UNIVERZA V LJUBLJANI

Fakulteta za elektrotehniko

Stanislav Kovačič

Vzporedni sistemi (študijsko gradivo)

Ljubljana, 3. november 2004

Kazalo

1 Klasifikacija računalnikov

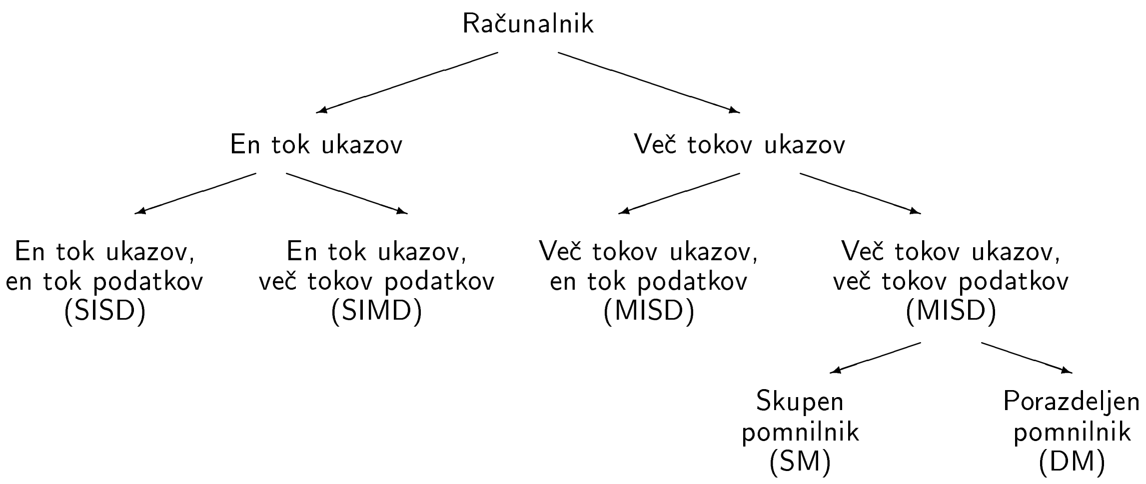
Vsak računalnik, sekvenčni ali paralelni, izvršuje ukaze nad podatki:

- zaporedje ukazov (tok ukazov) določa operacije, ki jih računalnik izvršuje eno za drugo (sekvenčno),
- zaporedje podatkov oziroma operandov (tok podatkov) določa, nad čim se operacije izvršujejo.

V danem trenutku je lahko v obdelavi eden ali več ukazov nad enim ali več operandi. Glede na število (vzporednih oziroma sočasnih) tokov ukazov in operandov razvrstimo računalnike v štiri skupine (Flynn 1966):

1. SISD (Single Instruction Single Data Stream),
2. SIMD (Single Instruction Multiple Data Stream),
3. MISD (Multiple Instruction Single Data Stream) in
4. MIMD (Multiple Instruction Multiple Data Stream).

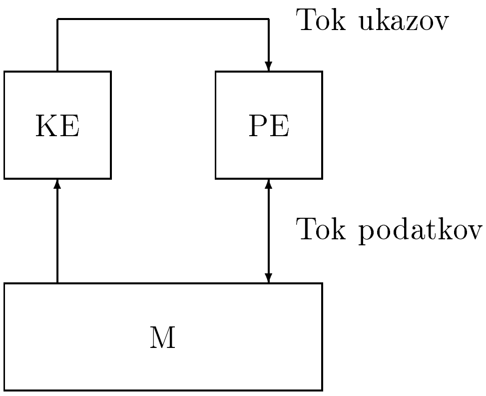
Obstaja še peta kategorija: SPMD (Single Program Multiple Data Stream).



Slika 1: Razvrstitev računalnikov po Flynnu.

1.1 SISD računalniki

SISD je sinonim za von Neumannov tip računalnika. To je klasičen sekvenčni računalnik, ki izvršuje ukaze enega za drugim. Z napredkom računalniških znanosti in tehnologij je ta tip računalnika doživel mnogo izboljšav. Današnji von Neumannov računalnik ima vgrajene različne oblike paralelizma (cevovodno, supercevovodno, skalarno, superskalarno procesiranje ukazov/podatkov).



KE - Kontrolna enota

PE - Procesna enota (element)

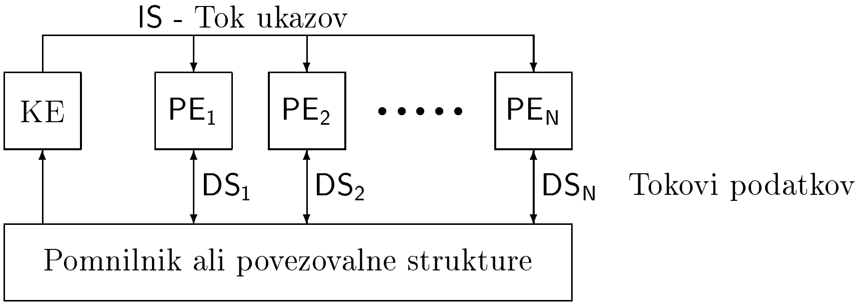
M - Pomnilnik

Slika 2: Von Neumannov tip računalnika.

V načelu je SISD računalnik primeren za izvrševanje sekvenčnih algoritmov. Na primer, po sekvenčnem algoritmu za seštevanje N števil mora na sekvenčnem računalniku procesor iz pomnilnika pridobivati podatek za podatkom, tekoči podatek prišteti k (delni) vsoti, t.j. zaporedoma izvršiti N seštevanj. Imamo eno zaporedje (tok) podatkov in eno zaporedje ukazov. Večopravilni operacijski sistemi na SISD računalnikih dajejo vtis navidezne sočasnosti - v določenem obdobju v sistemu obstaja več opravil, vendar v danem trenutku napreduje samo eno od teh opravil.

