti procesov delovanje v realnem čas ni nujno zajamčeno. Primer si bomo ogledali v naslednjem razdelku.

8.2.6 Obrat prioritet

Glavna zahteva v sistemih realnega časa je predvidljivost. Problem nepredvidljivosti je prisoten tudi v prioritetnem načinu razvrščanja. Pojavi se v primeru, ko proces z nižjo prioriteto zaseže sredstvo, ki ne dovoljuje sočasnega dostopa. Problem je znan kot 'obrat prioritet'(Ang. Priority inversion) in si ga bomo ogledali na primeru.

Denimo, da imamo tri procese, P_1, P_2 in P_3. Naj ima P_1 najvišjo prioriteto in P_3 najnižjo. Privzemimo, da si P_1 in P_3 delita skupno sredstvo D (na primer podatkovno strukturo) ter da proces z nižjo prioriteto P_3 zaseže D. Nadzor za dostop do D bi lahko uredili na primer s semaforjem ali z mutexom. Med tem naj proces P_1 postane pripravljen za izvrševanje. Ker ima višjo prioriteto, prevzame procesor in napreduje, dokler ne zahteva dostop do sredstva D. Ker je D še vedno zasežen in v uporabi procesa P_3 , dostopa ne more dobiti in torej čaka. Naj med tem proces P_2 izpolni pogoje za napredovanje. Ker ima višjo prioriteto kot P_3 , prevzame procesor, proces P_3 ostane (ali gre ponovno) v stanje čakanja, ki zato ne more sprostiti sredstva D. Posledično to preprečuje napredovanje procesu P_1, ki ima sicer

višjo prioriteto kot P_2 . Proces z nižjo prioriteto (P_3) je torej blokiral proces z višjo prioriteto (P_1), tako da ga lahko posledično procesi z nižjo prioriteto (na primer P_2) neomejeno prehitevajo.



Slika 8.9: *Proces P_3 zaseže D, a ga ne sprosti, ker mu proces P_1 z višjo piroriteto prevzame procesor. P_1 napreduje, dokler ne zahteva D, ki ga zaseda P_3. Zato napreduje P_2, ki ima višjo prioriteto od P_3, zato P_3 ne more sprostiti D ter tako posledično blokira P_1.*

Blokadi oziroma obratu prioritet se izognemo z 'dedovanjem' prioritet (Angl. Priority inheritance). To pomeni, da vsi procesi, ki dostopajo do skupnega sredstva, v času dostopa do skupnega sredstva začasno dobijo prioriteto procesa, ki ima izmed njih najvišjo prioriteto. Možnost blokade ali celo 'veriženja blokad' s strani procesa z nižjo prioriteto je tako preprečena. Druga rešitev je dvig prioritete procesu, ki zaseže skupno sredstvo na najvišjo prioriteto (Angl. Priority ceiling), zato se izvajanje procesa v kritičnem delu izteče brez prekinitve. Do obrata prioritet ne more priti.

8.2.7 Analiza odzivnosti

Imejmo množico periodičnih opravil $P_i, (i = 1, ..., N)$,

$$P_i =< \phi_i, C_i, D_i, T_i >,$$

pri čemer je ϕ_i trenutek, ko je opravilo P_i prvič sproščeno, C_i potrebni čas procesorja, D_i (relativni) skrajni rok opravila in T_i ponovljivost opravila. Privzamemo, da so vsi $\phi_i = 0$, kar pomeni, da so vsa opravila sproščena sočasno. Situacija se ponovi po času $T_c = NSM(T_1, T_2, \ldots, T_i, \ldots, T_N)$,

kjer NSM pomeni najmanjši skupni mnogokratnik vseh period. Naj bodo opravila razvrščena po pravilu RM in naj velja $D_i \leq T_i$. Zanimajo nas odzivni čas opravil, R_i. Odzivni čas opravila je čas, ki preteče od trenutka, ko je opravilo sproščeno, do trenutka, ko je opravilo opravljeno. Za delovanje v realnem času mora biti izpolnjen pogoj

$$R_i \leq D_i \leq T_i, (i = 1, 2, \ldots, N). \tag{8.6}$$

V primeru predopravilnega prioritetnega razvrščanja bo v vsakem trenutku napredovalo opravilo z najvišjo prioriteto. Zato je čakalni čas w_i opravila P_i enak

$$w_i = \sum_{j=1..i-1} \lceil \frac{w_i}{T_j} \rceil C_j \tag{8.7}$$

in odzivni čas

$$R_i = C_i + w_i = C_i + \sum_{j=1..i-1} \lceil \frac{w_i}{T_j} \rceil C_j. \tag{8.8}$$

Če po enačbi (8.7) določimo čakalne čase, lahko po enačbi (8.5) izračunamo odzivne čase in preverimo pogoj (8.6).

V enačbi (8.7) nastopa neznani w_i na obeh straneh enačbe. Enačba ni rešljiva analitično, da pa se jo rešiti numerično po iterativni metodi,

$$w_i^{k+1} = \sum_{j=1..i-1} \lceil \frac{w_i^k}{T_j} \rceil C_j, \quad (k = 0, 1, 2, \ldots).$$

Iteracija se ustavi, ko je $w_i^{k+1} = w_i^k$. Začnemo z $i = 1$, to je najprej izračunamo R_i za opravilno z najvišjo prioriteto.

Poglavje 9

Osnovni elementi sistemskega programiranja

To je uvodno poglavje v poglavja, ki bodo drugo za drugim podrobneje obravnavala programski vmesnik med jedrom Linux in aplikacijami. Ob tem bomo spoznavali funkcionalnost jedra in kako do nje priti.

V tem poglavju pa bomo na kratko spregovorili o sistemskem programiranju. Spoznali bomo koncept sistemskih klicev in funkcij ter razliko med njimi. Navedli bomo nekatere relevantne standarde s tega področja ter osvežili pomen pogostejših izrazov.

9.1 Kaj je sistemsko programiranje

Ko rečemo *sistemsko programiranje* imamo v mislih načrtovanje in razvoj programske opreme za sistemsko jedro ali pač načrtovanje in razvoj aplikacij z neposredno uporabo storitev jedra. V prvo skupino sodi denimo razvoj *modulov* jedra, to je gonilnikov vhodnih in izhodnih naprav. Razvoj modulov jedra zahteva znanja s področja operacijskih sistemov, arhitektur računalniških sistemov, predvsem pa temeljito poznavanje notranje strukture jedra in specifiko delovanja strojne opreme.

V drugo skupino sistemskega programiranja spada razvoj programov, ki so blizu jedru. Ti programi sicer delujejo v uporabniškem načinu, a temeljijo na podpori sistemskega jedra. Tudi ta nivo programiranja še vedno zahteva znanja s področja operacijskih sistemov, a poglobljeno znanje o notranji zgradbi jedra in strojni opremi ni nujno potrebno.

Ena od izstopajočih značilnosti sistemov UNIX je, da ne delajo kakšne posebne razlike med sistemskimi programi, kot so denimo `bash`, `ps`, `ls`, `cp` in aplikacijami. Zato bi lahko skoraj vse aplikacije v sistemih UNIX in Linux, ki so izdelane v programskem jeziku C, uvrstili v to skupino. Sem pa gotovo ne spadajo JavaScript ali PHP, spletne in podobne aplikacije.

Za začetek pa si osvežimo pomen nekaterih pogosto slišanih kratic.

9.2 API

API je kratica, ki jo tvorijo prve črke treh angleških besed Application Programming Interface. Kot pravi njen izvor, je API *programski vmesnik* med dvema kosoma programske opreme na nivoju aplikacij. Najpogosteje en kos programske opreme realizira nabor funkcij, ki so v skladu z zahtevami API in drugi kos programske opreme ali *aplikacija* te funkcije uporablja. API torej zagotavlja *združljivost* in posledično prenosljivost programske opreme na nivoju programskega jezika.

Dober primer vmesnika API je standard programskega jezika C in njegova implementacija v standardni knjižnici C. Poudarimo, API je specifikacija in knjižnica je njegova implementacija. Kot že povedano, se bomo v nadaljevanju osredotočali predvsem na API vmesnik s sistemskim jedrom.

9.3 ABI

Tudi ABI je kratica angleškega izvora, ki nastane iz začetnic besed Application Binary Interface. Naloga vmesnika ABI je zdržljivost programske opreme na nižjem, to je na *binarnem* nivoju. Vmesnik definira denimo

način klica funkcije. Vmesnik predpisuje kako se prenašajo parametri med klicno in klicano funkcijo in kateri registri procesorja se pri tem uporabljajo. Nadalje, vmesnik določa vrstni red bajtov, pravila povezovanja programskih modulov ter knjižnic, in podobno. Podprtost ABI je torej smiselno pričakovati samo v okviru iste arhitekture.

Za sistemskega programerja je poznavanje vmesnika ABI koristno, a še zdaleč ne nujno potrebno. Za usklajenost programske opreme na binarnem nivoju skrbijo razvojna orodja (Angl. Toolchain), to je prevajalniki, povezovalniki, nalagalniki. Poznavanje ABI postane pomemebno pri programiranju v zbirnem jeziku, povezovanju C-jevskih in zbirniških programov, in podobno.

9.4 ELF

Ko se izvorna koda prevede in poveže v izvršljiv program, se program shrani v datoteko, da se ga lahko kasneje izvrši. Vsebina datoteke mora imeti predpisano strukturo, ali kot rečemo *format*. Izvršljiva datoteka vsebuje poleg ukazov in začetnih vrednosti podatkov še *metapodatke*[1], ki so potrebni, da se program pravilno namesti v pomnilnik ter izvrši.

V preteklosti je bil v sistemih UNIX v uporabi tako imenovani `a.out` format (Assembler output). Večina današnjih sistemov UNIX in Linux pa za ta namen uporablja ELF. ELF je kratica in nastane iz prvih črk angleških besed Executable and Linking Format. ELF je splošno privzet format zapisa za izvršljive datoteke, objektne datoteke in knjižnice. Na primer, `gcc` na koncu prevajanja in povezovanja ustvari izvršljivo datoteko v formatu ELF. Nalagalnik sistemskega jedra pričakuje format ELF. Ko zahtevamo izvršitev programa, nalagalnik prebere vsebino datoteke, jo interpretira, temu primerno naloži v pomnilnik in program izvrši. Strukturo datoteke ELF lahko pregledujemo z ukazom

```
objdump <ime_datoteke>.
```

[1] Metapodatki so podatki o podatkih.

9.5 POSIX

POSIX[2] je kratica za Portable Operating System Interface. Črko X so dodali po vzorcu za poimenovanje sistemov UNIX. Nekaj takih primerkov imen je AIX, HP-UX, Xenix, MINIX in Linux.

POSIX je torej standard oziroma zbirka standardov, ki določajo vmesnik API med sistemskim jedrom in aplikacijami. Razvijal in uveljavil se je predvsem v sistemih s poreklom UNIX, pa tudi širše.

Poslanstvo POSIXa je skrb za *združljivost* med sorodnimi operacijskimi sistemi in za *prenosljivost* aplikacij na nivoju izvornega jezika. Standard sicer ne predpisuje kako naj bo vmesnik implementiran. To je prepuščeno razvijalcem sistema. Za operacijski sistem z vmesnikom po specifikaciji POSIX, se reče, da je *skladen* s POSIX.

POSIX ne specificira samo API za sistemske klice in funkcije, temveč tudi školjke, sistemske *ukaze* ter ukazno vrstico C-jevskega prevajalnika. Standard je nastajal in se razvijal več desetletij. Pobuda za razvoj standarda se je porodila sredi osemdesetih let prejšnjega stoletja iz potrebe po prenosljivosti programske opreme med sistemi UNIX. Razvoj številnih različic sistema je tedaj potekal povsem neuskajeno. Povečevale so se razlike med sistemi, trpela pa je prenosljivost aplikacij.

Razvoj in vzdrževanje standarda POSIX je pod okriljem mednarodnega združenja inženirjev `IEEE`. Prva različica, z uradnim nazivom IEEE Std 1003.1-1988, a bolj znana kot POSIX.1, je izšla v letu 1988. V prvi izdaji so zajeti predvsem temeljni sistemski klici in funkcije, kot so na primer sistemski klici za upravljanje procesov, medprocesne komunikacije ter podobno. Tekom devetdesetih let so sledile POSIX.1b razširitve za realni čas, POSIX.1c razširitve za niti, POSIX.1g razširitve za omrežne aplikacije, POSIX.1d in POSIX.1j dodatne razširitve za realni čas, ter sorodna specifikacija POSIX.2 za školjke in ukaze. Popravke in razširitve je na prelomu tisočletja zaokrožil standard IEEE Std 1003.1-2001. Dandanes se kratica POSIX nanaša na standard IEEE Std 1003.1-2008, ki je znan tudi kot

[2]Kratico je predlagal Richard Stallman. Izgovarjamo jo (približno) *paziks*.

POSIX.1-2008. V njem so združene posodobljene vsebine vseh predhodnih dokumentov.

9.6 SUS

POSIX-u sorodna skupina pomembnih tehničnih standardov za operacijske sisteme UNIX se imenuje Single Unix Specification in nosi kratko oznako SUS. Specifikacija SUS je pod okriljem združenja *The Open Group*. Prvi dokument iz te skupine z oznako SUSv1 datira v leto 1995. Nastal je na podlagi predhodne dokumetacije imenovane X/Open Portabilty Guide. To je tudi čas, ko sta se X/Open in Open Software Foundation preoblikovali v The Open Group.

Specifikacija SUS je širša kot POSIX. Predpisuje dodatne vmesnike in zahteva dodatno funkcionalnost, ki jo POSIX zgolj priporoča. Celotna zbirka vmesnikov se označuje tudi s kratico XSI (X/Open System Interface).

Sodelovanje med IEEE, ISO/IEC JTC 1 ter The Open Group leta 1999 je rodilo poenoten in dopolnjen dokument z oznako POSIX 1003.1-2001. Standard je znan tudi pod imenom SUSv3. Dokument predpisuje osnovne zahteve za POSIX skladnost in razširitve za SUSv3 skladnost. The Open Group, ki je lastnica blagovne znamke UNIX, zahteva kot podlago za podelitev pravice poimenovanja sistema z imenom UNIX (UNIX 03), popolno SUSv3 skladnost.

Podobno ugotovimo tudi za specifikacijo POSIX.1-2008 s popravkom iz leta 2013. Tudi ta predpisuje osnovno funkcionalnost za POSIX skladnost in XSI razširitve za SUSv4 skladnost, a dopolnitve naprav SUS3 niso tako bistvene.

9.7 POSIX, SUS in Linux

In kako je s skladnostjo s standardi v sistemu Linux? Razvijalci sistema Linux si prizadevajo za čimbolj dosledno skladost sistema Linux z vsemi

relevantnimi standardi, predvsem POSIX in SUS, a za enkrat še nobena od distribucij sistema Linux ni pridobila pravice za uporabo imena UNIX. Glavni razlog temu leži v hitrem razvoju, dolgih postopkih in stroških za pridobitev te pravice. Ne oziraje se na to lahko zaključimo, da je prenosljivost Linuxa in pa seveda Linux aplikacij izjemna. Sistem najdemo na najrazličnejših platformah in arhitekturah in tudi s prenosljivostjo aplikacij, če le upoštevajo relevantne standarde, ni težav.

Skladnost posameznih sistemskih funkcij in klicev z relevantnimi standardi je navedena na straneh uporabniškega priročnika za vsako funkcijo oziroma klic posebej. Tako za sistemski klic `open` ugotovimo

```
CONFORMING TO
       SVr4, 4.3BSD, POSIX.1-2001. ....
       .....
```

Klic je skladen z UNIX System V release 4 (SVr4), sistemom UNIX BSD verzije 4.3 ter POSIX.1-2001. Sledijo še posebnosti in izjeme, ki smo jih v navedbi opustili.

9.8 Programski jezik C in standardi

Nastanek in razvoj programskega jezika C je neločljivo povezan s sistemom UNIX. Glavnina jezika je dozorela pred letom 1973, to je do tedaj, ko je bilo moč UNIX skoraj v celoti prepisati v C. Prva knjiga *The C programming language* avtorjev B.W. Kernighan in D.M. Ritchie je izšla leta 1978. Ta knjiga je dolgo služila kot neformalna, a edina specifikacija programskega jezika. Sintaksa jezika iz te knjige se pogosto poimenuje tradicionalni C ali tudi K& R C in je še vedno prisotna. Markantna posebnost jezika tega časa je bila izrazita jedrnatost kode in v nasprotju denimo s programskim jezikom Pascal neverjetna svoboda pri delu s podatkovnimi tipi.

Prve dopolnitve programskega jezika so prišle v letu 1989. Prinesle so prototipe funkcij, podatkovni tip `void`, označevalca tipa `const` in `volatile` ter specifikacijo standardne knjižnice C. Novo različico jezika je še istega leta potrdila ameriška organizacija za standardizacijo ANSI in se spričo

tega pogosto poimenuje po njej ANSI C ali C89. Tu in tam se pojavlja poimenovanje ISO C90 in sicer po letu, ko je jezik potrdila še organizacija ISO.

Nadaljnje dopolnitve programskega jezika so sledile leta 1999 z ISO standardom ISO/IEC 9899:1999 ali krajše C99. C99 na primer dovoljuje C++ obliko komentarjev (//), predvsem pa prinaša številne dopolnitve standardne knjižnice. Zadnja posodobitev jezika in potrditev standarda pa datirata v leto 2011, od tu tudi poimenovanje s C11.

Skupna značilnost v seriji dopolnitev programskega jezika so bile vse strožje zahteve v pogledu podatkovnih tipov ter prizadevanja za večjo preglednost. Programski jezik C je razvojno gledano neločljivo povezan s sistemom UNIX. Vendar naj nas to ne zavede. C je povsem samostojen in od operacijskega sistema neodvisen programski jezik. Aplikacija v programskem jeziku C, ki uporablja zgolj funkcije standardne knjižnice C, bi načeloma morala delovati na vsakem računalniku.

9.9 Sistemske funkcije in sistemski klici

Storitve operacijskega sistema so dejavnosti, ki jih sistemsko jedro daje uporabniškim programom oziroma procesom. Do njih procesi dostopajo s sistemskimi klici in funkcijami. Jedro zagotavlja storitve z upravljanjem strojne opreme. Uporabniški programi nikoli ne dostopajo direktno do strojne opreme. Množica sistemskih klicev in funkcij definira vmesnik programov do sistemskega jedra. To je torej API med jedrom in aplikacijami.

Po namenu uporabe razdelimo sistemske funkcije/klice v grobem v štiri skupine,

- funkcije za upravljanje datotečnega sistema ter vhodno/izhodnih naprav, kot na primer `open`, `read` in `write` ter `close`.
- Funkcije za upravljanje procesov in niti, kot na primer `fork`, `execve`, `wait` in `exit`.

- Funkcije za komunikacijo med procesi, kot na primer `pipe`, `signal`, `socket` ter funkcije za semaforje, sporočila in deljeni pomnilnik.
- Funkcije za nadziranje in upravljanje sistema, kot denimo `time`, `uname`, `getrlimit` in podobno.

Imena nekaterih značilnih predstavnikov funkcij navajamo zgolj v orientacijo. V naslednjih poglavjih si bomo podrobno ogledali delovanje in uporabo važnejših sistemskih klicev in funkcij. Najprej pa si oglejmo osnovni koncept.

9.9.1 Sistemki klici

Pod izrazom sistemski klic načeloma razumemo prenos izvajanja procesa iz uporabniškega v sistemski način oziroma prehod izvajanja procesa iz *uporabniškega prostora* v sistemsko jedro. Ta prehod je realiziran s programsko prekinitvijo, poimenovano tudi past oziroma *izjema* (Angl. Software interrupt, trap, exception). V sistemu Linux najdemo seznam vseh sistemskih klicev na straneh priročnika *syscalls(2)*. Številka dve v oklepaju pomeni, da se opis nahaja v poglavju 2. Torej, ukazna vrstica

```
man 2 syscalls
```

izpiše ta seznam. Vsak sistemski klic ima svojo številko. Številke sistemskih klicev v sistemu Linux so zbrane v datoteki `sys/syscall.h`. Številke klicev so uporabnikom prikrite. Za to poskrbijo *ovojne* funkcije (Angl. Wrapper functions) v knjižnici C. Tipičen sistemski klic poteka takole:

- Uporabniški program kliče ovojno funkcijo, na primer `read`.
- Ovojna funkcija prenese številko dotičnega sistemskega klica v za to predvideni register procesorja. V x86 arhitekturah je to register `eax`. Sistemski klic `read` ima na primer številko tri.
- Ovojna funkcija prevzame argumente prek seznama argumentov funkcije in jih prenese v predvidene registre procesorja. Na primer `read(fd, buf, size)` ima tri argumente, ki gredo v registre `ebx`, `ecx` in `edx`.

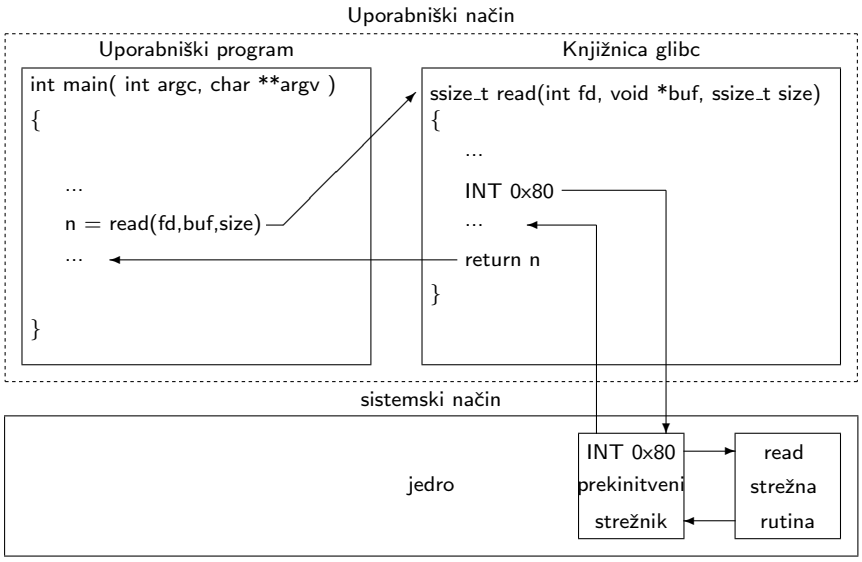
- Ovojna funkcija povzroči programsko prekinitev s klicem procesorskega ukaza za ta namen. V x86 arhitekturah je to zbirniški ukaz `INT 0x80`, ki povzroči izjemo prek vektorja s številko 0x80. Z izvršitvijo tega ukaza se spremeni način delovanja iz uporabniškega v sistemski način. Sedaj smo v prostoru jedra.
- Sistemsko jedro shrani registre procesorja na sklad, preveri veljavnost številke sistemskega klica in na podlagi te številke kliče ustrezno strežno rutino. V našem primeru je to strežna rutina za read.
- Ko se strežna rutina izteče, sledi obnovitev vrednosti registrov iz sklada in povratek v uporabniški način z ukazom `IRET`. Sistemski klic pusti rezultat izvršitve na skladu in stanje oziroma kodo napake v registru `eax`.
- Ovojna funkcija prevzame rezultat in stanje izvršitve ter stanje prepiše v globalno spremenljivko `errno`, ki v primeru napake, vsebuje kodo napake.
- Ovojna funkcija se vrne klicnemu programu. V primeru napake preko `return` vrne vrednost -1 in `errno` vsebuje kodo napake. Sicer vrne vrednost nič ali vrednost večjo od nič.

Grobi potek dogodkov je skiciran na sliki 9.1. Dejanska izvedba vmesnika sistemskih klicev sicer zavisi od arhitekture, a sam koncept je v grobem vedno enak.

Vsak sistemski klic je moč povzročiti tudi brez ovojne funkcije. To omogoča funkcija `syscall`. Tedaj moramo poznati oziroma navesti številko dotičnega sistemskega klica.

9.9.2 Sistemske funkcije

Pod izrazom sistemska funkcija načeloma razumemo podprogram – funkcijo, ki se izvede izključno v uporabniškem načinu ali po potrebi povzroči tudi sistemski klic. Sistemske funkcije, ali krajše tudi kar funkcije, so zbrane v sistemskih knjižnicah. Za nas je to standardna knjižnica C. V knjižnici C

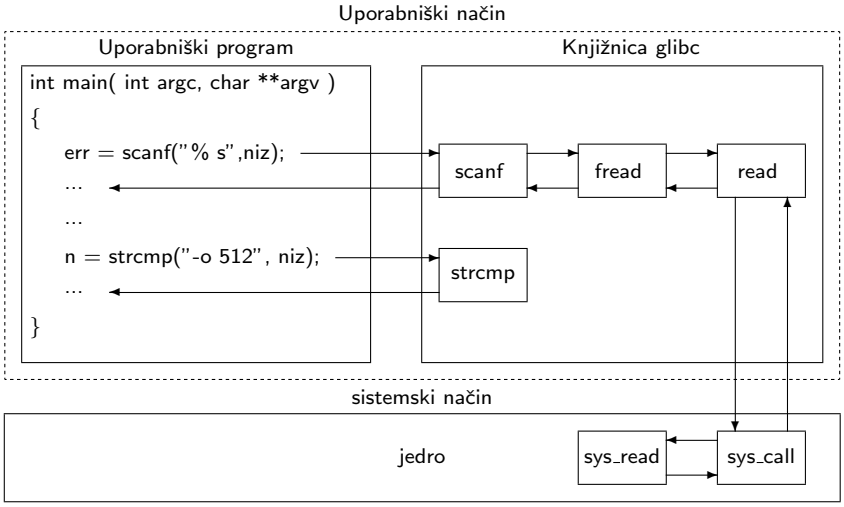


Slika 9.1: *Koncept izvedbe sistemskega klica.*

je zares veliko funkcij. Nekatere pod sabo sploh nimajo sistemskega klica, kot so na primer funkcije za delo z znakovnimi nizi. Veliko funkcij je zgolj ovojnih funkcij, klicu take funkcije vedno sledi sistemski klic. Nekatere druge funkcije pa izvršijo sistemski klic samo po potrebi. Take funkcije se, odvisno od razmer, izvedejo zgolj v uporabniškem načinu, ali pa opravijo tudi sistemski klic in posledično povzročijo spremembno načina. Na primer, funkcija za branje `fread` lahko klicnemu programu vrne podatke iz svojega izravnalnega pomnilnika, če jih že ima, sicer zahteva podatke od sistemskega jedra s klicem `read`. Funkcije v knjižnici lahko kličejo druge funkcije iz knjižnice. Na primer, funkcija `scanf` za branje s standardne vhodne datoteke posledično kliče funkcijo `fread`, ki posledično po potrebi kliče ovojno funkcijo `read` in sledi sistemski klic. Slika 9.2 ponazarja ta koncept.

S stališča programerja v jeziku C, vsaj kar zadeva programski jezik, se sistemske funkcije na videz ne razlikujejo od sistemskih klicev, zato je razlikovanje funkcij od klicev manj pomembno. Podrobnejši vpogled v vmesnik sistemskih klicev in funckij nam lahko razkrije razlike med sistemi. Denimo, kar je v enem sistemu sistemski klic, je lahko v drugem sistemu realizirano kot funckija. To je stvar implementaciji sistema. A dokler je vmesnik v

skladu s specifikacijo, se lahko zanesemo, da bo aplikacija prenosljiva med sistemi.



Slika 9.2: *Koncept sistemskih funkcij.*

9.10 Knjižnice

Programska knjižnica je zbirka funkcij v prevedeni, to je strojni obliki. Programu, ki potrebuje nekatere funkcije iz dane knjižnice, se v fazi povezovanja doda potrebne, a ne vse, funkcije iz knjižnice. Program se poveže s knjižnico.

Sistemi UNIX in Linux poznajo dva tipa knjižnic, *statične* in *dinamične* oziroma *deljene* knjižnice (Angl. Shared libraries).

9.10.1 Statične knjižnice

Statične knjižnice so tradicionalna izvedba zbirke funkcij. Tako knjižnico običajno sestavlja določeno število relativno obsežnih objektnih modulov. Imena knjižnic po dogovoru začenjajo s predpono `lib` in imajo končnico `.a`, ki pomeni *arhiv*. Na primer, `libc.a` je statična standardna knjižnica C in `libm.a` je statična matematična knjižnica. Ko se sklicujemo na dotično

knjižnico, predpono in končnico opustimo. Na primer, `m` pomeni matematično knjižnico. Če denimo glavni program povežemo s statično knjižnico, nastane izvršljiva koda, ki poleg glavnega programa vsebuje tudi vse potrebne objektne module iz knjižnice. Če so ti moduli veliki in če jih je še veliko, je prevedeni program lahko tudi zelo velik, četudi je glavni program zelo majhen. Je pa zato program celota, ki se da prenesti na drugo platformo in izvršiti, ne računajoč na podporo knjižnic na drugi platformi.

9.10.2 Deljene knjižnice

Za razliko od statičnih knjižnic se z dinamično oziroma deljeno knjižnico povezava dogaja v času izvrševanja programa (Angl. run-time). V fazi povezovanja glavnega programa s funkcijami iz knjižnice, se sicer uredijo povezave s funkcijami, a se same funkcije ne vključijo v izvršljivo datoteko. Nastala izvršljiva datoteka je zato velika približno toliko, kolikor je velik glavni program.

Ko želimo tak program izvršiti, se v pomnilnik namesti samo program, medtem ko se knjižnica namesti v pomnilnik posebej. Ker je program manjši, je tudi nameščanje programa hitrejše. Deljeno knjižnico si lahko hkrati deli več programov, zato tudi tako ime. Ker je v pomnilnik nameščena ena sama kopija deljene knjižnice, ne glede na to, koliko programov jo uporablja, so deljene knjižnice tudi bolj ekonomične v pogledu porabe pomnilnika. Drugače povedano, ista kopija deljene knjižnice je preslikana v naslovne prostore večih procesov.

Deljene knjižnice omogočajo tudi večjo fleksibilnost. V sistem lahko namestimo posodobljeno knjižnico. Od tedaj naprej bodo programi uporabljali novo knjižnico, ne da bi bilo potrebno ponovno prevajati programe.

Knjižnice tega tipa imajo v sistemu Linux končnico `.so`, kot na primer `libc.so`, kar sicer pomeni *shared object*. Odvisnost programa od deljenih knjižnic lahko preverimo z ukazom

```
ldd <ime_datoteke>
```

Deljene knjižnice so umeščene v naslovni prostor programa nekje pod skla-

dom in nad kopico.

9.10.3 Knjižnica GNU C

Vsak sistem UNIX ima standardno knjižnico C. Knjižnica je *standardna*, tako da je njena funkcionalnost skladna s standardom programskega jezika C. Skladnost je zagotovljena tudi s specifikacijo POSIX. Pravzaprav skladnost s POSIX že implicira tudi skladnost s standardom C. Knjižnico C ne uporabljajo samo C-jevski programi, temveč tudi programi v drugih programskih jezikih, kot na primer programi v programskem jeziku C++. Ta knjižnica je tako pomembna, da brez nje sistem sploh ne bi deloval, ali pa bi bilo delovanje sistema skrajno okrnjeno.

Na različnih sistemih UNIX in Linux obstajajo različne implementacije knjižnice C. A skladnost knjižnice s specifikacijo POSIX jamči, da bo programska oprema prenosljiva s sistema na sistem brez večjih posegov ali brez posegov v kodo, ne glede na implementacijo knjižnice.

Na sistemu Linux je običajno nameščena `GNU C glibc` knjižnica. Za manjše vgradne sisteme so primernejše manj razkošne knjižnice, kot je na primer `uClibc`. Če nas zanima verzija knjižnice, lahko do nje pridemo tako, da v ukazni vrstici tipkamo njeno ime, kot bi jo hoteli izvršiti. Na primer

```
/lib/libc.so.6
```

Sama knjižnica in seveda nobena od knjižnic samostojno *ni* izvršljiva.

9.11 Pridružene datoteke

Včasih se programske knjižnice zamenjuje s pridruženimi datotekami in obratno. Vzrok temu se najbrž skriva v tem, da se skoraj vedno pridružene datoteke in programske knjižnice pojavljajo skupaj. Tu in tam zasledimo, da se datoteki `stdio.h` reče knjižnica. To ni knjižnica. To je pridružena datoteka, ki se uporablja skupaj s standardno vhodno izhodno knjižnico, ki je del knjižnice `libc`.

Kot rečeno so programske knjižnice zbirke funkcij v strojni obliki. Knjižnice vsebujejo *definicije* funkcij in so pred tem, ko se jih rabi, že prevedene v strojno obliko. Čeprav so knjižnice v strojni obliki, pa knjižnična datoteka sama posebi ni izvršljiva. Vsebuje le dele potencialno izvršljivega programa. Za razliko od izvršljivih datotek ali programov, tem datotekam rečemo *objektne datoteke*.

Med tem ko so knjižjice v strojni obliki in vsebujejo definicije funkcij, pa pridružene datoteke vsebujejo *deklaracije* funkcij v izvornem programskem jeziku. Datoteke te vrste imajo po dogovoru končnico `.h` kot *header*. Deklaracije oziroma navedbe prototipov funkcij so potrebne prevajalniku, da lahko programsko besedilo pravilno prevede. Zato se `.h` datoteke *pridružijo* nekje na začetku pred samim programskim besedilom, ki kliče funkcije. Na primer,

```
#include <stdio.h>

int main( int argc, char **argv )
{
  printf("Dober dan\n");
  return 0;
}
```

Stavek `#include` zahteva, da se na mestu kjer se pojavi, in to je na začetku besedila, pred ostalo besedilo doda celotno vsebino datoteke `stdio.h`.

Pridružene datoteke vsebujejo razen deklaracij funkcij še različne definicije podatkovnih tipov, stavke za pogojno obdelavo besedila, *makro definicije* in pogosto vključujejo spet druge `.h` datoteke.

V postopku prevajanja se pragramskemu besedilu najprej doda vse in v navedenem zaporedju pridružene datoteke ter združeno besedilo pripravi za prevajanje. V tej fazi se *razširijo* makro definicije in to nalogo opravi makro predprocesor. Šele nato sledi prevajanje tako izoblikovanega programskega besedila.

9.12 Zbirka prevajalnikov gcc

GNU zbirka prevajalnikov (Angl. GNU Compiler Collection) ali s kratico `gcc` sestavlja razvojno verigo orodij in je del GNU/Linux distribucije. Včasih se je ime `gcc` razlagalo kot *GNU C compiler*, a s časom se je funkcionalnost razširila na druge jezike. Dandanes je `gcc` *čelni* program za prevajalnike C, C++, Java, Fortran in še nekatere druge. Na voljo so implementacije za različne platforme in arhitekture. `Gcc` podpira *križno* prevajanje, to je prevajanje na enem, to je razvojnem sistemu, za drugi, ciljni sistem ali napravo. Seveda ni treba, da bi imela razvojni in ciljni sistem enako arhitekturo.

`Gcc` ni le zbirka prevajalnikov, temveč tudi povezovalnikov in podpornih knjižnic. Uporaba prevajalnika je lahko preprosta kot tale,

```
gcc mprog.c
```

s čimer se izvorni program v jeziku C prevede in poveže v izvršljiv program z imenom `a.out`. Če želimo druačno ime, na primer `mprog`, ga podamo z izbiro `-o` kot *output*,

```
gcc -o mprog mprog.c
```

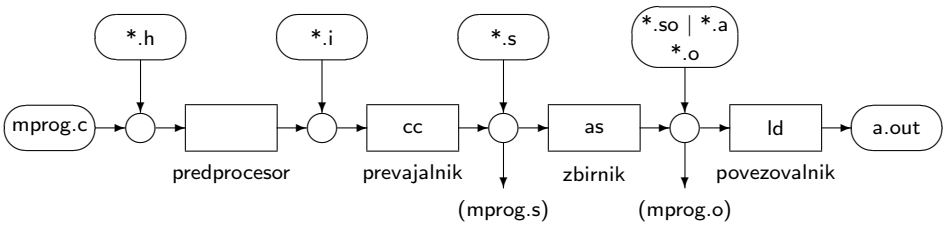
Načelni potek korakov v procesu prevajanja prikazuje slika 9.3. Ko rečemo *prevajanje*, mislimo v prenesenem pomenu besede na vse v obdelavo vključene operacije: makro predprocesiranje, C-jevsko in zbirniško prevajanje ter povezovanje. Možnosti prevajalnika so številne. Na primer,

```
gcc -o mprog mprog.c masm.s mfun.o -lm
```

generira izvršljivo datoteko `mprog` na osnovi C-jevske datoteke `mprog.c`, zbirniške datoteke `masm.s`, objektne datoteke `mfun.o` ter matematične knjižnice `libm.so`. Standardne knjižnice C ni potrebno posebej navajati.

9.13 Obravnavanje napak

Skoraj vsak sistemski klic ali funkcija nadzira potek izvršitve in javi napako, do katere morebiti pride tekom izvršitve. Uporabnikov program mora *vedno*



Slika 9.3: *Potek operacij v procesu prevajanja. Zvezdica pomeni dodatne vire, ki se po potrebi dodajo v vmesnih korakih. Oklepaj nakazuje možnost predčasnega zaključka z zbirniškim (*`.s`*) ali objektnim (*`.o`*) programom.*

preveriti stanje izvršitve klica in se temu primerno odzvati.

Večina sistemskih klicev in funkcij vrne vrednost 0 ali vrednost večjo od nič v primeru, ko se zahtevana operacija izvrši pravilno. V nasprotnem primeru funkcija vrne -1 in globalna spremenljivka `errno` vsebuje kodo napake. Uporabnikov program mora preveriti *vsaj* vrednost, ki jo funkcija vrne. Na primer, sistemski klic `open` odpre datoteko `mdat` za branje,

```
fd = open("mdat", O_RDONLY);
if (fd == -1)
  printf("napaka open()\n");
```

V primeru, da zaradi kakršnega koli razloga datoteke ni moč odpreti za branje, klic vrne -1 in v nadaljevanju sledi drugačno obravnavanje, kot če bi klic uspel.

Včasih nas zanima ne le, da je do napake prišlo, temveč tudi kakšen je njen vzrok. Tega dobimo s klicem funkcije `perror` (p se prebere print), ki na podlagi vrednosti spremenljivke `errno` izpiše tudi vzrok napake. Na primer

```
fd = open("mdat", O_RDONLY);
if (fd == -1)
  perror("napaka open()");
```

v primeru napake izpiše besedilo `napaka open()` in za njim kratko pojasnilo napake. Prototip funkcije `perror` je:

```
#include <stdio.h>
void perror( const char *msg );
```

Vrednost spremenljivke `errno` je veljavna samo neposredno po klicu, zato

se jo vedno preveri *takoj* po povratku funkcije.

Druga možnost za interpretacijo in izpis vrednosti spremenljivke `errno` daje funkcija `strerror()`.

```
#include <string.h>
char *strerror( int err );
```

Na primer,

```
#include <errno.h>
fd = open("mdat", O_RDONLY);
if (fd == -1)
  printf("napaka open(), %s\n", strerror( errno ));
```

Med funkcijami so tudi izjeme. Nekatere ne vračajo ničesar in nekatere ne postavljajo vrednosti `errno`. Kaj funkcija vrne in katere kode napak postavi v primeru napake, je natančno dokumentirano na straneh priročnika za vsako funkcijo posebej.

9.14 Funkcija main in ukazna vrstica

Vsak C-jevski program ima funkcijo `main()`. Prototip funkcije je naslednji:

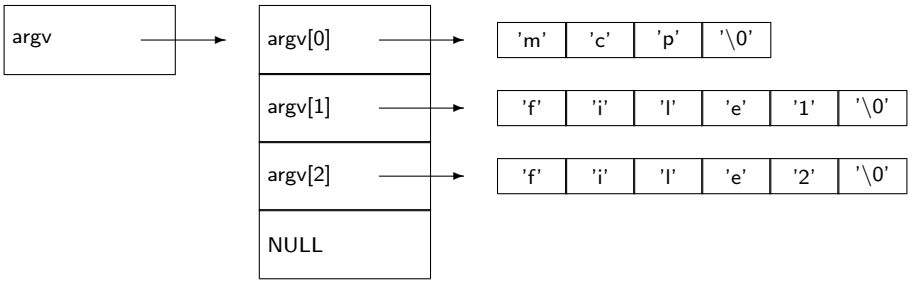
```
int main( int argc, char *argv[] );
```

Pri tem je `argc` število argumentov ukazne vrstice in `argv` je polje kazalcev na te argumente. Z 'ukazno vrstico' mislimo na vrstico, ki jo vtipkamo školjki in vsebuje določeno število znakovnih nizov (argumentov) ločenih s presledki. Kot bomo videli kasneje, je funkcija `main()` pravzaprav vmesnik med uporabnikovim programom in funkcijo sistemskega jedra `exec()`, ki dani program naloži v pomnilnik ter ga izvrši oziroma kliče funkcijo `main()` z dejanskimi vrednostmi argumentov funkcije. Na primer, ukazna vrstica:

```
mcp file1 file2
```

ima tri argumente, zato bi bila vrednost `argc = 3`. Elementi ulazne vrstice so znakovni nizi, medtem ko bi polje kazalcev na nize bilo videti takole, slika 9.4. Rečemo, da kažejo na z ničlo zaključene znakovne nize.

Funkcija `main()` je tipa `int` ter vrne celoštevilsko vrednost, ki jo podamo s



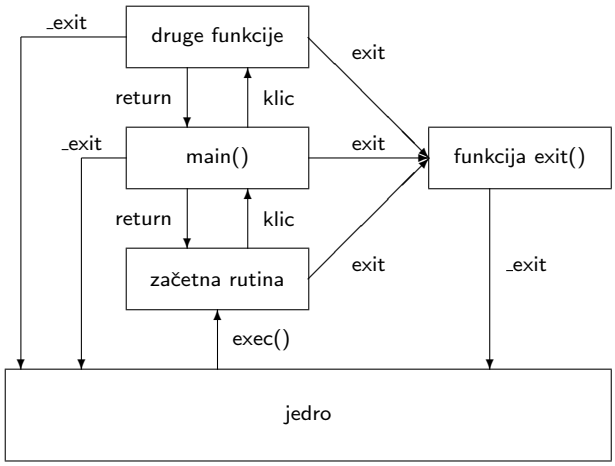
Slika 9.4: *Primer polja argumentov.* `argv` *kaže na polje kazalcev. Elementi polja so kazalci, ki kažejo na znakovne nize. Znakovni nizi so polja znakov, zaključeni z znakom nič. Število argumentov je 3. Zadnji kazalec polja argumentov je kazalec NULL (kazalec nič).*

C-jevskim stavkom `return n`. Kot bomo videli kasneje, je to ekvivalentno sistemskemu klicu `exit( n )`. Ogrodje programa nakazuje naslednja shema.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main( int argc, char *argv[])
{
  int fd;

  if( argc != 3){
    printf("Uporaba: %s <infile> <outfile>\n", argv[0]);
    exit( EXIT_FAILURE );
  }
  fd = open(argv[1], O_RDONLY );
  if( fd == -1){
    perror("napaka open()");
    exit( EXIT_FAILURE );
  }
  // in tako dalje
  return 0;
}
```

Program pričakuje tri argumente. Zato vrednost argumenta `argc` primerja s tri. Če vrednost ni enaka tri, izpiše način pravilne uporabe ter zaključi, sicer nadaljuje. Prvi argument `argv[0]` je ime programa samega, naslednja dva bi bila denimo interpretirana kot imeni datotek za branje in pisanje. Torej sledi odpiranje datoteke z imenom `argv[1]`, preverjanje napak in tako dalje.

In komu se funkcija `main()` sploh vrne? Povezovalnik pred prevedeno uporabnikovo funkcijo `main()` doda nekaj začetnih ukazov strojne kode (angl. 'C start-up routine'). V začetni rutini je ukaz, s katerim se bo program začel izvrševati, potem so ukazi, ki poskrbijo za prenos argumentov, nakar sledi klic funkcije `main()`. Ko se funkcija `main()` izteče ter vrne začetni rutini, začetna rutina kliče funkcijo `exit()` in sledi povratek k jedru. Celoten scenarij prikazuje slika 9.5.



Slika 9.5: *Načelna shema C-jevskega programa.*

Nalaganje in izvršitev programa opravi ena od sistemskih funkcij iz družine `exec()`. Začetna rutina kliče funkcijo `main()`, ta pa po potrebi poljubno mnogo uporabnikovih funkcij. Izvrševanje programa se zaključi z `return` v funkciji `main()` ter posledično s povratkom začetni rutini, lahko pa direktno s klicem funkcije `exit()` od kjerkoli. Funkcija `exit()` se vedno zaključi s sistemskim klicem `_exit()`, možen pe je tudi direkten klic in takojšni povratek v jedro s klicem `_exit()` od kjerkoli. Od začetka do zaključka procesa,

torej med napredovanjem procesa, je prek sistemskih klicem mogoče poljubno število prehodov med uporabniškim in sistemskim načinom. V sliki **??** tega nismo ponazorili.

9.15 Orodje make

Tu bi koristilo osnovno o make s primerom Makefile-a, maksimalno dve strani.

9.16 Nadaljnje čtivo

Tri knjige [1, 2, 3] so bile ključne za nastanek te knjige in še posebej za večino poglavij, ki bodo sledila. Prva je spričo časovne odmaknjenosti dandanes težko dosegljiva. Navajamo jo za referenco spričo njene pomembnosti tedaj, ko je izšla. Da se dobiti njena razširjena izdaja iz leta 2004.

Richard Stevens je avtor številnih uspešnic. Stevens je pisal o zapletenih stvareh na zgoščen, razumljiv in hkrati brezkompromisno dosleden način. Njegova knjiga o sistemskem programiranju v okolju sistema UNIX je gotova ena najboljboljših knjig s tega področja. Doživela je dosti ponatisov in spričo razvoja na področju tudi nekaj dopolnitev. Knjiga [2] je njegova zadnja knjiga v seriji teh knjig.

Knjiga Michaela Kerriska [3] je zagotovo najbolj popolno objavljeno delo o programskem vmesniku sistema Linux. To ne preseneča. Kerrisk je že vrsto let vzdrževalec priročnika `man` sistema Linux. Stilistično je knjiga podobna Stevensovi [2]. Zelo je primerna za samoučenje. Priporočamo jo vsem, ki jih to področje zanima.

Literatura

[1] M. J. Rochkind, *Advaned UNIX Programming*, Prentice Hall, 1985.

[2] W.R. Stevens, S. A. Rago, *Advanced Programming in the UNIX Environment, 3rd ed.*, Addison-Wesley, 2013.

[3] M. Kerrisk, *The Linux System Programming Interface*, No Starch Press, 2010.

Poglavje 10

Funkcije za upravljanje datotek

V tem poglavju si bomo ogledali osnovne sistemske klice in funkcije za upravljanje datotek in vhodno izhodnih prenosov. Spoznali bomo koncept *datotečnih deskriptorjev*. Za uvod pa sledi kratek pregled važnejših funckij.

10.1 Pregled funkcij

Osnovni sistemski klici oziroma funkcije za vhodne in izhodne prenose ter datotečni sistem so:

`open`, `creat`, `read`, `write`, `close` in `lseek`.

Z `open` datoteko odpremo in tako pridobimo dostop do obstoječe datoteke ali naprave. S `creat` ustvarimo novo datoteko, ki je sprva brez podatkov in zatorej prazna. Z `read` in z `write` beremo in pišemo vsebino datoteke, dokler je ne zapremo s `close`. Datoteka je sekvenčna podatkovna struktura, zato se z branjem ali pisanjem pomikamo po nizu podatkov naprej, lahko pa se s funkcijo `lseek` svobodno pomikamo vzdolž podatkov naprej in nazaj.

Z navedenimi funkcijami dostopamo direktno do storitev sistemskega jedra.

Večini programerjev v programskem jeziku C pa so bolj znane funkcije standardne knjižnice `stdio`:

`fopen, fread, fwrite, fseek` in `fclose`.

Funkciji `fread` in `fwrite` realizirati medpomnenje v okviru uporabniškega procesa, do prehoda v sistemsko jedro prek funkcij `read` in `write` pa pride samo ko je potrebno. To se odraža v manjšem številu prehodov med uporabniškim in sistemskim načinom in posledično hitrejšem napredovanju procesa.

Za delo z datotekami obstaja še veliko drugih klicev in funkcij, med njimi:

- `dup, dup2` za podvajanje datotečnih deskriptorjev,
- `fcntl, ioctl, stat` za upravljanje datotečnih določil,
- `link, unlink` za dodajanje in odvzemanje datotečnih povezav,
- `remove, rename` za brisanje in preimenovanje datotek,
- `mkdir, rmdir, opendir, readdir, closedir` za upravljanje direktorijev,
- `chmod, chown` za spreminjanje datotečnih določil ter lastnika datoteke,
- `mknod` za kreiranje posebnih datotek naprav in
- `mount, umount` za nameščanje in umikanje datotečnih sistemov.

Vseh si ne bomo ogledali.

10.2 Datotečni deskriptor

Vsaka datoteka, ki jo nek proces odpre, je za proces enolično določena z datotečno številko oziroma datotečnim `deskriptorjem`. Deskriptor je majhno nenegativno celo število. V starejših sistemih je bilo število deskriptorjev na posamezen proces, in s tem število sočasno odprtih datotek, omejeno na 20 in vrednosti deskriptorjev na $\{0, 1, ..., 19\}$. Standard SUS3 predpisuje, da je število sočasno odprtih datotek na proces najmanj 20, sicer pa je to parameter, ki je odvisen od konfiguracije sistema. V novejših sistemih je

dovolj velik, da nas to posebej ne skrbi. A če nas največje možno število sočasno odprtih datotek zanima, ga lahko preverimo z ukazom školjki

```
getconf OPEN_MAX
```

Število hkrati odprtih datotek je lahko nekaj tisoč in še več.

Po dogovoru so deskriptorji 0, 1 in 2 rezervirani za standardno vhodno datoteko (tipkovnico), standardno izhodno datoteko (zaslon) ter datoteko za sporočila napak (zaslon). Namesto dejanskih vrednosti deskriptorjev teh datotek se priporoča uporaba simboličnih konstant:

`STDIN_FILENO, STDOUT_FILENO in STDERR_FILENO.`

Te so definirane v *unistd.h*, imajo pa dejanske vrednosti $0, 1$ in 2.

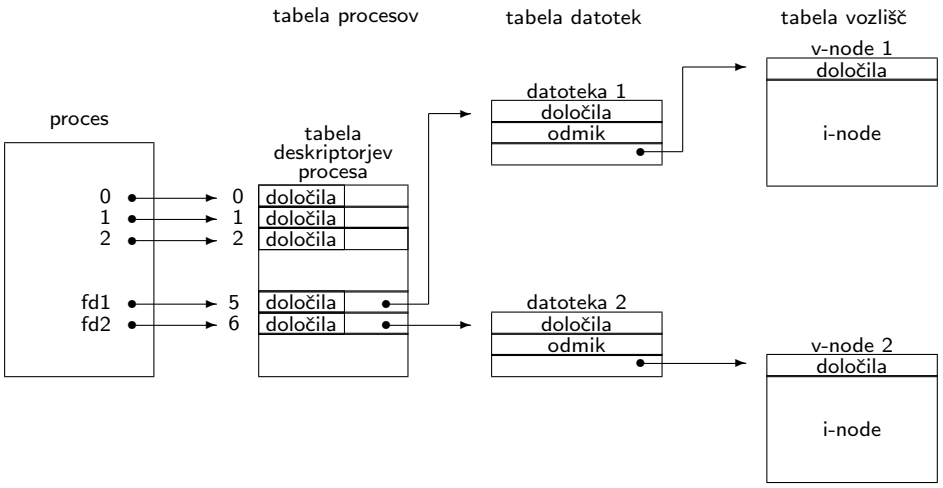
V okviru sistemskega jedra obstajajo tri strukture, v katerih je zabeležena vsaka odprta datoteka[1]:

- tabela procesov (Angl. Process table),
- tabela datotek (Angl. File table) in
- tabela datotečnih vozlišč(Angl. v-node table).

Relacije med tabelami ponazarja slika 10.1. V tabeli procesov je za vsak proces, poleg številnih drugih določil vezanih na proces, zabeležena tabela deskriptorjev odprtih datotek. V tej tabeli se za vsako odprto datoteko beležijo določila in kazalec na pripadajočo datotečno strukturo v tabeli datotek. Datotečna struktura vsebuje dodatna določila in pa *odmik* (Angl. File offset). Odmik določa položaj vzdolž niza podatkov datoteke, kjer bo delovala operacija nad vsebino datoteke. Na primer, z branjem 128 bajtov podatkov, se vrednost odmika poveča za število prebranih bajtov, v tem primeru za 128. Naslednja operacija bo delovala od tam naprej. V datotečni strukturi je še kazalec na pripadajoči `v-node`. Ta vsebuje kopijo indeksnega vozlišča (`i-node`), ki kaže naprej na podatkovne bloke ter dodatna določila.

Slika 10.1 prikazuje podatkovne strukture jedra za dve odprti datoteki. Deskriptor ene je pet in je shranjen v spremenljivki `fd1`. Deskriptor druge

[1]Na konceptualnem nivoju, dejanski izgled je odvisen od implementacije sistema.



Slika 10.1: *Podatkovne strukture odprtih datotek. Kazalci v tabelo datotek za deskriptorje* `STDIN_FILENO` *0,* `STDOUT_FILENO` *1,* `STDERR_FILENO` *2 niso skicirani.*

je šest in je shranjen v spremenljivki `fd2`.

10.3 Funkciji `open` in `creat`

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open( const char *path, int flags, ... /* mode_t mode */ );
```

Vrne deskriptor datoteke ali -1 v primeru napake.

Funkcija `open` odpre datoteko z imenom `path` in ji dodeli deskriptor. Deskriptor je majhno nenegativno število. `Open` vrne prvi prosti deskriptor, to je najmanjše prosto še nedodeljeno število, a se na to nikoli ne zanašamo. Tretji argument ni obvezen, zato navedba `...` . Argument `flags` predpiše eno od treh osnovnih operacij z datoteko (*beri*, *piši*, *beri in piši*). Simbolična imena in dejanske vrednosti konstant osnovnih operacij so:

- `O_RDONLY`: odpri datoteko samo za branje (dejanska vrednost = 0),
- `O_WRONLY`: odpri datoteko samo za pisanje (dejanska vrednost = 1),

- O_RDWR: odpri datoteko za branje in pisanje (dejanska vrednost = 2).

Na primer:

```
int fd; /* datotecni deskriptor */
...
if( (fd = open("/home/stanek/Vs/Moja.c", O_RDONLY )) == -1){
  printf("Napaka open\n");
}
```

odpre datoteko z imenom /home/stanek/Vs/Moja.c za branje. Če datoteka ne obstaja, vrne -1 in tako javi napako.

Skupaj z eno od treh osnovnih vrednosti argumenta `flags` so mogoča dodatna določila, denimo *ustvari, podaljšaj, odreži* in druge, ki jih dodajamo z operacijo ALI (|):

- O_CREAT: ustvari novo datoteko za pisanje
- O_APPEND: pripni novo vsebino datoteki, ki obstaja
- O_TRUNC: odreži datoteko, če obstaja, na dolžino 0

Kot bomo videli prav kmalu, je s funkcijo `open` in z določili `O_WRONLY | O_CREAT` moč ustvariti novo datoteko, pri čemer je obvezen tretji argument `mode`, ki določi, kdo sme datoteko brati, spremeniti in podobno.

Novo datoteko običajno ustvarimo s funkcijo `creat`. Včasih je bil to edini način za kreiranje nove datoteke. Kasneje je to funkcionalost pridobila funkcija `open`, tako da bi sedaj lahko shajali tudi brez klica `creat`.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat( const char *path, mode_t mode );
```

Klic vrne deskriptor novo datoteke ali -1 v primeru napake.

Klic `creat` ustvari novo datoteko z imenom `path` in vrne njen deskriptor. Datoteka še ne vsebuje podatkov in je *prazna*. Argument `mode` definira devet bitov dovoljenj nove datoteke. S temi biti *dovolimo* branje (`r`), pisanje (`w`) ter izvajanje (`x`) lastniku, skupini ter ostalim, kot sledi:

```
lastnik    skupina    ostali
r  w  x    r  w  x    r  w  x
```

Bit v stanju ena da dovoljenje in bit v stanju nič prepove operacijo. Najbolj pogosta vrednost argumenta `mode` je 0644. Vodilna ničla pravi, da gre za osmiški zapis. Vsakemu osmiškemu znaku ustrezajo trije biti, tako da je ekvivalentna binarna kombinacija 0b110100100. To dovoli lastniku datoteke branje in pisanje (števka 6), skupini iz katere je uporabnik branje (števka 4), in vsem ostalim tudi branje (števka 4). Lastnik je tisti, ki je datoteko ustvaril, skupina pa tista, v katero spada lastnik. Klic

```
fd = creat( "ime.dat", 0644 );
```

je ekvivalenten klicu

```
fd = open( "ime.dat", O_WRONLY | O_CREAT | O_TRUNC, 0644 );
```

Branje datoteke odprte s `creat` potemtakem ni mogoče. Če bi želeli datoteko ob tem tudi brati, bi jo morali naknadno odpreti s funkcijo `open` še za branje ali pa bi klic `creat` enostavno nadomestili z

```
fd = open( "ime.dat", O_RDWR | O_CREAT | O_TRUNC, 0644 );
```

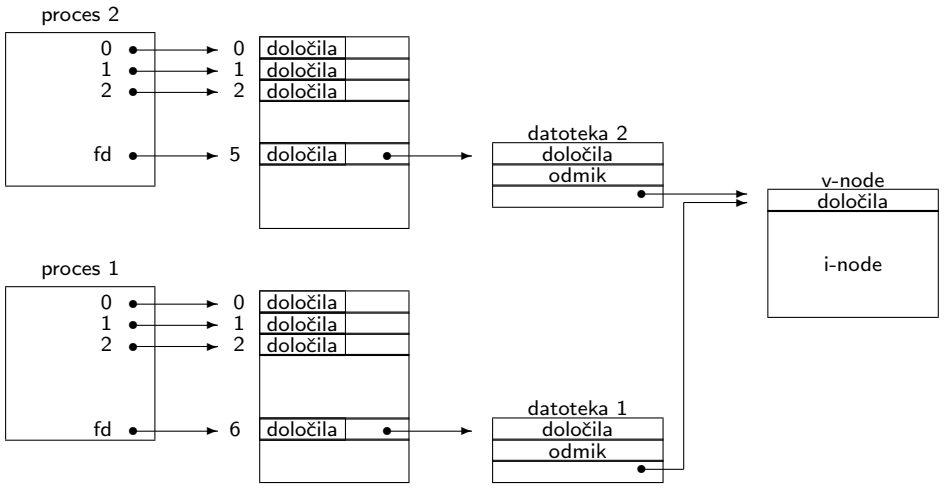
Do iste datoteke ima lahko sočasen dostop več programov oziroma procesov, seveda če določila datoteke to dovoljujejo. Tedaj, ko prvi proces datoteko odpre, se formira zanjo datotečna struktura v tabeli datotek in vozlišče v tabeli vozlišč, slika 10.2. Ko naslednji proces odpre isto datoteko, se v tabeli datotek formira nova datotečna struktura, ki kaže na isto datotečno vozlišče. Vsak od procesov torej uporablja svoj odmik za operacije nad vsebino datoteke. Isto datoteko bi seveda lahko na novo odprl tudi isti proces.

10.4 Funkcija `close`

```
#include <unistd.h>
int close( int fd );
```

Vrne 0 in datoteka je zaprta ali -1 v primeru napake.

Funkcija `close` zapre datoteko, ki ji je pridružen deskriptor `fd`. S tem



Slika 10.2: *Dostop dveh procesov do iste (skupne) datoteke. Vsak proces dostopa do datoteke na osnovi svojega deskriptorja. Deskriptorja nista povezana in sta različna. V tabeli datotek pripada vsakemu procesu po ena datotečna struktura za isto datoteko. Datotečni strukturi kažeta na skupno vozlišče.*

je datoteka zaključena oziroma operacije z datoteko niso več mogoče, a deskriptor postane ponovno prost. Ker proces, ki konča z `exit` oziroma bolje rečeno jedro, zapre vse odprte datoteke, se `close` često opušča. A na avtomatično zapiranje datotek se ne gre zanašati.

10.5 Funkciji `read` in `write`

S funkcijo `read` beremo ali sprejemamo in s funkcijo `write` pišemo ali sprejemamo. Čeprav se funkciji povezuje z operacijimi branja in pisanja na datotekah, sta funkciji enakovredno uporabni za delo z zunanjimi napravami, s cevmi in z vtičnicami, o čemer bomo več povedali kasneje.

```
#include <unistd.h>
ssize_t read( int fd, void *buff, size_t nbytes );
```

Vrne dejansko število prebranih bajtov ali -1 v primeru napake.

Funkcija `read` prebere iz datoteke `fd` v pomnilnik kamor kaže kazalec `buff`

največ nbytes bajtov. Če je podatkov manj, jih prebere toliko, kolikor jih je na razpolago. Klic vrne dejansko število prebranih bajtov ali -1 v primeru napake. Operacija read na koncu datoteke vrne 0. To sicer pomeni, da ni bil prebran noben podatek. Zato se ničlo, sicer nepravilno, včasih enači z znakom za konec datoteke.

Pozor: argument nbytes vedno pomeni število bajtov, ne glede na resnični tip branih podatkov.

Naslednji kos programa odpre za branje (flags = O_RDONLY) datoteko z imenom podatki in iz nje prebere NPODATKOV podatkov tipa float v polje mp.

```
...
#define NPODATKOV  16
...
int   fi, n_read, n_bajtov;
float mp[16];
...
if ((fi = open("podatki", O_RDONLY)) == -1){
  printf("Napaka pri odpiranju datoteke podatki\n");
  exit (1);
}
n_bajtov = NPODATKOV * sizeof(float);
n_read   = read(fi, mp, n_bajtov );

if (n_read == -1){
  printf("Napaka pri branju datoteke podatki\n");
  exit (1);
}
if (n_read != n_bajtov){
  printf("Prebranih samo %d bajtov\n", n_read);
  exit(1);
}
close (fi);
...
```

S funkcijo write zapišemo ali pošljemo izbrano število podatkov.

```
#include <unistd.h>
ssize_t write( int fd, const void *buff, size_t nbytes );
```

Vrne dejansko število zapisanih bajtov, ali -1 v primeru napake.

Funkcija `write` zapiše `nbytes` podatkovnih bajtov iz pomnilnika kamor kaže `buff` v datoteko `fd`. Vrednost, ki jo vrne funkcija, je v primeru uspešnega zapisa podatkov enaka `nbytes` in različna od `nbytes` v primeru napake. Vzrok je običajno pomankanje pomnilniškega prostora – diska, ali pisanje v datoteko, ki ne dovoljuje pisanja.

10.6 Funkcija `lseek`

Vsaki odprti datoteki pripada relativen *odmik* od začetka datoteke do trenutnega mesta pisanja ali branja. Odmik je zabeležen v tabeli datotek jedra, slika 10.1. Odmik se po vsaki operaciji branja ali pisanja poveča za prebrano število bajtov. Z operacijama branja ali pisanja se torej pomikamo naprej po datoteki. S funkcijo `lseek` se lahko dokaj svobodno pomikamo nazaj in naprej po datoteki ter datoteko ponovno prebiramo in po potrebi spreminjamo. Funkcija `lseek` omogoča direkten dostop do podatkov datoteke, ki je sicer po definiciji sekvenčnega značaja.

```
#include <sys/types.h>
#include  <unistd.h>
off_t lseek( int fd, off_t offset, int whence );
```

Vrne nov odmik, -1 v primeru napake.

Funkcija `lseek` spremeni datotečni odmik za `offset` relativno glede na začetek datoteke, ali glede na trenutno vrednost odmika ali glede na konec datoteke, kot to določa argument `whence`:

- `SEEK_SET` absolutni pomik glede na začetek datoteke,
- `SEEK_CUR` relativni pomik glede a trenutni položaj,
- `SEEK_END` absolutni pomik glede na konec datotek.e

Funkcija vrne novo vrednost odmika (v bajtih od začetka datoteke) ali -1 v primeru napake.

Poglejmo si nekaj primerov:

```
Kje   = lseek( fd, 0, SEEK_CUR );   /* vrne odmik, brez premika */
Start = lseek( fd, 0, SEEK_SET );   /* na zacetek datoteke      */
Back  = lseek( fd, -10, SEEK_CURR );/* nazaj za 10 bajtov       */
End   = lseek( fd, 0, SEEK_END );   /* na konec datoteke        */
CezEnd= lseek( fd, 10, SEEK_END );  /* 'preko' konca datoteke   */
```

10.7 Funkciji dup in dup2

Funkciji `dup` in `dup2` podvojita deskriptor `fd` že odprte datoteke. Ista datoteka je potem dostopna prek dveh enakovrednih deskriptorjev. In zakaj je podvajanje deskriptorjev sploh potrebno? No, na to bomo odgovorili v nadaljevanju, najprej pa deklaraciji funkcij:

```
#include <unistd.h>
int dup( int fd );
int dup2( int fd, int fdupd );
```

Vrneta nov deskriptor ali -1 v primeru napake.

Funkcija `dup` vrne prvi prosti deskriptor, funkcija `dup2` vrne zahtevani deskriptor `fdupd` in pred tem po potrebi zapre datoteko z deskriptorjem `fdupd`. Torej:

```
close( 0 ); dup( fd );
```

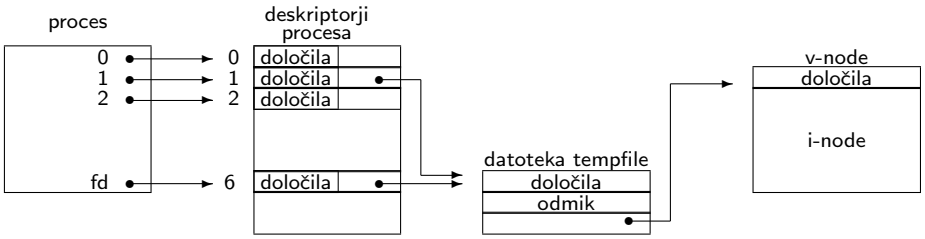
je identično

```
dup2( fd, 0 );
```

Po povratku funkcije je ista datoteka dosegljiva z dvema različnima deskriptorjema. Tipični primeri uporabe funkcije `dup` so preusmerjanje branja ali pisanja s standardnih datotek v izbrano datoteko ter cevi (Angl. pipes).

Naslednji kos programa preusmeri izpis s standardnega izhoda v datoteko z imenom `tempfile`, slika 10.3.

```
...
fd = creat( "tempfile", 0644 );
close( 1 ); dup( fd ); close( fd );
write(1, ...); /* izpis gre v datoteko tempfile */
...
```



Slika 10.3: *Dostop do datoteke s podvojenim deskriptorjem. Proces ima najprej odprte standardne datoteke za branje, pisanje in napake, z deskriptorji 0, 1 in 2. Nato odpre datoteko* `tempfile` *za pisanje. Potem zapre standardno izhodno datoteko, s tem sprosti deskriptor 1. Zatem z* `dup(fd)` *podvoji deskriptor* `fd`*, ki ima sicer vrednost 6 na prvi prosti deskriptor, ki je 1. Zato* `write(1,...)` *piše v datoteko* `tempfile`*. Rečemo, da smo izhod preusmerili s standardnega izhoda v datoteko. Nazadnje bi še zaprli deskriptor 6, torej* `close(fd)`*.*

To funkcionalnost ima vgrajeno školjka. Na primer, če želimo preusmeriti izpis ukaza `ls` z zaslona v datoteko `temp`, tipkamo > pred imenom datoteke:

```
ls >temp
```

10.8 Funkciji `fcntl` in `ioctl`

Funkciji `fcntl` in `ioctl` imata cel kup dodatnih določil in po potrebi modificirata vhodno izhodne operacije za odprto datoteko (ali terminal) ter lastnosti datoteke same. Teh funkcij tu ne bomo pojasnjevali, podroben opis pa se nahaja in je dosegljiv na straneh priročnika,

```
man fcntl
man ioctl
```

10.9 Velike datoteke

V klasičnem sistemu Linux na 32 bitnih aritekturah je datotečni odmik tipa `off_t` dejansko 32 bitno celo število s predzakom `long int`. To omejuje velikost datoteke na $2^{31} - 1$ ali 2 GB. Čeprav to ni malo, je včasih premalo. Rešitev je v 64 bitnem odmiku (`long long`). To dosežemo z makro definicijo

```
#define _FILE_OFFSET_BITS 64
```

v prvi vrstici izvornega programa. Na 64 bitnih arhitekturah z velikostjo odmika ni težav.

10.10 Program za prepis datoteke

Naslednji program prepiše datoteko z imenom argv[1] na datoteko z imenom argv[2]. Prepisovanje poteka v blokih velikosti, ki jo podamo s tretjim argumentom argv[3]. Če tretji argument opustimo, je privzeta velikost bloka NPOD bajtov. Glej tudi pomen bitov dovoljenj.

```
/* ---------------------- V-S ------------------------------
   Program: mcp
        program za prepis datoteke.

   Uporaba: mcp stara nova [velikost bloka]
            stara: ime obstojece datoteke
            nova : ime nove datoteke
    [velikost bloka]: stevila podatkov, ki se prepise v enem kosu

*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <malloc.h>

#define NPOD 512   /* privzeta velikost bloka za prepis z read/write*/


int main( int argc, char *argv[] )
{
  char    *pom;
  int     fi, fo;
```

```
  ssize_t n_pod, p_pod;


  if (argc == 4){
    sscanf(argv[3],"%ld", &n_pod );
  }
  else if (argc == 3){
    n_pod = NPOD;
  }
  else{
    printf("\nUPORABA: %s <vhodna dat> <izhodna dat> [velikost bloka]\n\n",argv[0]);
    exit(1);
  }

/* odpremo datoteko za branje, ime datoteke doloca argv[1]       */

  if ((fi = open(argv[1], O_RDONLY)) == -1){ /* O_RDONLY je seveda 0 */
    printf("%s: napaka open %s\n", argv[0], argv[1]);
    exit(2);
  }

/* odpremo(ustvarimo) datoteko za pisanje, ime doloca argv[2]    */
/* biti dovoljenj: rwxrwxrwx lastnik(rwx) skupina(rwx) vsi(rwx)  */
/* 1 = dovoli, 0 = prepove, r = read, w = write, x = execute     */
/* 0644(osmisko) = 110 100 100 (dvojisko) = rw-r--r--(dovoljenja)*/
/* rw za lastnika, r za skupino, r za vse uporabnike             */


  if ((fo = creat(argv[2], 0644)) == -1){
    printf("%s: napaka creat %s\n", argv[0], argv[2]);
    exit(3);
  }
/* Zagotovimo si pomnilnik za podatke, najvec Npodatkov    */

  if ((pom = malloc( n_pod )) == NULL){
    printf("%s: napaka malloc\n", argv[0]);
    exit(4);
  }
```

```
/* prepisujemo podatke do konca datoteke                    */

  printf("%s: velikost bloka %ld bajtov\n", argv[0], n_pod);

  while ((p_pod = read(fi, pom, n_pod)) != 0){
    if (p_pod == -1){
      printf("%s: napaka read %s\n", argv[0], argv[1]);
      exit(5);
    }
    if( write(fo, pom, p_pod) != p_pod){
      printf("%s: napaka write %s\n", argv[0], argv[2]);
      exit(6);
    }
  }
  if ((close(fi) == -1) || (close(fo) == -1)){
    printf("%s: napaka close\n",argv[0]);
    exit(7);
  }
  exit(0);

} /* Konec main() */
```

10.11 Funkcije fopen, fread, fwrite, fclose, fseek

To so funkcije standardne vhodno/izhodne knjižnjice za neformatirane binarne prenose podatkov. Izravnava (medpomnjenje) podatkov se opravlja razen v jedru tudi na nivoju uporabnikovega programa, sicer pa imajo funkcije podoben pomen kot sistemski klici `open`, `read`, `write`, `close`, `lseek`. Torej jih ne bomo podrobno obravnavali.

```
#include <stdio.h>

FILE   *fopen(const char *pathname, const char *type);
size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream);
int    fclose(FILE *stream);
```

```
int    fseek(FILE *stream, long int offset, int whence);
```

Funkcija `fopen` odpre datoteko oziroma *podatkovni tok* (Angl. *Stream*) z imenom `pathname`. Operacijo določa argument `type`: `''r''` za read, `''w''` za write, so pa še druge možnosti. Klic vrne datotečni kazalec na strukturo tipa `FILE` ali `NULL` v primeru napake. Ta kazalec je nato referenca na podatkovni tok za vse kasnejše operacije `fread, fwrite` in tako dalje. Datotečni kazalec ima torej podobno vlogo kot datotečni deskriptor.

Funkcija `fread` prebere in funkcija `fwrite` zapiše ustrezno število `nitems` podatkovnih enot velikosti `size` bajtov. Funkciji vrneta dejansko število prenešenih podatkov ali -1 v primeru napake.

Naslednji program prepiše datoteko z imenom argv[1] na datoteko z imenom argv[2]. Preverjanje napak pri branju in pisanju je opuščeno.

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

int main( int argc, char **argv )
{
  FILE *fp_in, *fp_out;
  int  mp[512];
  size_t n;

  if( argc != 3 ){
    printf("UPORABA: %s VhodnaDatoteka IzhodnaDatoteka\n", argv[0]);
    exit( 1 );
  }
  if( (fp_in = fopen( argv[1], "r" )) == NULL){
    printf("Napaka fopen na %s\n", argv[1]); exit( 2 );
  }
  if( (fp_out = fopen( argv[2], "w" )) == NULL){
    printf("Napaka fopen na %s\n", argv[2]); exit( 3 );
  }
  while( (n = fread( mp, sizeof(int), 512, fp_in )) > 0 ){
    fwrite( mp, sizeof(int), n, fp_out );
  }
  fclose( fp_in ); fclose( fp_out );
```

```
  exit( 0 );
}
```

10.12 Druge vhodno izhodne in sorodne funkcije

Funkciji `fread, fwrite` sta za branje in pisanje podatkov v binarni obliki – podatke prenašamo take kot so, brez pomenske interpretacije. Formatiran vhod/izhod opravita funkciji `fscanf` in `fprintf` oziroma v primeru standardne vhodne ter izhodne datoteke funkciji `scanf` in `printf`, glej `man fprintf` in `man fscanf`.

Sorodni tem funckijam sta funkciji `sscanf` in `sprintf`. Delujeta enako le nad znakovnim nizom namesto nad datoteko.

Za delo z znakovnimi nizi (Angl. String) in s posameznimi črkami, kot sta branje in pisanje, so na voljo funkcije `fgets, fputs, fgetc, fputc, gets, puts, getc, getchar, putc, putchar`.

Prepisovanje, primerjanje, združevanje, razčlenjevanje, itd., opravljajo funkcije s predpono `str`, na primer: `strcpy, strcmp, strcat, strtok, strlen`. Znankovni niz je poljubno zaporedje alfanumeričnih znakov z ničelnim bajtom na koncu. `strlen( niz )` vrne dolžino niza, nevštevši ničelni bajt.

Sorodne tem funckijam so funkcije za manipulacijo s kosi pomnilnika. Imena teh začenjajo s predpono `mem`. Na primer `memcpy, memcmp` in druge.

Naslednje programsko besedilo daje primer podprograma – funkcije getargs.c – za analizo 'ukazne' vrstice, ki jo odtipkamo na tipkovnici oziroma v terminalskem oknu. Vrstica – znakovni niz – se prebere s standardne vhodne datoteke s funkcijo `fgets` in analizira (razčleni na besede) s funkcijo `strtok`. Naša funkcija `getargs` vrne polje kazalcev na podnize (besede) v odtipkani vrstici.

```
/* ----------------------------------------------------------------
   Funkcija za analizo niza - ukazne vrstice

   int getargs( char **argv, int maxarg );
   char **argv;   polje kazalcev na argumente
```

```
   int  maxarg;   najvecje dovoljeno stevilo argumentov

   Vrne: >= 0 - dejansko stevilo argumentov v prebrani vrstici
         -1   - napaka, prevec argumentov;
         -2   - prazna vrstica;

   Primer uporabe:
   void main( void )
   {
     char *Argumenti[12];
     int  Status;
     while( (Status = getargs( Argumenti, 12 )) > 0){
       printf("OK\n");
     }
     if(Status == -2)exit(0);
   }
*/

#include <stdio.h>
#include <string.h>


#define MAX_LINE 128

int getargs( char **argv, int maxarg )
{
  static char vrstica[MAX_LINE];
  char   *p_vrstica;
  int    i;

  if( fgets( vrstica, MAX_LINE, stdin ) == NULL)
    return -2;

  vrstica[strlen(vrstica)-1] = '\0'; /* \n nadomestimo z \0 */

  p_vrstica = vrstica; /* vrstico razclenimo na nize, locene s presledkom */
  for(i = 0; i <= maxarg; i++){
    if((argv[i] = strtok(p_vrstica," ")) == NULL)  /* glej man strtok */
```

```
      break;
    p_vrstica =  NULL; /* to zahteva strtok() funkcija */
  }
  if( i > maxarg ){
    printf("Prevec argumentov v ukazni vrstici\n");
    return -1;
  }
  return i; /* stevilo argumentov */
}
```

Poglavje 11

Funkcije za upravljanje procesov

Doslej smo spoznali koncept procesa, stanja procesa in prehajanje med stanji procesa. V tem poglavju bomo spoznali osnovne sistemske klice in funkcije za upravljanje procesov. Najprej pa sledi pregled glavnih funkcij.

11.1 Pregled glavnih funkcij

Osnovni sistemski klici za upravljanje procesov v sistemu UNIX/Linux so:

`fork, exec, wait` in `exit`.

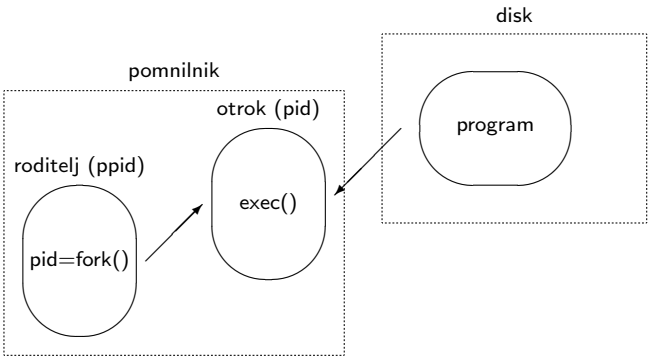
S `fork()` nastane nov proces, z `exec()` se proces inicializira iz programa in z `exit()` se proces konča, slika 11.1. Z `wait()` je moč počakati na konec procesa.

Proces, ki se izvaja, s `fork()` od sistemskega jedra zahteva, da ustvari nov proces. Zato temu procesu rečemo roditelj ali straš, novi proces pa je njegov otrok.

Enkrat, ko proces obstaja, se lahko inicializira iz programa. Program je izvršliva datoteka na disku, ki se kot posledica klica `exec()` namesti v po-

mnilnik in izvrši.

Regularen zaključek procesa se zgodi s funkcijo `exit`. Po potrebi lahko roditelj z `wait` čaka na konce otroka.



Slika 11.1: *Proces roditelj s* `fork()` *zahteva od sistemskega jedra naj ustvari nov proces. Jedro ustvari nov proces (otroka), ki po povratku* `fork()` *napreduje povsem neodvisno in asinhrono od roditelja. Otrok je skoraj identičen roditelju, dokler (če sploh) se z* `exec()` *ne inicializira iz programa.*

11.2 Številka procesa

Vsak proces je v sistemu enoznačno določen s številko procesa. Ko se sklicujemo na številko procesa, skoraj vedno uporabimo kratico `PID`. Kratica izvira iz angleškega izraza 'Process IDentity'. Številko roditelja napram otroku označujemo s `PPID`, kar je krajše za 'Parent PID'.

Številka procesa je tipa `pid_t`, sicer pa je pozitivno celo število. Noben proces nima številke nič. Najvišja številka procesa je odvisna od konfiguracije jedra in je običajno največja pozitivna vrednost v obsegu 16 bitnih predznačenih števil. S tem je navzgor omejeno število sočasnih procesov.

Ob začetnem zagonu ali ob ponovni vzpostavitvi sistema (Angl. Boot, Reboot) se v pomnilnik namesti sistemsko jedro. Sistemsko jedro na koncu vzpostavitve ustvari proces s številko PID=1 in imenom `init`. Proces `init` nadaljuje oziroma dokonča začetno vzpostavitev sistema ter nato pritajeno 'bedi' cel čas delovanja sistema. Vsi kasnejši procesi so bližnji ali dalnji

potomci procesa `init`.

Številke novih procesov v sistemu s časom praviloma naraščajo, dokler ne preplavijo. Ko se to zgodi, se novo nastalim procesom spet dodelujejo manjše številke od neke najmanjše številke navzgor, seveda če niso še v uporabi. Dejanske vrednosti niti niso pomembne, važno je, da so enoznačne oziroma različne. Izbira številk je v pristojnosti jedra.

11.3 Funkcija `fork`

Funkcija `fork()` daje edino možnost za nastanek novega procesa. Druge možnosti kot je klic funkcije `fork()` za nastanek procesa ni.

```
#include <unistd.h>

pid_t  fork( void );
```

Klic vrne 0 otroku, PID otroka roditelju ali -1 v primeru napake.

S `fork()` proces, ki mu rečemo roditelj (Angl. Parent) ustvari nov proces, ki mu rečemo otrok (Angl. Child). Otrok je nov, samostojen proces. Jedro mu dodeli unikatno številko (PID), sicer pa je skoraj identičen roditelju.

Jedro za otroka formira procesni deskriptor, ki je najprej kar kopija roditeljevega deskriptorja. Otrok je zaenkrat v stanju neprekinljivo ustavljen (Angl. Uninterruptable sleep). Jedro mu dodeli tabelo strani in prek nje naslovni prostor, ki ga podeduje od roditelja. To pomeni, da je kontekst pomnilnika otroka enak kontekstu pomnilnika roditelja, tedaj ko se `fork()` vrne. A zavedati se moramo, da sta roditeljev in otrokov naslovni prostor dva ločena naslovna prostora.

Otrok od roditelja podeduje tudi deskriptorje odprtih datotek in druge lastnosti roditelja, a sedaj je še prezgodaj, da bi govorili o njih. Jedro nato oseveži nekatere komponente procesnega deskriptorja, kot so številka procesa, čas nastanka, procesorski čas in še nekatere druge komponente, ki so predvsem statističnega značaja.

Ko se klic vrne, se izvajanje obeh procesov nadaljuje z ukazom, ki sledi

klicu `fork()`. Zato rečemo, da se `fork()` vrne obema, otroku in roditelju. Sicer pa oba procesa napredujeta povsem neodvisno eden od drugega ali asinhrono. Kateri od obeh morebiti dobi prej procesno enoto, je odvisno od pravila razvrščanja in izvedbe razvrščevalnika.

Naslednje ogrodje programskega besedila ponazarja uporabo funkcije `fork()`.

```
....
if( (pid = fork( )) == 0){
  printf("Otrok\n");  /* Fork vrne otroku vrednost 0       */

/* tu sledi poljubno programsko besedilo ter obicajno exec */

}
else if( pid == -1){
  printf("Napaka, proces ni ustvarjen\n");
}
else{
  printf("Roditelj, ustvaril sem proces s pid = %d\n", pid);

/* Roditelj lahko tu caka na konec otroka, ali pa enostavo nadaljuje */

}
```

11.4 Funkciji `exit` in `_exit`

Proces lahko končna regularno ali neregularno,

- *Regularno*: z `return` iz funkcije `main()`, ali od kjerkoli z `exit()` ali z `_exit()`, glej tudi sliko 9.5.
- *Neregularno*: na lastno pobudo s klicem `abort()` ali na zahtevo od zunaj s sprejemom *signala*.

Signal je neke vrste programska prekinitev, preko katere en proces zahteva od drugega procesa, da konča. O tem bomo govorili kasneje.

Klic _exit() povzroči takojšni konec procesa in povratek k jedru. Return iz funkcije main() in exit() sta si ekvivalentna in končata s klicem _exit(). Funkcija exit() lahko opravi pred povratkom z _exit() še določene zaključne operacije, ki so včasih potrebne (npr. čiščenje v/i medpomnilnikov). V primeru, da se eksplicitni klic ustrezne funkcije v main opusti, se privzame povratek return.

```
#include <stdlib.h>
void exit( int exitstatus ); /* ekvivalentno return( exitstatus ) */

ali

#include <unistd.h>
void _exit( int exitstatus );
```

Preko argumenta exitstatus lahko proces, ki konča z exit(), javi procesu, ki ga čaka z wait(), na kakšen način je končal. Funkcijo wait() bomo spoznali v nadaljevanju. Na primer,

```
int main( )
{
  ...
  exit(0); /* ali return(0) --- povratek z 0, roditelj dobi 0 */
}
```

Vrednost nič je dogovorjena za regularen konec procesa, pozitivne vrednosti $(1, 2, 3, \ldots, 255)$ pomenijo napake, vendar je to popolnoma stvar dogovora oziroma aplikacije. Za jedro je dejanska vrednost argumenta status nepomembna. Vrednosti argumenta status večje od 255 nimajo smisla, saj višje bite jedro maskira in vrne (status & 0xff).

11.5 Funkciji wait in waitpid

Funkcija wait() omogoča roditelju, ki ustvari otroka, da čaka oziroma se sinhronizira na njegov konec.

```
#include <sys/types.h>
#include <sys/wait.h>
```

```
pid_t wait( int *statloc );

pid_t waitpid( pid_t pid, int *statloc, int options );
```

Vrne -1, če ni otroka za čakanje, ali PID dočakanega procesa-otroka.

Načeloma lahko proces z večkratnim klicem `fork()` ustvari večje število procesov, vsi so njegovi otroci. Če je potrebno, lahko roditelj z `wait()` čaka, da se eden od otrok konča. Funkcija `wait()` ni selektivna. Z njo proces počaka na otroka, ki prvi konča. Če bi proces-roditelj moral počakati na konec vseh svojih otrok, bi to lahko opravil s ponavljanjem klica `wait()`, dokler le-ta ne vrne -1, kar pomeni, da proces več nima otrok. Ko se `wait()` vrne, vsebuje `statloc` način, kako je proces končal, in sicer:

- v primeru regularnega konca je spodnji bajt nič in zgornji bajt vsebuje vrednost, ki jo otrok pusti z `exit`.
- v primeru neregularnega konca je spodnji bajt različen od nič in vsebuje številko 'usodnega' signala, ki je pokončal dočakan proces.

Proces-roditelj z več procesi-otroki lahko z `waitpid()` selektivno čaka na konec procesa z dano številko `pid` ali točneje, z `waitpid()` proces čaka na spremembo stanja otroka. Tretji argument (`options`) daje dodatne možnosti za obravnavanje procesov. Ena od teh možnosti je, da proces-roditelj nadaljuje brez čakanja, če otrok ob oziroma pred klicem `waitpid` še ni zaključen.

Procesu-roditelju načeloma ni potrebno čakati na konec procesov, ki jih je ustvaril s `fork()`. Tipični primeri takih procesov so procesi 'v ozadju', tako imenovani pritajeni procesi ali 'demoni'. Demoni so dolgo živeči procesi, ki nimajo razloga za interakcijo z uporabnikom.

V primeru, ko roditelj ne čaka na konec otroka, lahko konča tudi prej, otrok tedaj postane 'sirota', ki ga posvoji proces s številko 1, to je proces `init`. Če pa otrok konča pred roditeljem, a roditelj še ne čaka ali sploh ne bo čakal na njegov konec, postane otrok 'opuščen' (angl. zombie) ali (angl. defunct). To je stanje procesa, ko je proces sicer sprostil dodeljena mu sredstva, kot je na primer pomnilnik, vendar ostane zabeležen v tabeli procesov, od kjer

bi lahko roditelj pridobil njegovo stanje z `wait()`.

Slika 11.2: *Struktura in pomen vsebine spremenljivke* `statloc`*, ki jo za proces-roditelj pusti dočakani proces-otrok z* `exit( n )` *oziroma številka signala, ki je povzročil konec procesa otrok.*

11.6 Funkcije `exec`

Družina funkcij `exec` poskrbi za inicializacijo procesa iz programa in njegovo izvršitev. Ko rečemo inicializacija, mislimo na namestitev programa v pomnilnik, posodobitev relevantnih komponent procesnega deskriptorja in potencialno izvršitev. Vendar pozor, z `exec` ne nastane nov proces. Z `exec` proces inicializira samega sebe.

Na voljo je šest različic klica `exec`: `execl, execv, execle, execve, execlp, execvp`. Funkcije se med sabo razlikujejo zgolj po tem, kako podamo argumente, a učinek funkcije je vedno isti. Edino `execve` je sistemski klic, medtem ko so ostale različice funkcije, ki na koncu kličejo `execve`.

```
#include <unistd.h>
extern char **environ;

int execl( const char *path, const char *arg0, ...
           /* const char *arg1, ..., const char *argn, (char *)0 */
         );

int execv( const char *path, char * const argv[]);

int execle( const char *path, const char *arg0, ...
            /* const char *arg1,..., const char *argn, (char *)0,
               char * const envp[] */
          );
```

```
int execve(const char *file, char * const argv[], char * const envp[]);

int execlp(const char *file, const char *arg0, ...
           /* const char *arg1,..., const char *argn, (char *)0 */
          );

int execvp(const char *file, char * const argv[]);
```

Funkcije vračajo -1 v primeru napake, kar pomeni, da inicializacija procesa ni uspela. Zvedati se moramo, da v primeru, ko `exec` uspe, do povratka sploh ne pride.

Klici in torej imena klicev se razlikujejo zgolj glede na način podajanja argumentov, na način podajanja imena programa, iz katerega se proces inicializia, in glede na prenos 'okolja' procesa (Angl. environment):

- l: podajanje argumentov v obliki seznama kazalcev (l kot list)
- v: podajanje argumentov v obliki polja kazalcev (v kot vector)
- p: upoštevanje spremenljivke PATH (p kot path)
- e: podajanje novega okolja v obliki polja kazalcev (e za environment), v nasprotnem primeru je "dedovanje" spremenljivk okolja avtomatično preko globalne spremenljivke `**environ`.

Naslednji kos programa nakazuje uporabo funkcije `execlp`.

```
...
int main( void )
{
  pid_t p;
  int   s;
  if( (p = fork( )) == 0){
    execlp("ls","ls", (char *)0 ); /* izvrsitev ukaza (programa) ls  */
    exit( 1 );                     /* ce exec ne uspe, sledi exit(1) */
  }else{
    p = wait( &s );                /* cakam, da otrok konca          */
    printf("Proces-otrok PID = %d koncan, exit status %d\n", p, s >> 8);
    exit( 0 );
```

```
  }
}
```

V zgornjem primeru smo opustili preverjanje, kaj `execlp` vrne. To smo naredili namenoma in z dobrim razlogom. Namreč, če klic uspe, se `execlp` sploh ne vrne. Če pa se vrne, potem je jasno, da inicializacija ni uspela.

11.7 Funkciji `getpid` in `getppid`

S funkcijo `getpid()` proces pridobi svojo številko.

```
#include <unistd.h>
pid_t getpid(void);
```

Funkcija `getppid()` vrne številko roditelja.

```
#include <unistd.h>

pid_t getppid(void);
```

Klic ene ali druge funkcije vedno uspe.

11.8 Funkcija `system`

Funkcija `system` daje večini uporabnikov najprikladnejši način za izvršitev ukazne vrstice s sintakso školjke (lupine) znotraj programa. V prenesenem smsilu realizira klic `fork, exec` in `wait`.

```
#include <stdio.h>
int system( const char *String );
```

Primer uporabe te funkcije ponazarja naslednje ogrodje programa.

```
  ...
  system( "ls -l" );    /* izpis tekocega direktorija  */
  ...
  system( "cp sss xxx"); /* prepis datoteke sss na xxx */
  ...
```

Razmislite o možni realizaciji funkcije `system`.

11.9 Preprosta školjka

V nadaljevanju je podan primer zares preproste školjke, ki bere ukazno vrstico, na podlagi ukazne vrstice ustvari proces ter ga, če program obstaja, inicializira. Če privzamemo ime datoteke z glavnim programom msh.c in ime datoteke s funkcijo za analizo ukazne vrstice getargs.c iz enega od prejšnjih poglavij, dobimo prevedeni program v datoteki msh z ukazom:

```
gcc -o msh msh.c getargs.c
```

Datoteka msh.c

```
/* Zelo enostavna skoljka */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define  N_ARGS 16

int main( void )
{
  int getargs( char **arg, int narg); /* nasa funkcija za analizo vrstice */

  char *args[N_ARGS];
  int status, line;
  int pid;
  line = 0;

  while( 1 ){ /* neskoncna zanka */

/* izpisi 'pozornik': pot tekocega direktorija:msh[stevilka vrstice] */

    printf("%s:msh[%d]>", getcwd(NULL, 0), ++line);
```

```
/* --- analiza 'ukazne' vrstice */

    status = getargs( args, N_ARGS );

    switch( status ){
      case  0:
        printf("Prazna vrstica\n");
    break;
      case -1:
        printf("Prevec argumentov\n");
        break;
      case -2:
        printf("Odjava\n");
        exit(0);
      default:

/* --- domnevamo, da gre za zahtevo za izvrsitev programa  */

        if( (pid = fork()) == 0 ){ /* ustvarimo nov proces */
          printf("Pid otroka = %d\n", getpid());
          execvp(args[0], args );  /* ga inicializiramo in izvrsimo */
          printf("Neuspel poskus EXEC\n");
          exit(255);
        }
        else if( pid == -1 ){
          printf("Neuspel fork()\n");
        }
        else{  /* pocakamo, da otrok konca */
          pid = wait( &status );
          if( (status & 0xff) == 0){
            printf("Otrok pid = %d je koncal z exit(%d)\n",pid, status >> 8);
          }
          else{
            printf("Otrok pid = %d je prejel signal %d\n", pid, status);
          }
        }
    }
```

```
  }
}
```

Poglavje 12

Funkcije za upravljanje niti

Niti ustvarjamo, razvrščamo, sinhroniziramo, zaključujemo, in še kaj. Skratka, niti upravljamo. Niti si delijo skupen naslovni prostor danega procesa. Tako kot ima vsak proces svojo unikatno številko (tipa `pid_t`), ima tudi vsaka nit svojo identiteto (ID) tipa `pthread_t`. ID niti mora biti unikatna v okviru istega procesa. ID niti je lahko sestavljena struktura ali kazalec na strukturo, kar je prepuščeno implementaciji. Najpogosteje je ID niti kar pozitivno ali nepreznačeno celo število, vendar se na to ne gre zanašati.

V nadaljevanju si bomo ogledali programski vmesnik za upravljanje niti po specifikaciji POSIX. Imena vseh funkcij začnejo s predpono `pthread_`. Črka 'p' pravi, da gre za 'POSIX sistem niti'. Glavne funkcije POSIX za niti so: `pthread_create`, `pthread_exit`, `pthread_join`, `pthread_detach`, `pthread_cancel`, `pthread_self`, `pthread_equal`. Skupno vsem funkcijam je, da vračajo nič v primeru uspeha in vrednost večjo od nič v primeru napake. Vrednost, ki jo funkcija vrne, je koda napake.

12.1 Funkcija `pthread_create`

Proces, enonitni ali večnitni, se začne izvrševati z eno ali edino nitjo. Med napredovanjem nit, ki napreduje, po potrebi zahteva nastanek nove niti. To se zgodi s klicem sistemske funkcije `pthread_create`.

```
#include <pthread.h>
int pthread_create( pthread_t *restrict tidp,
                    const pthread_attr_t *restrict attr,
                    void *(*start_fn,)( void *),
                    void *restrict arg);
```

Če ni napake, funkcija vrne nič ter ustvari novo nit z identifikacijsko oznako, na katero kaže `tidp`. Običajne lastnosti niti lahko spremenimo z določili, ki jih podamo prek reference `attr`, a običajno pustimo ta določila nespremenjena. Tedaj je dejanska vrednost tega argumenta `NULL`. V primeru napake nit ne nastane, funkcija `pthread_create` pa vrne številko napake.