1.2 SIMD računalniki

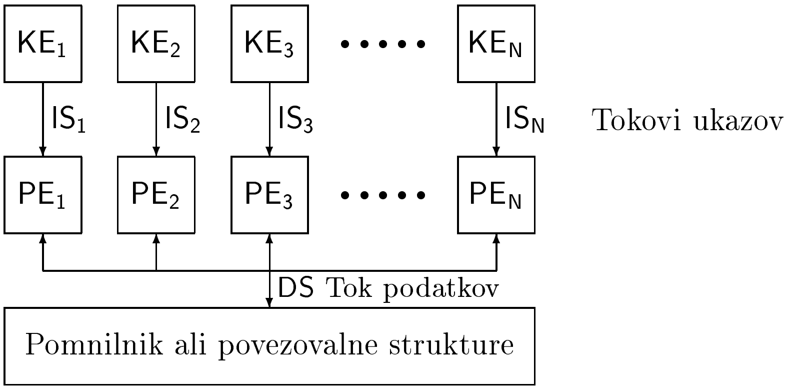
SIMD računalnik v danem trenutku izvršuje isto operacijo nad več kot enim podatkom. Na primer, sočasno lahko sešteje dva para števil. Večje število procesorjev (procesnih elementov PE) sinhrono deluje pod nadzorom ene kontrolne enote, ki definira eno zaporedje ukazov. V splošnem torej izvršuje eno zaporedje ukazov na več zaporedjih podatkov.
Računalniki tega tipa so posebej primerni za obdelavo digitalnih slik - matrik slikovnih elementov.



Slika 3: Računalnik tipa SIMD.

1.3 MISD računalniki

MISD računalnik lahko izvršuje več različnih tokov ukazov nad enim samim tokom podatkov. Zato je nad enim in istim podatkom v danem trenutku sposoben izvršiti več operacij. Komercialni računalniki te vrste danes ne obstajajo.

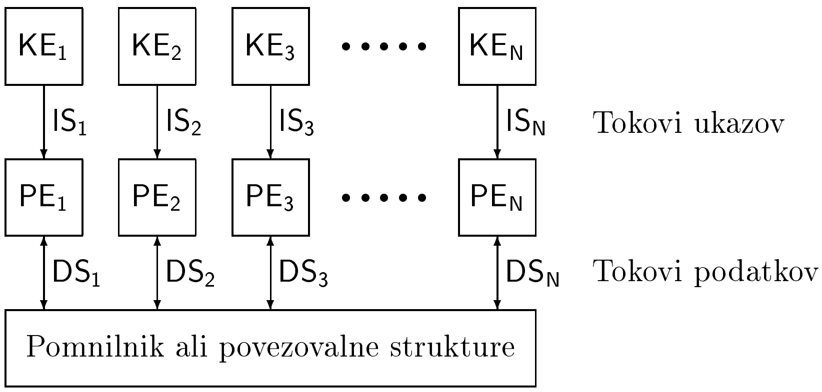


Slika 4: Računalnik tipa MISD.

Tak tip računalnika je učinkovit za reševanje problemov, pri katerih želimo nad istim podatkom opraviti več različnih operacij. Na primer, ena od možnosti, da ugotovimo ali je dano naravno število (N) praštevilo, je, da ga delimo z 2,3,...,N-1. Na MISD računalniku z N-2 procesorji bi lahko to naredili sočasno v enem koraku, na računalniku z manj procesorji ustrezno več, a še vedno v krajšem času kot na računalniku z enim samim procesorjem.

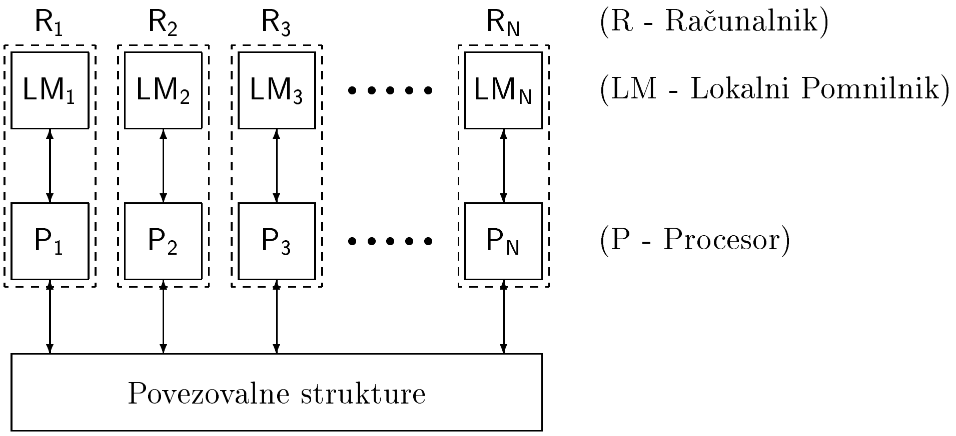
1.4 MIMD računalniki

MIMD računalniki lahko sočasno izvajajo več različnih tokov ukazov nad več različnimi tokovi podatkov. MIMD računalnik vsebuje več procesnih elemementov, od katerih vsak deluje pod nadzorom svoje kontrolne enote. Lahko rečemo, da MIMD računalnik sestavja večje število sekvenčnih računalnikov, ki so na nek način povezani med seboj in lahko sodelujejo pri reševanju danega problema.



Slika 5: MIMD računalnik s skupnim pomnilnikom.

Če naj sodelujejo, morajo med seboj komunicirati. Podlago za to daje bodisi skupen pomnilnik ali povezovalne strukture (povezovalno ali komunikacijsko omrežje). Sistemi s skupnim pomnilnikom so tesno sklopljeni paralelni sistemi. Običajno dostopajo procesorji do (globalnega) pomnilnika preko skupnega vodila. Za veliko število procesorjev postane vodilo ozko grlo. Zato se taki sistemi obnesejo za majhno število procesorjev (pod 30).
Sistemi s povezovalnim omrežjem so rahlo sklopljeni. Na ta način lahko povežemo načeloma poljubno mnogo procesorjev. Vsak procesor ima še svoj pomnilnik (lokalni pomnilnik), do katerega ima hiter dostop. Komunikacija s procesorji preko omrežja, je počasnejša. Dostop do pomnilnikov drugih procesorjev, ki poteka preko omrežja, je počasnejši. Lahko bi rekli, da imamo krajevno porazdeljen pomnilnik.



Slika 6: Računalnik tipa MIMD s porazdeljenim pomnilnikom.

V navadi je, da sistemu s skupnim pomnilnikom rečemo multiprocesorski sistem (multiprocesor). Sistem s povezovalnim omrežjem rečemo multiračunalniški sistem

(multiračunalnik). Vsi procesorji/računalniki so lahko popolnoma enaki ali pa se razlikujejo. V zvezi s tem govorimo o homogenih in heterogenih sistemih.
V uporabi je tudi izraz porazdeljeni sistemi, pri čemer mislimo na večje število krajevno porazdeljenih računalnikov, ki jih povezuje komunikacijsko omrežje.