Če hoče nit spoznati svojo identiteto, lahko to naredi s funkcijo `pthread_self`.

```
#include <pthread.h>
pthread_t pthread_self( );
```

Na primer, naslednji kos programa izpiše ID niti.

```
pthread_t tid, mtid;
int       err;

if( (err = pthread_create( &tid, ... )) > 0){
  printf("pthread err, %s\n", strerror( err )); exit(1);
}
mtid = pthread_self( );
printf("ThreadIDs = %ld, %ld\n", (long)tid, (long)mtid);
```

Čeprav takšen izpis povečini deluje, se ga z vidika prenosljivosti kode odsvetuje. Pravilen način za primerjanje ID niti daje funkcija `pthread_equal`, ki v primeru ujemanja identitet vrne vrednost različno od nič. Na primer,

```
if( pthread_equal( tid, mtid ) )
  printf("tid se ujema z mtid\n");
```

Nit, ki nastane s `pthread_create` se začne izvrševati s funkcijo `start_fn`. Nitna funkcija ima en sam argument, ki je kazalec 'brez tipa'. Kazalec seveda lahko kaže na strukturo z več komponentami. Zadnji argument funkcije `pthread_create` je argument nitne funkcije `start_fn`. Klic `pthread_create` je potemtakem ekvivalenten klicu

```
start_fn( arg );
```

s to bistveno razliko, da se nitna funkcija *izvršuje sočasno* s klicno funkcijo in seveda sočasno z vsemi drugimi nitnimi funkcijami. Argument `arg` je lahko tudi `NULL`. Običajno `arg` kaže na globalno ali statično spremenljivko, ki obstaja cel čas trajanja procesa. Ugotovimo še, da je funkcija `start_fn` tipa `void *`. Potem ko je nit ustvarjena, napreduje povsem asinhrono glede na nit, ki jo je ustvarila, torej pred ali za njo, seveda pa si z njo deli isti naslovni prostor in torej pomnilnik.

12.2 Funkciji `pthread_exit` in `pthread_cancel`

Če katera od niti izvrši funkcijo `exit`, se cel proces konča, kot bi tudi pričakovali. V večnitnem procesu pa povečini ne želimo, da bi se v trenutku zaključile vse niti, saj so lahko v različnih fazah napredovanja. Potreben je mehanizem, s katerim se konča samo posamezna nit, proces pa ostane.

Tako kot obstaja več možnosti za zaključek procesa, imamo tudi več možnosti za zaključek niti. Nit se lahko zaključi enostavno s klicem `return`, ki povzroči povratek iz nitne funkcije. Nitna funkcija se zaključi in niti ni več. Nit se lahko zaključi s klicem funkcije `pthread_exit`. Funkcija `pthread_exit` je analogna funkciji `exit`, le da prva zaključi nit in druga proces. Funkcija `pthread_exit` opravi direkten povratek iz vsake funkcije, ki jo pred tem po potrebi kliče nitna funkcija, ne glede na globino vgnezdenja. Tako kot je klic `exit` v funkciji `main` ekvivalenten klicu `return`, je tudi klic `pthread_exit` v nitni funkciji ekvivalenten klicu `return`. Prototip funkcije `pthread_exit` glasi:

```
#include <pthread.h>
int pthread_exit( void *rval_ptr );
```

Referenca `rval_ptr` naj kaže na vrednost, ki ostane definirana tudi potem, ko se funkcija `start_fn` konča. Sklad že ni primeren za to. Običajno se vrednost nahaja v podatkovnem segmentu globalnega značaja. Prek `rval_ptr` je namreč določena vrednost, ki jo vrne funkcija `start_fn`.

Nit lahko konča tako, da jo uniči druga nit s klicem funkcije `pthread_cancel`. S klicem te funkcije se nit lahko uniči tudi sama.

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

```
int pthread_cancel( pthread_t thread );
```

12.3 Funkcija pthread_join

Nit je moč *združiti* z drugo nitjo. To v bistvu pomeni, da ena od niti s `pthread_join` čaka, da se druga nit konča. Če bi čakana nit končala že prej, bi bil povratek funkcije takojšen. Na nek način je funkcija `pthread_join` sorodna funkciji `waitpid` za procese. Bistvena razlika je v tem, da se vsaka nit lahko združi z vsako nitjo, ne glede na to, katera nit je ustvarila katero nit, medtem ko `waitpid` deluje le za relacijo procesov roditelj-otrok.

Praviloma se mora združiti vsako nit, sicer postane 'opuščena', podobno kot velja za 'zombie' proces. To ni dobro, ker opuščena nit po nepotrebnem obremenjuje sistem. Le na nit, ki je bila pred tem 'odpeta' z `pthread_detach`, ali pa je bila takega tipa že ob nastanku, ni potrebno čakati.

```
#include <pthread.h>
int pthread_join( pthread_t thread, void **rval_ptr );
```

Funkcija `pthread_join` preko argumenta `**rval_ptr` dobi kazalec na vrednost, ki jo 'pusti' nit tedaj, ko konča, na primer z `return` iz nitne funkcije ali prek argumenta `*rval_ptr` funkcije `pthread_exit`. Argument je lahko tudi `NULL`.

Pa si poglejmo primer. V okviru naslednjega procesa iz programa `mtest.c` nastanejo tri niti. Začetna nit `main` ustvari dve novi niti. V okviru vsake od teh dveh novih niti se sicer izvrši ista nitna funkcija `nitFun`, a z drugačnimi vrednostmi argumentov. Argument nitne funkcije je referenca na strukturo z dvema komponentama. Po našem dogovoru naj pomeni prva komponenta številko niti in druga naj vsebuje 'sporočilni' znakovni niz. Nit ne naredi kaj dosti. Preprosto izpiše vrednosti komponent argumenta, spremeni dobljeni niz v velike črke in vrne kazalec na niz prek funkcije `pthread_exit`.

Glavna nit s `pthread_join` počaka na izvršitev in povratek nitnih funkcij. Nato zgolj za ilustracijo izpiše obe identiteti niti, čeprav se tak način izpisa ne priporoča. Izpiše tudi niza, ki ga vrneti nitni funkciji s `pthread_creat`.

Glavna nit nadaljuje in konča z `exit`. Program prevedemo z:

```
gcc -pthread -o mtest mtest.c
```

```
/* Testni program koncepta niti */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

#include <string.h>
#include <ctype.h>


void   *nitFun( void *nitniArg );  //nitna funkcija
struct narg_t{
  int  nid;      //moja oznaka niti
  char niz[128]; //info
};
struct  narg_t  arg1, arg2;

int main(void)
{
  int       err;
  void      *nout1, *nout2;
  pthread_t nid1, nid2;

  arg1.nid = 1; // moja oznaka za prvo nit
  strcpy(arg1.niz, "nitka 1");

  arg2.nid = 2; // moja oznaka za drugo nit
  strcpy(arg2.niz,"nitka 2");

  if( (err = pthread_create(&nid1, NULL, nitFun, &arg1)) > 0){
    printf("pthread_creat err, %s\n", strerror( err ));
    exit(1);
  }
  if( (err = pthread_create(&nid2, NULL, nitFun, &arg2)) > 0){
    printf("pthread_creat err, %s\n", strerror( err ));
```

```
    exit(1);
  }
  if( (err = pthread_join( nid1, &nout1)) > 0){
    printf("pthread_join err, %s\n", strerror( err ));
    exit(1);
  }
  if( (err = pthread_join( nid2, &nout2)) > 0){
    printf("pthread_join err, %s\n", strerror( err ));
    exit(1);
  }
  printf("NitId1: 0x%lx vrnila: %s\n",(ulong)nid1, (char *)nout1);
  printf("NitId2: 0x%lx vrnila: %s\n",(ulong)nid2, (char *)nout2);

  exit( 0 );
}

void *nitFun( void *arg )
{
  struct narg_t *narg = arg;
  char   *p;
  printf("nit %d sporoca: %s\n", narg -> nid, narg -> niz);
  for( p = narg -> niz; *p != '\0'; *p = toupper( *p ), p++);
  pthread_exit( narg -> niz );
}
```

Poglavje 13

Upravljanje časa

V tem poglavju se bomo posvetili času v povezavi z računalnikom. Obravnavali bomo realni čas, procesorski čas, ure in časovnike. Najprej bomo govorili o nadziranju realnega in procesorskega časa ter o funkcijah, ki to omogočajo. Potem si bomo na kratko ogledali še ure in časovnike, kot jih poznamo v sistemih UNIX/Linux in po specifikaciji POSIX.

13.1 Čas v računalniku

Ko govorimo o času v povezavi z računalniškim sistemom ali računalniškim procesom, sta v navadi dve pojmovanji časa:

- realni čas in
- procesorski ali s kratico CPE čas.

Realni čas je tisti čas, v katerem živimo in ga merimo od izbranega trenutka naprej. Za referenco lahko vzamemo koledarsko štetje in čas izražamo z letom, mesecem, dnem, uro, minuto in sekundo v formatu *datum* in *čas*. A za referenčni trenutek, to je trenutek ob času nič, lahko vzamemo zagon sistema ali, alternativno, nastanek procesa.

Za razliko od realnega časa, ki teče neprekinjeno in linearno, pa je procesorski čas vezan na napredovanje procesa. Procesorski čas je tisti čas, ko je procesor dodeljen procesu. Vsak proces ima torej vsaj dva načina za obravnavanje časa oziroma vsaj dva časovnika. Z enim časovnikom meri realni čas. Ta se stalno povečuje, denimo od trenutka nič ob nastanku procesa, do trenutka, ko je proces končan. Drugi časovnik pa se povečuje le tedaj, ko proces teče.

Procesorski čas delimo naprej na uporabniški in sistemski čas. K uporabniškemu času prispevajo intervali, ko procesor izvaja uporabniško kodo, med tem ko je sistemski čas tisti čas, ko proces napreduje v sistemskem jedru. To je na primer tedaj, ko proces izvede sistemski klic za branje, pisanje in podobno.

Vzdrževanje realnega časa je pomembno iz različnih razlogov. Denimo, pomembno je, da se zabeleži čas nastanka datoteke, čas spreminjanja datoteke, čas brisanja datoteke, čas začetka procesa in podobno. Posebej pomembno je upoštevanje realnega časa tedaj, kadar so procesi vezani na zunanje dogodke. Na primer, proces opravlja periodično odčitavanje ali postavljanje stanj stikal, vzorčenje ali generiranje signalov, predvajanje videoposnetkov in podobno.

Procesorski čas je koristen pri analizi ter optimizacijah procesov, pri proučevanju obremenjenosti ali izoriščenosti sistema, za delovanje v realnem času in posledično za razvrščanje procesov.

Vsak računalniški sistem ima vsaj en časovni števec (Angl. Timer) oziroma elektronsko vezje, ki služi za referenco pri merjenju časa, tako realnega časa kot procesorskega časa. Števec se enakomerno povečuje z urnim taktom, tako da je računalniški čas diskreten. Časovna ločljivost je eden od parametrov sistema oziroma sistemskega jedra in tipično znaša nekaj milisekund ali nekaj deset milisekund. Časovna ločljivost, ki je dana s strojno opremo, je višja in povečini tudi bistveno višja.

Linux deli čas na kratke časovne intervale (Angl. Jiffies). Čas merjen v tej enoti je tako imenovani 'softverski' oziroma 'programski čas' in načeloma je časovna ločljivost določena s trajanjem enega 'jiffy-ja'. Ta interval upora-

blja denimo razvrščevalnik pri krožnem (Angl. Round-Robin) razvrščanju. A povsem mogoče je, da sistem za potrebe realnega časa omogoča tudi boljšo časovno ločljivost.

Ako nas zanimajo časovne karakteristike procesa, jih lahko dobimo z ukazom školjke `time`. Na primer, izpis po izvršitvi programa `mproc` z ukazom

```
time mproc
```

bi lahko izgledal takole:

```
real    0m0.500s
user    0m0.105s
sys     0m0.025s
```

Iz izpisa bi razbrali, da je proces `mproc` trajal polovico sekunde in med tem porabil 0,130 sekunde časa procesorja, od tega v sistemskem jedru 25 milisekund.

Sedaj pa se posvetimo funkcijam za upravljanje časa.

13.2 Funkcije za upravljanje realnega časa

Sistemi UNIX/Linux imajo več funkcij za upravljanje koledarskega časa in za pretvarjanje med različnimi formati predstavitve časa:

- gettimeofday() in settimeofday(),
- time in stime,
- localtime, gmtime in mktime,
- ctime in asctime,
- strftime in strptime.

Nekatere si bomo ogledali v naslednjih razdelkih. Zavedati pa se moramo, da je nastavljanje ali spreminjanje sistemskega časa (`settimeofday, stime`) kritična operacija, zato jo sme opraviti samo uporabnik oziroma proces s primernimi pooblastili. Denimo, če bi čas pomotoma prestavili nazaj, bi dogodki, ki bi sledili, bili zabeleženi kot starejši dogodki od tistih v

preteklosti, kar bi lahko imelo škodljive posledice.

Sistemi UNIX/Linux obravnavajo interni realni čas kot predznačeno celoštevilsko spremenljivko, merjeno v sekundah, začenši s prelomnim 1. januarjem 1970, ki se pojmuje kot približni dan 'rojstva' sistema UNIX. Na 32-bitnih arhitekturah se bo 32-bitna spremenljivka časa iztekla leta 2038.

13.2.1 Funkciji `time` in `gettimeofday`

Funkcija `time` vrne število sekund od prelomnega datuma do klica.

```
#include <time.h>
time_t time(time_t *timep);
```

Funkcija vrne čas na dva načina, kot kazalec (`timep`) na spremenljivko s časom ali kot vrednost funkcije. V primeru napake vrne -1. Tipičen klic funkcije je zato naslednji:

```
time_t      realniCas;
realniCas = time( NULL );
```

Sistemski klic `gettimeofday()` vrne koledarski čas v strukturi na katero kaže `tv`,

```
#include <sys/time.h>
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

Klic vrne -1 v primeru napake, sicer pa 0. Struktura časa izgleda takole:

```
struct timeval{
  time_t tv_sec;  /* Sekunde po 00:00:00, 1 Jan 1970 UTC */
  suseconds_t tv_usec; /* dodatne mikrosekunde (long int)*/
};
```

Čeprav druga komponenta pomeni dodatno 'mikrosekundno' ločljivost, je dejanska ločljivost pogojena s strojno opremo. Drugi argument (`tz`) je bil namenjen poizvedbi po časovnem pasu, a je dandanes opuščen, zato naj bo ob klicu kar `NULL`,

```
struct timeval koledarskiCas;
gettimeofday( &koledarskiCas, NULL );
```

13.2.2 Funkcija `ctime`

Funkcija `ctime` pretvori interni celoštevilski zapis časa v človeku prijaznejšo obliko.

```
#include <time.h>
char *ctime(const time_t *timep);
```

Klic vrne kazalec na z ničlo zaključen znakovni niz. Niz vsebuje znakovno kodiran datum v standardnem formatu, upoštevajoč časovni pas ter zimsko/letni čas, ki je v veljavi. V primeru napake funkcija vrne `NULL`. Torej, naslednje vrstice programa izpišejo tekoči datum.

```
time_t  cas;
char   *cDatum;

cas    = time( NULL );
cDatum = ctime( &cas );
printf("Danasnji datum, %s", cDatum);
```

Naslednji program preizkusi klice obravnavanih funkcij.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sys/time.h>

int main( int argc, char **argv )
{
  time_t t;
  struct timeval tv;
  char   *datum;

  t = time( NULL );
  printf("%s, cas v [s] po 0:0:0, 1.1.1970 (time): %ld [s]\n",
                                                  argv[0], t);
  if( gettimeofday( &tv, NULL) == -1){
    perror("gettimeofday err");
    exit(EXIT_FAILURE);
  }
```

```
  printf("%s, cas v [s],[us] po 0:0:0, 1.1.1970 (gettimeofday):
          %ld [s], %ld [us]\n", argv[0], tv.tv_sec, tv.tv_usec);

  datum = ctime( &t );
  if( datum == NULL ){
    perror("ctime err");
    exit(EXIT_FAILURE);
  }
  printf("%s, razclenjen cas v znakovni niz (ctime): %s\n",
                                                  argv[0], datum);
  exit( EXIT_SUCCESS );
}
```

13.2.3 Funkciji `gmtime` in `localtime`

Funkcija `ctime` vrne datum kot znakovni niz, ki ga ponavadi rabimo zato, da ga izpišemo. A včasih nas zanima 'razčlenjeni čas' na ure, minute, sekunde in tako naprej. Pri tem nam pomagata funkciji `gmtime()` ter `localtime()`. Prva nam vrne razčlenjeni čas UTC (Universal Time Coordinated), druga lokalni čas, upoštevaje časovni pas in zimsko letni čas.

```
#include <time.h>
struct tm *gmtime(const time_t *timep);
struct tm *localtime(const time_t *timep);
```

Obe funkciji vrneta kazalec na strukturo `tm`, ali `NULL` v primeru napake.

```
struct tm {
  int tm_sec;   /* Sekunde     (0-60) */
  int tm_min;   /* Minute      (0-59) */
  int tm_hour;  /* Ure         (0-23) */
  int tm_mday;  /* Dan v mesecu(1-31) */
  int tm_mon;   /* Mesec       (0-11) */
  int tm_year;  /* Leto po      1900  */
  int tm_wday;  /* Dan v tednu (Nedelja = 0)       */
  int tm_yday;  /* Dan v letu (0-365; 1.Januar = 0)*/
  int tm_isdst; /* DST 'Daylight saving time'      */
};
```

13.2.4 Funkciji `asctime` in `mktime`

Če imamo čas v razčlenjeni obliki in shranjen v strukturi `tm`, lahko dobimo znakovni niz datuma s funkcijo `asctime`, medtem ko nam funkcija `mktime` pretvori razčlenjeni čas nazaj v celoštevilsko vrednost.

```
#include <time.h>
char   *asctime(const struct tm *timeptr);
time_t mktime(struct tm *timeptr);
```

Prva funnkcija nam vrne kazalec na datum kot znakovni niz in `NULL` v primeru napake. Druga funkcija vrne število sekund po polnoči 1. 1. 1970 ali -1 v primeru napake.

Na koncu sledi še preizkusni program obravnavanih funkcij.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

int main( void )
{
  time_t t;
  struct tm *tmg, *tml;

  t   = time( NULL );
  tml = localtime( &t );
  printf("(localtime) dd/mm/yyyy, hh:mm:ss, %d/%d/%d, %d:%d:%d\n",
              tml->tm_mday, tml->tm_mon + 1, tml->tm_year + 1900,
              tml->tm_hour, tml->tm_min, tml->tm_sec);
  printf("(asctime) od (localtime), LC cas: %s", asctime( tml ));
  tmg = gmtime( &t );
  printf("(asctime) od (gmtime), GM cas: %s", asctime( tmg ));
  exit( 0 );
}
```

13.3 Funkcije za upravljanje procesorskega časa

Kot rečeno je procesorski čas tisti čas, ki ga procesor nameni procesu, bodisi v uporabniškem načinu bodisi v sistemskem načinu. Procesorski čas je torej merjen glede na dotični proces, na posamezno nit, ali pač na skupino procesov ali na skupino niti. Pa si oglejmo nekatere funkcije za dostop do procesorskega časa.

13.3.1 Funkciji `times` in `clock`

Do procesorskega časa dostopamo s funkcijo `times()`.

```
#include <sys/times.h>
clock_t times(struct tms *buf);
```

Struktura, na katero kaže argument `buf`, ima naslednje komponente:

```
struct tms {
  clock_t tms_utime;  /* CPE cas, uporabniski nacin     */
  clock_t tms_stime;  /* CPE cas, sistemski nacin       */
  clock_t tms_cutime; /* CPE uporabniski cas vseh otrok */
  clock_t tms_cstime; /* CPE sistemski cas vseh otrok   */
};
```

Prvi dve komponenti torej vsebujeta procesorski čas porabljen v uporabniškem ter sistemskem načinu, zadnji dve komponenti pa procesorski čas v uporabniškem ter sistemskem načinu za vse potomce - otroke skupaj, ki so se zaključili in jih je dotični proces dočakal s klicem `wait`. Tip `clock_t` je dejansko celoštevilski. En tiktak ure traja delček sekunde. Število tiktakov na sekundo je parameter sistema, ki ga pridobimo s klicem `sysconf(_SC_CLK_TCK)`,

```
clock_t tikNaSek;
tikNaSek = sysconf( _SC_CLK_TCK )
```

in je za večino sistemov enak 100. Čas v sekundah torej dobimo tako, da število udarcev ure, ki jih vrne klic funkcije, delimo z vrednostjo, ki jo vrne `sysconf(_SC_CLK_TCK)`.

Funkcija `times()` vrne število udarcev ure, ki je preteklo od referenčnega trenutka enkrat v preteklosti, na primer od zagona sistema, ali -1 v primeru

napake. Funkcija torej vrne realni čas. Če bi želeli dobiti realni čas procesa od njegovega nastanka v trenutku nič, bi funkcijo `times()` klicali dvakrat, enkrat na začetku in enkrat na koncu procesa. Vendarle ne smemo pozabiti, da lahko števec v času trajanja procesa tudi preplavi. Pri 100 udarcih na sekundo se bo to zgodilo enkrat do dvakrat letno.

Ko nas zanima zgolj število udarcev ure za realni čas, Linux dovoljuje za vrednost argumenta funkcije `times()` kazalec `NULL`.

Naslednji program `mcpe.c` demonstrira uporabo navedenih funkcij. V njem je zanka `for`, ki zgolj troši procesorski čas v uporabniškem načinu in periodično vsakič ustavi proces za 2 milisekundi.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include <sys/times.h>
#include <sys/resource.h>

int main( void )
{
  clock_t rt_proc, rt_start, rt_end;
  int     ticks;

  struct  tms cpu;
  float   cpu_usr, cpu_sys, proc_elapsed;
  int     i;
  double  x=1.5;

  printf("Tiktakov na sekundo: %d\n", ticks = sysconf(_SC_CLK_TCK));
  rt_start  = times( &cpu );
  for(i=0; i < 5000; i++){
    x = sin( x ); x = log( x ); x = exp( x );
    usleep(2000);
  }
  rt_end  = times( &cpu );
  rt_proc = rt_end - rt_start;
  printf("Cas v tiktakih:\n");
```

```
  printf("Realni: %ld \nCPE_U : %ld \nCPE_S : %ld\n",
                    rt_proc, cpu.tms_utime, cpu.tms_stime);

  cpu_usr      = (float)cpu.tms_utime /(float)ticks;
  cpu_sys      = (float)cpu.tms_stime/(float)ticks;
  proc_elapsed = (float)(rt_proc)/(float)ticks;
  printf("Casi v sekundah\n");
  printf("Realni: %6.3f \nCPE_U : %6.3f \nCPE_S : %6.3f\n",
                            proc_elapsed, cpu_usr, cpu_sys);

  return 0;
}
```

Testna izvršitev programa `mcpe` z ukazom `time` je povzročila naslednji izpis,

```
time mcpe
Tiktakov na sekundo: 100
Cas v tiktakih:
Realni: 1075
CPE_U : 1
CPE_S : 2
Casi v sekundah
Realni: 10.750
CPE_U :  0.010
CPE_S :  0.020

real    0m10.750s
user    0m0.010s
sys     0m0.024s
```

Opazimo kar dobro ujemanje med tem kar izpiše naš proces in tistim, kar izpiše ukaz `time`.

Kadar nas zanima zgolj procesorski čas, lahko uporabimo funkcijo `clock()`. Funkcija sicer vrne vrednost tipa `clock_t` tako kot funkcija `times()`. V obeh primerih gre za celo število, enote pa so drugačne.

```
#include <time.h>
clock_t clock( void );
```

Funkcija torej vrne celoten procesorski čas procesa v enotah `CLOCKS_PER_SEC`,

ki je po standardu POSIX vedno 1,000,000. Da dobimo CPE čas v sekundah, moramo vrednost, ki jo vrne funkcija, deliti s to konstanto. Standard C ne zahteva, da bi bila vrednost spremenljivke, katere vrednost funkcija vrne, ob začetku programa enaka nič. Zavedati se moramo tudi, da vrednost, ki jo funkcija vrne, dokaj pogosto 'preplavi'. V primeru napake funckija vrne -1.

13.4 Klasični UNIX časovniki

V sistemih realnega časa je časovno soodvisno obravnavanje dogodkov in s tem v zvezi časovno pogojeno napredovanje procesov ključnega pomena.

Na primer, odčitavanje analognega signala (vzorčenje) z analogno digitalnim pretvornikom bi načeloma lahko potekalo takole:

- Izbira prvega ali edinega trenutka vzorčenja, za kar potrebujemo programljiv časovnik s to funkcionalnostjo.
- Če gre za večkratno periodično vzorčenje, nastavitev periode vzorčenja, na primer na 10 milisekund, za kar potrebujemo programljiv časovnik s to fukcionalnostjo. Časovnik oziroma časovni števec (Angl. Timer counter ali Interval timer) nastavimo na začetno vrednost, ki ustreza intervalu 10 milisekund, nakar se časovnik zmanjšuje z osnovnim taktom, ki je odvisen od strojne (in programske) opreme.
- Vedno, kadar se časovnik izteče, dobi proces (ali nit) za vzorčenje signal, na katerega se odzove na primer tako, da proži začetek analogno digitalne pretvorbe in odčita signal. Časovnik se v primeru periodičnega vzorčenja ponovno nastavi na začetno vrednost, dokler ga ne 'ugasnemo' in posledično zaključimo proces vzorčenja.

Drugi primer vzemimo s področja komunikacij. Denimo, v paketnem podatkovnem omrežju izvorno vozlišče (oddajnik) pošilja paket za paketom, medtem ko jih ponorno vozlišče (sprejemnik), potrjuje s povratnimi paketi. Vedno, kadar izvorno vozlišče pošlje paket, nastavi časovnik na primerno vrednost (skrajni rok, Angl. Deadline), nakar čaka na potrdilo sprejemnika.

V normalnih pogojih bo potrdilo sprejeto predno se časovnik izteče in sledilo bo pošiljanje naslednjega paketa. V primeru, da potrdilo ne pride, se časovnik izteče (Angl. Time-out), kar oddajnik obravnava kot napako in pošlje isti paket še enkrat.

V obeh primer potrebujemo enega ali več časovnikov, katerih delovanje je moč programsko nastaviti. K osnovni funkcionalnosti, ki naj jo zagotavlja sistem časovnikov, spadajo:

- več časovnikov v okviru istega procesa,
- primerna časovna ločljivost,
- možnost nastavitve prvega trenutka in periode,
- predvidljivost oziroma stabilnost, čim manjša variabilnost glede na okoliščine.

S časovniki so tesno povezani 'spalniki' (Angl. Sleepers). To so funkcije, s katerimi se da proces postaviti v stanje ustavljen ali točneje 'odstavljen' (Angl. Suspended). To je stanje, v katerem se proces ne poteguje za procesno enoto. Tudi pri spalnikih sta potrebni primerno fina časovna ločljivost in stabilnost. No, tudi to funkcionalnost se da v bistvu realizirati s časovniki.

V nadaljevanju si bomo najprej ogledali klasične UNIX intervalne časovnike in funkcije za upravljanje z njimi.

13.4.1 Funkciji `getitimer()` in `setitimer()`

S funkcijo `setitimer()` ustvarimo in nastavimo časovnik na začetno vrednost. Časovnik nato teče in ko se izteče proces prejme signal. Časovnik lahko deluje samo enkrat ali periodično.

```
#include <sys/time.h>

int setitimer(int which, const struct itimerval *new_value,
                              struct itimerval *old_value);
```

Vrne -1 v primeru napake, sicer nič.

Prvi argument `which` določi tip časovnika kot sledi:

- `ITIMER_REAL`: časovnik teče v realnem času. Ko se časovnik izteče, proces prejme signal `SIGALARM`.
- `ITIMER_VIRTUAL`: časovnik teče tedaj, ko se proces izvaja v uporabniškem načinu. Ko se časovnik izteče, dobi proces signal `SIGVTALRM`.
- `ITIMER_PROF`: časovnik teče tedaj, ko se proces izvaja ali v uporaniškem ali v sistemskem načinu (Angl. Profiling timer). Časovnik torej upošteva skupen procesorski čas. Ko se časovnik izteče, dobi proces signal `SIGPROF`.

Privzet način obravnavanja signalov je konec procesa, a se signal lahko prestreže.

Argumenta `new_value` in `old_value` sta kazalca na strukturo `itimerval`,

```
struct itimerval {
  struct timeval it_interval; /* Interval za periodicen casovnik*/
  struct timeval it_value;    /* Dejanska vrednost casovnika    */
};
```

Komponenti te strukture sta tudi strukturi kot sledi,

```
struct timeval {
  time_t      tv_sec;  /* Sekunde                 */
  suseconds_t tv_usec; /* Mikrosekunde (long int) */
};
```

Komponenta `it_interval` argumenta `new_value` določa periodo (v sekundah in mikrosekundah), a če je vrednost nič, to pomeni enkratno proženje. Komponenta `it_value` vsebuje preostanek časa, ko se časovnik izteče oziroma do ponastavitve. Zadnji argument, kazalec `old_value`, je lahko `NULL`, ali pa kaže na strukturo, ki vsebuje prejšni vrednosti za interval in vrednost časovnika. Vsak proces ima lahko le po en časovnik vsakega tipa. Čeprav nastavljamo časovnik v mikrosekundah, je dejanska ločljivost odvisna od dotične strojne in programske opreme ter je tipično vezana na interval *jiffie*.

S funkcijo `getitimer()` *pridemo do stanja časovnika. Vrednost je enaka kot jo vrne funkcija* `setitimer()` *z argumentom* `old_value`.

```
#include <sys/time.h>
int getitimer(int which, struct itimerval *curr_value);
```

Funkcija vrne -1 v primeru napake, sicer pa nič.

13.4.2 Funkcija `alarm()`

Funkcijo `alarm` smo žspoznali v poglavju o signalih. Funkcija omogoča isto kot funkciji `setitimer()` ter `getitimer()`, le da zgolj s sekundno ločljivostjo.

Klic `old_timer = alarm( new_timer )` nastavi časovnik na novo vrednost, funckija pa vrne staro vrednost časovnika. Klic `cur_time = alarm( )` vrne vrednost časovnika in klic `alarm( 0 )` 'ugasne ' časovnik. Če se časovnik izteče, sledi signal `SIGALRM`.

13.5 Spalniki

Skoraj enako pomemno kot da proces napreduje, je tudi to, da proces miruje oziroma *spi*, saj tedaj ne bremeni procesne enote. Poleg tega, da proces spi, pa je dostikrat važno, da spi natanko predpisan časovni interval. Časovne zakasnitve so potrebne aplikacijam. Niso redki primeri, ko se mora določeno dejanje zgoditi po natančno določenem časovnem intervalu. Funkciji `sleep` ter `usleep` smo že spoznali.

13.5.1 Funkcije `sleep`, `usleep` in `nanosleep`

Funkcija `sleep` postavi proces v stanje odstavljen (Angl. Sespended) za število sekund, kot določa vrednost argumenta. Funkcija `usleep` odspi najmanj predpisano število mikrosekund, ki je lahko malenkostno daljše spričo omejene časovne ločljivosti sistema.

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
int usleep(useconds_t usec);
```

V primeru, da proces medtem dobi signal, se spanje s `sleep` zaključi. Funkcija vrne 0 v primeru, ko se spanje izteče ali preostalo število sekund, če se je spanje zaključilo prej zaradi signala.

Funkcija `usleep` vrne 0 v primeru uspeha in -1 v primeru napake.

Funkcija `nanosleep`, kot bi glede na ime tudi pričakovali, daje boljšo ločljivost pri nastavitvi intervala,

```
#define _POSIX_C_SOURCE 199309
#include <time.h>
int nanosleep(const struct timespec *request,
                    struct timespec *remain);
```

Klic vrne 0 v primeru regularnega povratka ali -1 v primeru napake ali prekinitve.

Vrednost, na katero kaže prvi argument, določa trajanje spanja, definicija strukture pa je taka:

```
struct timespec {
       time_t tv_sec;   /* Sekunde     */
       long   tv_nsec;  /* Nanosekunde */
};
```

Drugi argument je lahko `NULL`, tedaj nas preostanek časa za spanje ne zanima. V nasprotnem primeru drugi argument kaže na preostanek spanja ob prekinitvi spanja spričo signala. Možen primer uporabe bi bil ponoven klic funkcije s preostankom časa za spanje, da se proces 'naspi do kraja'.

13.6 POSIX ure

POSIX ure (Angl. Clocks) so pripomočki za upravljanja časa s fino časovno ločljivostjo in na splošno z boljšo funkcionalnostjo kot klaisične UNIX/Linux funkcije. Glavne funkcije za upravljanje časa so:

- `clock_gettime()`: za poizvedbo po času,
- `clock_settime()`: za nastavitev časa,

- `clock_getres()`: za poizvedbo po časovni ločljivosti ure.

S temi funkcijami je moč upravljati z različnimi časovnimi viri: realnim časom, procesorskim časom procesa, procesorskim časom niti in drugimi. Prototipi funkcij so:

```
#define _POSIX_C_SOURCE 199309L
#include <time.h>

int clock_getres(clockid_t clk_id, struct timespec *res);
int clock_gettime(clockid_t clk_id, struct timespec *tp);
int clock_settime(clockid_t clk_id, const struct timespec *tp);
```

Definicija strukture za čas je enaka kot pri funkiji `nanosleep`,

```
struct timespec {
       time_t tv_sec; /* Sekunde     */
       long tv_nsec;  /* Nanosekunde */
};
```

Pri povezovanju/prevajanju je potreben dodatek `-lrt`, ki doda knjižnico za realni čas (librt.so). Funkcije vračajo 0 v primeru uspeha in -1 v primeru napake.

Prvi argument poda identiteto ure in je lahko

- `CLOCK_REALTIME` za uro realnega časa sistema,
- `CLOCK_MONOTONIC` za nenastavljivo monotono uro realnega časa,
- `CLOCK_PROCESS_CPUTIME_ID` za procesorski čas procesa,
- `CLOCK_THREAD_CPUTIME_ID` za procesorski čas niti.

Ura `CLOCK_REALTIME` je ura realnega časa, ki jo je moč ponastaviti. Za razliko od te ure se ure `CLOCK_MONOTONIC` ne da ponovno nastavljati. v sistemu Linux se ta ura nastavi na nič ob zagonu sistema, nakar monotono teče v realnem času. Ura `CLOCK_PROCESS_CPUTIME_ID` meri procesorski čas prosesa v uporabniškem in sistemskem načinu skupaj. Enako velja za niti.

Naslednji program preizkusi klic funkcij in izpiše uro realnega časa ter njeno ločljivost.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

#define errExit(msg)    {perror(msg); exit(EXIT_FAILURE);}

int main(int argc, char *argv[])
{
  struct timespec t;

  if( clock_gettime( CLOCK_REALTIME, &t ) == -1)
    errExit("clock_gettime err");
  printf("%s: REALTIME clock: %ld[s], %ld[ns]\n",
    argv[0], t.tv_sec, t.tv_nsec);
  printf("%s: calendar time : %s\n", argv[0], ctime( &t.tv_sec));
  if( clock_getres( CLOCK_REALTIME, &t ) == -1)
    errExit("clock_getres err");
  printf("%s: REALTIME resolution: %ld[s], %ld[ns]\n",
    argv[0], t.tv_sec, t.tv_nsec);
  exit(EXIT_SUCCESS);
}
```

Proces lahko pride do procesorskega časa drugega procesa tako, da pridobi identiteto njegove ure `clock_getcpuclockid` in nato s to identiteto kliče funkcijo `clock_gettime`.

```
#include <time.h>
int clock_getcpuclockid(pid_t pid, clockid_t *clock_id);
```

Funkcija vrne nič v primeru uspeha ali pozitivno vrednost, ki pomeni kodo napake. Tej sorodna funkcija za niti je funkcija `pthread_getcpuclockid`, ki ji namesto številke procesa kot prvi argument podamo identiteto niti.

Naslednji program sprejme PID procesa, ki ga vnesemo prek ukazne vrstive ter izpiše njegov procesorski čas. Če PID ne podamo, izpiše svoj procesorski čas.

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

```
#include <unistd.h>
#include <time.h>

#define  errExit( m )   {perror( m ); exit( EXIT_FAILURE );}

int main( int argc, char **argv )
{
  struct    timespec cpu;
  clockid_t clockid;
  pid_t     pid;

  pid = getpid( );
  if( argc == 2){
    pid = atoi( argv[1] );
  }
  if( clock_getcpuclockid( pid, &clockid ) == -1)
    errExit("clock_getcpuclockid err");
  if( clock_gettime( clockid, &cpu ) == -1)
    errExit("clock_gettime err");
  printf("%s: PID=%d, CPU time: %ld[s], %ld[ns]\n",
    argv[0], pid, cpu.tv_sec, cpu.tv_nsec);
  exit( EXIT_SUCCESS );
}
```

13.7 POSIX časovniki

Medtem ko so klasični intervalni časovniki (Angl. Timers, Interval timers) omejeni na po enega na proces, je POSIX časovnikov v okviru enega procesa lahko tudi več. Princip dela s časovnikom je naslednji. Časovnik najprej ustvarimo, nato nastavimo in nazadnje odstranimo. Vmes lahko časovnik ponovno nastavimo, ugasnemo ali, če nas zanima njegova trenutna vrednost, odčitamo. Časovnik lahko deluje v enkratnem ali večkratnem (periodičnem) načinu. Časovnik nastavimo na začetno vrednost, za periodično delovanje predpišemo še periodo proženja, nakar časovnik teče. Ko se časovnik izteče, smo o tem obveščeni. Glavne funkcije za delo s časovniki

so:

- `timer_create()`, ki ustvari nov časovnik in predpiše način obravnavanja časovnika, ko se le-ta izteče,
- `timer_settime()`, ki nastavi (Angl. arms) časovnik ali ustavi (Angl. disarms) časovnik,
- `timer_delete()`, ki odstrani časovnik, ko ni več potreben.

Pri prevajanju oziroma povezovanju je potrebno dodati knjižnico tt librt. V nadaljevanju si bomo na kratko ogledali navedene funkcije.

13.7.1 Funckija `timer_create`

S funkcijo `timer_create` ustvarimo nov časovnik. Prototip funkcije je:

```
#include <signal.h>
#include <time.h>
int timer_create( clockid_t clockid, struct sigevent *sevp,
                                              timer_t *timerid);
```

Če klic uspe, funkcija vrne 0, sicer -1. Prvi argument `clockid` je identiteta ure, ki jo izberemo enako kot pri funckijah `clock_`. Zadnji argument vrne identiteto časovnika, s pomočjo katere nato upravljamo časovnik. Z argumentom `sevp` določimo način obravnavanja časovnika, ko se ta izteče. Struktura `struct sigevent` je dokaj kompleksna. Za nas je zanimiva komponenta `sigev_notify`, ki določa, na kakšen način naj bo proces obveščen in kako naj se odzove, ko se časovnik izteče. Možnosti so:

- `SIGEV_NONE`: brez obvestila,
- `SIGEV_SIGNAL`: z izbranim signalom,
- `SIGEV_THREAD`: z novo nitjo z izbrano nitno funkcijo,
- `SIGEV_THREAD_ID`: s signalom, ki je poslan izbrani niti.

13.7.2 Funckiji `timer_settime` in `timer_gettime`

S funkcijo `timer_settime` nastavimo, ponastavimo ali ustavimo (ugasnemo) časovnik. S funkcijo `timer_gettime` odčitamo tekočo vrednost časovnika ter njegov ponovitveni interval. Prototipa funkcij sta:

```
#include <time.h>
int timer_settime( timer_t timerid, int flags,
                   const struct itimerspec *new_value,
                   struct itimerspec * old_value);

int timer_gettime( timer_t timerid, struct itimerspec *curr_value );
```

Če klic uspe, funkciji vrneta 0, sicer -1.

Funkcija `timer_settime()` upravlja časovnik z identiteto `timerid`. Z argumentom `new_value` določimo interval do prvega 'proženja' in v periodičnem delovanju tudi periodo. Če argument `old_value` ni enak `NULL`, je z njim moč dobiti staro nastavitev intervala proženja ter čas do prvega naslednjega proženja. Zgradba strukture `itimerspec` je naslednja:

```
struct itimerspec {
       struct timespec it_interval;  /* interval   */
       struct timespec it_value;     /* prvi potek */
};
struct timespec {
       time_t tv_sec;  /* Sekunde     */
       long   tv_nsec; /* Nanosekunde */
};
```

Argument `flags` dodatno določi pomen komponente `it_value`. Če je `flags = 0`, se vrednost obravnava `it_value` obravnava *relativno* glede na trenutek ob klicu funkcije `timer_settime`. Če je `flags = TIMER_ABSTIME`, se vrednost obravnava `it_value` obravnava *absolutno* glede na stanje ure, ki poganja časovnik. Na primer, v primeru `clockid = CLOCK_MONOTONIC`, bo `it_value` obravnavana glede na stanje te ure.

Časovnik ugasnemo preprosto tako, da funkcijo `timer_settime` kličemo vrednostjo 0 za interval in periodo proženja.

Funkcija `timer_gettime` ne potrebuje obširne razlage. Z njo enostavno dobimo trenutno stanje časovnika.

13.7.3 Funckija `timer_delete`

Časovnik odstranimo s funkcijo `timer_delete`. Njen prototip je:

```
#include <time.h>
int timer_delete( timer_t timerid);
```

Klic vrne 0 in časovnik je odstanjen, ali -1 v primeru napake.

Naslednji program demonstrira osnovno funkcionalnost časovnika. Program ustvari časovnik, ki je vezan na monotono uro realnega časa brez obveščanja, ko se izteče. Nato nastavi časovnik na prvo proženje po 4 sekundah in periodično proženje vsaki dve sekundi. Medtem ko časovnik teče vsake pol sekunde izpiše stanje časovnika. Nazadnje časovnik odstrani in konča.

```
/* mtimer.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <time.h>

#define errExit( m, e )    {perror( m ); exit(e);}

int main(int argc, char *argv[])
{
  timer_t  timerid;
  struct itimerspec it;
  struct sigevent   sev;
  int i;

  sev.sigev_notify = SIGEV_NONE;

  if( timer_create( CLOCK_MONOTONIC, &sev, &timerid ) == -1)
    errExit("timer_create err", EXIT_FAILURE);
```

```
  it.it_interval.tv_sec  = 2;
  it.it_interval.tv_nsec = 0;
  it.it_value.tv_sec     = 4;
  it.it_value.tv_nsec    = 0;

  if( timer_settime( timerid, 0, &it, NULL ) == -1)
    errExit("timer_settime err", EXIT_FAILURE);

  for( i = 0; i < 10; i++){
    if( timer_gettime( timerid, &it ) == -1)
      errExit("timer_gettime err", EXIT_FAILURE);
    printf("time: %ld[s], %ld[ns]\n", it.it_value.tv_sec, it.it_value.tv_nsec);
    usleep( 500000 );
  }
  if( timer_delete( timerid ) == -1)
    errExit("timer_delete err", EXIT_FAILURE);

  exit(EXIT_SUCCESS);
}
```

Naslednji program ilustrira koncept obveščanja niti, ko se časovnik izteče. Tokrat uporabimo uro za procesorski čas. Funkciji `timer_create` podamo naslov nitne funkcije `thredfun`. Argument tej nitni funkciji je kazalec na spremenljivko, ki vsebuje identiteto časovnika. Določil nitne funkcije ne spreminjamo. Časovnik nastavimo na enkratno proženje čez 5 sekund, nakar gre proces v zanko `while`, ki zgolj troši procesorski čas. Ko potroši pet sekund procesorja, se izvede nitna funkcija, ki izpiše identiteto časovnika in omogoči procesu izstop iz zanke `while`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <time.h>

#define errExit( m, e )    {perror( m ); exit(e);}

int i = 1;
```

```
static void threadfun(union sigval sv)
{
  timer_t *tp = sv.sival_ptr;
  printf("Casovnik %ld se je iztekel\n",(long)*tp );
  i = 0;
}

int main(int argc, char *argv[])
{
  timer_t  timerid;
  struct itimerspec it;
  struct sigevent   sev;

  sev.sigev_notify = SIGEV_THREAD;
  sev.sigev_notify_function  = threadfun;
  sev.sigev_value.sival_ptr  = &timerid;
  sev.sigev_notify_attributes= NULL;

  if( timer_create( CLOCK_PROCESS_CPUTIME_ID, &sev, &timerid ) == -1)
    errExit("timer_create err", EXIT_FAILURE);

  it.it_interval.tv_sec  = 0;
  it.it_interval.tv_nsec = 0;
  it.it_value.tv_sec     = 5;
  it.it_value.tv_nsec    = 0;

  if( timer_settime( timerid, 0, &it, NULL ) == -1)
    errExit("timer_settime err", EXIT_FAILURE);
  while( i );
  if( timer_delete( timerid ))
    errExit("timer_delete err", EXIT_FAILURE);
  exit(EXIT_SUCCESS);
}
```

Poglavje 14

Funkcije za komunikacije med procesi

V predhodnih pogavjih smo spoznali funkcije za upravljanje procesov in niti. Sedaj, ko imamo procese in niti, pa smo pripravljeni, da se posvetimo komunikacijam med njimi.

V nadaljevanju poglavja si bomo ogledali klasične oblike medprocesnih komunikacij v sistemih UNIX. To so cevi, cevi z imeni in signali.

V naslednjem poglavju 15 bomo obravnavali API vmesnik za komunikacije med procesi po načelu deljenega pomnilnika, sporočilnih vrst ter sinhronizacijo s semaforji.

Ker si niti delijo skupen naslovni prostor, je podlaga za komunikacijo po načelu deljenega pomnilnika med nitmi dana sama po sebi. Potrebna po sinhronizaciji med nitmi pa je še bolj izražena. Funkcije za sinhronizacijo niti si bomo približali v poglavju 16.

Komunikacije med krajevno porazdeljenimi procesi bomo pustili za zadnje poglavje 17 o medprocesnih komunikacijah.

Za uvod pa naštejmo glavne funkcije in na kratko povejmo čemo služijo.

14.1 Pregled funkcij

Za komunikacijo med procesi v sistemih UNIX obstaja precej možnosti. Osnovne oblike medprocesnih komunikacij in njim ustrezajoče funkcije so:

- cev (Angl. Pipes): funkcija `pipe`, ki napelje enosmerni komunikacijski kanal med procesoma v sorodstvu,
- poimenovana cev (Angl. Named pipes): funkcija `mkfifo`, ki napelje enosmerni komunikacijski kanal med poljubnima procesoma in
- signali (Angl. Signals): funkcije `signal, sigaction, kill, raise, alarm, pause`, ki javljajo ali prestrezajo redke nenavadne dogodke.

Cevi in poimenovane cevi ter signali so v sistemih UNIX/Linux klasična in najstarejša oblika komuniciranja. Cevi, kot nakazuje ime, omogočajo enosmerni pretok podatkov med dvema ali več v komunikacijo udeleženimi procesi. En proces pošlje oziroma zapiše podatke v cev in drugi proces te podatke na drugi strani cevi v enakem zaporedju sprejme oziroma prebere.

Signali so neka vrsta prekinitev. Praviloma se uporabljajo za javljanje redkih, tipično kritičnih dogodkov. Signali so asinhrone narave, kar pomeni, da se pojavljajo načeloma kadarkoli, *asihrono* glede na napredovanje procesov. Lahko se javljajo tudi kot stranski učinek napredovanja procesa.

Naslednja skupina funkcij daje podporo sporočilom, deljenemu pomnilniku ter sinhronizaciji:

- sporočila (Angl. Messages): funkcije `msgget, msgctl, msgsnd, msgrcv`,
- semaforji (Angl. Semaphores): funkcije `semget, semctl, semop`,
- skupen (deljen) pomnilnik (Angl. Shared Memory): funkcije `shmget, shmctl, shmat, msgdt`,

Sistemi semaforjev, sporočil ter skupen pomnilnik predstavljajo razširitev komunikacijskih možnosti. Ko govorimo o sistemu medprocesnega komuniciranja včasih mislimo prav na eno izmed teh treh oblik, pri čemer obravnavamo cevi kot poseben primer sporočil. Za razliko od cevi, ki obravna-

vajo podatke kot neprekinjeno zaporedje podatkov, pa sporočila ohranjajo strukturo sporočil in razmejitve med sporočili v zaporedju sporočil.

Semaforji so pripomoček za sinhronizacijo procesov. Denimo, na semaforju lahko en proces čaka, dokler mu mogoče nek drug proces ne omogoči napredovanje. Sinhronizacija procesov je pač ena od oblik medprocesnih komunikacij.

Deljen ali skupen pomnilnik je najhitrejša oblika komunikacije med procesi. Dvema ali več procesom je dodeljen isti del fizičnega pomnilnika, v katerega nek proces lahko piše in iz katerega lahko kakšen drug proces to vsebino preprosto bere.

V večnitnem procesu se morajo niti občasno časovno usklajevati. Funkcije, ki dajejo podporo za sinhornizacijo niti so:

- družina funkcij `pthread_mutex` v podporo mesebojnemu izkjučevanju,
- družina funkcij `pthread_rwlock` v podporo sočasnemu dostopu operacijam branja ob upoštevanju pisanja,
- družina funkcij `pthread_cond` za javljanje, da se je vrednost spremenljivke spremenila.

Komunikacijske vtičnice niso omejene na medprocesno komunikacijo na eni napravi. Nastale so iz potrebe po komunikaciji med krajevno porazdeljenimi procesi v omrežju – procesi na različnih napravah, neodvisno od krajevne namestitve, strojne opreme in operacijskega sistema. Razumljivo, da dopuščajo komuniciranjo med procesi na isti napravi.Osnovne funckije sistema komunikacijskih vtičnic so:

- `socket, bind, listen, accept, connect`.
- `receive_from, send_to`.

Te bomo pustili za nazadnje. Kot rečeno bomo najprej govorili o ceveh.

14.2 Cevi in sistemski klic `pipe`

Cev je *enosmerni* komunikacijski kanal, ki lahko obstaja med dvema ali več procesi v sorodstvu, denimo med roditeljem in otrokom ali med dvema otrokoma. Za obojesmerno komunikacijo med procesoma sta načeloma potrebni dve cevi. Po eni cevi gre prenos podatkov v eno smer in po drugi cevi tečejo podatki v obratno smer.

Prototip funkcije, s katero uporabnikov proces od sistemskega jedra zahteva komunikacijski kanal tipa cev, je naslednji,

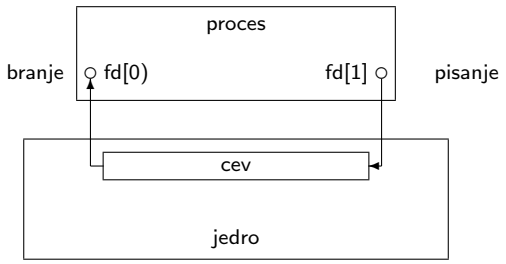
```
#include <unistd.h>
int pipe( int fd[2] );
```

Klic `pipe` vrne 0 in komunikacijski kanal je ustvarjen, ali -1 v primeru napake.

Funkcija `pipe` preko argumenta `fd`, ki je polje dveh elementov tipa `int`, vrne dva deskriptorja. Ta dva deskriptorja dajeta dostop do obeh koncev cevi, in sicer:

- `fd[0]` do konca za branje iz cevi in
- `fd[1]` do konca za pisanje v cev.

Sicer pa sta ta dva deskriptorja črpana iz zaloge datotečnih deskriptorjev.



Slika 14.1: *Koncept komunikacije prek cevi. Jedro na zahtevo* `pipe` *procesu dodeli enosmerni komunikacijski kanal – cev. Proces dobi dostop do obeh koncev cevi.*

Cev z obema koncema v istem procesu nima kakšne posebne koristi. Cev

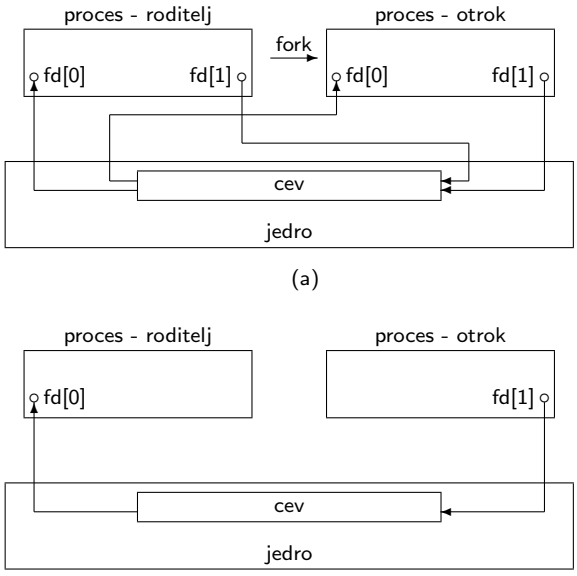
uporabimo tako, da jo 'napeljemo' med dva procesa, na primer takole, slika 14.2:

- odpremo cev s klicem `pipe()`.
- Ustvarimo nov proces s klicem `fork()`. Nov proces podeduje deskriptorja odprte cevi.
- En konec cevi, na primer za pisanje, pustimo odprt v enem procesu, drugi konec cevi, torej za branje, pustimo odprt v drugem procesu. Druga dva deskriptorja zapremo.

V cev pišemo s funkcijo `write(fd[1], ...)`, iz cevi beremo s funkcijo `read(fd[0], ...)`, dokler je ne zapremo s klicem `close()`. Cevi obravnavamo enako, kot da bi bile datoteke, le da operaciji `read/write` delujeta na dveh različnih koncih, ki sta dosegljiva prek dveh deskriptorjev. Branje iz cevi in pisanje v cev potekata asinhrono, na eni strani cevi proces bere, kar proces na drugi strani zapiše. Za izravnavo hitrosti poskrbi kar cev sama. Iz prazne cevi ni kaj brati, zato bralni proces z `read(fd[0],...)` na bralnem koncu cevi praviloma čaka na proces, ki piše, da kaj pošlje oziroma kaj 'napiše'. Če `read` vrne nič, to pomeni, da je bil konec za pisanje s `close(fd[1])` že zaprt in so bili prebrani vsi enkrat prej z druge strani poslani podatki.

Naslednje programsko besedilo najprej ustvari cev, nato ustvari nov proces, ki potem bere iz cevi, dokler ne bo cev zaprta za pisanje. Medtem roditelj bere znakovne nize s tipkovnice in jih pošilja v cev, dokler ni odtipkan znak za konec (<CTRL/D >). Tedaj zapre cev za pisanje, počaka na konec otroka in konča.