1.5 SPMD

V primeru SPMD tipa računalnika gre za izvrševanje enega in istega programa na večjem številu računalnikov, od katerih vsak obdeluje svoje podatke (sočasna večkratna izvršitev enega programa na različnih podatkih). SPMD je v tem smislu podoben SIMD računalniku, le da procesorji delujejo asinhrono. Sled ukazov, čeprav procesorji izvršujejo isti program, so lahko od procesorja do procesorja različne, saj je sled programa odvisna tudi od podatkov.

1.6 Model paralelnega računalnika

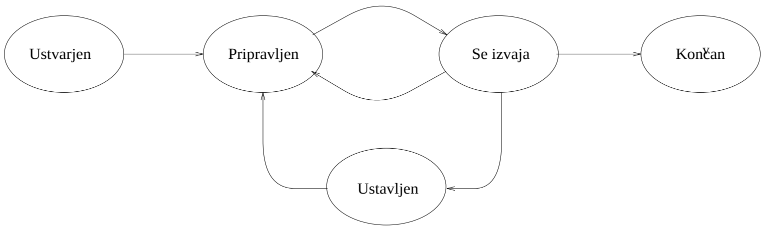
Von Neumannov tip računalnika se je izkazal kot dober model za razvoj sekvenčnih algoritmov in programov. Kakšen pa naj bi bil dober model paralelnega računalnika? Lahko bi rekli da tak, ki daje dobro podlago snovanju paralelnih algoritmov in paralelnemu programiranju. Paralelen algoritem je z zbirko navodil podan postopek za paralelno (sočasno) reševanje danega problema in je primeren za izvršitev na paralelnem računalniku.
Splošen model paralelnega računalnika, ki se zdi najprimernejši za obravnavanje paralelnih algoritmov, je multiračunalnik. Multiračunalnik sestavlja več na nek način med seboj povezanih sekvenčnih računalnikov, od katerih ima vsak svoj lokalni pomnilnik, do katerega ima hiter dostop. V takem multiračunalniku se lahko sočasno odvija več dejavnosti oziroma procesov ali opravil, ki po potrebi komunicirajo med seboj - izvršujejo paralelen algoritem. Eno opravilo se lahko odvija na enem procesorju ali na več procesorjih. Po drugi strani se lahko na enem procesorju odvija eno ali več opravil. Skratka, preslikava paralelnega algoritma na konkreten računalniški sistem, t.j. preslikava procesov na procesorje (dodelitev procesorjev procesom) je samo ena od nalog pri snovanju paralelnih algoritmov.

1.7 Operacijski sistem

Uporaben računališki sistem sestavljata strojna in programska oprema. Del sistemske programske opreme, ki skrbi za to, da se dejavnosti znotraj sistema pravilno odvijajo, da so sredstva sistema dobro izkoriščena in za komunikacijo sistema z okoljem (uporabnikom), imenujemo operacijski sistem (OS). Operacijski sistem nad strojno opremo ustvari za uporabnika primernejšo abstarkcijo sistema. S tega stališča je bolj pomemno, kaj se v sistemu dogaja kot pa na kakšen način, t.j. samo število procesorjev, pomnilnikov in povezav.

2 Proces

Osnovni objekt, s katerim se bomo ukvarjali večino časa, je proces. *Računalniški proces*, ali krajše kar proces, je program v izvrševanju. Za program v izvrševanju so v rabi tudi drugi izrazi, na primer opravilo (ang. Task). Mi bomo večinoma uporabljali izraz proces, ki se je uveljavil predvsem v sistemu UNIX.
Proces je torej program v izvrševanju. Proces nastane iz programa, ki je običajno shranjen v datoteki na disku. Ko se to zgodi, pravimo, da je proces ustvarjen. Med izvrševanjem proces spreminja stanje. Pravimo, da je proces aktiven, kadar ima zagotovljene vse pogoje za napredovanje. Sicer je ustavljen. Ustavljen je tedaj, ko nima vseh pogojev za napredovanje, na primer čaka na vhodno/izhodno operacijo. V aktivnem stanju se proces poteguje za procesno enoto ali pa mu je le-ta že dodeljena. Proces v resnici napreduje samo tedaj, ko ima procesno enoto. Pravimo, da *se izvaja*. V nasprotnem primeru je proces *pripravljen*. 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, dokler nazadnje ne konča.



Slika 7: Stanja in prehajanje stanj v času obstoja procesa.

V stanje "se izvaja" gre proces lahko samo iz stanja "pripravljen". V primeru, da proces potrebuje vhodno/izhodni prenos, gre proces iz stanja "se izvaja" v stanje "ustavljen". Ko je prenos opravljen, postane proces ponovno "pripravljen". V stanje "pripravljen" gre proces lahko tudi iz stanja "se izvaja" in sicer v primeru, ko mu poteče dodeljeni čas ali pa mu procesno enoto prevzame proces z višjo prioriteto. Glede na to, ali procesor sproščajo prosesi sami ali pa mu je le-ta prevzeta s strani drugega procesa, razlikujemo razvrščanje procesov s prevzemanjem in brez prevzemanje. Razvrščanje s prevzemanjem je bistvenega pomena za sisteme, ki morajo delovati v stvarnem času. Lastnost prevzema procesne enote s strani drugega procesa imenujemo tudi *predopravilnost*.
V večopravilnem (večprocesnem, tudi večprogramskem ali kar multiprogramskem) 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, a zaradi praktičnih razlogov navzgor omejeno, število procesov. Del operacijskega sistema, ki določa, kateri proces se bo izvajal naslednji oziroma kdaj in za koliko časa, imenujemo razvrščevalec procesov (ang. Scheduler). Del operacijskega sistema, ki procesu dodeli procesor oziroma opravi menjavo procesov, imenujemo dispečer. Menjavo procesov imenujemo tudi kontekstni (pomenski) preklop (ang. 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 več procesov. V enoprocesorskem sistemu pač v določenem obdobju napreduje (se izvaja) samo en proces, v večprocesorskem sistemu pa tudi več.
Naše področje zanimanja bi lahko opredelili na naslednji način. V sistemu obstaja več procesov. Ti procesi v splošnem napredujejo vsak s svojo in vnaprej nepredvidljivo hitrostjo oziroma asinhrono. Nekateri med njimi napredujejo samostojno in neodvisno drug od drugega. Taki procesi nas 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. Zanimali nas bodo prav problemi in njihove rešitve v zvezi z medprocesnimi komunikacijami.