```
/* Komunikacija dveh procesov prek cevi       */

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

int main( void )
```



Slika 14.2: *Komunikacija s cevmi med procesi v sorodstvu. (a) Proces-roditelj s* `pipe` *najprej ustvari cev, nato s* `fork` *ustvari nov proces, ki deduje deskriptorje cevi. (b) Eden od procesov (v tem primeru otrok) 'zapre konec cevi' za branje in ohrani deskriptor za pisanje. Komplementarno ravna drugi proces (roditelj). Enosmerni komunikacijski kanal (cev) je s tem vzpostavljen.*

```
{
  int     fd[2];
  pid_t   cpid;
  char    mp[256];
  ssize_t n;
  int     status;
  int     done = 0;

  if( pipe( fd ) == -1 ){    //ustvarimo cev
    perror("pipe err");
    exit(1);
  }
  cpid = fork( );            //ustvarimo nov proces
  if( cpid == -1 ){
    perror("Fork err");
    exit(2);
```

```
  }

  if( cpid == 0 ){           //smo v 'otroku'
    close( fd[1] );          //zapremo cev za pisanje
    while( (n = read(fd[0], mp, sizeof(mp))) > 0 ){
      mp[n] = '\0';
      printf("Proces: %d, sprejel: %s", getpid(), mp);
    }
    close( fd[0] );          //konec branja
    if( n == -1){
      perror("read err");
      exit( 3 );
    }
    exit( 0 );
  }
/* ---smo v 'roditelju' --- */
  close(fd[0]);              //zapremo cev za branje
  while( done != 1){         //beremo niz in ga posljemo v cev
    printf("Proces: %d, vnesi niz ali <CTRL/D>: ", getpid() );
    if( fgets(mp, sizeof(mp), stdin) != NULL ){
      if( write(fd[1], mp, strlen(mp)) == -1){
        close( fd[1] ); perror("write err");
        exit(4);
      }
      sleep(1);  //dajmo nekaj prednosti otroku
    }
    else{
      close( fd[1] );
      done = 1;
    }
  }
  if( (cpid = wait( &status )) == -1){
    perror("wait err");
    exit(6);
  }
  printf("\nProces %d, docakal %d, exit status %x\n",
                                  getpid(), cpid, status);
  exit( 0 );
```

```
}
```

Cev ima končno velikost izravnalnega pomnilnika in če je cev 'polna', potem proces na njej z `write( fd[1], ...)` čaka, da bo branje iz cevi sprostilo dovolj prostora za pisanje. Kapaciteta cevi je odvisna od sistema in je običajno dovolj velika, da do čakanja na koncu za pisanje komajda pride. V novejših sistemih Linux je maksimalna velikost cevi 1MB, velikost cevi pa je moč tudi programsko ponastaviti. A ne glede na to velja, da, kadar je cev napeljana samo med dva procesa, od katerih eden bere in drugi piše, nam zaradi omejene velikosti cevi ni treba skrbeti. Za sinhronizacijo med brajem in pisanjem v vsakem primeru poskrbi cev sama.

Proces, ki piše v cev, lahko opravi več zaporednih vpisov v cev predno bralni proces opravi kakšno branje. A za cev je značilno, da se meje med zaporedno vpisanimi nizi bajtov ne ohranjajo. Po cevi se pretaka neprekinjen tok podatkovnih bajtov. Skratka, bralni proces lahko bere kose podatkovnih nizov povsem po svoje. Če bi morala procesa ohranjati strukturo in dolžino podatkovnih nizov, bi moral med njima vnaprej obstajati dogovor o strukturi oziroma 'protokol'.

V isto cev lahko hkrati piše in iz nje bere tudi več procesov. Kako naj se vrstijo operacije vpisa in branja je prepuščeno aplikaciji. A cev vendarle poskrbi za to, da se zaporedni vpisi različnih procesov zgodijo zaporedno. Torej, posamični vpisi različnih procesov se med sabo zagotovo ne bodo prepletali, če le niso preveliki. Rečemo, da se posamezen vpis v cev zgodi časovno nedeljivo ali 'atomično'. Maksimalno število bajtov, ki se zagotovo zapiše v enem kosu, je parameter sistema in v sistemih Linux znaša 4096 bajtov.

Povejmo še, da je prek cevi sicer moč komunicirati tudi tako, da operacija branja `read(fd[0], ... )` iz prazne cevi ne blokira. To dosežemo tako, da postavimo zastavico `O_NONBLOCK)` s funkcijo `fcntl()`, a o tem ne bomo govorili. Nadalje, pisanje v cev z zaprtim deskriptorjem za branje ima za posledico pošiljanje signala, ki sporoča, da je bila cev 'pretrgana', a tudi o signalih kasneje.

Osnovno vprašanje v zvezi z medprocesnimi komunikacijami na sploh in

v tem primeru s komunikacijo prek cevi je, kako narediti oba konca cevi 'znana' dvema za komunikacijo zainteresiranima procesoma. V primeru prejšnjega programskega besedila nismo imeli težav. Program je najprej ustvaril cev in shranil oba deskriptorja cevi v polju `fd`. Otrok je dedoval deskriptorja cevi in vrednosti polja `fd` v trenutku nastanka. Prenos deskriptorjev je bil s tem zagotovljen. A kaj bi se dogodilo, če bi proces inicializirali iz nekega drugega programa? V tem primeru bi bila vsebina polja `fd` zanj izgubljena. Čeprav bi bila oba deskriptorja cevi zanj še vedno odprta, pa bi ostal brez njune dejanske vrednosti. Eno od možnosti za prenos deskriptorjev ponujajo argumenti funkcije `exec`. Na primer:

```
....
sprintf(arg, "%d", fd[0] );
execlp("prog","prog", arg, NULL); //arg nosi deskriptor
...
```

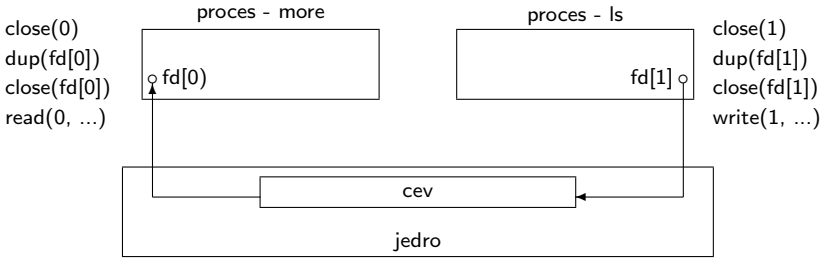
A za cevi se je iznašla elegantnejša rešitev, ki ji rečemo preusmerjanje. Pri tem izkoristimo pravilo oziroma dogovor, da se od procesa pričakuje, da bere s standardne vhodne datoteke, katere deskriptor je po vrednosti nič, ali simbolično `STDIN_FILENO`. Podobno proces piše na standardno izhodno datoteko, katere deskriptor je po vrednosti enak ena, ali simbolično `STDOUT_FILENO`. Torej, če bi bralni konec cevi 'spojili' na deskriptor 0, bi se branje z deskriptorja 0 dejansko nanašalo na branje iz cevi in ne več s standardne vhodne datoteke. Ekvivalentno bi dosegli pisanje v cev namesto na standardno izhodno datoteko, če bi pisalni konec cevi spojili na deskriptor z vrednostjo 1. Preusmerjanje omogočata funkciji `dup()` in `dup2()`, ki smo ju že spoznali v poglavju o vhodno izhodnih prenosih. Na primer, z zaporedjem:

```
close( 0 );             //ali close(STDIN_FILENO);
dup( fd[0] );
close( fd[0] );
read(0, ... );          //ali read(STDIN_FILENO, ... );
```

dosežemo, da se deskriptor konca cevi za branje `fd[0]` 'duplicira' oziroma izenači z deskriptorjem 0. Zato funkcija `read(0, ... )` od zdaj naprej bere iz cevi in ne več s standardne vhodne datoteke, ki je dejansko zaprta. Ekvivalentno ravnamo s koncem cevi za pisanje.

```
close( 1 );             //ali close(STDOUT_FILENO);
dup( fd[1] );
```

```
close( fd[1] );
write(1, ... );        //ali write(STDOUT_FILENO, ... );
```



Slika 14.3: *Proces "ls" zapre deskriptor standardne izhodne datoteke, podvoji deskriptor cevi za pisanje in zapre deskriptor cevi za pisanje. Proces "ls" sedaj z* `write(1, ...)` *piše v cev namesto na izhodno datoteko. Proces "more" zapre deskriptor standardne vhodne datoteke, podvoji deskriptor cevi za branje ter zapre deskriptor cevi za branje. Proces "more" z* `read(0, ...)` *sedaj bere iz cevi namesto iz vhodne datoteke, ki je tako zaprta.*

Osnovni koncept uporabe cevi s preusmerjanjem si poglejmo na primeru. Naslednji program oziroma proces najprej ustvari cev in nato še dva procesa (otroka), enega inicializira iz programa `ls` in drugega iz programa `more` ter med njima napelje cev. Kot vemo, ukaz `ls` izpiše vsebino želenega direktorija na standardno izhodno datoteko oziroma napravo, ki je zaslon. Ukaz `more`, ki mu rečemo 'filter', preprosto bere s standardne vhodne datoteke (tipkovnice) in prebrane podatke 'po straneh' izpisuje na zaslon. Zato v našem primeru za proces `ls` preusmerimo standardni izhod (deskriptor 1) v oddajni konec cevi, za proces `more` preusmerimo standardni vhod (deskriptor 0) na sprejemni konec cevi, slika 14.3. Rezultat preusmeritve je, da izpisovanje procesa `ls`, ki sicer uporablja deskriptor '1' za pisanje, poteka namesto na standardni izhod v cev. Podobno proces `more` namesto s standardnega vhoda bere iz cevi. S tem smo dosegli podoben učinek kot bi ga dosegli z ukazno vrstico školjke:

```
ls /etc | more
```

Pokončno črto v tem kontekstu beremo kar 'cev'. Proces `ls /etc`, ki se inicializira iz programa `/bin/ls`, bere vsebino direktorija `/etc` ter to pošilja v cev namesto na standardni izhod – zaslon. Standardni izhod smo pre-

usmerili v cev. Proces `more` se inicializira iz programa `more` ter namesto s standarne vhodne enote (tipkovnice) bere iz cevi ter prebrano 'po straneh' izpiše na standardno izhodno enoto, to je zaslon.

```
/* komunikacija dveh procesov prek cevi, brez preverjanja napak  */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main( void )
{
  int pfd[2];

  pipe( pfd );
  if( fork() == 0 ){
    close(1); dup( pfd[1] );  /* konec za pisanje ima deskriptor 1 */
    close( pfd[0] ); close( pfd[1] );
    execlp("ls", "ls", "/etc",(char *)0);
    exit( 1 );                /* za vsak slucaj, ce exec ne uspe   */
  }
  if( fork() == 0){
    close(0); dup( pfd[0] );  /* konec za branje ima deskriptor 0  */
    close( pfd[0] ); close( pfd[1]);
    execlp("more", "more", (char *)0);
    exit( 2 );                /* za vsak slucaj, ce exec ne uspe   */
  }
  close( pfd[0] ); close( pfd[1] );          /* roditelj zapre cev */
  exit( 0 );                            /* in zakljuci brez cakanja */
}
```

Posebej poudarimo, da je zapiranje neuporabljenih deskriptorjev cevi ne le priporočljivo ali primer dobre prakse, temveč pogosto nujnost za pravilno delovanje cevi.

Denimo, da bralni proces bere iz cevi nize bajtov, niz za nizom, dokler cev ni zaprta. Ko je cev zaprta, prebere niz dolžine nič in konča. Denimo, da pisalni proces piše nize bajtov, niz za nizom, dokler ne zapiše še zadnjega niza, zapre cev in konča. Vendarle, če bi cev po nesreči ostala odprta še v

kakšnem drugem procesu, bi za bralni proces cev ostala še naprej odprta. Zato bi proces 'obvisel' na `read` ter čakal na podatke, ki jih nikoli ne bo. Cev se namreč zares zapre šele, ko se zapre še zadnji odprti deskriptor.

Na koncu omenimo še možnost 'preusmerjanja' izpisa iz standardne izhodne datoteke v poljubno datoteko. To sicer nima nič skupnega s cevmi, ima pa s funkcijo `dup`. Uporabniki školjke poznajo sintakso ukazne vrstice, kot je denimo tale:

```
ls -l >listdir
```

Znak > v tem primeru zahteva preusmeritev izpisa, ki bi sicer šel na zaslon, v datoteko z imenom `listdir`. To je vgrajena funkcionalnost školjke, ki jo izvedena s klicem funckije `dup`. Školjka analizira ukazno vrstico ter naleti na znak 'večji'. Znak 'večji' razume kot zahtevano za preusmerjanje in preusmeri izpis. V načelu opravi naslednje zaporedje operacij:

```
...
fd = creat("listdir", 0644);
close(STDOUT_FILENO);
dup( fd );
close( fd );
write(STDOUT_FILENO, ... );
...
```

Na podoben način je realizirano preusmerjanje branja. Tedaj v ukazni vrstici navedemo znak 'manjši' pred imenom datoteke. To školjka razume kot zahtevo za branje iz poljubne druge datoteke namesto iz standardne vhodne datoteke.

14.3 Poimenovane cevi FIFO

Datoteka FIFO (First-In-First-Out) je poseben tip datoteke z dvema 'aktivnima' koncema: na enem koncu beremo kar na drugem koncu zapišemo. Kot pove ime se *prej zapisani* podatek *prej prebere*. Drugo ime za FIFO je cev z imenom ali *poimenovana cev* (Angl. 'Named pipe'). Poimenovane cevi omogočajo komunikacijo med poljubnimi procesi, ne le med procesi v sorodstvu. Prototip funkcije je naslednji:

```
#include <sys/stat.h>
int mkfifo(char *path, mode_t mode);
```

Funkcija vrne 0 in FIFO obstaja ali -1 v primeru napake.

Funkcija `mkfifo` je POSIX specifikacija za tvorjenje FIFO z `mknod` ('make node'). Argumenti k `mkfifo` imajo podoben pomen kot argumenti pri klicu `creat`. Klic `mkfifo` ustvari FIFO, to je datoteko tipa FIFO, ki jo zatem lahko odpremo z `open`.

Tipičen primer uporabe FIFO so komunikacije po načelu odjemalec–strežnik (Angl. client-server), pri čemer strežnik sprejema zahteve od večjega števila odjemalcev na vsem dobro znanem FIFO imenu. Ime FIFO ni seveda nič drugega kot ime datoteke z oznako 'p' nekje v datotečnem sistemu. Tako datoteko lahko naredimo tudi z ukazom `mknod` ali `mkfifo`, na primer:

```
mknod -m 0644 /tmp/pipeta p
ali enakovredno:
mkfifo -m 0644 /tmp/pipeta
```

Programček–proces `cevka.c`, ki bi lahko sprejemal sporočila drugih procesov–odjemalcev, bi bil v načelu potem videti takole:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define  JAVNI_FIFO "/tmp/pipeta"

int main( void )
{
  int r_fifo, n_read;
  char niz[128];

// na open proces blokira - caka na prvo zahtevo odjemalca
  if ( (r_fifo = open( JAVNI_FIFO, O_RDONLY)) == -1){
    perror("open err");
    exit(1);
```

```
  }
// proces sprejema zahteve odjemalcev
  while( (n_read = read(r_fifo, niz, sizeof(niz))) > 0){
   niz[n_read] = 0;
   printf("%s", niz);
 }
 return 0;
}
```

'Strežnik' sprejema znakovne nize ter jih izpisuje na zaslon, dokler še zadnji odjemalec ne zapre komunikacijskega kanala `fifo`. Preprost preizkus bi lahko izgledal takole, vsakega od procesov poženemo v svojem oknu.

```
cevka
```

```
cat >/tmp/pipeta
```

```
cat >/tmp/pipeta
```

Drugo vprašanje pa je, kako naj bi strežnik odgovarjal odjemalcem. Očitno bi za vsakega odjemalca potreboval svoj komunikacijski kanal. Odgovor na to vprašanje pa je že v domeni aplikacije in predhodnega dogovora med odjemalcem in strežnikom, torej komunikacijskega protokola. Ena od možnosti je naslednja. Odjemalec na primer ustvari privatno komunikacijsko cev in njeno ime sporoči strežniku po cevi z znanim naslovom. Strežnik se poveže na privatno cev odjemalca in komunikacija steče.

Naslednje programsko besedilo realizira 'strežnik', ki odgovarja na zahteve odjemalcev. Strežnik sprejema zahteve odjemalcev na 'javni' cevi, to je tisti datoteki FIFO, ketere ime poznajo vsi zainteresirani odjemalci (`JAVNI_FIFO`). Oblika sporočil med strežnikom in odjemalci je določena. Zahtevo odjemalca sestavljata `pid` odjemalca in kratek znakovni niz za kodo zahteve, ki je v našem primeru ena sama (`C_DATUM`). To naj bi bila denimo odjemalčeva poizvedba po datumu, ki ga ponuja strežnik.

Strežnik odjemalcu odgovori po odjemalčevi 'zasebni' cevi. Da je ime zasebne cevi za vsakega potencialnega odjemalca enoznačno, vsebuje poleg skupnega začetnega dela imena datoteke FIFO (`ZASEBNI_FIFO`) še `pid` odjemalca. Strežnik preveri veljavnost zahteve ter odjemalcu vrne svoj `pid` ter,

če je zahteva veljavna, tekoči datum.

```
/* ------------------------ VS -----------------------------
 Streznik: sfifo.c
   Streznik 'vecno' sprejema zahteve odjemalcev na 'dobro znanem naslovu'
   Dobro znani naslov je ime FIFO - JAVNI_FIFO
   Oblika zahteve odjemalca je dolocena in fiksna:
   pid_odjemalca+ascii znaki kode zahteve
   Streznik se odjemalcu odzove po zasebni fifo cevi odjemalca
   Ime fifo cevi je ZASEBNI_FIFO+pid odjemalca
   Streznik preprosto vrne svoj pid in tekoci datum

*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <strings.h>
#include <time.h>

#define  JAVNI_FIFO       "/tmp/s_pipeta"
#define  ZASEBNI_FIFO     "/tmp/c_pipeta_"
#define  L_C_MSG   4       //fiksna dolzina zahteve odjemalca
#define  L_S_MSG  32       //fiksna dolzina odgovora streznika

#define  C_DATUM  "101"    //naj bo to koda zahteve za datum

struct c_msg_t{
       pid_t pid;
       char  zahteva[L_C_MSG]; // L_C_MSG znakov zahteve
};

struct s_msg_t{
       pid_t pid;
```

```
        char  odgovor[L_S_MSG]; // L_S_MSG znakov odgovora
};


int main( void )
{
  int    c_fifo, s_fifo, s_read;
  char   c_name[128];
  mode_t umaska;
  struct c_msg_t c_msg;
  struct s_msg_t s_msg;
  time_t cas;
  char   *datum;

  umask( 0 );                 //da bo rezultat (umaska & ~0)
  umaska = S_IRUSR | S_IWUSR; //branje pisanje le za lastnika

// ustvarimo fifo za streznik, ce ne obstaja
  if ( mkfifo( JAVNI_FIFO, umaska ) == -1){
    if (errno != EEXIST){     //bo v redu, tudi ce fifo ze obstaja
      perror("serv mkfifo err");
      exit(1);
    }
  }
// odpremo fifo za branje, streznik blokira do prve povezave odjemalca
  printf("Streznik pripravljen, pid=%d\n", getpid());
  if ( (s_fifo = open( JAVNI_FIFO, O_RDONLY )) == -1){
    perror("serv fifo open err");
    exit(2);
  }
// odpremo isti fifo se za pisanje, tako da streznik ne konca zaradi close odjemalca
  if ( open( JAVNI_FIFO, O_WRONLY ) == -1){
    perror("serv fifo open err");
    exit(2);
  }
// beremo cev, da sprejmemo novo zahtevo odjemalca
  while( (s_read = read(s_fifo, &c_msg, sizeof(c_msg))) > 0){
    printf("Zahteva odjemalca %d, koda zahteve: %s\n", c_msg.pid, c_msg.zahteva);
```

```
    sprintf(c_name, "%s%5d", ZASEBNI_FIFO, c_msg.pid);
    if( (c_fifo = open(c_name, O_WRONLY)) == -1){
      perror("client fifo open err");
      exit(3);
    }
    s_msg.pid = getpid( ); // posljem svoj pid;
    if(strcmp(c_msg.zahteva,C_DATUM) == 0){ //veljavna koda zahteve
      cas = time( NULL );
      datum = ctime( &cas );
      strcpy(s_msg.odgovor,datum);
    }
    else{
      strcpy(s_msg.odgovor,"Neveljaven ukaz");
    }
    printf("Posiljam odgovor: %s\n", s_msg.odgovor);

    if ( write(c_fifo, &s_msg, sizeof( s_msg )) == -1){
      perror("client fifo write err");
      exit(4);
    }
    if ( close(c_fifo) == -1){
      perror("client fifo close err");
      exit(5);
    }
  }
  if( s_read == -1 ){
    perror("serv fifo read err");
    exit(6);
  }
  close( s_fifo ); //do tega pravzaprav ne bi smelo priti
  exit(0);
}
```

Komplementarno delujejo odjemalci. Programsko besedilo odjemalca je navedeno spodaj. Odjemalec najprej ustvari njemu lastno datoteko FIFO, ime datoteke sestavi iz dogovorjenega začetnega niza in svoje številke `pid`. Potem pošlje zahtevo strežniku po 'javni' cevi in pričqkuje njegov odgovor po 'zasebni' cevi. Ko odgovor pride, ga enostavno izpišena zaslon. Ker strežnik po oddaji odgovora z (`write(c_fifo, ...)`) s (`close(c_fifo)`) za-

pre zasebno cev odjemalca, odjemalec z `read( c_fifo, ...)` prebere 0, izstopi iz zanke, zapre cev in odstrani njeno ime.

```
/* ---------------------------- VS ----------------------------
 Odjemalec: cfifo.c
   Odjemalec od streznika zahteva naj mu poslje svoj pid ter tekoci datum
   Streznik sprejema zahteve odjemalcev na 'dobro znanem naslovu' - JAVNI_FIFO
   Oblika zahteve odjemalca je dolocena in fiksna:
   pid_odjemalca+nekaj ascci znakov za kodo zahteve
   Streznik se odjemalcu odzove po njegovi fifo cevi
   Ime fifo cevi je ZASEBNI_FIFO+pid odjemalca
   Streznik vrne svoj pid in tekoci cas (datum)

*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>

#define  JAVNI_FIFO       "/tmp/s_pipeta"
#define  ZASEBNI_FIFO     "/tmp/c_pipeta_"
#define  L_C_MSG   4       //fiksna dolzina zahteve odjemalca
#define  L_S_MSG  32       //fiksna dolzina odgovora streznika

#define  C_DATUM  "101"    //naj bo to koda zahteve za datum

struct c_msg_t{
       pid_t pid;
       char  zahteva[L_C_MSG]; // L_C_MSG znakov zahteve
};

struct s_msg_t{
       pid_t pid;
       char  odgovor[L_S_MSG]; // L_S_MSG znakov odgovora
};
```

```
int main( void )
{
  int    c_fifo, s_fifo, c_read;
  char   c_name[128];
  mode_t umaska;
  struct c_msg_t c_msg;
  struct s_msg_t s_msg;

  umask( 0 ); //da ne spreminjamo bitov dolocil, umaska & ~0
  umaska = S_IRUSR | S_IWUSR; //branje pisanje le za lastnika

  c_msg.pid = getpid();
  sprintf(c_name, "%s%5d", ZASEBNI_FIFO, c_msg.pid);

// ustvarimo zasebni fifo odjemalca
  if ( mkfifo( c_name, umaska ) == -1){
    perror("mkfifo err");
    exit(1);
  }
// odpremo streznikov javni fifo za pisanje
  if ( (s_fifo = open( JAVNI_FIFO, O_RDWR )) == -1){
    perror("serv_fifo open err");
    exit(2);
  }
// posljemo zahtevo strezniku
  strcpy(c_msg.zahteva, C_DATUM); // koda zahteve za datum
  if( write(s_fifo, &c_msg, sizeof(c_msg)) != sizeof(c_msg)){
    perror("serv_fifo write err");
    exit(3);
  }
// odpremo zasebni fifo odjemalca za branje
  if ( (c_fifo = open( c_name, O_RDONLY )) == -1){
    perror("client_fifo open err");
    exit(2);
  }
// beremo odgovor streznika
  while ((c_read = read(c_fifo, &s_msg, sizeof(s_msg))) > 0){
```

```
    printf("Odgovor streznika %d: %s\n", s_msg.pid, s_msg.odgovor);
  }
  if( c_read == -1 ){
    perror("client_fifo read err");
  }
  close( c_fifo );
  unlink( c_name );
  exit(0);
}
```

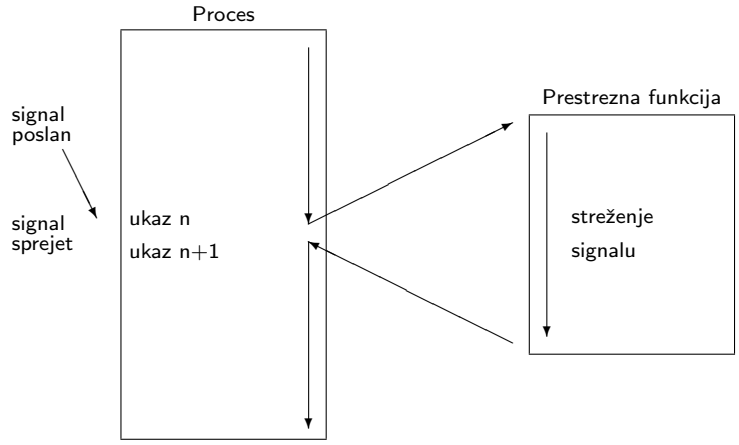
14.4 Signali

Signal je naznanilo procesu, da se je nekaj zgodilo. Signal je moč poslati. Signal lahko pošlje sistemskego jedro, proces drugemu procesu ali proces samemu sebi. Signal je moč sprejeti. Proces, ki signal sprejme, lahko signal obravnava na 'privzet' (Angl. default) način. Privzet ali pričakovan način obravnavanja signala je praviloma takojšen zaključek procesa. Izjemoma sme proces signal 'prezreti' (Angl. Ignore signal). Tedaj proces napreduje, kot da signala ne bi bilo. Signal je moč tudi 'presteči' (Angl. Catch signal). Proces v tem primeru obravnava signal po svoje. Kakšno dejanje bo sledilo kot posledica prestreženega signala, določi proces s *prestrezno funkcijo* (Angl. Signal handler).

Od trenutka, ko se signal pojavi, pa do trenutka, ko je procesu dostavljen, mine nekaj časa. V tem obdobju je signal 'v teku' (angl. Pending). Signal je moč začasno 'blokirati'. če se signal pojavi medtem ko je signal blokiran, ostane v teku do trenutka odblokade. Tedaj se signal dostavi procesu in ta se nanj ustrezno odzove.

Signali se pojavljajo *asinhrono* glede na napredovanje procesa, zato jih pojmujemo kot *programske prekinitve*, slika 14.4. Na signale pa je moč tudi čakati. Signali nosijo malo informacije in se uporaljajo predvsem za javljanje nenavadnih okoliščin ter nepravilnosti. Včasih se jih uporablja tudi za sinhronizacijo procesov.

Signali so bili v preteklosti na sistemih UNIX znani kot 'nezanesljivi', ker so se v nekaterih, sicer redkih okoliščinah, obnašali drugače kot je bilo predvideno s programom. Na primer, lahko bi se zgodilo, da bi proces signal spregledal. Ali, blokada signala bi bila lahko spregledana. Ta problem je bil v kasnejših sistemih v glavnem rešen. Standard POSIX.1 zahteva 'zanesljive' signale in uporabo funkcije `sigaction` namesto `signal`.



Slika 14.4: *Signal se pojavi kadarkoli, asinhrono z napredovanjem procesa. Proces se nanj odzove (prestreže signal), ko mu razmere to dopuščajo. Tedaj jedro v imenu procesa kliče prestrezno funkcijo. Ko se prestrezna funkcija vrne, proces nadaljuje na mestu prekinitve. Proces bi sicer lahko tudi čakal na signal.*

14.4.1 Tipi signalov

Ostaja več vrst signalov. V grobem jih delimo na 'standardne' in 'realnočasovne' (Angl. Real-time signals). Število signalov je z razvojem 'naraslo' na 31 (glej na primer signum.h), in sicer brez signalov za realni čas.

Seznam signalov, ki je povzet iz datoteke `signal.h (signum.h)`, sledi v spodnji tabeli:

```
/* Signals.  */
#define SIGHUP      1       /* Hangup                  (POSIX)  */
#define SIGINT      2       /* Interrupt               (ANSI)   */
#define SIGQUIT     3       /* Quit                    (POSIX)  */
#define SIGILL      4       /* Illegal instruction     (ANSI)   */
#define SIGTRAP     5       /* Trace trap              (POSIX)  */
#define SIGABRT     6       /* Abort                   (ANSI)   */
#define SIGIOT      6       /* IOT trap                (4.2 BSD)*/
#define SIGBUS      7       /* BUS error               (4.2 BSD)*/
#define SIGFPE      8       /* Floating-point exception  (ANSI)*/
#define SIGKILL     9       /* Kill, unblockable       (POSIX)  */
```

```
#define SIGUSR1    10       /* User-defined signal 1  (POSIX)  */
#define SIGSEGV    11       /* Segmentation violation (ANSI)   */
#define SIGUSR2    12       /* User-defined signal 2  (POSIX)  */
#define SIGPIPE    13       /* Broken pipe            (POSIX)  */
#define SIGALRM    14       /* Alarm clock            (POSIX)  */
#define SIGTERM    15       /* Termination            (ANSI)   */
#define SIGSTKFLT  16       /* Stack fault.                    */
#define SIGCLD     SIGCHLD  /* Same as SIGCHLD       (System V)*/
#define SIGCHLD    17       /* Child status has changed (POSIX)*/
#define SIGCONT    18       /* Continue               (POSIX)  */
#define SIGSTOP    19       /* Stop, unblockable      (POSIX)  */
#define SIGTSTP    20       /* Keyboard stop          (POSIX)  */
#define SIGTTIN    21       /* Background read from tty (POSIX)*/
#define SIGTTOU    22       /* Background write to tty (POSIX) */
#define SIGURG     23       /* Urgent condition on socket(4.2 BSD)*/
#define SIGXCPU    24       /* CPU limit exceeded     (4.2 BSD)*/
#define SIGXFSZ    25       /* File size limit exceeded  (4.2 BSD)*/
#define SIGVTALRM  26       /* Virtual alarm clock    (4.2 BSD)*/
#define SIGPROF    27       /* Profiling alarm clock  (4.2 BSD)*/
#define SIGWINCH   28       /* Window size change(4.3 BSD, Sun)*/
#define SIGPOLL    SIGIO    /* Pollable event occurred  (System V)*/
#define SIGIO      29       /* I/O now possible       (4.2 BSD)*/
#define SIGPWR     30       /* Power failure restart (System V)*/
#define SIGSYS     31       /* Bad system call.                */
#define SIGUNUSED  31

#define _NSIG      65       /* Biggest signal number + 1
                                  (including real-time signals)*/

#define SIGRTMIN        (__libc_current_sigrtmin ())
#define SIGRTMAX        (__libc_current_sigrtmax ())
```

Številke standardnih signalov sistema Linux gredo od 1 do 31. Vsak signal ima svoje ime oziroma makro definicijo, ki je pridružena številki signala. Zaradi prenosljivosti programov se priporoča uporaba simboličnih namesto dejanskih vrednosti konstant. Številske vrednosti nekaterih signalov se lahko od sistema do sistema namreč razlikujejo. Imena signalov začenjajo s

SIG, kot na primer SIGKILL. Signal SIGKILL ima številko 9. Nekatere izmed signalov pošlje sistemsko jedro. Taki primeri so denimo izpad napajanja SIGPWR, deljenje z nič SIGFPE, neveljaven ukaz SIGILL, neveljaven naslov SIGSEGV, ali napaka na vodilu SIGBUS. Ti signali so tako ali drugače povezani s strojno opremo, vendar jih lahko pošlje direktno tudi uporabniški proces. Drugi signali so softverskega značaja, kot na primer pisanje v cev z zaprtim koncem za branje (SIGPIPE), ali konec otroka oziroma sprememba stanja otroka (SIGCHLD).

Potem je še skupina signalov, ki so povezani s časovniki in skupina signalov za predčasen zaključek procesa. Nekateri signali oziroma številke signalov so proste za specifične potrebe uporabnikov, na primer signala SIGUSR1 in SIGUSR2.

Uporabniki sistema Linux dobro poznajo ukaz kill. Ukaz kill je koristen, kadar želimo pokončati proces, ki ne deluje kot bi želeli. Na primer, ukazna vrstica v katerikoli od oblik:

```
kill -9 pid
ali:
kill -SIGKILL pid
ali:
kill -KILL pid
ali:
kill -s SIGKILL pid
```

pošlje signal SIGKILL procesu pid. Signala SIGKILL ni moč blokirati, prezreti ali prestreči. Ko proces s številko pid sprejme signal, se mora brezpogojno zaključiti. Blažja in priporočena možnost za pokončanje procesa je signal SIGTERM. Signal SIGTERM je namreč moč prestreči. Dobro zasnovana aplikacija bo imela prestrezno funkcijo, ki bo v primeru signala SIGTERM najprej 'pospravila za sabo' in šele nato končala proces.

Večina uporabnikov pozna tudi 'kontrolne' tipkovne kombinacije. To so izbrane črkovne tipke, ki jih pristisnemo skupaj s tipko <CTRL>, tipka pa se izpiše kot ^. Na primer, naslednje tipkovne kombinacije tedaj, ko proces prevzame terminalsko okno, pošljejo procesu signal:

```
^C: signal SIGINT  (2)
```

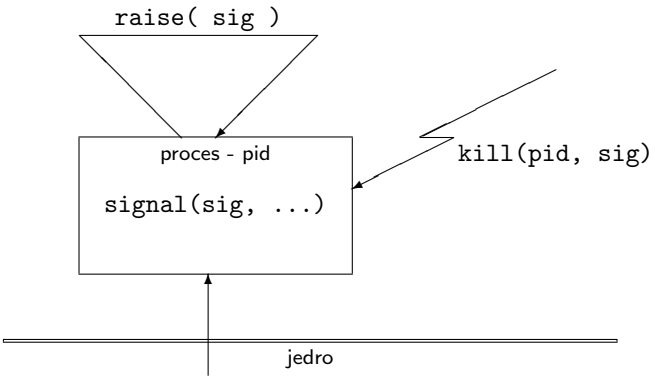
```
^\: signal SIGQUIT (3)
^Z: signal SIGTSTP (20)
```

Točneje, signal pošlje procesu gonilnik terminalskega okna. Signal `SIGTSTP` ustavi proces, podobno kot `SIGSTOP`, le da se kot posledica slednjega proces zagotovo ustavi. Signala `SIGSTOP` namreč ni moč blokirati, prestreči ali prezreti. Proces se ponovno postavi v stanje izvajanja, ko dobi signal `SIGCONT`. Uporabniki kombinacijo <`CTRL`>/`Z` sicer uporabljajo predvsem zato, da proces najprej ustavijo, zato nadzor terminala spet prevzame školjka in nato z ukazom školjki `bg` (kot 'background') od školjke zahtevajo, da proces nadaljuje v ozadju, podobno kot če bi proces izvršili z & v ukazni vrstici.

Signal `SIGQUIT` je koristen tedaj, kadar proces 'zaide' in ga hočemo pokončati. Večina uporabnikov je navajena, da proces predčasno prekine s kombinacijo <`CTRL`>/`C`, ki procesu pošlje signal `SIGINT`. Od procesa, ki sprejme signal `SIGINT` se pričakuje, da bo končal, a signal je moč tudi prestreči in obravnavati na drugačen način.

14.4.2 Funkcije za upravljanje signalov

V nadaljevanju si bomo ogledali sistemske funkcije za upravljanje s signali. Kot že povedano, lahko signal pošlje proces drugemu procesu, proces samemu sebi, ali procesu neposredno sistemsko jedro, slika 14.5. Signal pošlje proces drugemu procesu s funkcijo `kill`. Če hoče proces poslati signal samemu sebi, lahko to naredi s funkcijo `raise`. Sistemsko jedro pošlje signal kot stranski učinek nepravilnosti v napredovanju procesa, na primer pri deljenju z nič, napaki na vodilu, posegu iz naslovnega področja in podobno. Klic funkcije `signal` določi, kako naj proces signal obravna oziroma kako naj ga 'prestreže'. Namesto funkcije `signal` se sicer spričo prenosljivosti kode priporoča uporabo funkcije `sigaction`. Slednja ima tudi dodatno funkcijonalnost, ki je funkcija `signal` nima. Ne glede na to si bomo v nadaljevanju ogledali uporabo funkcije `signal`, ki je lažja za uporabo, medtem ko je osnovni koncep uporabe signalov ekvivalenten.



Slika 14.5: *Signal lahko pošlje procesu drug proces s* `kill`*, proces sam sebi z* `raise`*, ali sistemsko jedro. Obravnavanje signala predpišemo s funkcijo* `signal`*.*

Funkcija `signal`

```
#include <signal.h>

void (*signal(int sig, void (*action)(int)))(int);
```

Funkcija `signal` vrne prejšnjo vrednost naslova prestrezne funkcije ali `SIG_ERR` v primeru napake.

S funkcijo `signal` je mogoče procesu izbirati enega izmed treh načinov obravnavanja signala. Z argumentom `sig` se predpiše številko (ime) signala oziroma za kateri signal gre in `action` določa način njegovega obravnavanja. Možen je eden od treh načinov obravnavanja signala:

- `SIG_DFL` - privzet (Angl. default) način,
- `SIG_IGN` - prezri signal (Angl. Ignore),
- `action` - prestrezi signal (Angl. Catch signal).

Za večino signalov je privzeti način obravnavanja signala *konec* procesa. Tak način ponastavimo s klicem:

```
...
signal( SIGxxxx, SIG_DFL );
...
```

Nekatere signale, a ne vseh, je moč prezreti. Signalov `SIGKILL` in `SIGSTOP` se denimo ne da prezreti, ostale načeloma lahko. Klic funkcije

```
...
signal( SIGxxxx, SIG_IGN );
...
```

doseže tak način. Proces signala v resnici niti ne dobi.

V primeru, da izberemo tretjo možnost, in sicer da proces prestreže signal, navedemo kot drugi argument funkcije `signal` naslov prestrezne funkcije. Prestrezno funkcijo `action` določi načrtovalec programske opreme. Po povratku iz prestrezne funkcije se nadaljuje izvajanje procesa na mestu kjer je bil prekinjen v trenutku sprejema signala. Prestrezna funkcija ima en parameter `sig`, ki je tipa `int` in običajno služi za prenos številke signala.

Prototip funkcije `signal` je videti dokaj kompliciran. Pravi, da funkcija `signal` vrne kazalec na funkcijo, ki ne vrača ničesar (je tipa `void`) in funkcija, na katero kaže vrnjeni kazalec, ima en argument tipa `int`, zato zadnji `int` v deklaraciji.

Argumenta funkciji `signal` sta dva. Prvi argument je tipa `int` in pomeni številko signala (`sig`). Drugi argument je kazalec na funkcijo `action`. Funkcija `action` je tipa `void`, ne vrača ničesar in sprejme en argument tipa `int`. Če povzamemo, funkcija `signal` vrne naslov prestrezne funkcije, ki je bila v veljavi pred klicem in sprejme dva argumenta, številko signala in naslov nove prestrezne funkcije. Ogrodje prestrezne funkcije je spričo povedanega naslednje:

```
void action( int sig )
{
  //programsko besedilo za obravnavanje signala
}
```

Funkciji `kill` in `raise`

Signale pošiljamo s funkcijama `kill` in `raise`,

```
#include <sys/types.h>
#include <signal.h>
```

```
int kill(pid_t pid, int sig);
int raise(int sig);
```

Funkciji vrneta 0 in signal je poslan ali -1 v primeru napake.

Funkcija `kill` pošlje *signal* procesu ali skupini procesov. Številko procesa podamo z argumentom `pid`. Interpretacija vrednosti argumenta je naslednja:

- če je `pid` > 0, se signal pošlje temu procesu,
- če je `pid = 0`, se signal pošlje vsem procesom iz iste skupine kot je proces, ki pošilja signal (z enakim 'group ID'),
- če je `pid = -1`, se signal pošlje vsem procesom, ki jim proces sme poslati signal,
- če je `pid` $<$ -1 se signal pošlje vsem procesom v skupini s številko `-pid`.

Argument `sig` je številka signala ali nič. Pošiljanje 'nultega' signala ima poseben pomen. Naslednji klic:

```
...
err = kill( pid, 0 );
if( err == 0 )
  printf("proces obstaja in mu smem poslati signal");
if( err == -1 && errno == EPERM)
  printf("proces obstaja, a mu nimam pravice poslati signala");
if( err == -1 && errno == ESRCH)
  printf("tak proces ne obstaja");
...
```

ne pošlje signala, temveč preveri, če je signal procesu moč poslati. Odvisno od pravic procesa, ki pošilja signal, je signal preizkušanemu procesu dovoljeno poslati ali pa ne. Če klic uspe in vrne nič, potem proces obstaja in mu smemo poslati signal. Če proces obstaja, klic funkcije vrne -1 in spremenljivka `errno` je enaka `EPERM`. Če pa proces ne obstaja, klic ne uspe ter vrne -1, spremenljivka `errno` pa vsebuje `ESRCH`.

S funkcijo `raise` pošlje proces signal samemu sebi in je ekvivalenten klicu

```
...
kill( getpid(), sig );
...
```

Naslednji program iz ukazne vrstice prebere številko signala, ga prestreže in pošlje samemu sebi. Potem vrne obravnavanje signala nazaj na privzet način. Nekaterih signalov se seveda ne da prestreči.