3 Medprocesne komunikacije

Razlikujemo dve glavni obliki komunikacij med procesi oz. medprocesnih komunikacij (ang. Interprocess Communications - IPC):

- komunikacija s pomočjo skupnega (deljenega) pomnilnika. V tem primeru poteka komunikacija preko pomnilnika ali dela pomnilnika, do katerega je dovoljen dostop večjemu številu procesov. Komunikacija poteka na podlagi v naprej dogovorjenih podatkovnih struktur po načelu beri/piši.

- Komunikacija s pomočjo sporočil. V tem primeru poteka komunikacija po komunikacijskem kanalu s pomočjo operacij (funkcij) pošlji in sprejmi.

V prvem primeru je dana medprocesni komunikaciji elementarna podlaga, nad katero je možno nadgraditi druge oblike medprocesnih komunikacij, vključno s sistemom sporočil.
Kadar govorimo o *medprocesnih komunikacijah*, se običajno ne zanimamo, kako so procesorji (v primeru, da jih je več) v resnici povezani med seboj.
Kadar govorimo o *medprocesorskih komunikacijah* običajno postavljamo v ospredje povezovalne strukture, ki povezujejo procesorje (in pomnilnike).

3.1 Sinhronizacija – ena od oblik medprocesnih komunikacij

Govorili bomo o problemih in rešitvah teh problemov, ki se pojavijo tedaj, kadar večje število procesov dostopa do sredstva, ki ne dovoljuje sočasnega dostopa (skupen pomnilnik, del pomnilnika, datoteka, tiskalnik, ...). 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. Ko govorimo o medsebojnem izključevanju, mislimo v bistvu na mehanizme ozirimo 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.

3.1.1 Definicija problema

Imamo večje število sočasnih procesov. Procesi so sočasni, ker obstajajo ob istem času, v istem časovnem obdobju. Ti procesi napredujejo vzporedno ali navidez vzporedno v smislu multiprogramiranja (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. Naj bo trenutna vrednost x-a 10. Proces P_1 poveča x za 1, proces P_2 pa zmanjša x za 1.

<u>P1</u>	<u>P2</u>
int x = 10;	int x; /* skupna spremenljivka
...	...
...	...
x++	x– –
...	...
...	...

Izmed več možnih zaporedij si oglejmo naslednje zaporedje dogodkov (R je register 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

Spremenljivka x zavzame po izvršitve tega zaporedja vrednost 9. Ob drugačnem zaporedju dogodkov bi x dobil vrednost 11 ali 10. Očitno je dejanska vrednost spremenljivke odvisna od naključja; rezultat izvršitve operacij je nepredvidljiv, kar je nedopustno. V primeru, da se povečanje in zmanjšanje vrednosti x-a zgodi sekvenčno (eno za drugim, najprej zvečanje in nato zmanjšanje, ali obratno), je rezultat 10, kar bi tudi bilo pravilno. Da bi se to dogodilo vedno v vsakih okoliščinah, bi se morali obe operaciji (zvečanje/zmanjšanje) izvršiti časovno nedeljivo (atomočno). Z drugimi besedami, ko operacija enkrat začne, se mora končati brez prekinitve. Glede na to, da operaciji trajata malo časa, bi bila zahteva po atomičnosti operacij sprejemljiva rešitev. Če bi prišlo do sočasnega dostopa do spremenljivke x, bi se obe zahtevi razvrstili sekvenčno.
Na podagi povedanega pa lahko zaključimo, da se v splošnem obe operaciji medsebojno časovno izključujeta: če dovolimo eno, moramo prepovedati drugo, dokler se prva ne konča.

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

Imamo dva procesa, P_1 in P_2, ki dostopata do skupne datoteke D. Možno bi bilo naslednje zaporedje operacij:

1. P_1 delno spremeni datoteko D, za kar rabi 1 minuto (datoteko na primer spremeni do polovice),

2. P_2 bere delno spremenjeno datoteko D,

3. P_1 dokončno spremeni datoteko in njena vsebina je veljavna, za kar potrebuje še eno minuto.

4. $\cdots$

Tudi v tem primeru je problem očiten in je posledica dejstva, da se spreminjanje vsebine datoteke ne izvede v celoti od začetka do konca predno dobi možnost dostopa do datoteke drug proces.

Ena od možnosti: prepovemo prekinitev procesa – onemogočimo izvajanje drugih procesov. To bi bila preveč konzervativna rešitev (spreminjanje datoteke 2 minuti je dolga doba). Zadostuje, da ne dovolimo izvajanje procesa P_2 v primeru, če bi med tem zahteval dostop do datoteke. Sicer pa se tudi proces P_2 lahko poljubno izvaja.

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** (ang. critical section, critical region).
Tedaj, ko proces izvaja zaporedje ukazov, ki posega po skupnem sredstvu pravimo, da se izvaja v svojem kritičnem delu.

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.

3.2 Rešitev problema kritičnega področja (dela)

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

1. medsebojno izključevanje v vsakih okoliščinah (ang. Mutual exclusion),
2. ne sme priti do zastoja (zastoj je posebno stanje procesa, iz katerega ni regularnega izhoda. Proces je v stanju zastoj (ang. 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) (ang. 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/izvajanja procesa. Procesi napredujejo z različno in nepredvidljivo hitrostjo. Predpostavljamo le, da se posamezna operacija (ukaz) izvrši atomično (časovno nedeljivo – ko enkrat začne, se izvrši do konca brez prekinitve).
Pri načrtovanju splošne rešitve bomo za model vzeli naslednjo zgradbo procesa:

$\underline{P_i}$ /* 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
}
```

3.2.1 Rešitev 1 – Procesa se v K.D. izvajata izmenoma

Imamo 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.

```
Pi          /* 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.