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

static void prestreziSignal( int );

int main(int argc, char **argv)
{
  int signo;

  if (argc != 2){
    printf("Uporaba: %s signo\n", argv[0]);
    exit(1);
  }
  signo = atoi( argv[1] );
  signal( signo, prestreziSignal );
  printf("Posiljam signal %d\n", signo );
  kill( getpid(), signo ); //ekvivalentno raise( signo );
  return( 0 );
}
static void prestreziSignal( int sig )
{
  printf("Prestregel signal: %d\n", sig);
  signal( sig, SIG_DFL ); //in vrnemo na privzet nacin
  return;
}
```

V nadaljevanju sledi preprost programček `mkill.c`, ki sprejme dva argumenta. Prvi argument `signo` je celo število, ki ga program obravnava kot številko signala, ki naj ga pošlje. Drugi argument `pid` pomeni številko

procesa, kateremu pošlje signal. Če je številka signala veljavna in proces obstaja, programček `mkill` pošlje signal ter izpiše ime in številko poslanega signala.

```
/* mkill.c
   Uporaba: mkill signo pid
   poslje signal signo procesu pid
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <errno.h>

int main( int argc, char **argv )
{
  pid_t pid;
  int signo, err;

  if( argc != 3){
    printf("Uporaba: %s signo pid\n", argv[0] );
    exit(1);
  }
  signo = atoi(argv[1]);
  pid   = atoi(argv[2]);
  if( signo < 1 || signo >= _NSIG){
    printf(" %s: napacen signal %d?\n", argv[0], signo );
    exit(2);
  }
  if( pid < 1){
    printf("%s: neveljavna stevilka procesa, %d ?\n", argv[0], pid);
    exit(2);
  }
  err = kill( pid, 0); //preizkus procesa z 'nultim' signalom

  if( err == -1 && errno == EPERM){
    printf("%s: procesu %d nimamo pravice poslati signala\n", argv[0], pid);
    exit(3);
```

```
  }
  if( err == -1 && errno == ESRCH){
    printf("%s: proces %d ne obstaja\n", argv[0], pid);
    exit(4);
  }
  if( kill( pid, signo ) == -1){
    printf("%s: signal ni poslan, proces %d?\n", argv[0], pid);
    exit(3);
  }
  printf("%s: poslan signal %s (%d)\n", argv[0], strsignal(signo), signo);
  exit(0);
}
```

Izpis po izvršitvi

```
mkill signo pid
```

s signo = 8 in s pid procesa, ki se izvaja, bi bil videti takole:

```
mkill: poslan signal Floating point exception (8)
```

14.4.3 Funkciji `alarm` in `pause`

S funkcijo `alarm` je moč upravljati časovnik procesa.

```
#include <unistd.h>

unsigned int alarm(  unsigned int seconds );
```

Vrne 0 ali čas v sekundah do alarma.

Klic nastavi začetno vrednost časovnika. Ko se časovnik izteče, dobi proces signal `SIGALRM` in navadno konča, vendar se ta signal lahko prestreže in ukrepa drugače. Vsak proces ima svoj, a en sam alarmni časovnik. Funkcija `alarm` si v resnici deli časovnik s funkcijo `setitimer`, tako da uporaba obeh hkrati ni smiselna. Tudi uporaba funkcije `alarm` in funkcije `sleep` hkrati ni priporočljiva, ker implementacija funkcije `sleep` zelo verjetno temelji na istem časovniku.

Časovnik se 'ugasne' s klicem `alarm( 0 )`, medtem ko klic `alarm( )` brez

argumenta vrne čas do izteka časovnika ali nič, če časovnik ni bil nastavljen.

S funkcijo `pause` je procesu moč čakati na sprejem signala, torej vezati svoje nadaljevanje na nek dogodek v drugem procesu. Povedano drugače, proces je moč sihronizirati z drugim procesom.

```
#include <unistd.h>

int pause( void );
```

Klic funkcije postavi proces v stanje 'ustavljen', dokler proces ne dobi signala in se prestrezna funkcija vrne. Ko signal pride in se prestrezna funkcija vrne, se `pause` zaključi in vrne -1, spremenljivka errno pa je postavljena na `EINTR`.

Naslednji program prikazuje preprosto izvedbo funkcije `sleep`. Program si sam sebi pošlje signal `SIGALRM` in ga prestreže.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

static  void PrestreziAlarm( int );

int main( void )
{
  int spi( int );

  printf("grem spat..."); fflush(stdout);
  spi( 20 ); /* preiskusni klic */
  printf("...konec spanja\n");
  exit( 0 );
}
static void PrestreziAlarm( int sig )
{
  /* zgolj prestezemo signal */
  return;
}
int spi( int sekund )
```

```
{
  signal( SIGALRM, PrestreziAlarm );
  alarm( sekund );      /* poslji signal SIGALRM  */
  pause( );             /* cakaj na signal */
  return( alarm( 0 ) ); /* ugasni casovnik */
}
```

Vprašajmo se še, kaj se zgodi, če naslednjemu programu/procesu pošljemo signal `SIGINT`, bodisi z ukazom

```
kill -INT pid
```

ali s prekinitveno tipkovno kombinacijo, ki je običajno $< ctrl > /c$.

```
#include <stdio.h>
#include <signal.h>

static void MojIntStreznik( int );

void main( void )
{
  printf("Tipkaj CTRL/C vsaj trikrat\n");
  signal(SIGINT, MojIntStreznik );
  for( ; ; );   /* neskoncna zanka */

} /* End main */

void MojIntStreznik( int sig )
{
  static int i = 0;
  printf("To je moj SIGINT streznik, poskus %d\n", ++i);
  if( i == 3) signal(SIGINT, SIG_DFL );
}
```

14.4.4 Funkcija `abort`

S funkcijo `abort` pošlje proces signal `SIGABRT` samemu sebi. Signal pomeni zahtevo za predčasen konec procesa. Še pred zaključkom proces generira datoteko z imenom 'core', v katero shrani posnetek stanja procesa v tre-

nutku sprejema signala. Ta datoteka je kasneje lahko v pomoč pri analizi procesa in iskanju morebitnega vzroka napak v delovanju procesa. Kot vse ostale signale, je tudi signal `SIGABRT` moč poslati s funkcijo `kill`. Vendar je klic funkcije `abort` 'močnejši' kot denimo klic

```
kill(getpid(), SIGABRT);
```

Signal `SIGABRT` je namreč moč blokirati, prezreti ali prestreči. A od funkcije `abort` se zahteva, da morebitno blokado ali neupoštevanje signala izniči. V primeru, da proces prestreže signal, funkcija `abort` ponovno pošlje signal in proces se konča. Funkcija `abort` se nikoli ne vrne.

14.4.5 Signali realnega časa

Signali realnega časa so bili uvedeni s specifikacijo POSIX.1b. Standardni UNIX signali imajo namreč nekatere pomankljivosti, ki jih v sistemih realnega časa ne moremo spregledati. Bistvena pomankljivost standardnih signalov je ta, da je ponovitev istega signala pred streženjem prejšnji zahtevi spregledana. Ponovitve signala se ne beležijo. Signali realnega časa nimajo te pomankljivosti. Ponovitve istega tipa signala se beležijo v čakalni vrsti in so dostavljene v enakem zaporedju, kot so se pojavile.

Naslednja dobra lastnost signalov realnega časa je, da so signali razvrščeni po prioritetah od `SIGRTMIN` do `SIGRTMAX`. Višja številka pomeni višjo prioriteto signala. Vse številke signalov realnega časa so višje od številk standardnih signalov. Signali realnega časa, ki čakajo na streženje, bodo razvrščeni po prioritetah in streženi v zaporedju od najvišje proti nižjim prioritetam. Signali z enako prioriteto bodo streženi v enakem zaporedju, kot so se pojavili.

Nadalje, pri uporabi signalov na sploh in pri delu s signali realnega časa še toliko bolj, morajo biti funkcije, ki jih morebiti kličemo znotraj prestrezne funkcije, *ponovljive* (Angl. Reentrant). To pomeni, da ob morebitni prekinitvi funkcije med izvajanjem, dovoljujejo ponovni klic oziroma ponovni vstop v funkcijo. Prvo pravilo pri načrtovanju ponovljive funkcije je, da ne sme uporabljati statičnih spremenljivk. Niso vse sistemske funkcije

ponovljive. Denimo, vse funkcije standardne vhodno izhodne knjižnice že niso take. Uporaba funkcij standardne vhodno izhodne knjižnice znotraj prestrezne funkcije se zato odsvetuje, morebitno uporabo kljub svarilu pa označuje kot 'nevarno'.

Nevarnosti, da bi prišlo do nekonzistence sicer ne bi bilo, če bi bile vse klicane funkcije 'neprekinljive', kar pomeni, da se vedno izvedejo od začetaka do konca brez prekinitve. Drugo vprašanje pa je, kako bi bilo potem z odzivnostjo sistema. Na splošno pa vedno velja pravilo naj bo prestrezna funkcija kratka in enostavna.

Signale realnega časa praviloma pošiljamo s funkcijo `sigqueue`, a ni ovire, da ne bi signalov realnega časa pošiljali tudi s funkcijo `kill`. Če bo signal v območju številk realnega časa poslan s funkcijo `kill` tudi zares imel lastnosti signala realnega časa, pa je odvisno od implementacije jedra. Prestrezno funkcijo signala realnega časa pa nastavimo s funkcijo `sigaction`. Več o tem ne bomo govorili.

Poglavje 15

Semaforji, deljen pomnilnik in sporočila

V tem poglavju bomo obravnavali tri mehanizme medprocesnih komunikacij: deljen pomnilnik, semaforje in sporočila. Ogledali si bomo družino funkcij, ki so bile razvite v sistemu UNIX System V in so splošno znane kot *System V IPC*. Sorodna družina funkcij za medprocesno komunikacijo na podlagi deljenega pomnilnika, semaforjev in sporočil izhaja iz POSIX.1b razširitev za realni čas. Vmesnik API za medprocesne komunikacije, ki je skladen s specifikacijo POSIX, je zato poimenovan *POSIX IPC*.

Med programskima vmesnikoma System V IPC in POSIX IPC v pogledu funkcionalnosti ni velikih razlik. Oba dajeta podlago za komunikacijo med procesi po načelu prenašanja sporočil in po načelu dostopa do skupnega pomnilnika. Oba realizirata semaforje za potrebe medprocesne sinhronizacije. Vmesnika se bolj razlikujeta s programerskega stališča. POSIX IPC je konceptualno bližji datotečnemu sistemu. Denimo, POSIX sporočila odpremo za komunikacijo ter na koncu zapremo. Tudi semaforje POSIX najprej odpremo in nazadnje zapremo. Enako ugotovimo pri delu z deljenim pomnilnikom. Po drugi strani skuša System V IPC ustvariti enoten koncept obravnavanja tako semaforjev, deljenega pomnilnika kot sporočil, čeprav gre za korenito različne mehanizme komuniciranja.

S stališča programerja se oba koncepta najbolj razlikujeta v pogledu semaforjev. POSIX semaforji so preglednejši in dosti lažji za uporabo. Semaforji System V IPC so za večino praktičnih primerov morda celo preveč splošni. Tudi sporočilni sistem POSIX je nekoliko naprednejši od sporočil sistema UNIX V. Obe različici ohranjata strukturo sporočil in razmejitev med sporočili v zaporedju sporočil, a sporočilne vrste POSIX dajejo direktno podporo prioritetnemu obravnavanju sporočil. POSIX sporočila realizirajo razvrščanje sporočil po prioritetah ter dostavljanje sporočil glede na prioriteto, medtem ko sporočila sistema UNIX V prioritete sporočil sicer omogočajo preko tipa sporočila, a je prioritetni sistem sporočil potrebno nadgraditi.

Ne glede na določene prednosti, ki bi jih smeli pripisati vmesniku POSIX IPC, si bomo v nadaljevanju najprej ogledali bolj uveljavljene medprocesne komunikacije, ki jih poznamo iz sistema UNIX V.

15.1 Uvod v System V IPC

Še predno zabredemo v podrobnosti si oglejmo glavne lastnosti programskega vmesnika ter imenujmo glavne sistemske funkcije.

- *Semaforji* so namenjeni za sinhronizacijo procesov. Urejeni so v množice semaforjev. V množici je eden ali več semaforjev in vsak proces ima lahko eno ali več množic semaforjev. Operacije nad množico semaforjev so atomične. Glavne funkcije in njihove naloge so:
  - `semget` ustvari množico semaforjev,
  - `semctl` upravlja z množico, postavi začetne vrednosti semaforjev,
  - `semop` izvaja operaciji čakaj in javi.
- *Deljen pomnilnik* omogoča, da si večje število procesov deli isti del pomnilnika. Pomeni, da so isti okvirji strani preslikani v logična naslovna področja več procesov. Glavne funkcije za delo z deljenim

pomnilnikom in njihov pomen so:

  - `shmget` od jedra zahteva deljen pomnilnik,
  - `shmctl` upravlja z deljenim pomnilnikom,
  - `shmat` procesu 'pripne' pomnilnik in mu omogoči dostop,
  - `shmdt` deljeni pomnilnik 'odpne' in dostop do pomnilnika ni več mogoč.

- *Sporočila* ali sporočilne vrste omogočajo strukturirano komunikacijo s prioritetnim obravnavanjem sporočil. Ni treba, da bi bili procesi v sorodstvu. Glavne funkcije in njihov pomen so:
  - `msgget` ustvari sporočilno vrsto,
  - `msgctl` upravlja in tudi odstani sporočilno vrsto,
  - `msgsnd` pošlje sporočilo in
  - `msgrcv` sporočilo sprejme.

Čeprav se mehanizmi semaforjev, deljenega pomnilnika ter sporočil med sabo bistveno razlikujejo, pa je shema poimenovanja funkcij pri vseh enaka. To pomaga pri delu s funkcijami. Vse funkcije tipa `get`, ne glede na izbrani mehanizem komunikacije, ustvarijo komunikacijski objekt. Če objekt že obstaja, zagotovijo dostop do objekta. Na nek način so sorodne funkciji `open`. Funkcije vračajo unikatno številko objekta, prek katere je zagotovljen dostop do objekta, torej semaforja, deljenega pomnilnika ali sporočilne vrste. Na primer, klic

```
if( (semid = semget(key, 1, 0644)) == -1) perror("semget err");
```

vrne *identifikator* oziroma *prepoznavnik* množice semaforjev z enim semaforjem in določili dovoljenj 0644 za lastnika, skupino in ostale. `semid` je pozitivno celo število, enako kot velja za datotečni deskriptor. Za razliko od funkcije `open`, ki procesu vrne deskriptor in ta je veljaven v okviru procesa, pa funkcije `get` vrnejo identifikator, ki je unikaten in *veljaven na nivoju sistema*. Vsak proces, ki pozna `semid` lahko, seveda če ima dovoljenje, dostopa do tega objekta. To ima lahko dobre in slabe posledice, zato

je prav, da se tega zavedamo.

Prvi argument vseh funkcij `get` je `key`. Argument `key` je celo število, kar niti ni najbolj pomembno. Temeljna naloga klica `get` je, da preslika `key` v unikatni identifikator objekta. Jedro zagotavlja, da bo za dani `key` objekta, ki že obstaja, klic vedno vrnil enak identifikator. Če pa komunikacijski objekt še ne obstaja, bo jedro ustvarilo nov objekt in klic bo vrnil unikatni identifikator novega objekta. Če povzamemo, argument `key` zagotavlja procesom dostop do istega objekta in skuša preprečiti nenameren dostop do napačnega objekta.

Poglejmo primer. Denimo, da deljeni pomnilniški segment za dani `key` še ne obstaja. V tem primeru klic

```
if( (shmid = shmget( key, 4096, IPC_CREAT | 0644 )) == -1)
  perror("shmget err");
```

ustvari nov pomnilniški segment velikosti 4096 bajtov ter vrne unikatni identifikator segmenta `shmid`. V nasprotnem primeru, ko pomnilniški segment za podani `key` že obstaja, klic vrne identifikator obstoječega segmenta.

Ni narobe, če skuša več procesov, ki bi želeli med sabo komunicirati, ustvariti isti komunikacijski objekt. A v komunikacijah po načelu odjemalec/strežnik je v navadi, da objekt ustvari strežnik (klic z določilom `IPC_CREAT`), medtem ko odjemalci to določilo opustijo.

Dodatna možnost za identifikacijo komunikacijskega objekta je izbira `IPC_PRIVATE` za vrednost argumenta `key`. Na primer, klic

```
if( (shmid = shmget( IPC_PRIVATE, 4096, 0644 )) == -1)
  perror("shmget err");
```

ustvari nov pomnilniški segment 'zasebnega' značaja. Ta način je prikladen za procese v sorodstvu, ki si brez težave izmenjajo `shmid` in skrb za isti ter unikaten `key` je odveč. Tudi določilo `IPC_CREAT` tedaj ni potrebno. Klic v vsakem primeru ustvari nov pomnilniški segment.

V zvezi z medprocesno komunikacijo se *vedno* zastavlja osnovno vprašanje, kako zagotoviti dostop večjega števila procesov do istega komunikacijskega

kanala. Konkretno, kako identifikacijsko številko 'oznaniti' vsem udeleženim procesom. Ta problem je v primeru sorodstveno vezanih procesov lažje rešljiv z `IPC_PRIVATE`. Dodatno podporo pri izbiranju vrednosti argumenta `key` za procese izven sorodstva daje funkcija `ftok` (beri 'File To Key'). Tipičen primer uporabe funkcije nakazuje naslednji kos programa:

```
#define MY_MODE   O_CREAT|O_WRONLY|O_TRUNC
#define MY_FLAGS  S_IRUSR | S_IWUSR
  ...
  key_t key;
  int   ipcid;
  char  *filename = "./ipckey"
  ...
  //datoteka mora obstajati, torej jo ustvarimo, ce je se ni
  if(open(filename, MY_MODE, MY_FLAGS) == -1)
    perror("open err");
  ...
  //ustvarimo unikaten key
  if( (key = ftok(filename, 'k')) == -1)
    perror("ftok err");
  ...
  //ustvarimo mnozico semaforjev (ali deljen pom, ali sporocila)
  if( (ipcid = semget( key, ... )) == -1)
    perror("semget err");
  ...
```

Računanje vrednosti za `key`, ki ga opravi jedro, je dokaj zapleteno. V izračun vključi ime datoteke, ki mora obstajati, 8-bitni argument k funkciji, številko datotečnega vozlišča in še nekatere druge parametre datotečnega sistema, a to niti ni pomembno. Važno je, da za večino praktičnih primerov uporabe ni bojazni, da bi do 'trčenja' pri izbiri ključa prišlo.

Funkcije tipa `ctl` nadzirajo objekt oziroma objekt dodatno pripravijo za uporabo. Operacije, ki jih lahko z danim objektom počnemo, so sicer v veliki meri odvisne od vrste objekta. Namreč, ni vseeno ali imamo opravka z deljenim pomnilnikom ali s sporočili. A operacija `IPC_RMID`, s katero odstranimo objekt, je potrebna za vse, ne glede na vrsto objekta. Na primer, s klicem

```
if( msgctl( msgid, IPC_RMID, NULL) == -1)
  perror("message IPC_RMID err");
```

odstanimo sporočilno vrsto. Operacija je na nek način sorodna klicu `close`.

Vpogled v segmente deljenega pomnilnika, semaforje in sporočilne vrste ter njihove karakteristike, daje ukaz `ipcs`. Na nek način je podoben ukazu `ls`. Komunikacijski objekt odstranimo z ukazom `ipcrm`. Ukaza sta koristna za administriranje sistema. Prikladna sta tudi med razvijanjem programske opreme, ko, denimo, proces zaide ter za sabo pusti nerabljene objekte. Na primer, z

```
ipcrm -s semid
```

odstranimo opuščen semafor z identifikatorjem `semid`, ali z

```
ipcrm -S semkey
```

odstranimo semafor s ključem `semkey`. Podobno vplivamo na deljen pomnilnik (-m ali -M), sporočilne vrste (-q ali -Q), ali na vse objekte (-a).

Sedaj pa se posvetimo semaforjem.

15.2 Sistem semaforjev

Sistemi UNIX/Linux nudijo procesom na nivoju sistemskega jedra množice semaforjev (Angl. Semaphore Set). Za razliko od deljenega pomnilnika ali sistema sporočil semaforji ne prenašajo podatkov, temveč skrbijo za usklajevanje pri prenosu podatkov. Na primer, predno proces piše v pomnilnik, ga s semaforjem 'zaklene'. Ko je zapis končan, proces s semaforjem 'odklene' pomnilnik.

Semaforji so lahko števni ali binarni. Operacije z UNIX System V semaforji so splošnejše kot to zahteva formalna definicija semaforja, a je dodatna funkcionalnost včasih koristna. Vrednost semaforja se sme namreč z eno samo operacijo zvečati ali zmanjšati za več kot ena. Dodatna dobra lastnost množice semaforjev je tudi ta, da se operacije na semaforjih znotraj množice izvedejo atomično. Na primer, enega od semaforjev povečamo, drugega zmanjšamo, vse v enem samem, časovno nedeljivem kosu. Sistemski klici

za delo s semaforji so: `semget, semctl in semop`.

15.2.1 Funkcija `semget`

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

Vrne identifikator množice semaforjev ali -1 v primeru napake.

Funkcija `semget` ustvari množico semaforjev. Za množico semaforjev skrbi sistemsko jedro. Klic vrne identifikator množice, prek katerega proces oziroma vsi v komunikacijo udeleženi procesi dostopajo do množice. Argument `nsem` pomeni število semaforjev v množici. Vsak semafor znotraj množice je določen z zaporedno številko: $0, 1, 2$ in tako dalje, do `nsem`-1. Argument `semflg` združuje določila – bite dovoljenj, ki imajo enak pomen kot pri datotekah. Argument `key` je:

```
IPC_PRIVATE za komunikacijo med procesi v sorodstvu,
ftok()      za komunikacijo med nesorodstvenimi procesi.
```

15.2.2 Funkcija `semctl`

Funkcija `semctl` služi za upravljanje množice semaforjev `semid`. Pozor, funkcija ni namenjena operacijam nad semaforjem kot sta 'čakaj' in 'javi'. Funkcija omogoča nastavljanje začetnih vrednosti semaforjev, preverjanje vrednosti semaforjev, in podobno. Klic funkcije uporabimo tudi zato, da se množica semaforjev odstrani, ko je ne potrebujemo več.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ... /* arg */ );
```

Klic funkcije v primeru napake vrne -1. Prvi argument `semid` je identifikator množice. Naslednji argument `semnum` izbere semafor ali vse semaforje v

množici in argument `cmd` predpiše dejanje z izbranim semaforjem ali z vsemi semaforji. Nekatere možnosti za argument `cmd` so:

```
GETVAL          vrne vrednost semaforja
SETVAL          postavi vrednost semaforja na arg
GETALL          v arg vrne vrednosti vseh semaforjev v mnozici (atomicno)
SETALL          postavi vrednost vseh semaforjev v mnozici na arg (atomicno)
```

Funkcija sprejme tri ali štiri argumente. Število argumentov je odvisno od vrednosti argumenta `cmd`. Naslednji primer ustvari množico dveh semaforjev za komunikacijo med procesi v sorodstvu (klic `semget` z `IPC_PRIVATE`), postavi prvi semafor na nič in drugi semafor na 1 (klic `semctl`). Nato atomično preveri vrednosti obeh semaforjev (ukaz `GETALL`) in vrednosti vrne v polju `semArray`. Na koncu odstrani množico semaforjev s klicem `semctl` in z ukazom `IPC_RMID`.

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEM_1    0    /* definiramo imena - zgolj zaradi preglednosti */
#define SEM_2    1

int main( void )
{
  unsigned short int semArray[2];
  int semId;

  if( (semId = semget( IPC_PRIVATE, 2, 0644 )) == -1) exit(1);/* Napaka */
  if( semctl( semId, SEM_1, SETVAL, 0 ) == -1)  exit(2);      /* Napaka */
  if( semctl( semId, SEM_2, SETVAL, 1 ) == -1)  exit(3);      /* Napaka */
  if( semctl( semId, 0, GETALL, semArray) == -1)exit(4);      /* Napaka */
  if( semctl( semId, 0, IPC_RMID, 0) == -1) exit(5);          /* Napaka */
  exit(0);
}
```

15.2.3 Funkcija `semop`

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

```
int semop( int semid, struct sembuf *sops, unsigned int nsops );
```

Vrne 0, če je operacija pravilna, -1 v primeru napake.

Funkcija `semop` atomično realizira operacije (čakaj, javi) na enem ali več semaforjih v množici `semid`. Operacije podamo z argumentom `sops`, ki kaže na polje struktur tipa `sembuf`. Vsaka struktura se nanaša na eno operacijo. Argument `nsops` pomeni število struktur v polju in torej tudi število operacij. Struktura `sembuf` je definirana takole:

```
struct sembuf{
        ushort    sem_num;      /* stevilka semaforja     */
        short     sem_op        /* operacija na semaforju */
        short     sem_flg;      /* dolocila               */
}
```

Komponenta `sem_num` določa številko semaforja v množici, `sem_op` je vrednost, ki jo želimo prišteti vrednosti tega semafora. Pozitivna vrednost `sem_op` (na primer 1) ustreza operaciji *javi* – semafor 'odklenemo'. Negativna vrednost `sem_op` (na primer -1) ustreza operaciji *čakaj* – semafor 'zaklenemo'. Vrednost argumenta ni omejena na vrednosti ± 1.

Naslednji kos programa izvede operacijo *čakaj*.

```
...
sops[0].sem_num = SEM_1;  /* izberemo semafor SEM_1    */
sops[0].sem_op  = -1;     /* operacija "cakaj"         */
sosp[0].sem_flg =  0;
semop( semid, sops, 1 );
...
```

Proces poskusi semaforju `SEM_1` iz množice `semid` prišteti -1. V primeru, da je vrednost semaforja pred tem večja od 1, se semaforju -1 enostavno prišteje oziroma ker je vrednost negativna, se semafor zmanjša za ena, proces pa nadaljuje z izvajanjem. V nasprotnem primeru, ko je vrednost semaforja manjša od ena, bo proces s klicem `semop` sam sebe postavil v stanje ustavljen. Vrednost semaforja *se nikoli* ne zmanjša pod nič. Proces bo na semaforju čakal,

- dokler drugi proces ne dvigne vrednosti semaforja (to je regularen način, ki pomeni operacijo *javi*), ali

- dokler drugi proces ne odstrani semaforja z `SEM_RMID` (saj v nasprotnem primeru preti zastoj), ali
- dokler proces ne sprejme signal.

Razumljivo, tisti proces, ki prvi zmanjša vrednost semaforja na nič, zahteva od drugih procesov, da na semaforju čakajo, medtem ko on napreduje.

Operacijo *javi* realizira naslednji kos programa:

```
...
sops[0].sem_num = SEM_1;  /* izberemo semafor SEM_1    */
sops[0].sem_op  =  1;     /* operacija "javi"          */
sosp[0].sem_flg =  0;
semop( semid, sops, 1 );
...
```

Proces preprosto poveča vrednost semaforja za ena in nadaljuje z izvajanjem. S tem dovoli drugim procesom, da napredujejo, če so čakali na semaforju. Vsak proces izmed procesov, ki so morda čakali na tem semaforju, kar pomeni, da je bila prejšnja vrednost semaforja enaka nič, dobi sedaj možnost za napredovanje. Jedro jih postavi v stanje *pripravljen*.

Z argumentom `sem_flg` strukture tipa `sembuf` je moč spremeniti osnovni pomen operacij na semaforju, kar pride v praksi redko kdaj v poštev.

Naslednja dva programa oziroma procesa s semaforji sinhronizirata komunikacijo prek skupne datoteke. V datoteko eden od procesov piše in drugi proces iz nje bere. Ko eden od njiju dostopa do datoteke, prepreči dostop drugemu. Dostop do datoteke je v tem primeru izmeničen in to rešimo z dvema semaforjema. Pisalni proces prepreči napredovanje bralnemu procesu dokler piše. Šele zatem, ko zapiše vso vsebino, dovoli napredovanje bralnemu procesu. Če bi bralni proces zahteval branje datoteke še predno so podatki pripravljeni, bi moral počakati. V nasprotnem slučaju nemoteno napreduje brez čakanja. Potem, ko je pisalni proces pripravil podatke in jih je bralni proces tudi prebral, bralni proces ponovno dovoli napredovanje pisalnemu procesu.

Pisalni proces ustvari množico dveh semaforjev in semaforja inicializira. Vrednosti semaforjev postavi tako, da z enim dovoli pisanje sebi in z dru-

gim prepreči branje bralnemu procesu. Pisalni proces sprejme znakovni niz, ki ga vnesemo prek zaslonskega okna. Natipkani niz znakov nato zapiše v datoteko, ki jo vedno ustvari 'na novo' (`O_CREAT | O_TRUNC`) ter odpre za pisanje. Bralni proces preprosto čaka na dovoljenje za dostop do datoteke. Ko dovoljenje dobi, prebere vsebino datoteke ter ponovno dovoli pisanje pisalnemu procesu. Pisanje in branje se izmenjujeta, dokler pisalnemu procesu ne vnesemo praznega niza.

Od procesa, ki piše, se pričakuje, da se bo začel izvrševati prvi, zato je primerno, da množico dveh semaforjev ustvari ta proces in takoj zatem semaforje inicializira – postavi začetne vrednosti semaforjev za branje in pisanje. Ker bo semaforje zadnji uporabljal bralni proces, je smiselno, da semaforje ta proces tudi odstrani. Drugi scenariji potencialno peljejo v zastoj ali nepričakovan zaključek procesov.

Ker procesa nista v sorodstvu, potrebujemo množico semaforjev z identifikacijsko številko, ki ima skupno poreklo. Možnosti za to je več. Uveljavljeno rešitev daje funkcija `ftok`. Druga, manj splošna rešitev je ta, da za 'ključ' izberemo neko število, ki ga uporabljajo vsi procesi, ki uporabljajo isto množico semaforjev. Pri tem tvegamo, a ne prav veliko, da bo enako število po naključju izbrano za drugo rabo. Tretja, zasilna rešitev, je uporaba semaforjev privatnega značaja (`IPC_PRIVATE`) in izmenjava identifikatorja množice semaforjev na primer prek skupne datoteke.

V prikazanem primeru smo se poslužili preprostejše možnosti in si za ključ izmislili število 1234 (`SEM_KEY_NO = 1234`). Uporaba funkcije `ftok` je zgolj nakazana z definicijo makro in jo v prikazanem primeru preprosto izberemo tako, da vrednost prvega argumenta funkcije `semget` nadomestimo s `SEM_KEY`.

Za preizkus procesov je najbolje, da odpremo dve terminalski okni ter vsak proces izvršimo v svojem oknu.

```
/* SemWrite.c demo semaforji
   Program pise v datoteko, od kjer drugi proces bere
   Operaciji pisi/beri se izmenjujeta
   Usklajevanje je reseno s semaforji

*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>

#define  DOLZINA  0x100          /* 256 bajtov maks. dolzina vrstice*/
#define  SEM_KEY  ((key_t)ftok("./SemKey",'k'))      /* File to Key */
#define  SEM_KEY_NO (key_t)1234
#define  SEM_READ  0
#define  SEM_WRITE 1

int main( int argc, char **argv )
{
  int   fd, SemId;
  char  Vrstica[DOLZINA];
  char  *Status;
  unsigned short SemArray[2];
  struct sembuf Semaphore;

  if( argc != 2){
    printf("Uporaba: %s <ime datoteke>\n",argv[0]); exit(1);
  }

  if( (SemId = semget(SEM_KEY_NO, 2, IPC_CREAT | 0644)) == -1){
    printf("%s: napaka semget\n", argv[0]);
    if( semctl( SemId, 0, IPC_RMID, 0 ) == -1)
      printf("%s: napaka, semctl IPC_RMID ni uspel\n", argv[0]);
    exit(1);
  }

  SemArray[SEM_READ] = 0;
  SemArray[SEM_WRITE]= 1;

  if( semctl( SemId, 0, SETALL, SemArray) == -1){
     printf("%s: napaka semctl - inicializacija ni uspela\n", argv[0]); exit(1);
```

```
  }

  do{
// Semafor wait -  write lock
    Semaphore.sem_num = SEM_WRITE;
    Semaphore.sem_op  = -1;
    Semaphore.sem_flg =  0;
    semop(SemId, &Semaphore, 1);

    if( (fd = open(argv[1], O_WRONLY | O_TRUNC | O_CREAT, 0644)) == -1){
      printf("%s: napaka create na %s\n", argv[0],argv[1]); exit(1);
    }
    printf("Vnesi besedilo: ");
    Status = fgets(Vrstica, DOLZINA, stdin);
    if( Status != NULL){
      if( write(fd, Vrstica, strlen(Vrstica)) != strlen(Vrstica) ){
        printf("%s: napaka write na %s\n", argv[0],argv[1]); exit(1);
      }
    }
    if( close(fd) == -1){
      printf("%s: napaka close na %s\n", argv[0],argv[1]); exit(1);
    }

// Semafor signal - unlock read
    Semaphore.sem_num = SEM_READ;
    Semaphore.sem_op  = 1;
    Semaphore.sem_flg = 0;
    semop(SemId, &Semaphore, 1);
  } while (Status != NULL);

  exit( 0 );
}


/* SemRead.c demo semaforji
   Program bere iz datoteke, v katero drugi proces pise
   Operaciji beri/pisi sta izmenicni
   Usklajevanje je reseno s semaforji

*/
#include <stdio.h>
#include <stdlib.h>
```

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>

#define  DOLZINA  0x100          /* 256 bajtov maks. dolzina vrstice*/
#define  SEM_KEY  ((key_t)ftok("./SemKey",'k'))      /* File to Key */
#define  SEM_KEY_NO (key_t)1234
#define  SEM_READ  0
#define  SEM_WRITE 1

int main( int argc, char **argv )
{
  int   n, fd, SemId;
  char  Vrstica[DOLZINA];

  struct sembuf Semaphore;

  if( argc != 2){
    printf("Uporaba: %s <ime datoteke>\n",argv[0]); exit(1);
  }

  if( (SemId = semget(SEM_KEY_NO, 2, IPC_CREAT | 0644)) == -1){
    printf("%s: napaka semget\n", argv[0]); exit(1);
  }

  do{
// Semafor wait -  read lock
    Semaphore.sem_num = SEM_READ;
    Semaphore.sem_op  = -1;
    Semaphore.sem_flg =  0;
    semop(SemId, &Semaphore, 1);

    if( (fd = open(argv[1], O_RDONLY)) == -1){
      printf("%s: napaka open na %s\n", argv[0],argv[1]); exit(1);
    }
    if( (n = read(fd, Vrstica, DOLZINA)) > 0){
      Vrstica[n] = 0;
```

```
      fputs(Vrstica, stdout);
    }
    close( fd );

// Semafor signal - unlock
    Semaphore.sem_num = SEM_WRITE;
    Semaphore.sem_op  = 1;
    Semaphore.sem_flg = 0;
    semop(SemId, &Semaphore, 1);
  }while ( n > 0);

  if( semctl( SemId, 0, IPC_RMID, 0 ) == -1){
    printf("%s: napaka, semctl IPC_RMID ni uspel\n", argv[0]);
  }
  exit( 0 );
}
```

15.3 Deljen (skupen) pomnilnik

Deljen pomnilnik je najhitrejša oblika medprocesne komunikacije. Sistemsko jedro na zahtevo aplikacije s funkcijo `shmget` pripravi del fizičnega pomnilnika za skupno rabo. V navadi je, da rečemo temu delu pomnilnika segment, a pozor, v tej pomenski zvezi nima čisto nič skupnega s segmeniranim navideznim pomnilnikom.

Pomnilniški segment z deljenim dostopom je enoznačno določen z identifikatorjem. Identifikator je unikaten. Vsak proces, ki pozna indetifikator in mu je dovoljen dostop, bi lahko dostopal do izbranega segmenta. Proces, ki potrebuje dostop do deljenega pomnilnika, si ga pripne s klicem funkcije `shmat`. Oziroma, sistemsko jedro 'pripne' pomnilniški segment v logočni naslovni prostor procesa. Enako postopajo vsi procesi, ki bi hoteli med seboj komunicirati prek deljenega pomnilnika. V sistemu Linux je za deljen pomnilnik v logičnem naslovnem prostoru procesa predviden naslovni prostor pod skladom in nad prostorom za kup, ki je rezerviran za dinamično dodeljevanje pomnilnika. Logični naslovi istega dela fizičnega pomnilnika, ki omogoča deljeni dostop, so seveda v različnih procesih praviloma različni.

Potem, ko je pomnilniški segment pripravljen za skupno rabo, teče komunikacija *brez* posredovanja sistemskega jedra. Eden ali več procesov v pomnilnik piše in drugi procesi iz njega berejo. Za to ni potrebno nobenih sistemskih funkcij, nobenih sistemskih klicev, nobenih prehodov med jedrom in aplikacijo. Če pa je potrebno usklajevanje pri dostopu do pomnilnika, morajo za usklajevanje poskrbeti procesi sami.

Ko proces več ne potrebuje dostopa do pomnilnika, ga 'odpne' iz svojega naslovnega področja. S tem izgubi dotop do pomnilnika, a pomnilniški segment še obstaja. Ko to naredi še zadnji proces, se segment 'uniči' (`IPC_RMID`) in s tem sprosti fizični pomnilnik.

Osnovne sistemske funkcije oziroma klici, ki se nanašajo na deljen pomnilnik, so: `shmget, shmctl, shmat` in `shmdt`. Z `shmget` zahtevamo od sistemskega jedra naj procesu dodeli pomnilnik za deljen dostop, z `shmctl` ga upravljamo, z `shmat` procesu 'pripnemo' in z `shmdt` 'odpnemo' deljen pomnilnik.

15.3.1 Funkcija `shmget`

```
#include <sys/shm.h>

int shmget( key_t key, size_t size, int shmflg );
```

Vrne identifikator dodeljenega pomnilnika ali -1 v primeru napake.

S funkcijo `shmget` proces od sistemskega jedra zahteva naj mu preskrbi pomnilnik potrebne velikosti, do katerega bo možen deljen dostop. Jedro dodeljuje pomnilnik v kosih, ki so po velikosti enaki mnogokratniku velikosti strani. Na primer, za velikost strani 4K in `size = 5000` bajtov, bi klic vrnil segment velikosti 8K (dve strani).

Če pomnilniški segment za dani `key` še ne obstaja, ga ustvari in vrne nov identifikator. Če pa pomnilniški segment za dani `key` že obstaja, vrne identifikator obstoječega segmenta. V tem primeru argument `size` nima vpliva na velikost segmenta, a ne sme biti večji od velikosti obstoječega segmenta. Zadnji argument `shmflg` določa pravice pri dostopu do segmenta za lastnika, skupino ter ostale. Na primer, s klicem

```
shmId = shmget( IPC_PRIVATE, 4096, 0600 );
```

proces zahteva $4k$ pomnilnika, do katerega lahko dostopajo (berejo/pišejo) procesi v sorodstvu (ključ `IPC_PRIVATE`) z enakim lastnikom (konstanta 0600). Dodatna določila pomnilniškega segmenta so zabeležena v sistemski strukturi segmenta, na katero se da vplivati s klicem `shmctl`. Klic `shmget` vrne identifikacijsko številko `shmId` preko katere je pomnilnik potem 'znan' in ga je moč pripeti procesu s funkcijo `shmat`.

V primeru, da potrebujemo nov deljen pomnilnik med sorodstveno nevezanimi procesi, pa:

```
shmId = shmget( ftok(".','k'), 4096, IPC_CREATE | 0600 );
```

V primeru, da segment še ne obstaja, bo ustvarjen. Če segment že obstaja, klic vrne identifikator obstoječega segmenta. Ako si slednjega ne želimo, dodamo k zadnjemu argumentu še `IPC_EXCL`,

```
shmId = shmget( ftok(".','k'), 4096, IPC_CREATE | IPC_EXCL | 0600 );
```

Tedaj klic vrne napako in postavi spremenljivko `errno` na `EEXIST`.

V zadnjih primerih smo za prvi argument funkcije `ftok` izbrali datotečno ime 'pika'. Pika je (nadomestno) ime tekočega direktorija. To ime v direktoriju vedno obstaja. Zato pa morajo imeti vsi procesi, ki si hočejo deliti pomnilnik, isti tekoči direktorij.

15.3.2 Funkcija `shmctl`

```
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Funkcija `shmctl` realizira nadzorno operacijo `cmd` nad pomnilniškim segmentom z identifikatorjem `shmid`. Dovoljene so naslednje operacije:

- `IPC_STAT` v strukturi na katero kaže `buf` vrne vrednosti podatkovne strukture sistemskega jedra, ke je pridružena segmentu z identifikacijo `shmid`.
- `IPC_SET` definira vrednosti naslednjih komponent sistemske strukture

pridružene segmentu `shmid`. Vrednosti podamo v strukturi na katero kaže `buf`:

```
shm_perm.uid     /* ID uporabnika          */
shm_perm.gid     /* ID skupine             */
shm_perm.mode    /* samo spodnjih 9 bitov  */
```

- `IPC_RMID` odstrani dodeljeni pomnilniški segment. Segment se dokončno uniči, ko ga sprosti (`shmdt`) zadnji proces.
- `SHM_LOCK SHM_UNLOCK` zaklene/odklene segment, ki potem ni/je izpostavljen sistemu upravljanja navideznega pomnilnika. Operacija je dovoljena samo priviligiranemu uporabniku. Zaklenjen segment se ne umakne na disk.

Najpogostejši klic funkcije, s katerim odstranimo deljeni pomnilniški segment, izgleda takole:

```
if( shmctl( shmId, IPC_RMID, 0 ) == -1){
  perror("shmctl IPC_RMID err");
}
```

Klic funckije se vrne takoj, a delovanje zahteve je lahko zadržano. V primeru, ko vsi procesi odpnejo pomnilniški segment iz svojega naslovnega področja, je učinek klica takojšen. V nasprotnem primeru se zahteva odloži, dokler še zadnji proces ne odpne segmenta s klicem `shmdt`.

15.3.3 Shmop: funkciji `shmat` in `shmdt`

```
#include <sys/shm.h>

char *shmat( int shmid, void *shmaddr, int shmflg );

int   shmdt( void *shmaddr );
```

Funkcija `shmat` vrne kazalec na segment ali -1 v primeru napake.
Funkcija `shmdt` vrne nič ali -1 v primeru napake.

S funkcijo `shmat` si proces 'pripne' pomnilniški segment `shmid` v svoje na-

slovno področje in vrne logični naslov segmenta, ki je odvisen od argumenta `shmaddr` takole:

- če je `shmaddr` nič, določi naslov segmenta jedro. To je priporočen način.
- če je `shmaddr` različen od nič in `shmflg != SHM_RND` je segment 'pripet' na naslov `shmaddr`.
- če je `shmaddr` različen od nič in `shmflg = SHM_RND` je segment 'pripet' na naslov `shmaddr` zaokrožen navzdol na mnogokratnik števila `SHMLBA`; ta je odvisen od sistema.

Če je `shmflg = SHM_RDONLY` je pomnilniški segment dostopen samo za branje, sicer je dostopen za branje in pisanje.

Ko pomnilniški segment ni več potreben, ga sprostimo s `shmdt` (ang. Detach), segment pa še obstaja, dokler se ga ne uniči s klicem `shmctl` z operacijo `IPC_RMID`.

Naslednja dva programa realizirata enostavno komunikacijo – brez sinhronizacije – preko deljenega pomnilnika. Za sinhronizacijo bi lahko uporabili semafor ali več semaforjev. Procesa nista v sorodstvu, zato je prvi argument funkcije `shmget` kar klic funkcije `ftok()`. Predvideno je, da se začne prvi izvrševati `ShmWrite`. Proces vsako sekundo v skupni pomnilnik vpiše tekoči čas in datum. Proces `ShmRead` enostavo bere iz pomnilnika, dokler `ShmWrite` ne vpiše prazen niz.

```
/* ShmWrite.c demo */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <time.h>
#include <string.h>
```

```
#define  VELIKOST  0x40   /* 64 bajtov */
#define  MINUTA    60

#define  SHM_KEY  (key_t)ftok(".",'k')

int main( int argc, char **argv )
{
  int    shmId;
  char   *shmWrite;
  time_t cas, casEnd;

  if( (shmId = shmget( SHM_KEY, VELIKOST, IPC_CREAT | 0600 )) == -1 ){
    perror("shmget err"); exit(1);
  }

  if( (shmWrite = shmat( shmId, NULL, 0 )) == (void *) -1 ){
    perror("shmat err"); exit(2);
  }
  printf("%s: shmaddr od %p do %p\n", argv[0], &shmWrite[0], &shmWrite[VELIKOST]);

  casEnd   = time( NULL ) + MINUTA;

  do{
    cas = time( NULL );
    strcpy(shmWrite, ctime( &cas));   // spremenimo cas v znakovni niz
    printf("%s: %s", argv[0], shmWrite);
    usleep(1000000);
  } while (cas < casEnd );

  shmWrite[0] = 0; /* prazen niz */

  if( shmctl( shmId, IPC_RMID, 0 ) == -1){
    perror("shmctl IPC_RMID err");
  }
  exit( 0 );
}

/* ShmRead.c demo */

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

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

#define  VELIKOST  0x40   /* 64 bajtov */

#define  SHM_KEY  (key_t)ftok(".",'k')


int main( int argc, char **argv )
{
  int    shmId;
  char  *shmRead;

  if( (shmId = shmget( SHM_KEY, VELIKOST, IPC_CREAT | 0600 )) == -1 ){
    perror("shmget err"); exit(1);
  }

  if( (shmRead = shmat( shmId, NULL, 0 )) == (void *) -1 ){
    perror("shmat err"); exit(2);
  }

  printf("%s: shmaddr od %p do %p\n",argv[0], &shmRead[0], &shmRead[VELIKOST]);

  do{
    usleep( 600000 );
    printf("%s: %s", argv[0], shmRead);
  } while ( strlen(shmRead) > 0 ); /* prazen niz */

  if( shmctl( shmId, IPC_RMID, 0 ) == -1){
    perror("shmctl SHM_RMID err");
  }
  exit( 0 );
}
```

15.4 Sistem sporočil

Sistem sporočil omogoča komunikacijo med procesi preko 'sporočilnih vrst' (Angl. message queues). Dostop do sporočilne vrste si proces pridobi s klicem funkcije `msgget`. Sporočilna vrsta je povezan seznam sporočil. S funkcijo `msgsnd` proces *odda* sporočilo v sporočilno vrsto in z `msgrcv` nek drug proces sporočilo iz vrste *sprejme*. Sistem sporočil ohranja strukturo sporočil – ko konča (je oddano) eno sporočilo, začne drugo sporočilo. Sporočilo ima svoj začetek in konec. Dolžina sporočil je poljubna, navzgor omejena samo s konfiguracijo sistema. Proces, ki sprejema, lahko z eno operacijo sprejme samo eno sporočilo. Ni moč sprejeti pol sporočila ali več sporočil hkrati.

Proces, ki skuša sprejeti sporočilo iz 'prazne' vrste gre (običajno) v stanje ustavljen, dokler kakšen proces v vrsto ne odda sporočila. Vrsto upravljamo in odstranimo s funkcijo `msgctl`.

Razen vsebine sporočila uporabnik določi tudi njegov tip. *Tip* sporočila omogoča sprejememu procesu 'selektivno' obravnavanje sporočil. S primerno izbiro tipov sporočil se da izdelati prioritetni sistem sporočil (sporočila višje/nižje prioritete) ali multipleksirati več komunikacijskih kanalov na isto sporočilno vrsto.

Uporabo/delovanje sistema sporočil bomo opisali precej površno.

15.4.1 Funkcija `msgget`

```
#include <sys/msg.h>

int msgget( key_t key, int msgflg );
```

Vrne nenegativno razpoznavno številko sporočilne vrste (deskriptor) ali -1 v primeru napake.

Klic funkcije ustvari dostop do sporočilne vrste in vrne njen deskriptor. Vrsti je v sistemskem jedru pridružena struktura, preko katere lahko uporabnik določa dodatne lastnosti vrste. Teh ne bomo obravnavali; zato nas

tudi komponente omenjene strukture ne bodo zanimale. Če je `key` enak `IPC_PRIVATE` je vrsta predvidena za komunikacijo med procesi v sorodstvu. Za komunikacijo med poljubnimi procesi določimo `key` s funkcijo `ftok`.

15.4.2 Funkcija `msgctl`

```
#include <sys/msg.h>

int msgctl( int msqid, int cmd, struct msqid_ds *buf );
```

Vrne 0 v primeru da ni napake, -1 v primeru napake.