3.2.2 Rešitev 2 – Petersonova rešitev

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.

```
Pi          /* 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.

3.2.3 Problem medsebojnega izključevanja za N procesov, komentar

Dijkstra je prvi predlagal rešitev problema kritičnega dela za primer sinhronizacije N procesov (1966) in Knuth jo je leto za tem izboljšal tako, da je preprečil nepredvidljivo dolgo odlaganje. S tem je postal problem K.D. za N procesov "vroča" tema in sledili so predlogi z vse krajšim odzivnim časom. Eisenberg in McGuire (1972) sta prišla do rešitve, ki N procesom jamči dostop najkasneje po N-1 poskusih. Rešitev za N procesov, ki je primerna predvsem za krajevno porazdeljene sisteme, je predlagal Lamport (1974, "take a ticket" algoritem).

3.2.4 Podpora sinhronizaciji na nivoju strojne opreme – 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 (npr. ukaz z imenom TST (TestAnd-Set) ali kako drugače imenovan ukaz).

Posebnost ukaza TST je v tem, da realizira R-M-W (Read-Modify-Write) operacijo vedno atomočno – od začetka do konca – brez prekinitve. 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:

```
Pi          /* 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.

3.2.5 Sinhronizacija z aktivnim čakanjem

Obravnavani primeri rešujejo problem K.D. z aktivnim čakanjem, t.j. proces, ki ne sme v kritični del, aktivno čaka v zanki in procesor ponavlja iste (nekoristne) ukaze, dokler se mu ne dovoli vstop v kritični del. Bolj ekonomično pa je, da se procesu, ki ponavlja ukaze v zanki, prevzame procesor in ga dodeli drugemu procesu (če ta obstaja).

3.3 Semafor

Semafor je pripomoček, ki se uporablja za sinhronizacijo procesov. Uporabo semaforja je predlagal Dijkstra sredi šestdesetih let.
Semafor je celoštevilčna spremenljivka, na kateri sta poleg inicializacije možni dve operaciji: *čakaj* in *javi* (tudi zakleni in odkleni, postavi in spusti, preveri in postavi). Tipično se operacijo čakaj (ang. Wait) označi s **P**, operacijo javi (ang. 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".
Za naše potrebe definirajmo nov podatkovni tip Sem, ki ustreza definiciji semaforja.

3.3.1 Operacija P

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)- -;
}
```

3.3.2 Operacija V

Operacija V javlja, da proces zapušča kritični del.

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

3.3.3 Binarni in števni semafor

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.

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.

Rešitev problema K.D. s semaforjem je pregledna:

```
Pi        /* 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
  }
```

Primer

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.

Primer

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

Proces P_0

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

Proces P_1

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

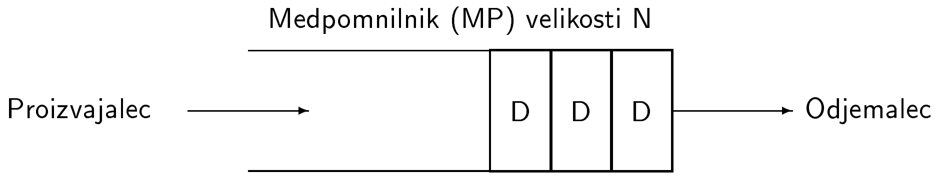
V razmislek: v zgoščeni obliki zapišite krožen dostop do skupnega sredstva s pomočjo semaforjev za N procesov.

3.4 Klasični primeri medprocesne sinhronizacije

3.4.1 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 8: 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 preprečiti 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
}
```

3.4.2 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 številov drugih bralnih procesov, pisanje procesom P_{W_j} pa je tedaj prepovedano.

Možna bi bila naslednja rešitev:

$\underline{P_{W_i}}$

```
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
}
```

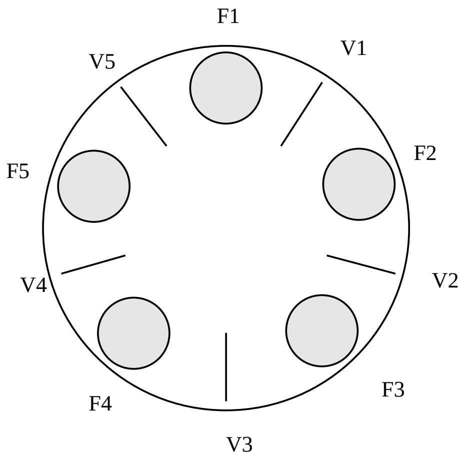
$\underline{P_{R_j}}$

```
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
}
```

3.4.3 Problem petih filozofov (pri kosilu)

Problem petih filozofov pri kosilu (ang. 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.



Slika 9: Filozofi pri kosilu.

3.5 Dogodkovni števnik

Tudi dogodkovni števnik (ang. Event Counter) je pripomoček za sinhronizacijo procesov (Reed in Kanodia 1979). Nad dogodkovnim števnikom so definirane tri operacije: beri, povečaj in čakaj:

- operacija Beri(E) vrne trenutno vrednost dogodkovnega števnika E,
- operacija Povečaj(E) ga poveča za ena (časovno nedeljivo),
- operacija Čakaj(E, V) čaka toliko časa, dokler števnik E ne doseže (ali preseže) določeno vrednost V.

Ena od zanimivih lastnosti dogodkovnega števnika je, da se vrednost števnika samo povečuje (šteje "dogodke"). Druga pomembna lastnost števnika je, kar bo razvidno iz primera, da lahko njegovo vrednost spreminja en proces, medtem ko jo drugi proces samo bere.

Za naše potrebe si definirajmo nov podatkovni tip *dogodkovni števnik* Ec.

3.5.1 Rešitev problema proizvajalca/porabnika z dogodkovnim števnikom

Pa poglejmo, kako bi rešili problem proizvajalca in odjemalca s pomočjo dogodkovnega števca. Medpomnilnik je organiziran kot krožni pomnilnik. Števnik Vstavi povečuje samo proizvajalec, preverja pa ga porabnik. Števnik Vzemi povečuje porabnik, preverja pa ga proizvajalec. Kritična operacija (spreminjanje) je tako v domeni enega procesa.