Funkcija `msgctl` omogoča upravljanje vrste sporočil. Ukaze določamo z argumentom `cmd`. Na primer, za `cmd = IPC_RMID` vrsto odstanimo, z `IPC_STAT` dobimo vrednosti komponent sistemske strukture, ki je prirejena vrsti, z `IPC_SET` definiramo vrednost nekaterih komponent te strukture.

Msgop: funkciji `msgsnd` in `msgrcv`

S tema funkcijama pošiljamo in sprejemamo sporočila.

```
#include <sys/msg.h>

int msgsnd( int msqid, const void *msgp, size_t msgsz, int msgflg );

int msgrcv( int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg );
```

Funkcija `msgsnd` vrne 0 v primeru uspeha, -1 v primeru napake.
Funkcija `msgrcv` vrne dolžino sprejetega sporočila, -1 v primeru napake.

S funkcijo `msgsnd` pošljemo sporočilo v sporočilno vrsto `msqid`. Argument `msgp` kaže na sporočilo, ki ga uporabnik poda v tej obliki:

```
long  mtype;     /* tip sporocila - kot definira uporabnik */
char  mtext[];   /* sporocilo poljubnega tipa              */
```

`mtype` je (majhno) pozitivno celo število, po katerem sprejemni proces razlikuje sporočila. `mtext` je zaporedje podatkov poljubnega tipa dolgo `msgsz` bajtov in je lahko tudi dolžine 0. Argument `msgflg` določa dodatna sporočila, ki nas ne zanimajo.

Funkcija `msgrcs` sprejme sporočilo iz vrste `msgid`. Sporočilo se nahaja tam, kamor kaže kazalec `msgp`. `msgsz` je predvidevana dolžina sporočila, dejanska dolžina pa je lahko manjša. Argument `msgtyp` določa *tip* sporočila, ki ga hočemo sprejeti:

- `msgtyp = 0` Prvo sporočilo v vrsti
- `msgtyp > 0` Prvo sporočilo tipa `msgtyp`
- `msgtyp < 0` Prvo sporočilo tipa *tip* <=`msgtyp`.

Povedano drugače: proces lahko čaka na sporočilo z oznako `msgtyp`. Argument `msgflg` predpiše dopolnilna določila in nas ne zanima.

Naslednja dva programa komunicirata preko sporočilne vrste; prvi odda sporočilo in drugi sporočila sprejema – do prekinitve. Procesa nista v sorodstvu.

```
/* Msg Oddajnik */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/types.h>
#include <sys/msg.h>

#define MSG_TYPE    1       /* To je (izbran) tip mojih sporocil     */

struct MsgSend_t{
  long MsgType;             /* To je struktura mojih sporocil        */
  long SenderId;
  char Message[128];
};

int main( int argc, char **argv )
{
  int MsgId, MsgSendSize;
  struct MsgSend_t MsgSend;
```

```
  if( argc != 2 ){ /* Sporocilo - tekst - podamo v ukazni vrstici    */
    printf("Uporaba: %s sporocilo\n", argv[0] );
    exit( 1 );
  }
  if( (MsgId =  msgget( ftok("msgkey", 'k'), 0644 )) == -1 ){
    printf("%s: Napaka, msgget ni uspel\n", argv[0]);
    exit( 2 );
  }
  MsgSend.MsgType   = MSG_TYPE;       /* Naj bo tip sporocila tak    */
  MsgSend.SenderId  = getpid( );      /* To naj je moja oznaka - PID */
  strcpy( MsgSend.Message, argv[1] ); /* To je sporocilo             */

  MsgSendSize       = sizeof(MsgSend.SenderId) + strlen( argv[1] );

  if( msgsnd( MsgId, &MsgSend, MsgSendSize, 0 ) == -1 ){ /* Oddaja      */
    printf("%s: Napaka, msgsnd ni uspel\n", argv[0]);
    exit( 2 );
  }
  exit( 0 );
}
```

Sprejmni proces v neskončni zanki čaka na sporočilo.

```
/* Msg Sprejemnik */

#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/types.h>
#include <sys/msg.h>

#define MSG_TYPE  1    /* Sprejemam samo sporocila tega tipa */

struct MsgReceive_t{   /* Pricakovana stuktura sporocil      */
  long MsgType;
  long SenderId;
  char Message[128];
};
```

```
int main( int argc, char **argv )
{
  int MsgId, MsgReceiveSize, Sprejel;
  struct MsgReceive_t MsgReceive;

  if( (MsgId =  msgget( ftok("msgkey", 'k'), IPC_CREAT | 0644 )) == -1 ){
    printf("%s: Napaka, msgget ni uspel\n", argv[0]);
    exit( 1 );
  }
  MsgReceiveSize       = sizeof(MsgReceive) - sizeof( long );

  for( ; ; ){
    if((Sprejel=msgrcv(MsgId, &MsgReceive, MsgReceiveSize, MSG_TYPE, 0)) == -1){
      printf("%s: Napaka, msgrcv ni uspel\n", argv[0]);
      exit( 2 );
    }
    MsgReceive.Message[Sprejel- sizeof(long)] = 0;
    printf("Tip sporocila: %4d \n", MsgReceive.MsgType );
    printf("Id Oddajnika : %4d \n", MsgReceive.SenderId );
    printf("Sporocilo: %s\n", MsgReceive.Message );
  }
  msgctl( MsgId, IPC_RMID, 0 ); /* to se nikoli ne izvrsi */
  exit( 0 );
}
```

Poglavje 16

Sinhronizacija niti

Niti napredujejo sočasno in asinhrono, povečini neodvisno in vsaka s svojo hitrostjo. Niti si delijo skupen naslovni prostor istega procesa. Kadar dve ali več niti hkrati dostopa do istega dela pomnilnika, to je do spremenljivk, seznamov, tabel, ali drugačnih podatkovnih struktur, je dostop načeloma potrebno časovno uskladiti oziroma sinhronizirati. Na primer, ko ena nit spreminja pomnilniško besedo, morajo druge niti, ki bi želele dostopati do te pomnilniške besede, počakati.

Za sinhronizacijo niti obstajajo trije sorodni mehanizmi in tri skupine funkcij: `pthread_mutex`, `pthread_rwlock` in `pthread_cond`.

16.1 Funkcije `pthread_mutex`

Sočasen dostop do sredstev, ki ne dovoljujejo sočasnega dostopa, preprečimo s funkcijami `pthread_mutex` (skovanka mutex pride od 'Mutual Exclusion'). Spomnimo se, da smo dele procesov, ki posegajo po sredstvih, ki ne dopuščajo sočasnega dostopa, imenovali kritično področje. Te funkcije torej poskrbijo, da se izvrševanje niti v kritičnem področju med seboj časovno izključuje. Prototipi funkcij so:

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

```
int pthread_mutex_init( pthread_mutex_t *restrict mtx,
                        const pthread_mutexattr_t *restrict attr );

int pthread_mutex_lock( pthread_mutex_t *mtx );

int pthread_mutex_trylock( pthread_mutex_t *mtx );

int pthread_mutex_lock( pthread_mutex_t *mtx );

int pthread_mutex_unlock( pthread_mutex_t *mtx );

int pthread_mutex_destroy( pthread_mutex_t *mtx );
```

Kadar ni napake, vse funkcije vračajo nič. S funkcijami `pthread_mutex` se da zagotoviti medsebojno izključevanje niti, ki dostopajo do skupnih sredstev, kot sta branje in pisanje pomnilniške besede, konceptualno enako kot s semaforji. S funkcijo `int pthread_mutex_init` pridobimo pomožno spremenljivko z imenom `mtx`, ali seveda s poljubno drugačnim imenom. Spremenljivka, ki ji večkrat rečemo kar 'mutex', je lahko v enem od dveh stanj, v stanju *zaklenjena* ali v stanju *odklenjena*. Ko je mutex inicializiran, je najprej v odklenjenem stanju. S funkcijo `pthread_mutex_lock` mutex zaklenemo. Če je bil mutex pred tem v odklenjenem stanju, bo napredovanje niti oziroma povratek iz funkcije `pthread_mutex_lock` takojšen, mutex pa bo šel v zaklenjeno stanje. Če pa je bil mutex ob klicu funkcije `pthread_mutex_lock` že zaklenjen, bo nit na mutex-u čakala, dokler ga s `pthread_mutex_unlock` ne odklenemo. Odklepati mutex, ki ni zaklenjen, ni primerno, kot ni primerno odklepati mutex-a, ki ga je pred tem zaklenila druga nit. Iz povedanega sledi, da so `mutex`-i sorodni semaforjem, a je njihova uporaba bolj specifična, saj so namenjeni za reševanje problema kritičnega področja.

Če si nit ne more privoščiti, da bi čakala na dostop do skupne spremenljivke, preveri stanje s klicem `pthread_mutex_trylock`. Nazadnje po potrebi spremenljivko odstranimo s `pthread_mutex_destroy`.

Osnovni koncept mutex-ov spremenimo z dodatnimi določili v argumentu `attr` k funkciji `pthread_mutex_init`. S temi modifikacijami postanejo mutex-i še bolj podobni semaforjem.

Naslednji program zgolj prikazuje klasičen način uporabe. Glavna nit ustvari dve niti, od katerih ena povečuje in druga zmanjšuje vrednost skupne spremenljivke `skupenX`. Nitni funkciji nimata oziroma ne uporabljata argumenta. Nitni funkciji sicer vračata referenco na konstanto, a se zanjo glavna nit ne zanima. Vsaka od niti v okviru nitne funkcije zgolj spreminja in izpisuje vednost spremenljivke `skupenX`, ki jo pred tem in za tem izpiše tudi glavna nit. Vrednost `skupenX` se v eni od niti poveča za ena in v drugi niti zmanjša za ena, tako da je na koncu enaka kot na začetku.

Večkratna ponovna izvršitev programa verjetno ne bo vedno povsem enaka. Mogoče je, da se bo včasih prej izvršila druga nit in posledično tt skupenX najprej zmanjšal. A na koncu bo vrednost spremenljivke `skupenX` vedno enaka.

```
/* Test mutex  */

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

int skupenX   = 10;
pthread_mutex_t mtx;

void *nFun1( void *narg ), *nFun2( void *narg );

int main( void )
{
  pthread_t nid1, nid2;

  printf("Nit 0, x= %d\n", skupenX);

  if( pthread_mutex_init( &mtx, NULL ) > 0 ){
    printf("Napaka mtx init"); exit( 1 );
  }
  if( pthread_create( &nid1, NULL, nFun1, NULL) > 0 ){
    printf("Napaka create 1"); exit( 2 );
  }
```

```
  if( pthread_create( &nid2, NULL, nFun2, NULL) > 0 ){
    printf("Napaka create 2"); exit( 2 );
  }
  if( pthread_join( nid1, NULL ) > 0 ){
    printf("Napaka join 1"); exit( 3 );
  }
  if( pthread_join( nid2, NULL ) > 0 ){
    printf("Napaka join 2"); exit( 3 );
  }
  printf("Nit 0, x= %d\n\n", skupenX);
  exit( 0 );
}


void *nFun1( void *arg )
{
  pthread_mutex_lock( &mtx );
  skupenX++;
  printf("Nit 1, x= %d\n", skupenX);
  pthread_mutex_unlock( &mtx );
  return (void *)1;
}

void *nFun2( void *arg )
{
  pthread_mutex_lock( &mtx );
  skupenX--;
  printf("Nit 2, x= %d\n", skupenX);
  pthread_mutex_unlock( &mtx );
  return (void *)2;
}
```

16.2 Funkcije pthread_rwlock

Dejstvo je, da je operacija branja manj kritična od pisanja. S stališča zagotavljanja celovitosti skupne spremenljivke, lahko dovolimo njeno sočasno branje večjemu številu niti ob pogoju, da nobena od niti med tem tudi ne

piše. Takšen scenarij dosežemo s funkcijami `pthread_rwlock` (Angl. Reader / Writer Lock). S temi funkcijami inicializiramo, zaklenemo za pisanje pred branjem, zaklenemo za branje in pisanje pred pisanjem, odklenemo in na koncu uničimo.

```
#include <pthread.h>

int pthread_rwlock_init( pthread_rwlock_t *restrict rwlock
                         const pthread_rwlockattr_t *restrict attr );

int pthread_rwlock_destroy( pthread_rwlock_t *restrict rwlock );

int pthread_rwlock_rdlock( pthread_rwlock_t *rwlock );

int pthread_rwlock_wrlock( pthread_rwlock_t *rwlock );

int pthread_rwlock_unlock( pthread_rwlock_t *rwlock );
```

Vse funkcije vrnejo nič, če ni napake.

Funkciji `pthread_rwlock_rdlock` in `pthread_rwlock_wrlock` si zaslužita dodatno pojasnilo. Klic funkcije `pthread_rwlock_rdlock( &rwlock )` nit blokira, če je bil pred tem `rwlock` zaklenjen zaradi pisanja s `pthread_rwlock_wrlock`, dokler ni odklenjen z `pthread_rwlock_unlock`. V obratnem primeru nit napreduje in `rwlock` zaklene za pisanje, medtem ko je sočasno branje še naprej dovoljeno. Ponovni klic `pthread_rwlock_rdlock` torej vedno uspe. Vsako zaklepanje za branje zahteva odklepanje. Torej, odklepanj mora biti ravno toliko, kolikor je bilo zaklepanj. Bralne niti potem izgledajo takole:

```
pthread_rwlock_rdlock( &rwlock );
// branje skupne spremenljivke
pthread_rwlock_unlock( &rwlock );
```

Pisalne niti so načeloma take:

```
pthread_rwlock_wrlock( &rwlock );
// spreminjanje skupne spremenljivke
pthread_rwlock_unlock( &rwlock );
```

Povedano na kratko, `pthread_rwlock_wrlock` blokira vse, `pthread_rwlock_rdlock` pa samo pisalce.

16.3 Pogojne spremenljivke in funkcije pthread_cond

Tretji in zadnji od obravnavanih pripomočkov za sinhronizacijo niti so *pogojne spremenljivke* (angl. Condition Variables). Le-te omogočajo 'zmenek' niti. Prek pogojne spremenljivke lahko ena nit signalizira drugim nitim, da se je vrednost dane skupne spremenljivke spremenila, ali v splošnem stanje skupnega sredstva, in po drugi strani omogoča nitim, da čakajo na to obvestilo.

Pogojne spremenljivke se uporabljajo v povezavi s spremenljivkami mutex. Mutex zagotavlja izključen dostop do skupne spremenljivke, medtem ko pogojna spremenljivka omogoča obveščanje o spremembi skupne spremenljivke.

Pogojno spremenljivko je moč inicializirati s `pthread_cond_init` ter kdaj kasneje, ko ni več potrebna, uničiti s `pthread_cond_destroy`. S funkcijama `pthread_cond_wait` in `pthread_cond_timedwait` je moč čakati do tedaj, ko je izpolnjen pogoj za napredovanje ali pa do tedaj, ko poteče skrajni rok čakanja. S funkcijo `pthread_cond_signal` se da signalizirati, da je pogoj za napredovanje ene niti izpolnjen in s funkcijo `pthread_cond_broadcast` je moč signalizirati, da lahko napredujejo vse na dotični pogoj vezane niti. Prototipi funkcij so:

```
#include <pthread.h>

int pthread_cond_init( pthread_cond_t *restrict cond,
                       const pthread_condattr_t *restrict attr );

int pthread_cond_signal( pthread_cond_t *cond );

int pthread_cond_broadcast( pthread_cond_t *cond );

int pthread_cond_wait( pthread_cond_t *restrict cond,
                       pthread_mutex_t *restrict mutex );

int pthread_cond_timedwait( pthread_cond_t *restrict cond,
                            pthread_mutex_t *restrict mutex,
                            const struct timespec *restrict abstime );
```

```
int pthread_cond_destroy( pthread_cond_t *cond );
```

Poglejmo shemo uporabe pogojnih spremenljivk. Denimo, da si niti delita skupno spremenljivko `skupenY`. Spodnja nit spreminja vrednost spremenljivke v kritičnem področju in zatem naznani spremembo takole:

```
pthread_mutex_lock( &mtx );
skupenY = skupenY + delta;
pthread_mutex_unlock( &mtx );
pthread_cond_signal( &cond );
```

Naslednja nit čaka na 'pravo' spremembo in deluje nad skupno spremenljivko v kritičnem področju takole:

```
pthread_mutex_lock( &mtx );
while( skupenY != praviY )
  pthread_cond_wait( &cond, &mtx );
skupenY = skupenY - delta;
pthread_mutex_unlock( &mtx );
```

Klic funkcije `pthread_cond_wait` deluje takole:

1. odkleni `mtx`,
2. postavi nit v stanje 'ustavljena',
3. blokiraj in čakaj na naznanilo na pogojni spremenljivki `cond`,
4. ko naznanilo pride, zakleni `mtx`.

Bistvo zgornjega kosa programa je v tem, da se prek pogojne spremenljivke `cond` izognemu 'aktivnemu' čakanju na spremenljivki `skupenY`. Aktivno čakanje obremenjuje procesor. Če pogoj za napredovanje v stavku `while` ni izpolnjen, klic `pthread_cond_wait` odklene `mtx` in s tem dovoli drugim nitim dostop do skupne spremenljivke. Nit gre v stanje pasivnega čakanja, dokler neka druga nit ne naznani spremembe in `mtx` se takoj ponovno zaklene. S tem je konzistenca spremenljivke `skupenY` zajamčena.

Poglavje 17

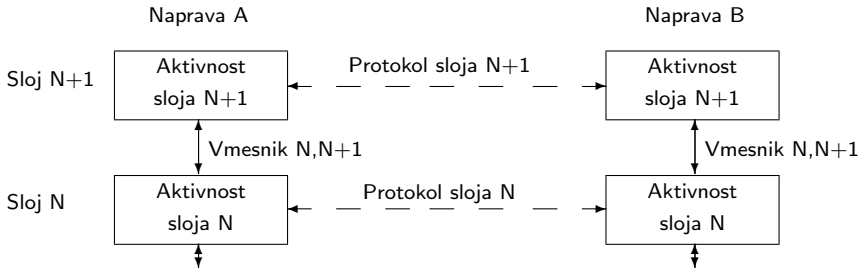
Krajevno porazdeljeni procesi

V predhodnih poglavjih smo spoznali osnove cevi, sporočil, signalov, semaforjev in deljenega pomnilnika. Ogledali smo si pripomočke za sinhromizacijo niti. Vsi našteti mehanizmi omogočajo komunikacije med procesi v okviru iste naprave. V nadaljevanju si bomo ogledali komunikacijske vtičnice (Angl. sockets), ki dajejo podlago komunikacijam med oddaljenimi procesi, to je med procesi, ki obstajajo na različnih krajevno porazdeljenih napravah. Še prej pa se bomo ozrli na nekatere osnovne lastnosti komunikacijskih omrežij.

17.1 Komunikacijska omrežja

Podlago za komunikacije med krajevno porazdeljenimi procesi dajejo povezovalne strukture in naprave, ki tvorijo komunikacijsko omrežje. Komunikacijska omrežja spadajo med kompleksne in izrazito obsežne sisteme. Zato so zasnovana modularno in hierarhično. Osrednji element v hierarhični zgradbi omrežja je sloj (Angl. Layer), poimenovan tudi plast. Sloj si lahko predstavljamo kot množico krajevno porazdeljenih modulov s sorodno funkcionalnostjo, ki se razprostirajo po napravah širom omrežja. V

okviru modula znotraj naprave se dogajajo aktivnosti, ki sodelujejo z oddaljenimi aktivnostmi sorodnih modulov drugih naprav. Zato med sabo komunicirajo. Koncept slojnosti omrežja ponazarja slika 17.1.



Slika 17.1: *Načelo slojnosti.*

Slika 17.1 prikazuje enega izmed slojev omrežja. Aktivnost sloja N na eni napravi komunicira s sorodno aktivnotjo sloja N na drugi napravi. Pravimo, da potekajo komunikacije v omrežju 'horizontalno' med istorodnimi aktivnostmi (Angl. Peer-To-Peer). Pravila in dogovore, ki jih pri tem upoštevata istorodni aktivnosti na obeh straneh na sloju N, imenujemo *komunikacijski protokol* sloja N.

Sloji oziroma dejavnosti slojev niso same sebi namen. Krajevno porazdeljene dejavnosti sloja N se dogajajo zato, da sloj N zagotavlja *komunikacijske storitve* neposredno višjemu sloju N+1. Med sosednima slojema je vmesnik. Vmesnik med slojem N in naslednjim višjim slojem N+1 določa storitve, ki jih sloj N daje sloju N+1, pa tudi operacije za izvedbo teh storitev.

Sloje najbolje opredeljujejo prav naloge, ki jih opravljajo, torej njihova funkcionalnost. Denimo, eden od slojev skrbi za dostop do prenosnih poti, drugi skrbi za zanesljiv prenos, tretji izbira primerne prenosne poti, četrti skrbi za pretok podatkov, in tako dalje.

Slojev je v splošnem več. Od slojev je eden najnižji in eden najvišji. Zgornji vmesnik najvišjega sloja je vmesnik komunikacijskega omrežja s končnim uporabnikom komunikacijskih storitev. Spodnji vmesnik najnižjega sloja

je vmesnik s prenosnim sredstvom.

Čeprav poteka komunikacija po protokolu v 'vodoravni' smeri, pa teče resnični pretok podatkov znotraj iste naprave v 'navpični' smeri. Višji sloj na vmesniku med slojema preda podatke nižjemu sloju, ta pa še nižjemu. Končno pridejo podatki do najnižjega sloja, nakar se po prenosnih poteh prenesejo na drugo stran. Na drugi strani gredo podatki spet navpično navzgor, od sloja do sloja, dokler ne pridejo do navišjega sloja, kjer se dostavijo končnemu uporabniku podatkov.

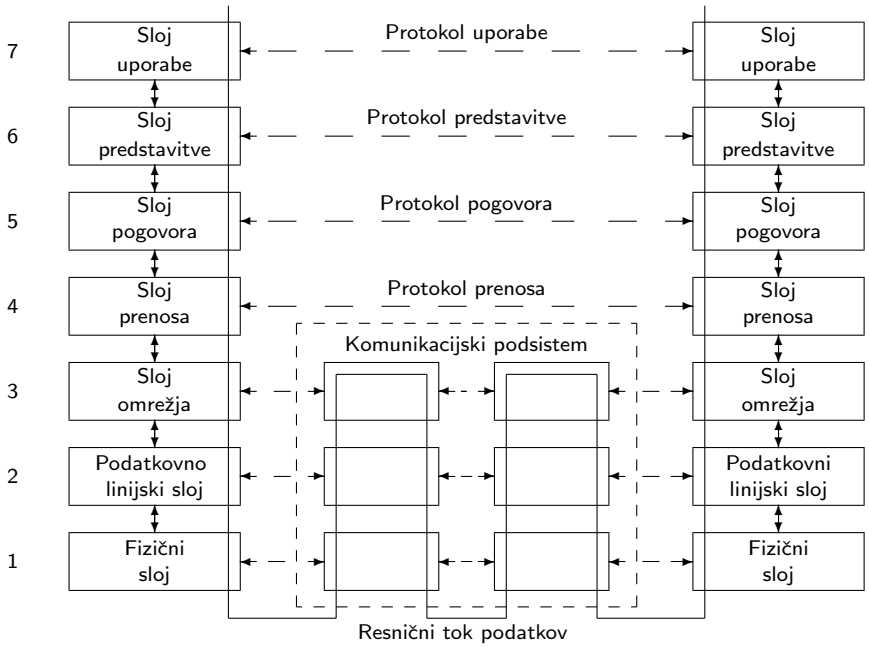
17.1.1 Referenčni model ISO OSI

Mednaroda organizacija za standardizacijo ISO je v začetku osemdesetih let preteklega stoletja predlagala model, po katerem naj bi gradili odprte informacijske sisteme.[1] Referenčni model OSI (Angl. Open Systems Interconnection), kot pove njegovo ime, se nanaša na sisteme, ki so odprti za komunikacijo z drugimi odprtimi sistemi. Arhitekturni model OSI obsega sedem slojev. Model je grafično ponazorjen na sliki 17.2. Slika prikazuje dve poljubni končni vozlišči, ki ju povezuje komunikacijski podsistem. V komunikacijski podsistem prištevamo vmesna vozlišča, kot so na primer usmerjevalniki (Angl. Router) in komunikacijska stikala (Angl. Communication switch).

Fizični sloj (Angl. Physical Layer) skrbi za prenos informacijskih signalov po komunikacijskem kanalu. Sloj določa mehanske, električne in postopkovne lastnosti naprav, signalov in tokokrogov. Tipična vprašanja v zvezi s tem slojem so napetostni nivoji signalov, hitrost prenosa, oblike signalov, vrste modulacij, konektorji in število priključkov na konektorjih, uporabljeni prenosni medij, čeprav sam ni del sloja.

Podatkovno linijski sloj ali linijski sloj (Angl. Data Link Layer) skrbi za visoko zanesljivost prenosa podatkov med sosednjimi vozlišči. Zato deli daljša zaporedja bitov na manjše okvire (Angl. Frames) ter odkriva in

[1] ISO 7498-1984: Information Processing Systems – Open Systems Interconnection Basic Reference Model.



Slika 17.2: *Referenčni model ISO OSI.*

popravlja napake na okvirih, če do njih pride. V podatkovno linijski sloj spada tudi nadzor nad dostopom do prenosnega sredstva (Angl. Media Access Control Sublayer – MAC).

Omrežni sloj (Angl. Network Layer) zagotavlja pot prenosa od izvornega do ponornega vozlišča. To doseže z naslavljanjem vozlišč, usmerjanjem paketov od vozlišča do vozlišča (Angl. Routing) in povezovanjem omrežij med seboj. Osnovna podatkovna enota tega sloja je paket (Angl. Packet).

Iz slike 17.2 razberemo, da vmesna vozlišča opravljajo naloge samo spodnjih treh slojev.

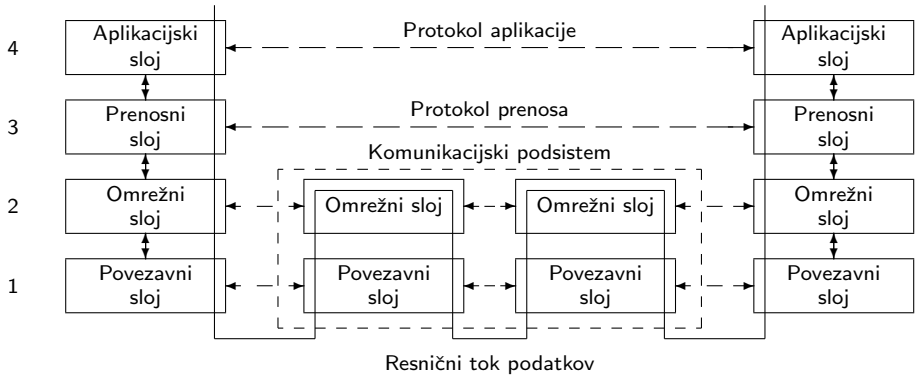
Prenosni sloj (Angl. Transport Layer) skrbi za prenos podatkov od enega končnega vozlišča do drugega, oddaljenega, končnega vozlišča. Iz slike 17.2 razberemo, da aktivnost na enem končnem vozlišču komunicira 'direktno' z istorodno aktivnostjo na oddaljenem vozlišču.

Pogovorni sloj (Angl. Session Layer) omogoča 'pogovor' med oddaljenimi

procesi. Medtem ko prenosni sloj skrbi za prenos podatkov med enim in drugim končnim vozliščem, skrbi pogovorni sloj za pogovor med istorodnima procesoma enega in drugega končnega vozlišča.

Predstavitveni sloj (Angl. Presentation Layer) opravlja kodiranje in prekodiranje, šifriranje in dešifriranje, zgoščevanje podatkov in podobno.

Sloj uporabe (Angl. Application Layer) zagotavlja storitve končnim uporabnikom. Zgornji vmesnik sloja uporabe je vmesnik med komuniacijskim sistemom in končnim uporabnikom. Od tega vmesnika je odvisen izgled omrežja, kot ga vidi končni uporabnik.



Slika 17.3: *Model TCP/IP.*

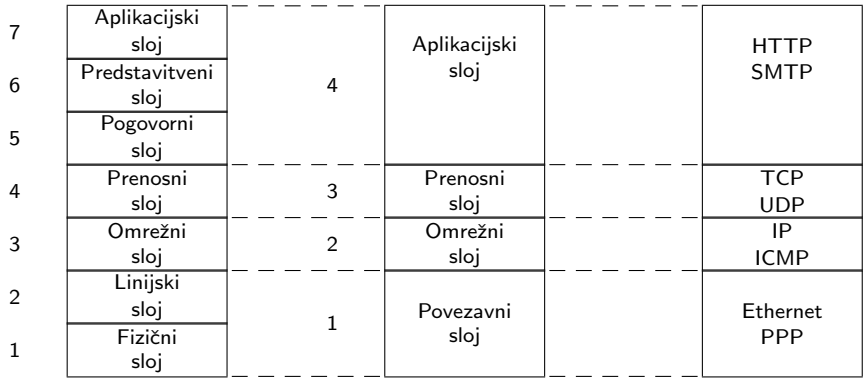
17.1.2 Model TCP/IP

Sedemslojni arhitekturni model OSI je nastajal v času, ko so bili zametki danes vsenavzočih Internetnih tehnologij že v uporabi in se je v marsičem tudi zgledoval po njih. Vendarle je slojnost omrežij dobila primeren teoretični poudarek šele s pojavom modela OSI, model OSI pa je korenito spremenil tudi način razmišljanja.

Omrežje Internet temelji na štirislojnem arhitekturnem modelu. Model in posledično arhitektura se dandanes največkrat poimenujeta kar model ali arhitektura TCP/IP, po dveh ključnih protokolih omrežne arhitekture. Slika 17.3 prikazuje arhitekturni model TCP/IP. Prvi sloj skrbi za do-

stop naprav do prenosnih poti, drugi sloj zagotavlja povezljivost naprav in omrežij, tretji sloj opravlja transport in četrti skrbi za podporo aplikacijam.

Slika 17.4 vzporeja model ISO OSI z modelom TCP/IP. Povezavni sloj modela TCP/IP ustreza fizičnemu in podatkovno linijskemu sloju, omrežni sloj omrežnemu, tansportni sloj transportnemu ter zgornji trije sloji ISO OSI modela aplikacijskemu sloju TCP/IP. Slika 17.4 desno prikazuje nekatere najvidnejše protokole današnjih omrežij.



Slika 17.4: *Primerjava med ISO OSI in TCP/IP modelom ter nekaj primerov protokolov TCP/IP modela.*

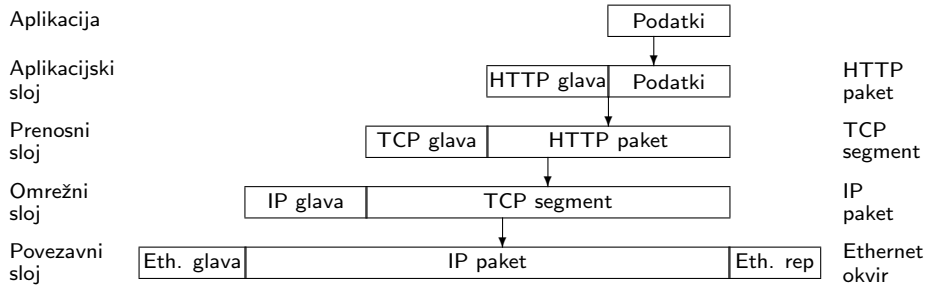
17.1.3 Načelo ovojnice

Eno temeljnih načel današnjih omrežij je načelo 'enkapsulacije' ali, lepše rečeno, načelo ovijanja. Koncept prikazuje slika 17.5. Primarna naloga omrežja je, da podatke aplikacije prenese na drugo stran. Na primer, ko uporabnik dostopa do spletnih vsebin na oddaljenem vozlišču, se morajo zahteva za dostop do spletnih vsebin in posledično vsebine spletnih strani prenesti z ene strani na drugo stran. Komunikacija med spletnima odjemalcem in strežnikom poteka po protokolu aplikacijskega sloja HTTP (HyperText Transmission Protocol). Protokol določa način prenašanja HTTP paketov v obe smeri. Paket HTTP sestavljata koristna vsebina končnega uporabnika ali 'tovor' (Angl. Payload) ter spremljajoča kontrolna informacija

ali 'glava' (Angl. Header). Glava ima predpisano strukturo. Vsebuje več kontrolnih polj, preko katerih se sporazumevata oddaljeni strani. Struktura glave in njen pomen sta določena s protokolom HTTP.

Za prenos paketov HTTP z ene na drugo stran je zadolžen prenosni sloj. V obravnavanem primeru skrbi za prenos HTTP paketov protokol TCP. Komunikacija po protokolu TCP poteka v obliki paketov, ki jim rečemo *TCP segmenti*. Segment sestavljata koristna vsebina ter TCP glava, prek katere poteka komunikacija po protokolu TCP. Vsebina TCP segmenta je HTTP paket. Rečemo, da smo HTTP paket 'ovili' v TCP segment.

Da pridejo segmenti TCP skozi omrežje na drugo stran, skrbi omrežni sloj. Prenosno pot izbira in nadzira protokol IP. Paket IP, imenovan tudi *datagram*, sestavljata glava in koristna vsebina. Koristna vsebina paketa IP je TCP segment. Nazadnje se IP paket ovije v *ethernet okvir* in pošlje na prenosni medij. Ko pridejo podatki po prenosnem sredstvu na drugo stran, se pričnejo vzpenjati po slojih navzgor. Pri tem se sproti odvijajo v obratnem vrstnem redu ovijanja, dokler končno koristni podatki ne pridejo do uporabnika, slika 17.5.



Slika 17.5: *Načelo ovojnice na primeru HTTP, TCP, IP, Ethernet. Paket višjega sloja se prenaša kot vsebina (tovor) paketa nižjega sloja.*

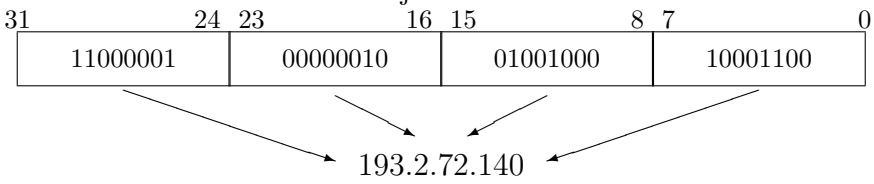
17.1.4 Omrežni naslovi

Vsako vozlišče v omrežju ima naslov, po katerem je znano drugim vozliščem v omrežju. *Omrežni naslov* v omrežjih TCP/IP imenujemo tudi IP naslov.

IP naslov oddajne naprave ter IP naslov sprejemne naprave se prenašata v glavi IP paketa. Naslov po protokolu IP verzije štiri (IPv4) obsega 32 bitov, slika 17.6. V navadi je, da se 32-bitni naslov zapiše z desetiškimi vrednostmi štirih bajtov in vrednosti loči s piko. Na primer, desetiški zapis IP naslova enega od računalnikov na Fakulteti za elektrotehniko je

```
193.2.72.140
```

in ekvivalentni 32-bitni IP naslov je `11000001000000100100100010001100`.



Slika 17.6: *Predočanje 32-bitnega IP naslova.*

V omrežjih, ki uporabljajo protokol IP verzije šest (IPv6), so omrežni naslovi 128-bitni. Po dogovoru se IPv6 naslov zapiše z osmimi 16-bitnimi šestnajstiško kodiranimi vrednostmi, ki se jih loči z dvopičji. Obliko zapisa predoča naslednji hipotetični IPv6 naslov,

```
a000:0999:0088:0000:0000:0000:0007:0006.
```

Ker so naslovi IPv6 zelo 'dolgi', se sme vodilne ničle in zaporedne skupine ničel opuščati. Gornji naslov bi smeli zapisati krajše:

```
a000:999:88::7:6.
```

Okrajšan zapis je moč nedvoumno vrniti v prvotno obliko. Nazaj do polnega IPv6 naslova pridemo tako, da vsako skupino 16-bitov dopolnimo z ustreznim številom vodilnih ničel in dvojno podpičje razširimo v toliko ničel, kolikor je potrebno, da dobimo 128-bitni naslov.

Obstaja tudi možnost kombiniranja IPv4 in IPv6 naslovov. Aplikacija, ki temelji na IPv6 naslavljanju, lahko komunicira z IPv4 vozlišči tako, da uporabi IPv4 'preslikan' IPv6 naslov. Na primer, naslov IPv4

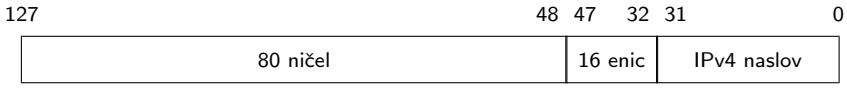
```
193.2.72.140
```

ima naslednji IPv6 ekvivalent:

```
::FFFF:193.2.72.140
```

Naslov IPv4 se zapiše v desetiški obliki, doda se mu predpono 16-ih enic

ter 80-ih ničel, kot ponazarja slika 17.7.



Slika 17.7: *Preslikava IPv4 naslova v IPv6 naslov.*

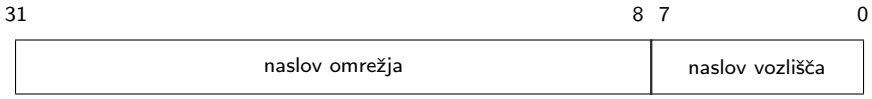
Omrežni naslov se pomensko deli na naslov omrežja ter na naslov vozlišča znotraj omrežja, slika 17.8. Zgornji biti naslova pripadajo omrežju, preostali spodnji biti pomenijo naslov vozlišča v tem omrežju. Delitev naslova na omrežni del ter na ostanek za vozlišče določa *naslovna maska*, kar se na kratko v tako imenovanem CIDR formatu zapiše takole:

```
nn.nn.nn.nn/mm
```

Denimo, zapis

```
193.2.72.0/24
```

pravi, da predstavlja 24 zgornjih bitov 32-bitnega naslova naslov omrežja. V našem konkretnem primeru je naslov omrežja 193.2.72.0. Preostalih spodnjih osem bitov določa eno od vozlišč v tem omrežju. Najvišji naslov vozlišča v tem omrežju je 193.2.72.255. Ta naslov je tako imenovani splošni naslov (Angl. Broadcast address). Karkoli je poslano na ta naslov, sprejmejo *vsa* vozlišča v tem omrežju. Najnižji naslov je naslov omrežja. Ta je 193.2.72.0. Ostale kombinacije so naslovi vozlišč. Naše omrežje ima potemtakem lahko največ 254 vozlišč. Delitev naslova na omrežni 'prefiks' ter naslov vozlišča v omrežju je pomemben za usmerjanje paketov skozi omrežja.



Slika 17.8: *Format in pomen naslova IPv4.*

17.1.5 Številke vrat

Na posamezni napravi sočasno obstaja več procesov, ki želijo komunicirati z oddaljenimi procesi na drugih napravah v omrežju. Za komunikacijo med

enim od procesov na eni napravi z enim od procesov na oddaljeni napravi omrežna naslova obeh naprav še nista dovolj. Potrebno je dodatno določilo, ki je specifično za posamezen proces v okviru iste naprave.

V omrežjih TCP/IP je to rešeno s številkami *komunikacijskih vrat* (Angl. Communication port). Številko komunikacijskih vrat (Angl. Port number) smemo razumeti kot naslov priključnega mesta, kamor se pripne proces in s tem dobi dostop do komunikacijskega kanala oziroma *komunikacijskih storitev*. Proces, ki želi komunicirati z oddaljenim procesom na drugi napravi, mora poznati oboje, omrežni naslov naprave in številko komunikacijskih vrat, na katera je 'priključen' oddaljeni proces. Omrežni naslov in številko vrat zapišemo skupaj in ločimo z dvopičjem. Na primer, vrata 23 na napravi 192.2.72.140 zapišemo skupaj takole,

```
192.2.72.140:23
```

Spomnimo se, da se omrežni naslov prenaša v glavi IP paketa, medtem ko se številka vrat prenaša v glavi protokola prenosnega sloja, torej v glavi enega od protokolov TCP, UDP, SCTP ali DCCP. Številka vrat je šestnajstbitno število. Števila 1 do 1023 so dodeljena s strani IANA (Internet Assigned Numbers Authority) in so rezervirana 'za dobro znane' storitve. Denimo, vrata 21 so za ftp strežnik, vrata 23 so za telnet, vrata 25 za poštni smtp strežnik, vrata 80 so za spletni strežnik. Številke od 1024 do 49151 so 'registrirana' vrata za aplikacije, ki jih registrira IANA. Števila nad 49151 so svobodna za prosto, začasno, rabo. Številke vrat se vezane na uporabljeni protokol prenosnega sloja. Tako imamo TCP vrata in UDP vrata. Številki TCP in UDP vrat sta seveda lahko enaki, pa vendar gre za različna vrata.

17.1.6 Vrstni red bajtov

Kadar poteka prenos podatkov med dvema napravama kot 'golo' zaporedje bajtov, nas sestavljene stukture, ki so nadgrajene na zaporedja bajtov in jim posledično določajo njihov pomen, posebej ne zanimajo. Tudi kadar poteka komunikacija med sistemi, ki obravnavajo 'vrstni red' ali *redosled bajtov* na enak način, nas njihovo strukturiranje posebej ne zanima. Če pa gre za

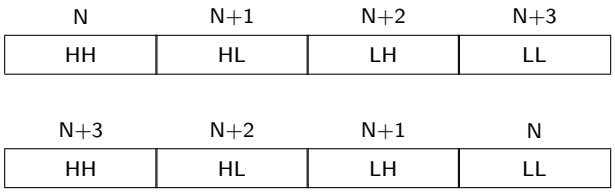
prenašanje podatkov med sistemi, ki temeljijo na različnih interpretacijah vrstnega reda bajtov, moramo poskrbeti tudi za pretvorbo med različnimi interpretacijami zapisa.

Današnji računalniki, ali bolje rečeno računalniške arhitekture, uporabljajo enega od dveh načinov zapisa večbajtnih podatkovnih enot, na primer dva-intridesetbitnih celih števil, v pomnilniku:

- zaključek na višjem naslovu ali pravilo 'debelega konca' (Angl. Big endian),
- zaključek na nižjem naslovu ali pravilo 'tankega konca' (Angl. Little endian).

Koncept prikazuje slika 17.9. Dokler obravnavamo sestavljene podatke v okviru enake arhitekture, nas sam način zapisa ne skrbi in dejansko se ga sploh ne zavedamo. V primeru prenašanja podatkov med sistemi z različno arhitekturo, pa moramo poskrbeti tudi za spremembo vrstnega reda bajtov. Pri prenosu podatkov z enega na drugo vozlišče imamo načeloma opravka z naslednimi interpretacijami vrstnega reda bajtov:

- vrstni red na izvornem (to je oddajnem) vozlišču,
- vrstni red med prenosom (Angl. Network ordering),
- vrstni red na ponornem (to je sprejemnem) vozlišču.



Slika 17.9: *Načelo debelega in tankega konca. Debeli konec: zadnji bajt na najvišjem naslovu (Angl. Big endian). Tanki konec: zadnji bajt na najnižjem naslovu (Angl. Little endian).*

Omrežje uporablja pravilo debelega konca. Vrstna reda bajtov izvornega in ponornega vozlišča sta odvisna od dotičnih arhitektur ter sta lahko enaka ali različna. Lahko sta sicer enaka, a drugačna od vrstnega reda omrežja. Zato

je najbolje, da pred oddajo spremenimo oziroma zagotovimo pravilni vrstni red bajtov iz lokalnega v 'pravi' vrstni red, to je vrstni red za prenašanje. Obratno ravnamo na sprejemni strani.

Podporo za prilagoditev vrstnega reda bajtov dajejo naslednje sistemske funkcije:

```
#include <arpa/inet.h>

/* h-host; n-network; l-long; s-short */

uint32_t htonl( uint32_t hostint32 );
uint16_t htons( uint16_t hostint16 );
uint32_t ntohl( uint32_t netint32 );
uin162_t ntohs( uint16_t netint16 );
```

Funkcije pretvarjajo 32-bitne ('long') in 16-bitne ('short') vrednosti iz zapisa, ki je v veljavi na vozlišču ('host'), v zapis, ki je v veljavi med prenosom ('network') ter obratno. Številke vrat so že tak 16-bitni primer in IPv4 naslovi so 32-bitni. Torej, za IPv4 naslov bi imeli

```
// oddajna stran - pred oddajo
networkIPv4Addr = htonl( hostIPv4Addr );
...
// sprejemna stran - po sprejemu
hostIPv4Addr = ntohl( networkIPv4Addr );
```

17.1.7 Pretvorbe med predstavitvami naslovov

V uporabi so torej naslovi vozlišč, desetiški zapisi IPv4 naslovov vozlišč in šestnajstiški zapisi IPv6 naslovov vozlišč. Poleg tega pa imamo še številke vrat in njim pridružena tako imenovana imena 'storitev'. Zelo pogosto se pojavi potreba, ko moramo naslov iz enega formata pretvoriti v drugačen format. Na primer, imamo 32-bitni IPv4 naslov in potrebujemo njegovo desetiško 4-bajtno predstavitev. Za pretvarjanje med različnimi oblikami zapisa omrežnih naslovov imamo na voljo kar precej funkcij. Dve izmed teh sta:

- `inet_ntoa()` (beri 'Network to Alpha'): funkcija za podani 32-bitni IPv4 naslov v binarni obliki vrne znakovno kodiran 4-bajtni decimalni zapis s piko,
- `inet_aton()`: funkcija za podani 4-bajtni znakovno kodiran decimalni zapis s piko vrne 32-bitni IPv4 naslov v binarni obliki.

Prototipa funkcij pa sta:

```
#include <arpa/inet.h>

int inet_aton( const char *stringPtr, struct in_addr *addrPtr );
    Vrne 1 v primeru uspeha in 0 v primeru napake.

char *inet_ntoa( struct in_addr inAddress );
    Vrne kazalec na znakovni decimalni zapis ali NULL v primeru napake.
```

Na primer, z znanim IPv4 naslovom vozlišča v znakovno kodiranem decimalnem zapisu pridemo do 32-bitnega IPv4 naslova v omrežnem redosledu takole,

```
struct in_addr hostIPv4Addr;
....
inet_aton( "192.168.90.32", &hostIPv4Addr );
....
```

Ali, 32-bitni IPv4 naslov pretvorimo v znakovni decimalni zapis,

```
char dotDecIPv4[16];
struct in_addr hostIPv4Addr;
...
hostIPv4Addr.s_addr = htonl(0x7f000001); //127.0.0.1
dotDecIPv4 = inet_ntoa( hostIPv4Addr );
...
```

Težava s funkcijami za pretvarjanje oblik naslova je predvsem v tem, da moramo poznati definicije naslovnih struktur in imena komponent teh struktur. Ne smemo pozabiti tudi na pravilen vrstni red bajtov.