Proizvajalec (z dogodkovnim števcem)

```
Ec Vstavi = Vzemi = 0;      /* Dogodkovna števca */
int v = 0; /* Lokalna spremenljivka */
while ( 1 ){      /* Neskončna zanka */
  Pripravi podatek D
  v++;
  Čakaj( Vzemi, v-N ); /* Je MP že poln ? */
  Vstavi podatek D v krožni MP, mp[ (v-1) % N ] = D;
  Povečaj( & Vstavi );
  Splošni (nekritični) del
}
```

Porabnik (z dogodkovnim števcem)

```
Ec Vstavi, Vzemi;      /* Dogodkovna števca */
int w = 0;
while ( 1 ){      /* Neskončna zanka */
  Nekritični del
  w++;
  Čakaj( Vstavi, w );
  Vzemi podatek D iz MP, D = mp[ (w-1)% N ];
  Povečaj( & Vzemi );
  Porabi D;
  Preostanek - nekritični del
}
```

3.6 Monitor

Monitor je programski objekt, ki definira podatkovne strukture in operacije nad njimi. Koncept monitorja so razvili Dijkstra, Hansen in Hoare sredi sedemdesetih let.
Osnovna zamisel monitorja je v tem, da je dostop do skupnega sredstva možen izključno skozi monitor. Na primer, če želi proces do skupne datoteke, mu je ta možnost dana preko monitorja. V splošnem dostopa do skupnega sredstva več procesov, vendar monitor dopušča vstop samo enemu, ostali morajo čakati izven monitorja. Bistvena lastnost monitorja je v tem, da definira strukturo podatkov in operacije nad njimi. Operacije, kot tudi podatki, so sestavni del monitorja, ki na ta način tvorijo objekt. Podatki so dostopni samo preko operacij, ki jih definira monitor, druge operacije ne obstajajo. Če želi določen proces opraviti določeno operacijo nad podatki, mora koristiti ustrezno operacijo monitorja. Ker se v monitorju lahko izvaja samo en proces, je s tem medsebojno izključevanje zagotovljeno.

3.7 Glavna literatura

1. R.J. Baron, K. Higbie, Computer Architecture, Addison-Wesley, 1992.

2. H.M. Deitel, An Introduction to Operating Systems, Second Edition, Addison-Wesley , 1990.

3. A. Tanenbaum, Modern Operating Systems, Prentice Hall, 1992.

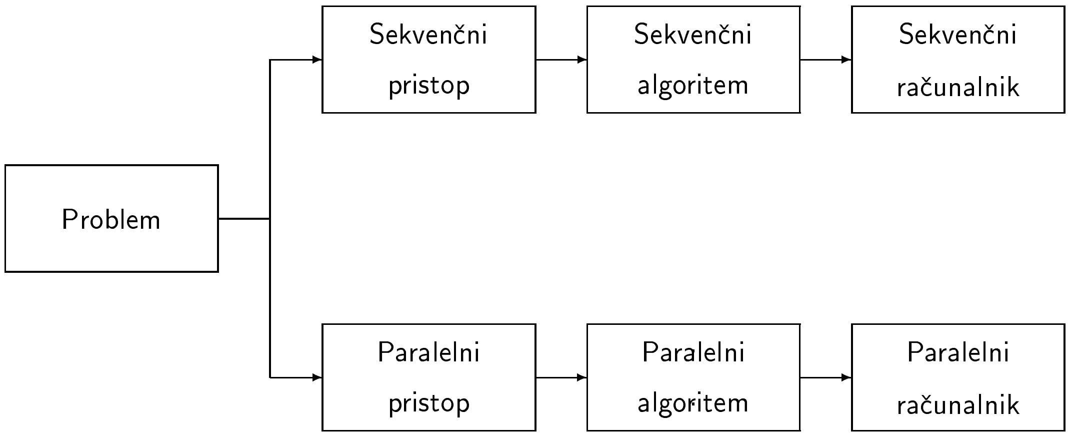
4. A. Silberschatz, J. Peterson, Operating Systems Concepts, Addison-Wesley, 1991.

4 Paralelizmi

Algoritem je z nizom navodil podan postopek za rešitev danega problema. Če je problem rešljiv, ga lahko v splošnem rešimo na več načinov. Za rešitev problema obstaja več algoritmov.

Ni vsako zaporedje navodil tudi algoritem. Algoritem mora biti elementaren, nedvoumen in ustavljiv. Algoritem je elementaren, če je podan s takimi navodili, da jih tisti, ki mu je namenjen, razume. Mora biti nedvoumen, kar pomeni, da se pravilno izvrši v vsakih okoliščinah. Končati pa se mora v končnem številu korakov - biti mora ustavljiv.

Pri snovanju algoritmov imamo načeloma dve možnosti. Lahko uberemo sekvenčni pristop in iščemo algoritem za izvršitev na sekvenčnem računalniku, t.j. *sekvenčni algoritem*. Alternativna možnost sekvenčnemu je paralelni pristop, t.j. iščemo *paralelen (vzporeden) algoritem* za izvršitev na paralelnem računalniku.



Slika 10: Alternativna pristopa k reševanju problema.

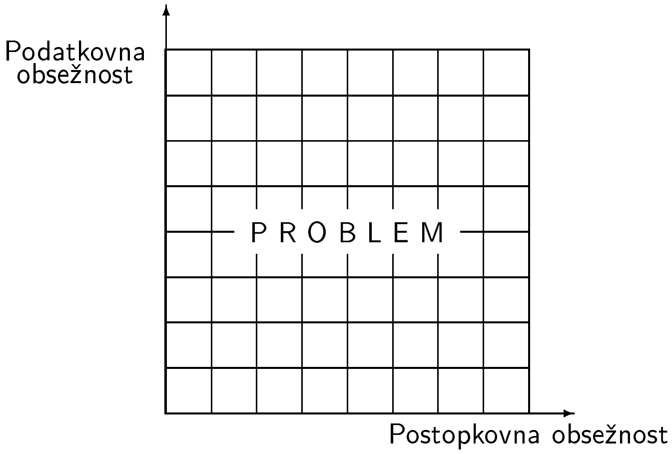
4.1 Oblike paralelizmov (oblike vzporednosti)

Vzporednost algoritmov se odraža v dveh osnovnih oblikah[1]:

- v številu operacij, ki se jih da opraviti sočasno in
- v številu podatkov, ki se jih da obdelati sočasno.

[1] Izvzeli smo "trivialno" obliko paralelizma. Pri trivialnem paralelizmu tečejo različne ali enake dejavnosti na enakih ali različnih podatkih medsebojno neodvisno - komuniciranje ni potrebno. Na primer: izvrševanje istega programa/procesov na različnih podatkih.

K tem dvem oblikam lahko dodamo še tretjo obliko vzporednosti (asinhrona vzporednost), ki je manj očitna in krši osnovno pravilo nedvoumnosti algoritma, vendar se izkaže, da v nekaterih okoliščinah ravno tako privede do želenega rezultata.