Navedeni funkciji delujeta za IPv4 naslove. Naslednji funkciji `inet_pton` in `inet_ntop` sta univerzalni, delujeta tako za IPv4 kot za IPv6 ('p' se bere kot 'presentation' in 'n' pride od 'network').

```
#include <arpa/inet.h>

int inet_pton( int family, const char *stringPtr, void *addrPtr );
Vrne 0 ali -1 v primeru napake in 1 v primeru uspeha

const char *inet_ntop( int family, const void *addrPtr,
                                   char *stringPtr, size_t len );
Vrne NULL v primeru napake ali kazalec na znakovni niz.
```

Na primer, naslednji klic pretvori IPv4 desetiški zapis v binarni IPv4 32-bitni naslov v omrežnem redosledu.

```
#include <arpa/inet.h>
...
struct in_addr hostIPv4Addr;

err = inet_pton(AF_INET, "127.0.0.1", &hostIPv4Addr); //loopback IPv4 addr.
...
```

Za pretvorbo IPv6 šestnajstiškega zapisa v IPv6 128-bitni naslov pa imamo:

```
#include <arpa/inet.h>
...
stuct in6_addr hostIPv6Addr;

err = inet_pton(AF_INET6, "::1", &hostIPv6Addr); //loopback IPv6 addr.
...
```

Definiciji obeh naslovnih struktur sta naslednji:

```
stuct in_addr{
  in_addr_t s_addr;    // 32-bitni IPv4 naslov
}

struct in6_addr{
  uint8_t s6_addr[16]; // 128-bitni IPv6 naslov
}
```

Primer pretvorbe naslovov IPv4 ter IPv6 iz desetiškega in šestnajstiškega zapisa v binarno obliko v omrežnem redosledu in nazaj pa prikazuje naslednji kos programa.

```
#include <arpa/inet.h>
...
struct in_addr  hostIPv4;
struct in6_addr hostIPv6;
char   dotDecIPv4[16];
char   hexDecIPv6[46];

int    errIPv4, errIPv6;
const  char *errPtrIPv4, *errPtrIPv6;
...
errIPv4 = inet_pton(AF_INET, "127.0.0.1", &hostIPv4);
errIPv6 = inet_pton(AF_INET6, "::1", &hostIPv6);
...
errPtrIPv4 = inet_ntop(AF_INET, &hostIPv4, dotDecIPv4, 16);
errPtrIPv6 = inet_ntop(AF_INET6, &hostIPv6, hexDecIPv6, 46);
...
printf("IPv4: %s\n", dotDecIPv4);
printf("IPv6: %s\n", hexDecIPv6);
...
```

Konstanta 16 pomeni maksimalno dolžino znakovnega niza za IPv4 naslov. Ta obsega štire bajte, tri znake na bajt, tri ločilne pike in dodatni zadnji ničnі bajt, $4 \times 3 + 3 + 1 = 16$. Podobno velja za konstanto 46 in maksimalno dolžino znakovnega niza za šestnajstiško kodiran IPv6 naslov.

17.1.8 Omrežna imena

Vsaki napravi v omrežju je pridruženo ime. Imena so v rabi zgolj zato, ker si jih ljudje lažje zapomnimo kot številke. Denimo, lažje si zapomnimo ime naprave kot njen omrežni naslov. Prostor imen je urejen hierarhično. Format imena je predpisan. Ime je sestavljeno iz znakovnih nizov, ki so ločeni s pikami. Polno *domensko ime* je sestavljeno iz imena naprave (Angl. host name) in zaporedja imen domen. Na primer:

`vision.fe.uni-lj.si`

je polno ime računalnika z naslovom 193.2.72.140. Pri tem je `vision` njegovo ime, `si` je ime vrhnje domene (Angl. Top-level domain, TLD) za Slovenijo,

`uni-lj` je ime poddomene za Univerzo v Ljubljani znotraj domene `si`, `fe` je ime poddomene Fakultete za elektrotehniko znotraj `uni-lj.si`.

Uporabniki običajno poznajo imena naprav, a za delovanje omrežja so pomembnejši IP naslovi. Za preslikavo med imeni in IP naslovi skrbijo imenski strežniki in porazdeljena podatkovna baza DNS (Angl. Domain Name System). Strežniki imen (Angl. Name Servers), ki so napravam znani, na poizvedbo z danim imenom vrnejo pripadajoči IP naslov.

17.1.9 Pretvorbe med imeni in naslovi

Največkrat poznamo ime naprave, a potrebujemo njen naslov, ali obratno. Funkciji, ki rešita problem, sta:

- `gethostbyname()`: za dano ime naprave vrne IPv4 naslov v omrežnem redosledu,
- `gethostbyaddress()`: za dani IPv4 naslov vrne ime naprave.

Na primer, naslednji klic

```
struct hostent *host;
in_addr_t      *hostAddr;
  ....
if( (host = gethostbyname("vision.fe.uni-lj.si")) == NULL) exit( 1 );
hostAddr  = host -> h_addr;
  ....
```

vrne 32-bitni IPv4 naslov v omrežnem redosledu. Sicer pa se uporabo funkcij `gethostbyname()` in `gethostbyaddress()` odsvetuje ter priporoča sodobnejši funkciji `getaddrinfo()` in `getnameinfo()`. V nadaljevanju si bomo obe funkcije tudi pobliže ogladali, a še prej se posvetimo sistemu komunikacijskih vtičnic.

17.2 Komunikacijske vtičnice – socket

Sistem komunikacijskih vtičnic (Angl. Socket) izvira iz BSD (Berkeley Software Distribution) veje sistemov UNIX. Pojavil se je leta 1983 v sistemu 4.2BSD. Skozi leta se je izpopolnjeval, a konceptualno razmeroma malo spreminjal. Do danes se je v izpopolnjeni obliki uveljavil domala povsod. V večini sistemov UNIX in v večini UNIX-u podobnih sistemov je bil vmesnik vtičnic zasnovan na podlagi BSD implementacije. POSIX.1 specifikacija programskega vmesnika API (Application Programming Interface) v glavnem temelji na 4.4BSD specifikaciji. Vtičnice sistema Linux pa so bile realizirane od začetka oziroma iz 'nič'.

Sistem vtičnic omogoča enotno obravnavanje medprocesne komunikacije neodvisno od krajevne porazdelitve procesov in zatorej predstavlja bistveno razširitev zmožnosti medprocesnega komuniciranja. Vtičnice služijo dvosmernemu komuniciranju med procesi, ki so poljubno porazdeljeni v omrežju. Razumljivo, da je na enak način možna komunikacija med procesi tudi v okviru iste naprave.

Komunikacijska vtičnica je osnovni gradnik medprocesnega komuniciranja. *Vtičnica* (Angl. Socket) je ime za priključno mesto enega od obeh koncev dvosmernega komunikacijskega kanala. Storitve sistema vtičnic so dane na zgornjem vmesniku prenosnega sloja. Če želita dva procesa komunicirati med seboj, mora vsak od njiju dobiti dostop do enega od koncev kanala – do ene od med seboj povezanih vtičnic – para vtičnic. Osnovni problem komuniciranja procesov v omrežju je povezan z vprašanjem, kako med seboj 'povezati' par vtičnic na oddaljenih napravah in na vtičnice 'pripeti' procese. Sistem vtičnic reši tudi ta problem.

Vtičnice obstajajo znotraj vnaprej predpisanih *komunikacijskih domen*. Komunikacijsko domeno definirata *naslovna družina* ter *protokolovna družina*. Z naslovno družino je opredeljen naslovni prostor. Naslovna družina določa pomen in format naslovov vtičnic. Protokolovno družino tvorijo protokoli, ki izvirajo iz dane omrežne arhitekture. Protokoli dajejo podlago za izvedbo komunikacijskih storitev, ki so dane na vtičnici. Znane naslovne družine in

s tem povezane protokolovne družine oziroma kar komunikacijske domene, so:

- `AF_UNIX` za komunikacije med procesi v okviru ene naprave,
- `AF_INET` za komunikcije v omrežjih temelječih na protokolu IPv4,
- `AF_INET6` za komunikacije v omrežjih temelječih na protokolu IPv6 in
- `AF_UNSPEC`.

Naslovna družina `AF_UNIX` historično opredeljuje komunikacijsko domeno v okviru sistema UNIX in posledično komunikacije med procesi na istem vozlišču. Oblika in pomen naslovov sta določena z datotečnim sistemom naprave oziroma shemo poimenovanja datotek. V nekaterih sistemih in po specifikaciji POSIX je sinonim te domene `AF_LOCAL`.

Naslovna družina `AF_INET` obstaja v omrežju Internet in temelji na protokolu IP verzije 4, IPv4. S tem sta določena pomen in oblika naslovov v komunikacijski domeni IPv4.

Naslovna družina `AF_INET6` obstaja v omrežju Internet in temelji na protokolu IP verzije 6, IPv6. S tem sta opredeljena oblika in pomen naslovov v domeni IPv6.

Naslovna družina `AF_UNSPEC` pravzaprav ni družina, temeč prepušča izbiro komunikacijske domene drugim dejavnikom.

Skupna predpona `AF_` pride od 'Address Family'. V literaturi tu pa tam srečamo še sorodna simbolična imena, ki začnejo s `PF_`, kar se bere 'Protocol Family'. Sistem vtičnic je bil v zgodnejših fazah razvoja namreč zasnovan zelo splošno. Predvideval je poljubne kombinacije protokolovnih družin z naslovnimi prostori. No, kasnejši razvoj je pokazal, da je tako zastavljena shema preveč splošna in v bistvu odveč. Denimo, ko izberemo naslovno družino `AF_INET`, sta s tem protokolovna družina (IP, TCP, UDP) in posledično tudi komunikacijska domena že določeni. Zato konstantam AF_UNIX, AF_INET, AF_INET6 običajno rečemo kar UNIX domena, Internet oziroma IP (IPv4) domena, in IPv6 domena. Simbolične konstante `AF_` ter `PF_` pa obravnavamo kot sopomenke.

Vtičnica ima svoj tip. *Tip* je določen z lastnostmi komunikacijskega kanala, konkretneje s protokolom oziroma storitvami, ki jih le-ta zagotavlja. Mogoči *tipi* vtičnic so:

- `SOCK_STREAM` za podatkovni tok,
- `SOCK_DGRAM` za datagram,
- `SOCK_RAW` za 'goli' tip,
- `SOCK_SEQPACKET` za sporočilni tip.

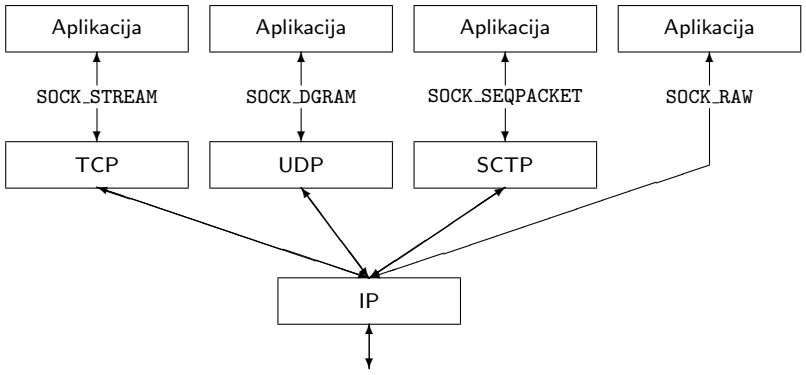
Vtičnice tipa `SOCK_STREAM` zagotavljajo *povezaven* način komuniciranja. To pomeni, da med obema koncema komunikacijskega kanala – vtičnicama – obstaja cel čas komuniciranja 'navidezna' povezava. Povezava se najprej vzpostavi, nakar se med prenašanjem podatkov povezava vzdržuje, dokler se na koncu ne sprosti ali 'podre'. Podatki gredo od enega do drugega konca zaporedno drug za drugim brez podvajanja ali izgubljanja, kot da bi bila med oddaljenima stranema v času prenašanja podatkov resnično napeljana direktna fizična povezava. *Podatkovni tok* ali 'podatkovod' pomeni, da se podatki prenašajo kot neprekinjen niz podatkov brez razmejitev med sporočili. Ta koncept smo že spoznali, ko smo obravnavali cevi. V domeni `AF_INET` se vtičnice tipa `SOCK_STREAM` realizirajo s protokolom TCP.

Vtičnice tipa `SOCK_DGRAM` ponujajo *nepovezaven* način komuniciranja oziroma *datagram* način. Nepovezaven način komuniciranja pomeni, da med oddaljenima stranema ne obstaja povezava, tako kot pri vtičnicah tipa `SOCK_STREAM`. Do vzpostavitve povezave sploh ne pride. V domeni `AF_INET` poteka prenos po protokolu UDP. Oddajna stran oddaja UDP datagrame, pri čemer predstavlja vsak oddani datagram samostojno celoto, ki gre skozi omrežje neodvisno od predhodnega in naslednjega datagrama. To posledično pomeni, da gredo podatki od enega konca do drugega konca načeloma celo po različnih prenosnih poteh in so spričo tega lahko dostavljeni tudi v spremenjenem vrstnem redu. Nista izključeni niti možnosti podvajanja ali izgubljanja. Skratka, vtičnica ne zagotovalja, da bodo podatki pravilno dostavljeni in v enakem zaporedju, kot so bili oddani. Po drugi strani pa vtičnice tipa `SOCK_DGRAM` posredno zagotavljajo razmejitev

med sporočili. Namreč, posamezen datagram je sprejet v enaki obliki kot je bil oddan.

Vtičnice tipa `SOCK_RAW` omogočajo dostop do storitev nižjih slojev komunikacijske arhitekture. V omrežju TCP/IP se prenosni sloj enostavno obide. S tem je omogočen direkten dostop do protokolov omrežnega sloja, torej protokolov IP, ICMP, ali IGMP. Uporaba vtičnic `SOCK_RAW` zahteva priviligirana pooblastila.

Vtičnice tipa `SOCK_SEQPACKET` zagotavljajo sekvenčen, zanesljiv, povezaven način, ki ohranja meje med oddanimi in sprejetimi segmenti podatkov. So torej najbolj podobne vtičnicam tipa `SOCK_STREAM`, s to razliko, da ohranjajo meje med sporočili. Odnos med tipi vtičnic in protokoli v domeni `AF_INET` prikazuje slika 17.10.



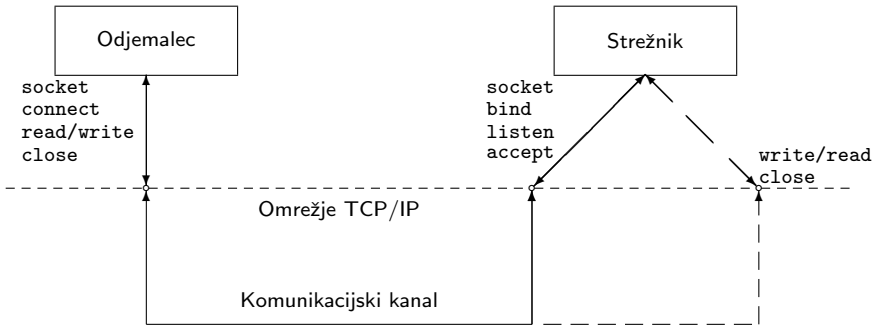
Slika 17.10: *Tipi komunikacijskih vtičnic v domeni* `AF_INET` *glede na protokole prenosnega sloja.*

17.2.1 Uvod v funkcije sistema vtičnic

Osnovne funkcije programskega vmesnika komunikacijskih vtičnic so: `socket`, `bind`, `listen`, `accept` in `connect`. S programskega stališča je vtičnica objekt, ki jo ustvari sistemski klic `socket`, vendar komunikacija lahko začne šele, ko je vtičnica z `bind` 'pripeta' na naslov Komunikacija med procesi je asimetrična po načelu 'odjemalec / strežnik' (Angl. Client/Server), kot ponazarja slika 17.11. Strežnik se z `listen` postavi v stanje 'pasivnega' čakanja ter

nato z `accept` na znanem naslovu vtičnice sprejema zahteve odjemalcev, ki s `connect` poskušajo priti do storitev strežnika – skušajo vzpostaviti zvezo z njim.

Če povezava (accept - connect) uspe, strežnik 'odpre' novo vtičnico, na kateri potem streže odjemalcu sam ali pa nalašč za to ustvari nov proces. Komunikacija med odjemalcem in strežnikom na novi vtičnici nato teče v obe smeri, bodisi z `read` in `write` bodisi s `send` in `recv`, dokler eden od partnerjev ali oba ne zapreta vtičnice s `close`. Na splošno večina funkcij za upravljanje datotek, a ne vse, deluje tudi z vtičnicami. Opisani scenarij je značilen za povezaven tip vtičnic. Podobno, a brez vzpostavljanja zveze, delujejo nepovezavne vtičnice. V nadaljevanju si bomo pobliže ogledali nekatere važnejše funkcije za upravljanje vtičnic ter na enostavnih primerih ilustrirali njih uporabo.



Slika 17.11: *Komunikacija po načelo odjemalec/strežnik na osnovi komunikacijskih vtičnic v povezavnem načinu.*

17.2.2 Funkcija `socket`

Funkcija `socket` ustvari en konec dvosmernega komunikacijskega kanala, ki ga preprosto poimenujemo vtičnica. Klic vrne deskriptor vtičnice. Denimo,

```
sd = socket( domena, tip, 0 );
```

ustvari vtičnico v predvideni naslovni *domeni*, ki je želenega *tipa* in uporablja privzeti komunikacijski protokol. Zato ima zadnji argument vrednost nič. Na primer:

```
sd = socket( AF_INET, SOCK_STREAM, 0 );
```

ustvari vtičnico v komunikacijski domeni IPv4 povezavnega tipa na podlagi protokola TCP. Prototip funkcije pa je:

```
#include <sys/socket.h>

int socket( int af, int type, int protocol );
```

Funkcija vrne deskriptor vtičnice, preko katerega je vtičnica dosegljiva, ali -1 v primeru napake.

Argument `af` določa naslovno družino in je lahko `AF_INET`, `AF_INET6`, `AF_UNIX` ali `AF_LOCAL` ter `AF_UNSPEC`.

Obstajajo še druge naslovne družine, a podpora le-tem je odvisna od implementacije. Definicije v sistemu GNU/Linux so zbrane v datoteki

`/usr/include/bits/socket.h`.

Nekaj primerov drugih naslovnih družin je `AF_IPX`, `AF_APPLETALK`, `AF_CAN` in denimo `AF_PACKET`. `AF_PACKET` daje podlago za direkten dostop do storitev povezavnega–linijskega sloja, a o tem ne bomo govorili.

Argument `type` predpiše *tip* vtičnice. Pomen tipa vtičnice smo že spoznali, možnosti izbire pa so `SOCK_STREAM`, `SOCK_SEQPACKET`, `SOCK_DGRAM` in `SOCK_RAW`.

Argument `protocol` predpiše komunikacijski protokol, ki je odvisen od komunikacijske domene. Če je vrednost argumenta nič in to je priporočen način, sistem vtičnic izbere 'privzeti' protokol sam. Na primer, za naslovno družino oziroma komunikacijsko domeno `AF_INET` ter povezaven tip vtičnic tipa `SOCK_STREAM`, je privzeti protokol prenosnega sloja protokol TCP. Za nepovezavne vtičnice `SOCK_DGRAM` v domeni `AF_INET` je privzeti protokol prenosnega sloja protokol UDP (Angl. User Datagram Protocol). Komunikacijska domena `AF_INET` tipa `SOCK_SEQPACKET` uporablja protokol SCTP. (Angl. Stream Control Transmission Protocol). Kot že rečeno, je razlika med `SOCK_STREAM` in `SOCK_SEQPACKET` prenosom v tem, da slednji ohranja razmejitev sporočil. Povedano drugače, sporočilo, ki je bilo poslano in obsega

denimo 128 bajtov, bo tako tudi sprejeto. Po drugi strani protokol TCP ne ohranja mej med oddanimi in sprejetimi enotami. Oddajnik oddaja različno dolga zaporedja bajtov, medtem ko jih sprejemnik sprejema kot neprekinjen niz podatkov.

Simbolične konstante protokolov za naslovno družino `AF_INET` pa so:

- `IPPROTO_TCP`: protokol TCP (Transmission Control Protocol),
- `IPPROTO_UDP`: protokol UDP (User Datagram Protocol),
- `IPPROTO_SCTP`: protokol SCTP (Stream Control Transmission Protocol).

Sledi še nekaj primerov klicev funkcije ter vtičnic:

```
...
/* za domeno Internet, protokol TCP */
sd = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP );
...
/* za domeno Internet, protokol UDP */
sd = socket( AF_INET, SOCK_DGRAM, 0 );
...
/* za domeno UNIX, povezavni tip    */
sd = socket( AF_UNIX, SOCK_STREAM, 0 );
...
```

17.2.3 Funkcija `bind`

S klicem `socket` nastane vtičnica brez naslova. Vtičnica sicer obstaja, vendar ji je pred uporabo potrebno pripeti še naslov. To nalogo opravi funkcija `bind`, ki vtičnici z znanim deskriptorjem priduži naslov. Na primer, klic:

```
bind( sd, naslov, sizeof(naslov));
```

vtičnici z deskriptorjem *sd* pridruži *naslov*. Vprašanje pa seveda ostane, kakšen je *naslov*. Format ter pomen naslova sta seveda odvisna od komunikacijske domene. In morda prav v definicji naslova, v vrstnem redu bajtov ter v različnih implementacijah tiči največja nadloga pri delu z vtičnicami. Prototip funkcije je:

```
#include <sys/socket.h>
#include <netinet/in.h>     /* za AF_INET */
#include <sys/un.h>         /* za AF_UNIX */

int bind( int s, const struct sockaddr *addr, socklen_t addrlen );
```

Funkcija vrne 0, če klic uspe, sicer -1.

Ker je format naslova odvisen od domene, je tudi dejanski tip adresne strukture, ki se ob klicu funkcije poda kot drugi argument, odvisen od domene. V definiciji funkcije `bind` pa nastopa 'generični' tip adresne strukture. Nobena dejanska naslovna družina ne uporablja generičnega tipa naslovne strukture. Generični tip v definiciji funkcije zgolj nadomešča vse mogoče dejanske naslovne strukture. Njegova definicija je:

```
struct sockaddr{
  sa_family_t sa_family;    /* naslovna druzina        */
  char        sa_data [14]; /* naslov dolocene dolzine */
};
```

Drugi argument funkcije `bind` je torej *referenca* na naslovno strukturo, katere dejanski pomen je odvisen od dotične komunikacijske domene. Elegantejša definicija tipa drugega argumenta funkcije `bind` bi sicer bila `void *`, a funkcija `bind` je še iz časov, ko programski jezik C tega tipa ni poznal.

Konkretno, za domeno `AF_UNIX` imamo naslednjo definicijo naslovne strukture `struct sockaddr_un`,

```
struct  sockaddr_un {
  short   sun_family;             /* AF_UNIX   */
  char    sun_path[108];          /* path name */
};
```

Definicijo strukture `sockaddr_un` najedemo v `/usr/include/sys/un.h`. Klic funkcije `bind` za vtičnico v komunikacijski domeni `AF_UNIX` pa bi bil načeloma videti takole:

```
  ...
  struct sockaddr_un unixAddress; /* ime je v bistvu pot */
  ...
  unixAddress.sun_family = AF_UNIX;
  unixAddress.sun_path   = "MojaPot";
```

```
  bind( sd, (struct sockaddr *)&unixAddress, sizeof( unixAddress ) );
  ...
```

Za vtičnico v domeni `AF_INET` je naslovna struktura tipa `struct sockaddr_in`, definirana je v datoteki `/usr/include/netinet/in.h`, sicer pa je videti takole:

```
struct sockaddr_in {
  sa_family_t    sin_family;
  in_port_t      sin_port;  /* stevilka vrat, 16 bitov, nepredzanceno */
  struct in_addr sin_addr;  /* IPv4 naslov, 32 bitov, nepredznaceno   */
  char           sin_zero[8];/* dopolnilo */
};
struct in_addr {
  in_addr_t     s_addr;        /* IPv4 naslov, 32 bitov, nepreznaceno */
};
```

Naslov vtičnice torej sestavljata številka vrat (TCP ali UDP), na katera je pripet proces ter omrežni naslov (IPv4) vozlišča.

Naslednji kos programa na vtičnico z deskriptorjem `sd` pripne naslov:

```
 struct sockaddr_in mojServ;
 ....
 bzero((char *)&mojServ,sizeof(mojServ));
 mojServ.sin_family        = AF_INET;
 mojServ.sin_port          = htons( 80 );       /* stevilka vrat     */
 mojServ.sin_addr.s_addr   = htonl(0xc0a86a20); /* 192.168.90.32 IPv4,
                                                   omrezni vstni red */
 bind( sd, (struct sockaddr *)&mojServ, sizeof(mojServ));
 ....
```

Elegantneje bi seveda bilo uporabiti funkcijo za pretvorbo desetiško podanega IP naslova v binarno obliko

```
 ...
 struct in_addr  hostIPv4Addr;
 ...
 inet_aton( "192.168.90.32", &hostIPv4Addr );
 mojServ.sin_addr = hostIPv4Addr;
```

....

17.2.4 Funkciji `accept` in `listen`

Potem, ko na vtičnico uspešno pripnemo veljaven naslov, je za začetek komunikacije potrebna pripravljenost dveh procesov, strežnika in odjemalca. Strežnik z `accept` čaka na `connect` odjemalca. Pred tem strežnik z `listen` predpiše maksimalno število čakajočih zahtev odjemalcev. Na primer, naj bo to število 5:

```
listen(sd,  5 );
```

Klic `listen` je potreben oziroma smiseln le pri povezavnem tipu vtičnic, torej ko gre za prenos po protokolu TCP ali SCTP. Funkcija `listen` dejansko postavi vtičnico v stanje pasivne pripravljenosti na zahtevo za vzpostavitev povezave oddaljenega odjemalca.

Strežnik, ki z `accept` pasivno čaka na zahtevo za vzpostavitev zveze enega od odjemalcev, lahko po želji pridobi njegov naslov, na primer:

```
sn = accept( sd, (struct sockaddr *)&naslovOdjemalca, &dolzinaNaslova );
```

ali pa se zanj ne zanima:

```
sn = accept( sd, NULL, NULL );
```

Seveda je oblika naslova in s tem v zvezi tip strukture drugega argumenta odvisen od domene. Prototip funkcije `accept` je:

```
#include <sys/socket.h>

int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

Funkcija vrne nov deskriptor ali -1 v primeru napake.

Funkcija `accept` se uporablja skupaj z vtičnicami tipa `SOCK_STREAM` ali `SOCK_SEQPACKET`. Z `accept` strežnik pasivno čaka na zahtevo za vzpostavitev zveze `connect` odjemalca. Klic vrne nov deskriptor nove vtičnice, na kateri strežnik komunicira z odjemalcem.

17.2.5 Funkcija `connect`

```
#include <sys/socket.h>
#include <netinet/in.h>  /* samo za AF_INET */
#include <sys/un.h>      /* samo za AF_UNIX */

int connect(int s, const struct sockaddr *addr, socklen_t addrlen);
```

Funkcija vrne 0 v primeru uspeha, -1 v primeru napake.

S funkcijo `connect` odjemalec skuša na vtičnici `s` vzpostaviti zvezo z oddaljenim procesom. Argument `addr` je kazalec na adresno strukturo vtičnice, ki mora vsebovati naslov oddaljene vtičnice, in `addrlen` je velikost te strukture. Tip adresne strukture je odvisen od domene, na primer za `AF_INET` domeno je `sockaddr_in` in za `AF_UNIX` je `sockaddr_un`.

Če na lokalno vtičnico v `AF_INET` domeni še ni pripet naslov, naslov ob uspelem `connect` izbere sistem sam.

17.2.6 Funkciji `getaddrinfo` in `getnameinfo`

Funkciji `getaddrinfo` in `getnameinfo` omogočata različne pretvorbe ali pač poizvedbe po imenih in naslovih. Sta zelo splošni, zato so možnosti uporabe številne, po drugi strani pa spričo univerzalnosti zahtevne. Prototip funkcije `getaddrinfo` je naslednji:

```
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo( const char *restrict Host,
                 const char *restrict Service,
                 const struct addrinfo *restrict hint,
                 struct addrinfo **restrict res ),
    Vrne: <> 0 v primeru napake, sicer 0.
```

Funkcija sprejme ali ime vozlišča ali ime storitve ali oboje. Če podamo kot argument samo eno ime, mora biti drugi argument NULL. Z 'imenom storitve' tu mislimo na simbolično ime komunikacijskih vrat. Definicije so zbrane v datoteki `/etc/services`. Klic funkcije povzroči DNS poizvedbo ter

pregledovanje relevantnih sistemskih datotek. Kot rezultat vrne kazalec na povezan seznam struktur tipa `addrinfo`. Definicija strukture je naslednja:

```
struct addrinfo {
       int              ai_flags;
       int              ai_family;
       int              ai_socktype;
       int              ai_protocol;
       socklen_t        ai_addrlen;
       struct sockaddr *ai_addr;
       char            *ai_canonname;
       struct addrinfo *ai_next;
};
```

Komponenta `ai_next` kaže na naslednjo strukturo v povezanem seznamu struktur. Ob klicu funkcije `getaddrinfo` lahko podamo argument `hint`, s katerim omejimo poizvedbo ali 'filtriramo' zadetke. Relevantne komponente strukture `addrinfo` so samo `ai_flags`, `ai_family`, `ai_protocol`, `ai_socktype`. Pomen komponent je samoumeven in ga razberemo iz imena, medtem ko komponenta `ai_flags` dodatno prilagodi poizvedbo. Ostale komponente morajo biti postavljene na vrednost nič in pa kazalci na NULL.

Funkcija `getnameinfo` je obratna funkcije `getaddrinfo`. Njen prototip pa je:

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *sa, socklen_t salen,
                char *host, size_t hostlen,
                char *serv, size_t servlen, int flags);
```

Naslednji primer ponazarja delovanje funkcije `getaddrinfo`. V ukazni vrstici podamo ime naprave ter po želji še ime storitve. Izvršitev programa izpiše informacijo o napravi in storitvi.

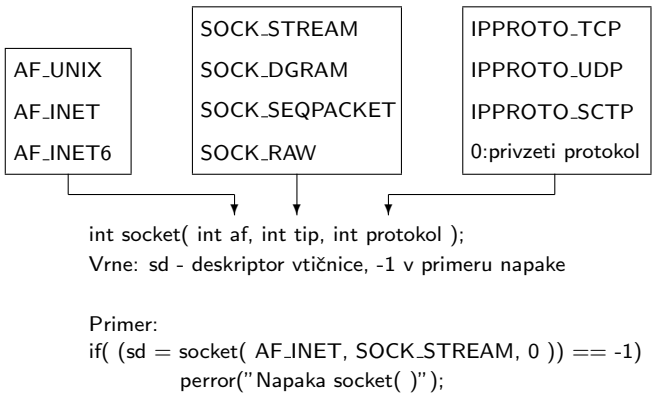
```
/* Primer poizvedbe po naslovu in st. vrat */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
```

```
#include <sys/socket.h>
#include <netdb.h>
#include <string.h>

#include <netinet/in.h>
#include <arpa/inet.h>

int main( int argc, char **argv )
{
  struct addrinfo hints;
  struct addrinfo *res;
  struct addrinfo *resp;
  struct sockaddr_in *addr;
  char   *serv;
  int err;

  if( argc == 2 ){
    serv = NULL;
  } else if( argc == 3){
    serv = argv[2];
  }else{
    printf("Uporaba: %s hostname servname\n", argv[0]);
    exit( 1 );
  }
  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_flags    = AI_CANONNAME; //zelimo pravo ime naprave
  hints.ai_addr     = NULL;
  hints.ai_next     = NULL;
  hints.ai_family   = AF_INET;      //zanima nas le IPv4 domena

  if( (err = getaddrinfo(argv[1], serv, &hints, &res)) != 0){
    printf("%s\n", gai_strerror( err ));
    exit(1);
  }
// tu gremo skozi povezan seznam odgovora funkcije
  for( resp = res; resp != NULL; resp = resp -> ai_next ){
    printf("%s: Host name   = %s\n", argv[0], argv[1]);
    printf("%s: Host cann   = %s\n", argv[0], resp -> ai_canonname);
```

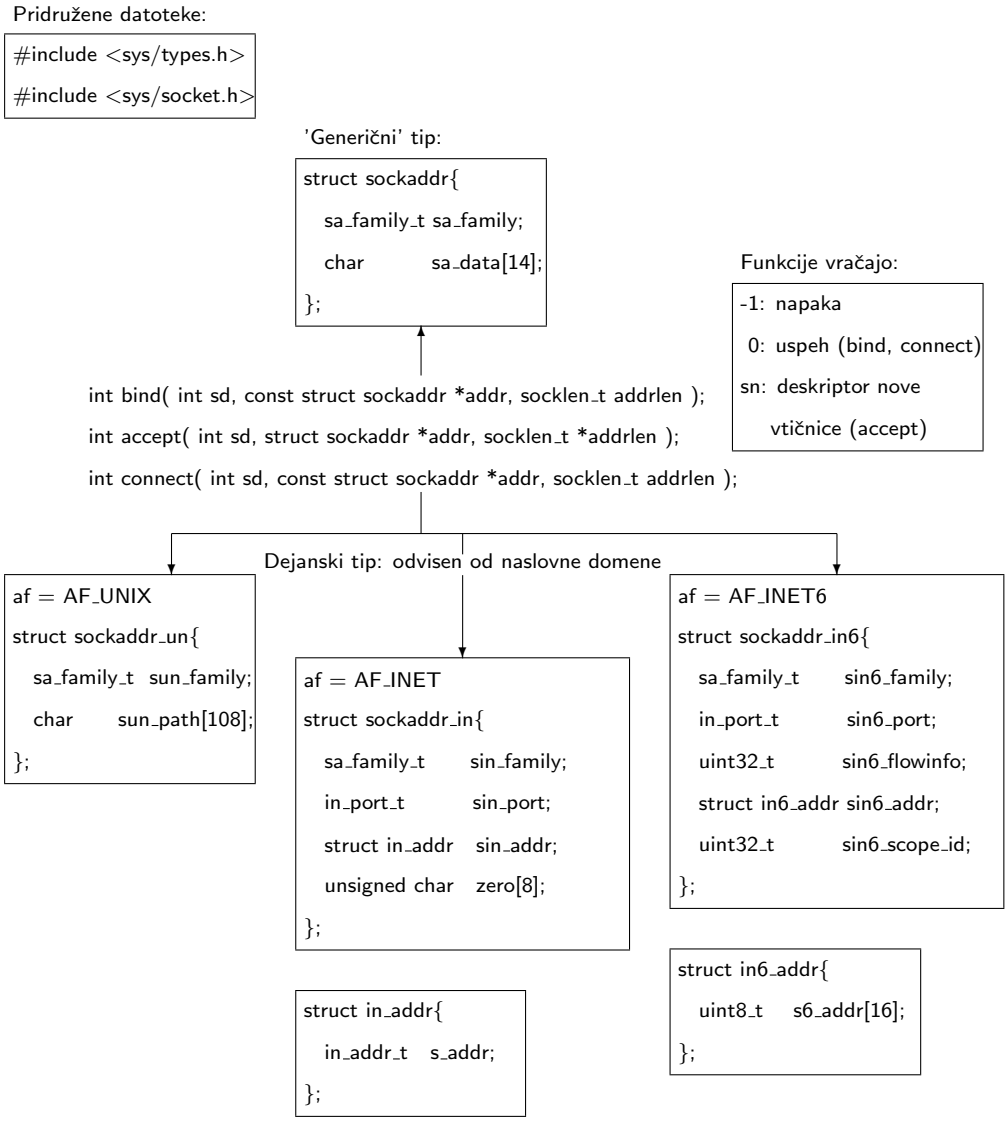
```
    printf("%s: Addr family = %d\n", argv[0], resp -> ai_family);
    printf("%s: Sock type   = %d\n", argv[0], resp -> ai_socktype);
    printf("%s: Sock prot   = %d\n", argv[0], resp -> ai_protocol);
    addr = (struct sockaddr_in *)resp -> ai_addr;
    printf("%s: Host addr   = %s\n", argv[0], inet_ntoa(addr -> sin_addr));
    printf("%s: Serv port   = %d\n", argv[0], ntohs(addr -> sin_port));
  }
  return 0;
}
```

17.2.7 Povzetek važnejših funkcij

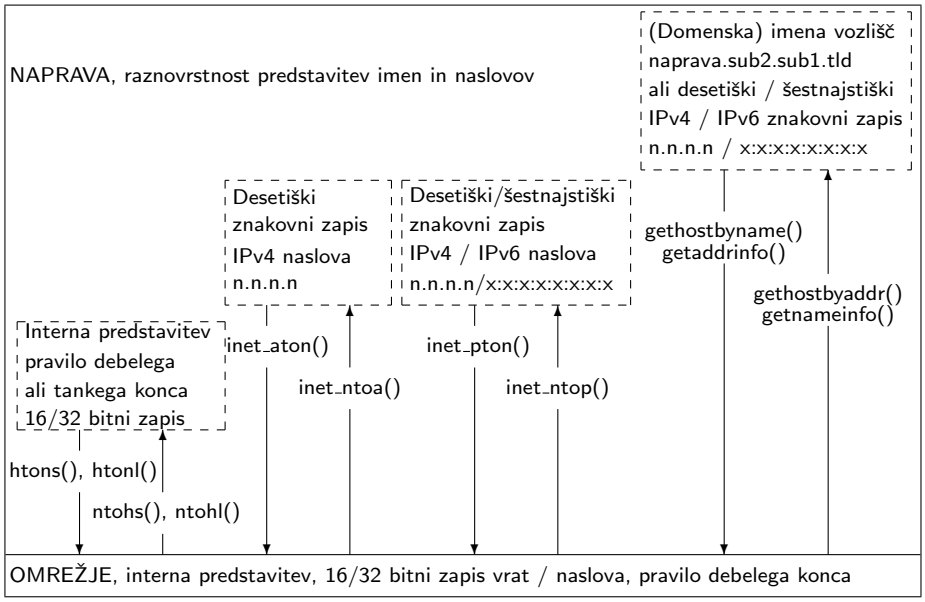
V tem poglavju so povzete nekatere funkcije programskega vmesnika komunikacijskih vtičnic, ki zaslužijo nekaj dodatne pozornosti. Zgoščen opis funckij naj služi kot opomnik neveščemu programerju pri uporabi funkcij.



Slika 17.12: *Zgoščen prikaz klica* `socket()`.



Slika 17.13: *Zgoščen prikaz klicev in podatkovnih struktur* `bind()`, `accept()` in `connect()`.



Slika 17.14: *Zgoščen prikaz preslikav med predstavitvami naslovov.*

17.2.8 Povezavna strežnik in odjamalec

V povezavnem načinu po načelu odjemalec / strežnik se pred dejanskim prenosom podatkov povezava najprej vzpostavi, nato se vzdržuje oziroma teče prenos podatkov v obe smeri, dokler se na koncu povezava ne sprosti oziroma 'zapre', slika 17.15. Strežnik odpre vtičnico, nanjo pripne svoje ime (IP naslov svojega vozlišča in svojo številko vrat), nakar gre z `listen` in `accept` v stanje pasivnega čakanja na zahtevo za vzpostavitev povezave s strani odjemalca. V omrežju se zaenkrat ne zgodi ničesar.

Povezavno obliko storitve (Angl. Connection-oriented service) transportnega sloja zagotavlja protokol TCP. Po protokolu TCP se povezava vzpostavi s tako imenovanim trikratnim usklajevanjem (Angl. Three-way handshake), ki začne na pobudo odjemalca za vzpostavitev povezave, torej kot posledica `connect`. Protokol TCP na stani odjemalca pošlje tako imenovani TCP SYN segment. Segment je podatkovna enota protokola TCP, ki ima v tem primeru v glavi segmenta postavljeno zastavico SYN. Protokol TCP

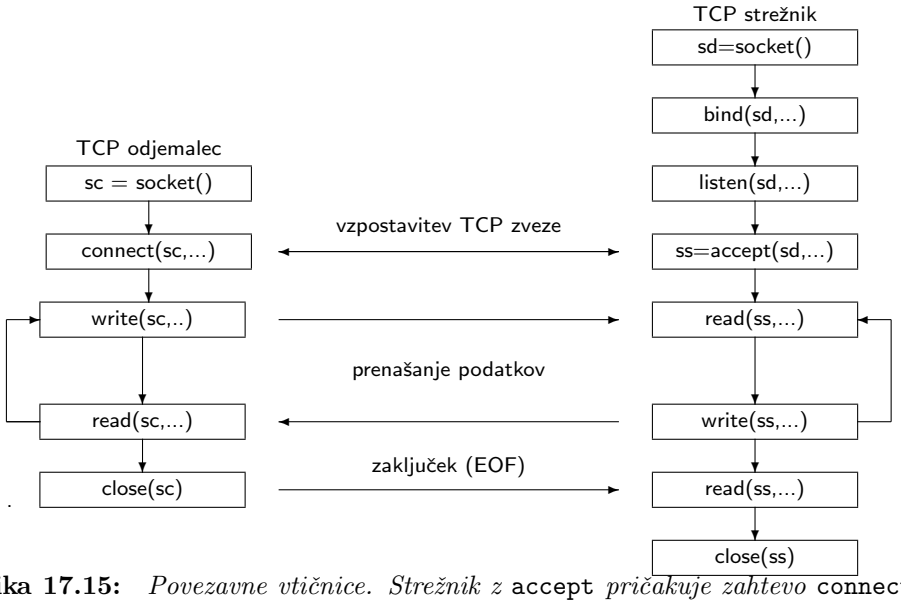
na strani strežnika kot posledica `accept` pristane na vzpostavitev povezave, zato v odgovor pošje TCP SYN segment, na katerega odjemalec odgovori s potrditvijo TCP ACK in povezava je vzpostavljena. S TCP ACK smo poimenovali TCP segment s postavljeno zastavico ACK (Angl. Acknowledgement) v glavi segmenta. Za vzpostavitev povezave so se prenesli trije segmenti TCP, od tu tudi ime 'trikratno usklajevanje'. Lahko bi rekli, da sta s tem odjemalec in strežnik sinhronizirana na oddajo in sprejem, ki poteka sočasno v obe smeri. V eni in drugi smeri teče prenos podatkov, a hkrati v obratni smeri potekajo potrditve. Podatki se prenašajo (oddajajo) kot vsebina TCP segmentov in če je v glavi segmenta postavljena zastavica ACK (angl. Acknowledge), je to hkrati potrditev sprejema podatkov iz druge strani.

Prenos podatkov se lahko zaključi na eni ali drugi strani, oziroma pobudo za sprotitev povezave lahko da ali odjemalec ali strežnik s klicem `close`. Ko na primer odjemalec zaključi oddajanje podatkov, napravi `close()`, kar povzroči oddajo segmenta TCP s postavljeno zastavico FIN (Angl. Final). Strežnik sprejme segment, odgovori s TCP FIN segmentom in naznani odjemalcu, da je konec sprejemanja podatkov. Enkrat kasneje tudi strežnik zapre vtičnico, TCP na strežnikovi strani odda FIN in odjemalčeva stran se odzove s segmentom FIN. Sproščanje povezave tako v celoti sestavljajo štirje segmenti TCP.

17.2.9 Primer TCP odjemalca in strežnika

Naslednja programa vsebujeta osnovne elemente 'strežnika' in 'odjemalca'. Komunikacija je povezavnega tipa. Povezava se vzpostavi, nato vzdržuje, dokler se na koncu ne podre. Podlago za komunikacijo daje protokol TCP. Strežnik na vratih 55000 čaka na vzpostavitev povezave odjemalca, dokler ne poteče z alarm nastavljeni čas (ena minuta). Ko se povezava z odjemalcem vzpostavi, strežnik s `fork` ustvari nov proces, ki sprejema podatke od odjemalca ter jih vrača kot 'odmev', dokler odjemalec s `close()` ne prekine povezave. Nazadnje povezavo zapre še strežnik s `close(sn)`.

Odjemalec je preprost. Najprej ustvari dostop do vtičnice in nato s `connect`



Slika 17.15: *Povezavne vtičnice. Strežnik z* `accept` *pričakuje zahtevo* `connect` *odjemalca za vzpostavitev zveze. Ko je zveza vzpostavljena, teče prenos podatkov v obe smeri, dokler se povezava ne zapre oziroma sprosti.*

skuša vzpostaviti povezavo z oddaljenim strežnikom. Po vzpostavitvi zveze bere vrstice besedila s tipkovnice dokler ni odtipkan kontrolni znak <CTRL>/D. Odtipkani vrstici doda predpono 'c->' in vrstico pošlje strežniku. Ko dobi odgovor strežnika, odgovor izpiše na zaslon. Povezava se prekine s `close( sd )`.

```
/* Povezaven - TCP streznik */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <string.h>

#define PORT_NUMBER 55000
```

```
int main( int argc, char **argv )
{
  struct sockaddr_in cliAddr;
  socklen_t size;
  char      buf[128];
  int       initServer( char * );
  int       sd, sn, n;

  if( argc != 2 ){
    printf("Uporaba: %s ime_naprave\n", argv[0]);
    exit( 1 );
  }
  if( (sd = initServer( argv[1] )) < 0){
    printf("Napaka: init server\n"); exit( 1 );
  }
  listen( sd, 5 );
  alarm( 60 ); /* koncaj po eni minuti */

  while( 1 ){
    size = sizeof(cliAddr);
    memset( &cliAddr, 0, size );

    if( (sn = accept(sd, (struct sockaddr *)&cliAddr, &size)) < 0){
      perror("accept err"); exit( 2 );
    }
    /* zveza je vzpostavljena, ustvari strezni proces */
    if( fork() == 0 ){
      printf("odjemalec: %s:%d\n",
        inet_ntoa( cliAddr.sin_addr ), ntohs( cliAddr.sin_port ) );
      while( (n = read( sn, buf, sizeof( buf ))) > 0 ){
    memcpy(buf,"s-> ",4);
        if( write(sn, buf, n) == -1)
          perror("write err");
      }
      printf("odjemalec: %s:%d je prekinil povezavo\n",
        inet_ntoa( cliAddr.sin_addr ), ntohs( cliAddr.sin_port ));
      close( sn );
      exit( 0 );
```

```
    }
  }
}

int initServer( char *hostName )
{
  struct sockaddr_in  servAddr;
  struct hostent     *host;
  int    sd;

  if( (host = gethostbyname( hostName )) == NULL)
    return( -1 );
  memset( &servAddr, 0, sizeof(servAddr) );
  memcpy( &servAddr.sin_addr, host->h_addr, host->h_length );
  servAddr.sin_family  = host->h_addrtype;
  servAddr.sin_port    = htons( PORT_NUMBER );

  printf("streznik: %s, ", host -> h_name);
  printf("%s:%d\n", inet_ntoa(servAddr.sin_addr), PORT_NUMBER );

  if( (sd = socket(AF_INET, SOCK_STREAM,0)) < 0 )
    return( -2 );
  if( bind(sd, (struct sockaddr *)&servAddr, sizeof( servAddr )) < 0)
    return( -3 );

  return( sd );
}


/* Povezaven - TCP odjemalec */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <string.h>
```

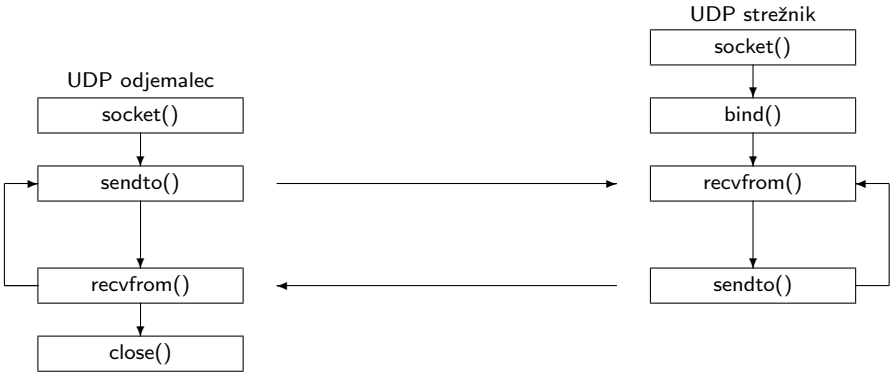
```
#define PORT_NUMBER 55000

int main( int argc, char **argv )
{
  int    n, sd, initClient( char * );
  char   buf[128];

  if( argc != 2 ){
    printf("Uporaba: %s ime_streznika\n", argv[0]);
    exit( 1 );
  }
  if( (sd = initClient( argv[1] )) < 0 ){
    printf("napaka init\n");  exit( 1 );
  }
  else{
    printf("tipkaj karkoli, ^D za konec\n");
    while( fgets(&buf[4], sizeof(buf), stdin ) != NULL ){
      memcpy(buf,"c-> ",4);
      printf("%s", buf);
      if( write(sd, buf, strlen(buf)) == -1)
        perror("write err");
      if( (n = read(sd, buf, sizeof(buf))) == -1)
        perror("read err");
      buf[n] = 0;
      printf("%s", buf);
    }
    close( sd );
  }
  exit( 0 );
}

int initClient( char *hostName )
{
  struct sockaddr_in  servAddr;
  struct hostent     *host;
  int    sd;
```

```
  if( ( host = gethostbyname( hostName )) == NULL)
    return( -1 );
  memset( &servAddr, 0, sizeof(servAddr));
  memcpy( &servAddr.sin_addr, host->h_addr, host->h_length );
  servAddr.sin_family = host->h_addrtype;
  servAddr.sin_port   = htons( PORT_NUMBER );

  printf("streznik: %s, ", host -> h_name);
  printf("%s:%d\n", inet_ntoa( servAddr.sin_addr ), PORT_NUMBER);
  if( (sd = socket(AF_INET,SOCK_STREAM,0)) < 0 )
    return( -2 );
  if( connect(sd, (struct sockaddr *)&servAddr,sizeof(servAddr)) < 0)
    return( -3 );

  return( sd );
}
```

17.2.10 Primer UDP odjemalca in strežnika

V prejšnjem razdelku smo si ogledali vtičnice povezavnega tipa. V nasprotju s povezavno usmerjenimi vtičnicami, temelječimi na protokolu TCP, pri katerih se povezava najprej ustvari, nato pa vzdržuje dokler se na koncu povezava ne sprosti, pa pri nepovezavnih vtičnicah do povezave med odjemalcem in strežnikom sploh ne pride. Podlago nepovezavnim vtičnicam daje protokol UDP. Vsaka transakcija med odjemalcem in strežnikom je zaključena celota in je neodvisna od drugih transakcij. Komunikacija poteka bolj hitro, je pa manj zanesljiva.

Slika 17.16 prikazuje nepovezaven koncept komunikacije odjemalca s strežnikom. Odjemalec se ne 'poveže' s strežnikom. Namesto `connect` enostavno pošlje podatke s `sendto` na naslov strežnika. Podobno tudi strežnik ne pričakuje oziroma ne čaka na vzpostavitev zveze z `accept`, temveč z `recvfrom` enostavno čaka na podatke enega od odjemalcev. Funkcija skupaj s sprejetimi podatki 'vrne' tudi naslov odjemalca.



Slika 17.16: *Nepovezavne vtičnice prek protokola UDP. Strežnik z* `recvfrom` *pričakuje podatke od potencialnega odjemalca in se po potrebi s* `sendto` *odzove.*

Funkciji `recvfrom` in `sendto` sta precej podobni standardnim funkcijam `read/write` in sta povzeti spodaj:

```
#include <sys/socket.h>

ssize_t recvfrom( int sockfd, void *buff, size_t nbajt, int flags,
                  struct sockaddr *from, socklen_t *addrlen );

ssize_t sendto( int sockfd, void *buff, size_t nbajt, int flags,
                  struct sockaddr *to, socklen_t addrlen );
```

Obe vrneti število poslanih ali sprejetih bajtov, ali pa -1 primeru napake. Prvi trije argumenti teh funkcij imajo podoben pomen kot pri funkcijah `read/write`: deskriptor vtičnice, kazalec na pomnilnik, kjer so ali bodo podatki ter dejansko število podatkov za pošiljanje ali pričakovano število bajtov za sprejem. Argument `flags` po potrebi spremeni delovanje funkcij in ker tega običajno ne želimo, je njegova vrednost navadno nič. Zadnja dva argumenta vsebujeta naslov, od kjer se sprejema ali kamor se pošilja ter dolžino naslova.

V nadaljevanju sta ilustrirana preprosta UDP strežnik in UDP odjemalec. Strežnik ustvari vtičnico v protokolovni družini Internet (AF_INET) tipa datagram (SOCK_DGRAM). Nato formira naslovno podatkovno strukturo s številko UDP vrat 50000 ter s svojim IP naslovom (INADDR_ANY) ozi-

roma vsemi svojimi naslovi, če bi jih bilo več. Nato naslov z `bind` pripne na vtičnico. Zatem z `recvfrom` pričakuje podatke odjemalca. Ko ti pridejo, izpiše omrežni naslov odjemalca ter številko vrat, spremeni prve štiri črke sprejetega niza in niz vrne nazaj odjemalcu. Strežnik torej enostavno vrne 'odmev' s spremenjenim začetkom odmeva. Tako se ve, da je odmev res vrnil strežnik.

Argument flags je nič, s čimer pustimo običajen način delovanja oddajne in sprejemne funkcije.

```
/* Nepovezaven UDP streznik */
#include <stdio.h>
#include <stdlib.h>

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <string.h>

#define MAXLINE     4096
#define SERVPORTNO 50000

int main( void )
{
  int sockfd,n;
  struct sockaddr_in servaddr,cliaddr;
  socklen_t len;
  char mesg[MAXLINE];

  if( (sockfd=socket(AF_INET,SOCK_DGRAM,0)) == -1){
    perror("socket err");
    exit(1);
  }
  bzero(&servaddr,sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
  servaddr.sin_port=htons( SERVPORTNO );
```

```
  if( bind(sockfd,(struct sockaddr *)&servaddr,sizeof(servaddr)) == -1){
    perror("bind err");
    exit(1);
  }
  while( 1 ){
    len = sizeof(cliaddr);
    n = recvfrom(sockfd,mesg,MAXLINE,0,(struct sockaddr *)&cliaddr,&len);
    printf("client: %s:%d\n", inet_ntoa( cliaddr.sin_addr ),
                              ntohs( cliaddr.sin_port));
    memcpy(mesg,"s-> ",4);
    sendto(sockfd,mesg,n,0,(struct sockaddr *)&cliaddr,sizeof(cliaddr));
  }
}
```

Tudi odjemalec je enostaven. Odjemalec v ukazni vrstici pričakuje desetiški zapis IPv4 naslova strežnika. Nato formira naslov vtičnice strežnika in s tipkovnice pričakuje znakovni niz. Znakovnemu niz na začetku doda 'c ->' in ga pošlje strežniku ter pričakuje njegov odgovor. Ko odgovor pride, ga izpiše na zaslon.

```
/* Nepovezaven UDP odjemalec */
#include <stdio.h>
#include <stdlib.h>

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <string.h>

#define MAXLINE 4096
#define SERVPORTNO 50000

int main(int argc, char **argv)
{
  int sockfd,n;
  struct sockaddr_in servaddr;
  char sendline[ MAXLINE ], recvline[ MAXLINE ];
```

```
  if (argc != 2){
    printf("usage: udpcli <IP address>\n");
    exit(1);
  }
  if( (sockfd=socket(AF_INET,SOCK_DGRAM,0)) == -1){
    perror("socket err");
    exit(1);
  }
  bzero(&servaddr,sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr=inet_addr(argv[1]);
  servaddr.sin_port=htons(SERVPORTNO);

  while (fgets(&sendline[4], MAXLINE,stdin) != NULL){
    memcpy(sendline,"c-> ", 4);
    fputs(sendline,stdout);
    n = sendto(sockfd,sendline,strlen(sendline),0,
            (struct sockaddr *)&servaddr,sizeof(servaddr));
    if( n == -1 ){
      perror("sendto err");
      exit(1);
    }
    n=recvfrom(sockfd,recvline,MAXLINE,0,NULL,NULL);
    if( n == -1 ){
      perror("recvfrom err");
      exit(1);
    }
    recvline[n]=0;
    fputs(recvline,stdout);
  }
  exit( 0 );
}
```

17.2.11 Primer odjemalca in strežnika v domeni UNIX

Naslednja dva programa realizirata komunikacijo po načelu odjemalec-strežnik v komunikacijski domeni UNIX. Strežnik čaka na zahtevo odjemalca ter ob vzpostavitvi zveze ustvari nov proces, ki izpisuje sprejeto sporočilo, dokler odjemalec ne zapre kanala. Ko zvezo vzpostavijo trije odjemalci, strežnik konča.

Odjemalec vzpostavi zvezo s strežnikom in mu pošilja sporočila, ki jih bere s standardne vhodne datoteke.

```
/* Socket streznik v UNIX domeni */
#include <stdio.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <sys/errno.h>

int main( int argc, char **argv )
{
  int     SocketNumber, NewSocketNumber;
  int     Len, Nodjemalcev = 0;
  struct  sockaddr_un  ServerAddress, ClientAddress;
  char    Message[512];
  int     Nread;

  if( (SocketNumber =   socket( AF_UNIX, SOCK_STREAM, 0 )) == -1){
    printf("%s: napaka socket\n", argv[0]);
    exit(1);
  }
  ServerAddress.sun_family = AF_UNIX;
  strcpy( ServerAddress.sun_path, "sockname");
  unlink(ServerAddress.sun_path); /* za vsak slucaj */

  if(bind( SocketNumber, &ServerAddress, sizeof(ServerAddress)) == -1){
    printf("%s: napaka bind\n", argv[0]);
    unlink(ServerAddress.sun_path);
    exit(1);
  }
```

```
  if( listen( SocketNumber, 5 ) == -1){
    printf("%s: napaka listen\n", argv[0]);
    exit(1);
  }
  while( Nodjemalcev++ < 3){ /* po treh vzpostavitvah zveze koncam */
    Len = 128;
    if( (NewSocketNumber=accept( SocketNumber, &ClientAddress, &Len)) >= 0){
      if( fork( ) == 0 ){
    while( (Nread = read(NewSocketNumber, Message, 100)) > 0){
      Message[ Nread ] = 0;
      printf("Sprejel: %s\n", Message );
    }
    close( NewSocketNumber );
    exit( 0 );
      }
    }
    else{
      printf("%s: napaka accept\n", argv[0] );
      exit( 2 );
    }
  }
  unlink(ServerAddress.sun_path);
  exit(0);
}
```

```
/* Socket Odjemalec v UNIX domeni */
#include <stdio.h>
#include <sys/socket.h>
#include <sys/un.h>

main( int argc, char **argv )
{
  int     SocketNumber;
  struct  sockaddr_un    ClientAddress, ServerAddress;
  char    Message[512];

  if( (SocketNumber =  socket( AF_UNIX, SOCK_STREAM, 0 )) == -1){
    printf("Error on socket\n");
    exit(1);
```

```
  }
  ServerAddress.sun_family = AF_UNIX;
  strcpy( ServerAddress.sun_path, "sockname");

  if( connect( SocketNumber, &ServerAddress, sizeof(ServerAddress) ) == -1){
    printf("%s: napaka connect\n", argv[0]);
    exit(2);
  }
  while(1){
    printf("Vnesi sporocilo: ");
    gets( Message );
    if( strlen( Message ) == 0){
      close( SocketNumber );
      exit(0);
    }
    write(SocketNumber, Message, strlen(Message));
  }
}
```

17.3 Nadaljnje čtivo

O zgradbi in delovanju komunikacijskih omrežij je dosti dobrih knjig. Priporočamo predvsem [1] ali [2] ter [3] za TCP/IP.

Za tiste, ki bi se želeli poglobiti v programiranje omrežnih aplikacij priporočamo [5, 4].

Literatura

[1] A. S. Tanenbaum, D. J. Wetherall, *Computer Networks, 5th ed.*, Pearson, Prentice Hall, 2011.

[2] J. F. Kurose, K. W. Ross, *Computer Networking, A top-down approach, 5-th ed.*, Pearson, Addison-Wesley, 2010.

[3] W.R. Stevens, *TCP/IP Illustrated, Vol. 1*, Addison-Wesley, 1994.

[4] W. R. Stevens, B. Fenner, A. M. Rudoff, *UNIX Network Programming, Vol. 1, 3rd ed.*, Addison-Wesley, 2004.

[5] W. R. Stevens, *UNIX Network Programming, Vol. 2, 2nd ed.*, Prentice Hall, 1999.

Poglavje 18

Razvrščanje opravil in Linux

V tem poglavju bomo obravnavali razvrščanje opravil v sistemu Linux. Ogledali si bomo tradicionalen pristop k delitvi procesorskega časa med večje število opravil, ki je v sistemih UNIX/Linux pristoten že od nastanka sistema. Srečali bomo *prijaznost* procesa in vrednost `nice`. Spoznali bomo sistemske funkcije, s katerimi vplivamo na delež procesorskega časa, ki ga dobi posamezen proces.

Sledil bo opis razvrščevalnika Linux v novejših sistemih in vmesnika POSIX za razvrščanje opravil v realnem času. Razložili bomo pravila, prioritete in algoritme razvrščanja ter sistemske funkcije za upravljanje prioritet. Za uvod pa bomo ponovili glavne lastnosti sistema Linux, ki so neposredno povezane z razvrščevalnikom opravil.

18.1 Nekaj lastnosti sistema Linux

Linux je večopravilni sistem. Zato v danem obdobju v sistemu sočasno obstaja več procesov. Linux je tudi večniten. V okviru istega procesa lahko zato hkrati obstaja več niti.

Razvrščevalnik sistema Linux obravnava niti enako kot procese. Tako za procese kot niti veljajo enaka pravila razvrščanja. Procese in niti bomo

zato poimenovali opravila. S stališča razvrščevalnika ni nobene razlike, ali je opravilo ena od niti večnitnega procesa ali je opravilo (enonitni) proces. Na proces je zato moč gledati tudi kot na skupino niti in na nit kot osnovno razvrstljivo dejavnost oziroma opravilo. Uporabniška vmesnika za niti in procese se seveda razlikujeta.

Linux realizira prioritetno razvrščanje opravil. Opravilo z višjo opravilo bo vedno prej na vrsti kot opravilo z nižjo prioriteto. Linux realizira različna pravila razvrščanja. Pravilo razvrščanja je odvisno od tipa opravila. Opravila po tipu delimo na:

- opravila realnega časa,
- interaktivna opravila in
- paketna opravila.

Opravila realnega časa imajo prioritete realnega časa in se razvrščajo po pravilih, ki so primerna za realni čas. Ker so časovno kritična, imajo vedno prednost pred interaktivnimi in paketnimi opravili. Interaktivna in paketna opravila niso tako zelo časovno kritična ter se obravnavajo v okviru navadnih prioritet. A med interaktivnimi in paketnimi opravili obstaja pomembna razlika. Paketna opravila so praviloma računsko intenzivna in za napredovanje potrebujejo procesor. Med računanjem večino časa ne rabijo interakcije z okoljem. Po drugi strani interaktivna opravila ne zahtevajo veliko procesorskega časa, so pa zahtevna v pogledu interakcije z okoljem in potrebujejo hiter odziv. Interaktivni procesi zahtevajo pogostejšo, čeprav kratkotrajno, pozorost procesorja.

Linux je predopravilni sistem (Angl. Preemptive multitasking). To pomeni, da lahko opravilu, ki se izvaja, procesno enoto prevzame eno izmed pripravljenih opravil z višjo prioriteto. Prioritetni predopravilni sistem zagotavlja, da bo v sistemu vedno napredovalo opravilo z najvišjo priporiteto. To je za sisteme realnega časa bistvenega pomena.

Linux daje podporo večprocesorskim oziroma večjedrnim sistemom po načelu simetričnega multiprocesiranja (Angl. SMP - Symmetric multiprocessing). To pomeni, da so vsa procesorska jedra in vsi procesorji obravnavani

enakovredno.

Če ima sistem več procesorskih jeder in morebiti tudi več procesorjev, se lahko v danem obdobju resnično sočasno izvaja večje število procesov in niti. Tedaj govorimo o paralelnem oziroma vzporednem napredovanju opravil (Angl. Parallel processing).

18.2 Krožno razvrščanje procesov in vrednost `nice`

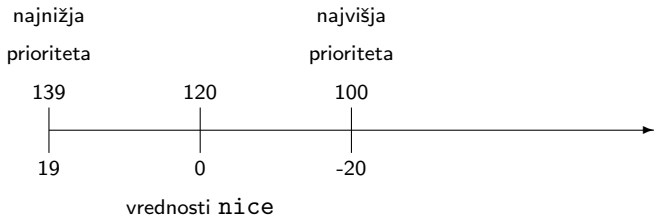
Standarden način razvrščanja v sistemih UNIX in Linux je že od nekdaj krožno razvrščanje z delitvijo časa (Angl. Round robin time sharing). V tem načinu dobi vsako opravilo sorazmeren delež procesorskega časa. Procesorski čas je deljen na kratke intervale ali časovne rezine. Nobeno od opravil se ne more polastiti procesorja za dlje kot to določa časovna rezina. Opravila se vrstijo krožno, drugo za drugim. Način je pravičen in ob primerni izbiri trajanja rezine tudi primerno odziven.

Navadna opravila, to so tista, ki niso realnočasovna, so v sistemu Linux razvrščena po prioritetah od 100 do 139. Prioritetni nivo 100 je najvišji in 139 najnižji. Privzeta prioriteta navadnih opravil je 120. Na delež procesorskega časa, ki ga je deležno posamezno opravilo, lahko uporabnik vpliva prek vrednosti `nice`. Prioriteti 120 ustreza parameter `nice` z vrednostjo nič. Tej prioriteti bi zato lahko rekli tudi nevtralna prioriteta. Vrednosti `nice` ležijo na intervalu od -20 do +19, kot prikazuje slika 18.1. Višja vrednost pomeni nižjo prioriteto. Od tu tudi ime parametra. Opravilo z višjo vrednostjo `nice` je bolj 'prijazno' do drugih opravil. Visoka vrednost parametra je primerna za dolgotrajna, računsko intenzivna opravila, ki nimajo posebnih časovnih zahtev. Privzeto vrednost je mogoče spreminjati navzgor ali navzdol. Povečati `nice` in posledično zmanjšati prioriteto je dovoljeno vsakemu opravilu. Zmanjšati `nice` in posledično zvišati prioriteto je dovoljeno le priviligiranim opravilom.

Vrednost `nice` pravzaprav niti ni prioriteta. Prek te vrednosti je namreč moč vplivati na *delež* procesorskega časa, ki ga razvrščevalnik dodeli posameznemu opravilu. Opravilo z višjo vrednostjo `nice` in torej z nižjo priori-

teto bo dobilo relativno manjši delež procesorjevega časa in bo posledično napredovalo počasneje od drugih opravil.

Prioriteto navadnih opravil določi in spreminja uporabnik s parametrom `nice`. Vrednost `nice` dedujejo otroci, ki nastanejo s `fork` in se ohranja tudi prek `exec`. A na prioriteto navadnih opravil vpliva razvrščevalik tudi sam. Zato rečemo, da so prioritete navadnih procesov oziroma opravil *dinamične*. Razvrščevalnik vzame za izhodišče *statično prioriteto* in to spremeni navzgor ali navzdol. S spreminjanjem prioritete želi zagotoviti 'popolnoma pravično' delitev procesorja med pripravljena opravila. Pri tem skuša zagotoviti dobro odzivnost interaktivnim opravilom in primerno hitrost napredovanja računsko zahtevnim opravilom. Spričo tega je algoritem poimenovan CFS (Angl. Completely Fair Scheduler).



Slika 18.1: *Prioritete navadnih opravil in parameter* `nice`*. Interval navadnih prioritet sega od 100 do 139. 'Nevtralna' vrednost parametra* `nice` *je nič in se preslika v prioriteto 120. Vrednosti* `nice` *gredo od +19 (najnižja prioriteta) do -20 (najvišja prioriteta).*

Pri krožnem razvrščanju z delitvijo časa se čas praviloma deli na enako dolge rezine. V sistemu Linux ni tako. Trajanje časovne rezine (kvanta) ni konstantno in ni enako za vsa opravila. Dolžina kvanta je odvisna od statične prioritete (SP) in je določena po naslednjem pravilu:

$$kvant = \begin{cases} (140 - SP) \times 20, & SP < 120 \\ (140 - SP) \times 5, & SP \geq 120 \end{cases}$$

Po tem pravilu razvrščevalnik nameni več procesorjevega časa pomembnejšim opravilom. Nekaj primerov časovnih rezin prikazuje naslednja tabela.

	Statična prioriteta (SP)	Vrednost nice	kvant [ms]
Najvišja prioriteta	100	-20	800
Višja prioriteta	110	-10	600
Normalna prioriteta	120	0	100
Nižja prioriteta	130	+10	50
Najnižja prioriteta	139	+19	5

Da dobijo važnejša opravila več procesorjevega časa, ne more biti sporno. Kadar so v sistemu samo računsko intenzivna opravila, se zdi samoumevno, da pomembnejša opravila napredujejo hitreje, za kar potrebujejo večji delež procesorja in to jim zagotavlja višja prioriteta. A v večini sistemov sočasno obstajajo računsko intenzivna in interaktivna opravila. Interaktivna opravila ne potrebujejo veliko procesorskega časa, pričakujejo pa primeren odziv, za kar potrebujejo višjo prioriteto. Vezati dolžino kvanta direktno na prioriteto se zdi prav s tega stališča napačno. Dodatna težava je tudi v tem, da razvrščevalnik vnaprej ne ve, katera opravila so interaktivna in katera paketna. Zato je potrebna korekcija prioritete, pa tudi mehanizem, s katerim je moč prepoznati interaktivno ter paketno naravo opravil.

Razvrščevalnik sistema Linux naredi korekcijo statične prioritete na podlagi relativnega povprečnega časa, ki ga opravilo prebije v stanju čakanja na procesno enoto. Popravek se imenuje *bonus* in je izračunan na podlagi časa, ki ga opravilo prebije v stanju ustavljen, zmanjšanem za čas, ko se opravilo izvaja. Daljši je ta čas, večji je *bonus*. Bonus je število z intervala [0..10] ter vpliva na dinamično prioriteto (DP) procesa po naslednjem pravilu,

$$DP = \max(100, \min(SP - bonus + 5, 139))$$

Vrednost pet za *bonus* je nevtralna, $bonus > 5$ dvigne prioriteto in $bonus < 5$ jo zniža. Hkrati je *bonus* podlaga, da se dotično opravilo obravnava bodisi kot interaktivno bodisi kot paketno opravilo. Za interaktivna opravila velja naslednje pravilo,

$$bonus - 5 \geq SP/4 - 28.$$

Desna stran neenačbe se imenuje 'interaktivni delta'. Večja je vrednost statične prioritete SP, kar pomeni nižjo prioriteteo, težje postane opravilo

interaktivno. Razlog je očiten. Interaktivna opravila, ki zahtevajo hiter odziv, potrebujejo višjo prioriteto. Po drugi strani interaktivna opravila večino časa ne potrebujejo procesne enote in so v stanju čakanja. Zato se jim dinamična prioriteteta povečuje. Če povzamemo, opravilo, ki dosti čaka in malo obremenjuje procesor, pridobiva na prioriteti.

Sedaj pa si oglejmo osnovne sistemske funkcije za upravljanje s prioritetami navadnih procesov.

18.2.1 Funkciji `getpriority` in `setpriority`

Funkcija `getpriority()` vrne in `setpriority()` nastavi prioriteto oziroma vrednost `nice` izbranih procesov, ali -1 v primeru napake.

```
#include <sys/time.h>
#include <sys/resource.h>

int getpriority(int which, int who);
int setpriority(int which, int who, int prio);
```

Vrednost argumenta 'who' je odvisna od vrednosti argumenta 'which' kot sledi:

- 'which' je `PRIO_PROCESS` in 'who' je številka procesa,
- 'which' je `PRIO_PGRP` in 'who' je številka skupine,
- 'which' je `PRIO_USER` in 'who' je številka uporabnika.

Vrednost nič za 'who' pomeni številko klicnega procesa, klicne skupine ali klicnega uporabnika. Vrednost argumenta 'prio' je v območju od -20 do 19, privzeta vrednost pa je nič.

Klic `getpriority()` vrne najmanjšo vrednost (najvišjo prioriteto) izmed s parametrom 'which' in 'who' izbranih procesov.

Klic `setpriority()` nastavi novo vrednost prioritete z 'which' in 'who' izbranih procesov. Samo priviligirani uporabnik lahko zviša prioriteto (torej zmanjša vrednost).

Denimo, naslednji program sam sebi ali drugemu procesu istega lastnika (drugi argument ukazne vrstice) spremeni prioriteto za privzeto vrednost 1 ali za vrednost, ki jo podamo s tretjim argumentom ukazne vrstice.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <errno.h>

#define  PRIO_INC   1  //privzeto povecanje nice

int main( int argc, char **argv )
{
  int prio_get, prio_set, prio_inc;
  pid_t pid;

  prio_inc = PRIO_INC;
  pid      = getpid( );
  switch( argc ){
    case 3: prio_inc = atoi( argv[2] );
    case 2: pid      = atoi( argv[1] );
  }
  errno = 0;
  prio_get = getpriority( PRIO_PROCESS, pid );
  if (prio_get == -1 && errno != 0){
    perror("getpriority err");
    exit(EXIT_FAILURE);
  }
  printf("Sedanja prioriteta: %d\n", prio_get ); /* verjetno 0 */
  prio_set = prio_get + prio_inc;  /* spremenimo vrednost nice */
  if (setpriority( PRIO_PROCESS, pid, prio_set ) == -1){
    perror("setpriority err");
  }
  errno = 0;
  prio_get = getpriority( PRIO_PROCESS, pid );
  if (prio_get == -1 && errno != 0){
```

```
    perror("getpriority err");
    exit(EXIT_FAILURE);
  }
  printf("Nova prioriteta: %d\n", prio_get );
  exit(EXIT_SUCCESS);
}
```

18.2.2 Funkcija `nice`

```
#include <unistd.h>
int nice(int inc);
```

Funkcija `nice()` prišteje vrednost argumenta `inc` k vrednosti `nice` procesa. Večja vrednost pomeni nižjo prioriteto. Samo priviligirani uporabnik lahko poda negativno vrednost argumenta in s tem zviša prioriteto. Če klic uspe, vrne novo vrednost `nice`, torej novo prioriteto procesa. Vrednost `nice`, ne da bi jo spremenili, lahko preverimo s klicem

```
moj_nice = nice( 0 );
```

Funkcija `nice` bi bila spričo predhodnih dveh funkcij odveč, je pa enostavnejša za uporabo.

Obstaja tudi sistemski ukaz z enakim imenom, to je ukaz `nice`. Z ukazom `nice` se da vrednost `nice` procesu nastaviti ob izvršitvi programa, a le priviligirani uporabnik lahko poda tudi negativno vrednost. Na primer, s spodnjo ukazno vrstico izvršimo program `mproc` z za 10 znižano prioriteto.

```
nice -n 10 mproc
```

Obstaja še sistemski ukaz `renice`. Z njim je moč procesu spremeniti vrednost `nice`, a je to spet dovoljeno le uporabniku s primernimi pooblastili.

18.3 Razvrščanje opravil realnega časa

V sistemih, v katerih prevladujejo interaktivne aplikacije v kombinaciji z računsko intenzivnimi, a ne realnočasovnimi aplikacijami, ustvarja standardni krožni način razvrščanja primerno okolje za napredovanje opravil. V

sistemih s strožjimi časovnimi zahtevami pa razvrščanje opravil na osnovi vrednosti `nice` in principa enakovrednih možnosti ni dovolj.

Razvrščanje opravil v novejših verzijah jedra Linux temelji na prioritetnem razvrščanju. V nasprotju z načelom pravičnosti in razvrščevalnikom CFS, prioritetno razvrščanje v osnovi ni 'pravično'. Prioritetni način razvrščanja s predopravilnostjo zagotavlja, da se v vsakem trenutku izvaja opravilo z najvišjo prioriteto. Zato opravila z nižjo prioriteto ne bodo dobila možnosti za napredovanje, dokler obstaja opravilo z višjo prioriteto.

Sodobni razvrščevalnik sistema Linux je zasnovan modularno. Ker je zasnovan modularno, omogoča razvrščanje opravil ne le po enem pravilu, temveč po več različnih pravilih (Angl. Scheduling classes, policies). Eno od pravil razvrščanja je pravilo razvrščanja navadnih opravil. Pravilo razvrščanja navadnih opravil, torej nerealnočasovnih, je pravilo CFS in smo ga spoznali v predhodnem podpoglavju. Linux pa daje podlago tudi razvrščanju opravil v realnem času.

Opravila realnega časa se razvrščajo po pravilih, ki so primerna za opravila realnega časa. Vsa opravila realnega časa imajo absolutno prednost pred opravili navadnih prioritet. Prioriteta opravila realnega časa je določena s številom z intervala [0..99]. Jedro Linux obravnava vrednost nič kot najvišjo in vrednost 99 kot najnižjo prioriteto realnega časa. To je skladno s prioritetnimi nivoji navadnih opravil, ki jemljejo prioritete z intervala [100..139]. Iz povedanega sledi, da ima Linux 140 prioritetnih nivojev, kot ponazarja slika 18.2.

Sistem Linux daje podporo aplikacijam realnega časa na podlagi vmesnika API, ki je skladen s specifikacijo POSIX. Sama specifikacija POSIX in skladnost sistema s specifikacijo POSIX sicer ne zahteva podpore aplikacijam s strogimi časovnimi zahtevami (Angl. Hard real-time applications). Novejše in primerno konfigurirane verzije jedra Linux s predopravilnostjo v jedru pa zagotavljajo dobro in predvidljivo odzivnost, čeprav mogoče ne ravno za najzahtevnejše aplikacije realnega časa. Koncept skrajnih rokov za izvršitev opravil namreč ni vgrajen v jedro.

Slika 18.2: *Linux pozna 140 prioritetnih nivojev. Interval prioritet sega od 0 do 139. Nič je najvišja in 139 najnižja prioriteta. Prioritete realnega časa so od 0 do 99 in jih vmesnik API za realni čas preslika na interval od 99 do 1. Navadne prioritete so od 100 do 139. Če jim odštejemo 100, dobimo 'uporabniške' prioritete na intervalu od 0 do 39. Vrednost* `nice` *je od 19 do -20.*

Čeprav jedro obravnava številke prioritet tako, da manjša vrednost pomeni višjo prioriteto, pa vmesnik API preslika prioritete realnega časa na interval [1..99], tako da večja vrednost pomeni višjo prioriteto. Vrednost 99 potem pomeni najvišjo prioriteto.

Razpon prioritet realnega časa je torej 1 do 99, a se na to ne gre zanašati. Minimalno in maksimalno vrednost prioritete dobimo s funkcijama

```
minprio = sched_get_priority_min( pravilo_razvrstitve )

maxprio = sched_get_priority_max( pravilo_razvrstitve )
```

Višja vrednost pomeni višjo prioriteto. Torej ima opravilo s prioriteto 99 prednost pred vsemi opravili.

Razvrščevalnik zagotavlja, da se od pripravljenih opravil vedno izvaja opravilo z najvišjo prioriteto. Pravilo razvrščanja in prioriteto opravila nastavimo s funkcijo `sched_setscheduler`. Prioriteta opravil realnega časa mora biti višja od nič. Vsa realnočasovna opravila imajo prednost pred navadnimi opravili. V splošnem ima lahko več opravil enako prioriteto. Kako si realnočasovna opravila z enako prioriteto delijo procesor določa pravilo razvrščanja (Angl. Scheduling policy ali Scheduling class). Obstajati dve pravili razvrščanja realnočasovnih opravil,

- `SCHED_FIFO`: prvi pride prvi strežen,
- `SCHED_RR`: krožo razvrščanje.

V obeh primerih so pripravljena opravila urejena po prioritetah v tabeli prioritet. Zato čakalna vrsta pripravljenih opravil dejansko sploh ni vrsta, temveč tabela. Ker je skupaj z navadnimi prioritetami 140 prioritetnih nivojev, vsebuje tabela 140 komponent. Komponente kažejo na začetek dvojno povezanega seznama opravil z enako prioriteto. Tedaj, ko opravilo nastane (Angl. Release time), se ga uvrsti na konec seznama. Klic funkcije `sched_setscheduler()` uvrsti izbrano opravilo na začetek seznama. Samo opravilo na začetku seznama se poteguje za procesno enoto. Ko to opravilo dobi procesor, se izvaja do konca oziroma dokler ima pogoje za napredovanje. Pred tem ga lahko prekine samo opravilo z višjo prioriteto. Če se to zgodi, razvrščevalnik prekine opravilo, ki se izvaja in ga uvrsti ponovno na začetek seznama opravil z enako prioriteto ter dodeli procesor novemu opravilu z višjo prioriteto. To pomeni, da bo prekinjeno opravilo nadaljevalo izvajanje takoj, ko ne bo več opravil z višjo prioriteto. V primeru, da opravilo zahteva v/i prenos, gre v stanje 'ustavljen' in se umakne s seznama pripravljenih opravil. Ko opravilo postane ponovno pripravljeno, se uvrsti na seznam pripravljenih opravil za dano prioriteto, in sicer na konec seznama. Obstaja še možnost, da opravilo sprosti procesno enoto samostojno oziroma na lastno pobudo z ukazom `sched_yield()`. V tem primeru se opravilo uvrsti na konec sezama pripravljenih opravil z enako prioriteto. Celoten postopek se ponavlja dokler opravilo ne konča.

Razvrščanje po pravilu `SCHED_RR` se od `SCHED_FIFO` razlikuje le v tem, da se opravilo, ki se izvaja in mu je potekel dodeljeni čas, uvrsti na konec seznama pripravljenih opravil z isto prioriteto. Dolžina intervala je parameter sistema, ki tipično znaša nekaj deset milisekund in ga je moč ugotoviti s funkcijo `sched_rr_get_interval()`.

Opravila realnega časa imajo fiksno oziroma 'statično' prioriteto. Prioriteta opravila je odvisna od konkretne aplikacije, je določena vnaprej in je jedro ne spreminja. Navadna opravila se izvajajo samo, če ni nobenega opravila realnega časa. Navadna opravila spadajo v enega od treh razredov in odvisno od tega za njih velja primerno pravilo razvrščanja,

- `SCHED_OTHER (ali SCHED_NORMAL)`: privzeto pravilo razvrščanja,

- `SCHED_BATCH`: za dolgotrajne računsko intenzivne, neinteraktivne procese,
- `SCHED_IDLE`: zares nizka in najnižja prioriteta.

Normalnim opravilom (`SCHED_NORMAL`) se prioriteta dinamično spreminja in je odvisna od njihovega napredovanja. Privzeto pravilo razvrščanja navadnih opravil je pravilo 'enakih' možnosti CFS (Angl. Completely Fair Scheduling) ob dobri odzivnosti za interaktivne procese, ki smo ga spoznali v prejšjem podpoglavju.

Nižja prioriteta je primerna za neinteraktivne (paketne) računsko intenzivne dolgotrajne procese. Razred `SCHED_BATCH` pomaga, da razvrščevalnik to upošteva pri dodeljevanju procesorja. Za procese tipa `SCHED_IDLE` vrednost `nice` nima pomena in se izvajajo le tedaj, ko je procesor prost (Angl. Idle). Njihova prioriteta je nižja od +19.

Razvrščevalnik sistema Linux deli čas na 'epohe' ali razdobja. Vsakemu od procesov, ki obstajajo znotraj epohe, se dodeli sorazmerni delež procesorja. Sprva, na začetku epohe, imajo procesi enako prioriteto, a prioriteta se jim z napredovanjem spreminja glede na to, koliko procesorskega časa so bili dejansko deležni. Proces, ki obremenjuje procesor relativno manj, bo pridobival višjo prioriteto, medtem ko proces, ki močno obremenjuje procesor, izgublja prioriteto. Poglejmo primer. Imejmo dva procesa, eden naj bo interaktiven (na primer urejevalnik) in drugi računsko intenziven (na primer Matlab). Na začetku epohe imata oba procesa enako možnost oziroma potrebo po procesorju, torej 50 %. Matlab izdatno obremenjuje procesor, medtem ko urejevalnik večino časa čaka na pritisk tipke na tipkovnici in ne potrebujeje procesorja. A ko je pritisnjena tipka, uporabnik pričakuje hiter odziv. Ob pritisku tipke se razvrščevalnik odloča na osnovi še neporabljenega vnaprej dodeljenega časa. V primerjavi z Matlabom je urejevalnik doslej porabil relativno malo časa procesorja (preostali čas je velik), zato mu razvrščevalnik dodeli procesor.

Vpogled v bistvene lastnosti procesa ponuja Linux ukaz `ps`. Z ukazom `ps` je moč ugotoviti številko procesa (pid), številko roditelja (ppid), stanje procesa (state), lastnika (uid), uporabljeni pomnilnik (size), procesorski čas

(time), prioriteto realnega časa (rtprio), dinamično prioriteto (pri), prioriteto nice (nice), ime programa (name), ukazno vrstico procesa (cmd), in drugo. Priročni možnosti sta `ps -l` in `ps -f`, ki izpišeti večino lastnosti, ki nas običajno zanimajo. Izpis lahko po želji oblikujemo z opcijo '-o'. Na primer:

```
ps -eo pid,rtprio,priority,nice,cmd
```

izpiše seznam vseh procesov (opcija 'e') v sistemu, številke procesov, prioritete realnega časa za procese realnega časa, uporabniške prioritete ter vrednosti `nice` za navadne procese in ukazne vrstice procesov.

18.4 Funkcije za razvrščanje procesov

Funkcije za upravljanje s prioritetami in s pravili razvrščanja so:

- `sched_getscheduler( )`: vrne pravilo razvrščanja danega procesa
- `sched_setscheduler( )`: nastavi pravilo razvrščanja danega procesa
- `sched_get_priority_min( )`: vrne minimalno prioriteto za izbrano pravilo razvrščanja
- `sched_get_priority_max( )`: vrne maksimalno prioriteto za izbrano pravilo razvrščanja
- `sched_rr_get_interval( )`: vrne časovni interval RR pravila razvrščanja
- `sched_getparam( )`: vrne prioriteto procesa
- `sched_setparam( )`: nastavi prioriteto procesa
- `sched_yield( )`: sprosti procesor, proces gre v stanje pripravljen

V predhodnem razdelku pa smo že spoznali in jih ponavljamo tukaj,

- `nice( )`: spremeni prioriteto navadnemu procesu
- `getpriority( )`: vrne prioriteto procesa
- `setpriority( )`: nastavi prioriteto procesa

Upravljanje s prioritetami realnega časa je dovoljeno le uporabnikom s pooblastili. Tudi nižanje vrednosti `nice` je dovoljeno le priviligiranim procesom.

18.4.1 Funkciji `sched_setscheduler` in `sched_getscheduler`

Funkcija `sched_setscheduler()` nastavi pravilo razvrščanja in prioriteto, funckija `sched_getscheduler()` vrne pravilo razvrščanja izbranega procesa.

```
#include <sched.h>

struct sched_param {
  int sched_priority;
};
int sched_setscheduler(pid_t pid, int policy,
                               const struct sched_param *param);
int sched_getscheduler(pid_t pid);
```

Pravilo razvrščanja je lahko:

```
SCHED_FIFO    realni cas, prvi pride prvi strezen
SCHED_RR      realni cas, enako kot FIFO, a z delitvijo casa RR

SCHED_OTHER   normalna prioriteta v RR nacinu
SCHED_BATCH   'paketni' nacin
SCHED_IDLE    najnizja prioriteta
```

Vrednost parametra `sched_priority` je odvisna od pravila razvrščanja in mora biti za procese realnega časa `SCHED_FIFO` ali `SCHED_RR` vsaj 1 in največ 99. Za procese z običajno prioriteto je vrednost `sched_priority` enaka nič.

Na primer, v spodnjem kosu programa proces sebi (pid = 0) nastavi najvišjo prioriteto realnega časa in razvrščanje po pravilu `SCHED_FIFO`.

```
struct sched_param prio;
....
prio.sched_priority = sched_get_priority_max( SCHED_FIFO );

sched_setscheduler( 0, SCHED_FIFO, &prio );
....
```

18.4.2 Funkciji sched_get_priority_min in sched_get_priority_max

Funkciji sched_get_priority_min( ) ter sched_get_priority_max( ) vrneti najmanjšo in največjo vrednost prioritete procesov realnega časa ter -1 v primeru napake.

```
#include <sched.h>
int sched_get_priority_max( int policy );
int sched_get_priority_min( int policy );
```

Linux omogoča prioritetne nivoje realnega časa na intervalu [1..99]. Prioritete procesov realnega časa so 'statične'. Statične prioritete so določene vnaprej, se med napredovanjem procesov ne spreminjajo oziroma v njih razvrščevalnik ne posega. POSIX.1-2001 zahteva (vsaj) 32 prioritetnih nivojev realnega časa. Običajni procesi imajo prioriteto realnega časa nič.

18.4.3 Funkcija sched_rr_get_interval

Funkcija sched_rr_get_interval vrne časovni interval (trajanje časovne rezine oziroma kvanta) pri razvrčanju po pravilu SCHED_RR.

```
#include <sched.h>

struct timespec {
       time_t tv_sec;     /* seconds */
       long   tv_nsec;    /* nanoseconds */
     };
int sched_rr_get_interval(pid_t pid, struct timespec * tp);
```

Denimo, naslednji kos programa

```
#include <sched.h>

int min_pri, max_pri;
struct timespec  tp;
...
min_pri = sched_get_priority_min( SCHED_RR );
max_pri = sched_get_priority_max( SCHED_RR );
...
```

```
sched_rr_get_interval(0, &tp);
...
printf("Min pri: %d, Max pri: %d, RR-s: %d, RR-ns: %d\n",
                          min_pri, max_pri, tp.tv_sec,tp.tv_nsec);
...
```

vrne najnižjo in najvišjo prioriteto razvrščanja po pravilu `SCHED_RR` ter RR interval (sekunde, nanosekunde).

18.4.4 Funkcija `sched_yield`

```
#include <sched.h>
int sched_yield(void);
```

Funkcija vrne nič v primeru uspeha, -1 v primeru napake. S funkcijo `sched_yield()` proces (ali nit) sam sprosti procesno enoto in se postavi v stanje pripravljen na izvajanje. Procesor prevzame naslednji proces z višjo ali enako prioriteto. Če takega procesa ni, proces nadaljuje z izvajanjem.

18.5 Afiniteta procesorja

Linux podpira večprocesorske sisteme po načelu SMP. Razvrščanje opravil na posameznem procesorju poteka praviloma avtonomno. Zato vsak procesor razpolaga s svojim seznamom pripravljenih opravil. Spričo uravnotežanja bremena pa je mogoče, da bo določeno opravilo preseljeno z enega procesorja na drugi procesor.

Linux implementira dva nestandardna sistemska klica, s katerima je moč vplivati na 'afiniteto' opravila do specifičnega procesorja. Na ta način lahko dosežemo, da bo izbranemu opravilu prednostno dodeljen na primer prvi procesor. Prototipa funkcij za spreminjanje afinitete sta:

```
#define _GNU_SOURCE
#include <sched.h>

int sched_setaffinity(pid_t pid, size_t cpussize, cpu_set_t *mask);
```

```
int sched_getaffinity(pid_t pid, size_t cpussize, cpu_set_t *mask);
```

Funkciji vrneta nič v primeru uspeha in -1 v primeru napake. Prva izmed funckij dodaja in druga odvzema afiniteto do izbranih procesorjev. Prvi argument je številka procesa ali nič. V slednjem primeru se operacija nanaša na proces, ki kliče funkcijo. Drugi argument pomeni velikost tretjega argumenta in bo zato enak `sizeof(cpu_set_t)`.

Procesorje se dodaja in odvzema z makro definicijami

```
#define _GNU_SOURCE
#include <sched.h>

void CPU_ZERO(cpu_set_t *set);
void CPU_SET(int cpu, cpu_set_t *set);
void CPU_CLR(int cpu, cpu_set_t *set);
int  CPU_ISSET(int cpu, cpu_set_t *set);
```

Procesorje se številči začenši z nič. Na primer, z naslednjo kodo nastavi proces afiniteto do procesorjev 3 in 4 samemu sebi.

```
#define _GNU_SOURCE
#include <sched.h>

cpu_set_t set;

CPU_ZERO( &set );
CPU_SET( 3, &set );
CPU_SET( 4, &set );
sched_setaffinity( 0, sizeof( set ), &set );
```

Če bi se proces dotlej izvajal na procesorju nič, bi se posledično selil na drugi procesor.

Naslednji primer odstrani afiniteto opravila do procesorja 4.

```
#define _GNU_SOURCE
#include <sched.h>

cpu_set_t set;
```

```
CPU_CLR( 4, &set );
sched_setaffinity( 0, sizeof( set ), &set );
sched_getaffinity( 0, sizeof( set ), &set );
if( CPU_ISSET( 4, &set) ){
  ce vrne 1 je procesor se vedno v mnozici, zelo malo verjetno.
}
```

Literatura

[1] C.L. Liu, J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment", *Journal of ACM*, Vol. 20, No.1, January 1973, pp. 4651.

[2] S. J. Leffler, R.S. Fabry, W.N. Joy, "A 4.2 BSD Interprocess Communication Primer", Draft of August 31, 1984, Computer Systems Research Group, University of California, Berkeley, 1984.

[3] —, *Real-Time Programming Manual, Series 800 HP-UX*, Hewlett Packard, 1986.

[4] J. A. Stankovic et al, "Strategic Directions in Real-Time and Embedded Systems", *ACM Computing Surveys*, vol. 28, no. 4, pp. 751-763, December 1996.

[5] B. W. Kernighan, D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

[6] B. W. Kernighan, D. M. Ritchie, *The C Programming Language, 2-nd Ed., ANSI C*, Prentice-Hall, 1988.

[7] M. J. Bach, *The Design of the UNIX Operating System*, Prentice Hall, 1986.

[8] D. P. Bovet, M. Cesati, *Understanding the Linux Kernel, 3-rd ed.*, O'Reilly, 2006.

[9] R. Love, *Linux Kernel Development, 3-rd ed.*, Pearson, Addison-Wesley, 2010.

[10] M. J. Rochkind, *Advaned UNIX Programming*, Prentice Hall, 1985.

[11] M. J. Rochkind, *Advaned UNIX Programming, 2-nd Ed.*, Addison-Wesley, 2004.

[12] M. Mitchell, J. Oldham, A. Samuel, *Advanced Linux Programming*, New Riders, (prost dostop: http://www.advancedlinuxprogramming.com/alpfolder/alptoc.pdf),

2001.

[13] W.R. Stevens, S. A. Rago, *Advaned Programming in the UNIX Environment, 2nd ed.*, Addison-Wesley, 2005.

[14] W.R. Stevens, S. A. Rago, *Advaned Programming in the UNIX Environment, 3rd ed.*, Addison-Wesley, 2013.

[15] R. Love, *Linux System Programming*, O'Reilly, 2007.

[16] M. Kerrisk, *The Linux System Programming Interface*, No Starch Press, 2010.

[17] W. R. Stevens, B. Fenner, A. M. Rudoff, *UNIX Network Programming, Vol. 1, 3rd ed.*, Addison-Wesley, 2004.

[18] W. R. Stevens, *UNIX Network Programming, Vol. 2, 2nd ed.*, Prentice Hall, 1999.

[19] W.R. Stevens, *TCP/IP Illustrated, Vol. 1*, Addison-Wesley, 1994.

[20] A. S. Tanenbaum, D. J. Wetherall, *Computer Networks, 5th ed.*, Pearson, Prentice Hall, 2011.

[21] J. F. Kurose, K. W. Ross, *Computer Networking, A top-down approach, 5-th ed.*, Pearson, Addison-Wesley, 2010.

[22] A. Silberschatz, J. Peterson, *Operating Systems Concepts*, Addison-Wesley, 1991.

[23] A. Silberschatz, P.B. Galvin, G. Gagne, *Operating System Concepts, 9th Ed.*, John Wiley & Sons, 2012.

[24] A. S. Tanenbaum, *Modern Operating Systems, 3rd Ed.*, Pearson Education, Prentice Hall, 2009.

[25] D. M. Dhamdhere, *Operating Systems, A Concept-Based Approach, 2-nd Ed.*, McGraw-Hill, 2006.

[26] A. S. Tanenbaum, A. S. Woodhull, *Operating Systems, The MINIX Book, 3rd Ed.*, Pearson Education, Prentice Hall, 2009.

[27] D. Abbott, *Linux for Emmbedded and Real Time Applications, 2nd Ed.*, Elsevier, 2006.

[28] M. Barr, A. Massa, *Programming Embedded Systems, with C and GNU Development Tools*, O'Reilly, 2007.

[29] C. Hallinan, *Embedded Linux Primer*, Pearson, Prentice Hall, 2007.

[30] B. Gallmeister, *POSIX.4: Programming for the real world*, O'Reilly, 1995.

[31] R. Kamal, *Embedded Systems: Architectures, Programming, and Design*, McGraw Hill, 2008.

[32] T. Noergaard, *Embedded Systems Architecture*, Elsevier, 2005.

[33] Jane W.S. Liu, *Real-Time Systems*, Prentice Hall, 2000.

[34] A. Burns, A. Wellings, *Real-time Systems and Programming Languages*, Pearson, Addison-Wesley, 2009.

[35] Machtelt Garrels, *Introduction to Linux, A Hands on Guide*, CreateSpace Independent Publishing Platform, 2007. Prost dostop na `http://tille.garrels.be/training/tldp/`.

[36] Ellen Siever, Jessica P. Hackman, Stephen Spainhour, Stephen Figgins, *Linux in a Nutshell*, O'Reilly, 2009.

[37] Lars Wirzenius, Joanna Oja, Stephen Stafford, Alex Weeks, The Linux System Administrator's Guide, 2004.