S tem v zvezi govorimo o treh oblikah vzporednosti in dekompozicije problema:

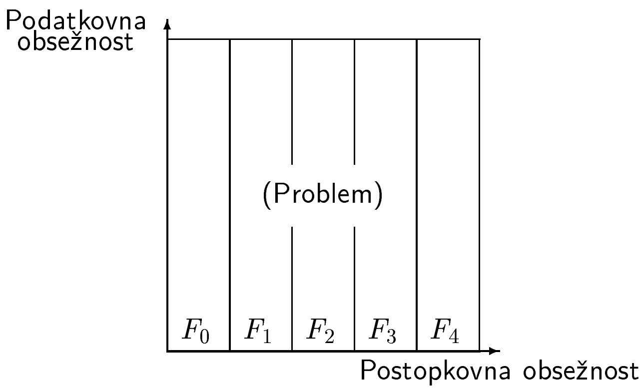
- Postopkovni (algoritmični ali cevovodni) paralelizem,
- Podatkovni (operandni ali geometrijski) paralelizem,
- Asinhroni (sproščeni) paralelizem ali relaksacija.

4.1.1 Postopkovni paralelizem

Pri postopkovni obliki paralelizma razbijemo operacijo F na operandu oz. podatku X na določeno število zaporednih delnih operacij ali podoperacij F_i,

$$F(X) = F_{s-1}(F_{s-2}(\ldots(F_1(F_0(X)))\ldots)).$$

ki zaporedoma pripeljejo do istega rezultata. Pri tej obliki paralelizma služi rezultat (izhod) iz operacije F_i za vhod v operacijo F_{i+1}. Na ta način se lahko vsaka od podoperacij realizira z njej dodeljeno strojno opremo (cevovodno stopnjo) PS_i. Sedaj lahko vsaka od cevodvodnih stopenj opravlja svojo (njej dodeljeno) podoperacijo sočasno z ostalimi cevovodnimi stopnjami, seveda nad drugimi operandi (oz. na operandih v različnih fazah obdelave). Dokler je tok (zaporedje) operandov tekoč, se izvršuje toliko sočasnih (pod)operacij kolikor je cevovodnih stopenj. Cevovodni algoritem sestavlja urejena množica opravil, pri čemer izhod iz enega opravila predstavlja vhod v naslednjega.



4.1.2 Podatkovni paralelizem

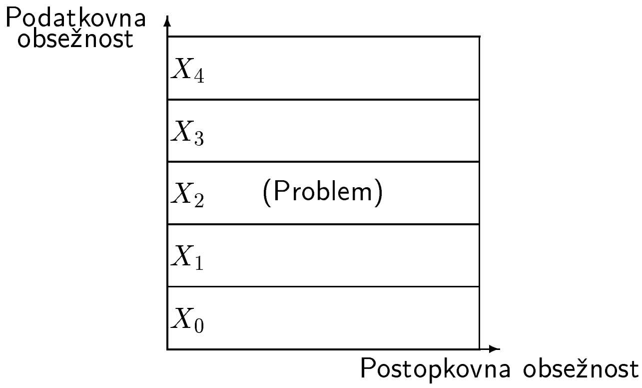
Pri podatkovnem paralelizmu razbijemo množico podatkov X, nad katero moramo opraviti operacijo F, na manjše množice, podmnožice X_i,

$$X = [X_0, X_1, \ldots, X_{d-1}].$$

Tako imamo d opravil,

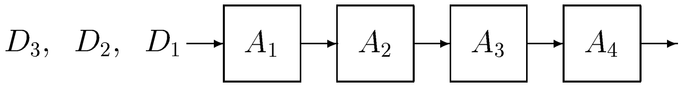
$$F(X) = F([X_0, X_1, \ldots, X_{d-1}]) = [F(X_0), F(X_1), \ldots, F(X_{d-1})].$$

Če imamo d procesorjev, lahko vsakemu od procesorjev dodelimo manjšo (pod)množico podatkov in s tem načeloma pospešimo izračun d-krat.

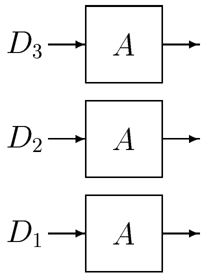


Primer:

Imamo algoritem, ki ga sestavljajo operacije A in deluje na podatkih D. Operacije sestavljajo štirje deli: A_1, A_2, A_3 in A_4. Podatke sestavljajo trije deli: D_1, D_2 in D_3. V primeru postopkovnega (cevovodnega) paralelizma imamo štiri procesorje - cevovodne stopnje, od katerih vsaka opravlja eno od operacij na vsakem in vseh podatkih (D).



V primeru podatkovnega paralelizma imamo tri opravila, od katerih vsako opravlja (vse) operacije na delu podatkov.



- V primeru postopkovnega paralelizma moramo poiskati delitev operacije na podoperacije.
- V primeru podatkovnega paralelizma delimo operande na podoperande.
- V primeru postopkovnega paralelizma porazdelimo postopek po procesorjih.
- V primeru podatkovnega paralelizma porazdelimo podatke po procesorjih.

V obeh primerih pridemo prej do rešitve problema. V enem in drugem primeru je v splošnem potrebno procese občasno sinhronizirati, kar vzame svoj čas (komunikacijsko breme). Pohitritev je zaradi tega manjša.

Primer:

Vzemimo operacijo seštevanja in seštejmo M števil $v_i, (i = 0, 1, \ldots, M-1)$,

$$V = \sum_{i=0}^{M-1} v_i.$$

Sekvenčno naredimo to tako, da akumuliramo vsoto z zaporednim prištevanjem podatka za podatkom k delni vsoti, dokler ne seštejemo vseh števil.
Izračun vsote bi lahko paralelizirali s podatkovno dekompozicijo. M števil delimo na enake (ali čimbolj enake) podmnožice velikosti M/N in jih dodelimo N procesorjem. Vsak procesor izračuna delno vsoto,

$$V_j = \sum_{i=j\times M/N}^{(j+1)\times M/N-1} v_i, \quad (j = 0, \ldots, N-1).$$

Nato N delnih vsot seštejemo in dobimo rezultat,

$$V = \sum_{j=0}^{N-1} V_j.$$

To seštevanje moramo narediti sekvenčno:

- v primeru komunikacije na podlagi sistema sporočil/povezovalnega omrežja bo $N-1$ procesov/procesorjev poslalo svojo delno vsoto enemu od procesov/procesorjev. Ta procesor izračuna končni rezultat.

- V primeru komunikacije na podlagi skupnega pomnilnika vsak proces/procesor prišteje svojo delno vsoto skupni vsoti - spremenljivki V. Prištevanja morajo biti narejena v kritičnem delu - potrebna je sinhronizacija.

4.1.3 Asinhroni paralelizem

V primeru asinhronega paralelizma se skuša držati procese/procesorje cel čas zaposlene brez sinhronizacije, potreba po medprocesni komunikaciji (sinhronizaciji) se enostavno prezre. Procesorji ne čakajo na nove podatke (rezultate), ampak obdelujejo tiste, ki jih imajo ali jih lahko dobijo, t.j. najbolj sveže oz. razpoložljive podatke. Čeprav se zdi, da na ta način ni mogoče priti do želenega rezultata, pa se v praksi pokaže, da obstaja veliko problemov, kjer je asinhroni paralelizem sprejemljiv – dobro izražen. Tipične primere najdemo pri numeričnem reševanju velikega sistema enačb, ki nastanejo z diskretizacijo (sistemov) (parcialnih) diferencialnih enačb, na primer Gauss Seidel-ova, Jacobi-jeva iteracija, "relaksacija".
Take algoritme je zelo težko analizirati.

4.2 Pohitritev in učinkovitost

Paralelen algoritem je algoritem, ki predvideva sočasno izvrševanje dveh ali več dejavnosti. Motivacija za vzporednen pristop k reševanju problema je *pohitritev*. Čim več operacij se opravi sočasno nad čim več podatki, tem prej bomo rešili problem. Pri snovanju paralelnih algoritmov nas zanima odgovor na vprašanje:

> kako čim prej, ali vsaj v doglednem času, z danimi sredstvi priti do želenega rezultata.

Iščemo take algoritme, ki nas pripeljejo do želenega cilja prej kot strogo zaporedno reševanje problema (operacije se izvršuje zaporedno nad zaporedjem podatkov).
Denimo, da izvršitev sekvenčnega algoritma za rešitev danega problema na sekvenčnem računalniku traja čas T_S, izvršitev paralelnega algoritma na paralelnem računalniku z N procesorji pa čas T_{P_N}. Pohitritev S_N definiramo kot razmerje obeh časov,

$$S_N = \frac{T_S}{T_{P_N}} \tag{1}$$

Primer:
Naj bo $T_S = 12$ in $T_{P_5} = 3$ (imamo pet procesorjev). Pohitritev je $S_5 = 12/3 = 4$. Z enim procesorjem torej rešimo problem v 12 enotah časa, s petimi procesorji pa štirikrat prej.

Primer:

Realizirajmo operacijo F cevovodno. Naj bo T čas za izvedbo ene od podoperacij. Naj bo s število podoperacij za realizacijo operacije F. V primeru sekvenčnega izvrševanja, bo M podatkov obdelanih v času $T_s = s \times T \times M$. V primeru cevovodnega paralelizma bo ta čas $T_P = s \times T + T \times (M - 1)$, pohitritev pa je

$$S = \frac{(s \times M)}{(s + M - 1)} \approx \frac{s}{1 + s/M},$$

ki gre proti s, ko M narašča.
Pohitritev ni zastonj. Čim manj dodatnih sredstev je potrebnih za pohitritev, bolj je paralelni algoritem *učinkovit.* Zato iščemo take rešitve, ki dosežejo čim večjo pohitritev ob čim nižjih stroških "paralelizacije". Učinkovitost definiramo kot pohitritev na en procesor,

$$E_N = \frac{S_N}{N}. \tag{2}$$

Za naš prvi primer je učinkovitost $E_5 = 4/5$. Načeloma skušamo doseči linearno pohitritev in učinkovitost ena, kar pomeni, da hočemo z N procesorji v primerjavi z enim procesorjem doseči N-kratno pohitritev. Zaradi številnih omejevalnih faktorjev je to težko doseči. Vsekakor si lahko mislimo, da poljubna pohitritev ni možna in da le-ta tudi zahteva dodatna sredstva. Predvsem pa se moramo zavedati, da pohitritev, ki jo želimo doseči, ni odvisna samo od naših hotenj (in naporov), ampak od narave problema. Nekateri problemi so sami po sebi izrazito sekvenčne narave, t.j. izkazujejo visoko stopnjo sekvenčnosti.

Definicija pohitritve (en. 1) zahteva dodaten komentar. Možen bi bil naslednji pomislek:

> pohitritev je definirana z razmerjem časov. Čas, ki je potreben za izvršitev algoritma pa ni odvisen samo od algoritma, ampak tudi od dotičnega računalnika.

Kot absolutno merilo bi lahko vzeli izvršitev najhitrejšega znanega sekvenčnega algoritma na najhitrejšem sekvenčnem računalniku, vendar je v praksi to težko izvedljivo. Možni alternativi sta:

- T_S - izvršitev sekvenčnega algoritma na enem procesorju paralelnega računalnika,
- T_S - izvršitev paralelnega algoritma na enem procesorju paralelnega računalnika.

Dilema je pravzaprav v tem, da ni skupnega soglasja, s čim naj paralelni algoritem primerjamo. Kaj vzamemo za referenco je odvisno od konkretnega primera in namena.

4.2.1 Resnična pohitritev

V tem primeru primerjamo paralelen algoritem z najbolšim (znanim) sekvenčnim algoritmom, oba pa izvršimo na izbranem (svojem) paralelnem računalniku, razmerje dobljenih časov je merilo za pohitritev:

$$S_N(n, N) = \frac{\text{Sekvenčni algoritem na enem procesorju paralelnega računalnika}}{\text{Paralelni algoritem na N procesorjih paralelnega računalnika}}. \tag{3}$$

Dobljena vrednost je v pogledu pohitritve pristranska, t.j. naklonjena paralelnim arhitekturam. V splošnem je lažje graditi hitrejše sekvenčne (enoprocesorske) računalnike kot je posamezen procesor paralelnega računalnika.

4.2.2 Relativna pohitritev

Včasih nas zanima, koliko hitreje rešimo dan problem na svojem (razpoložljivem) paralelnem računalniku z N procesorji v primerjavi s sekvenčnim računanjem. Zato primerjamo čas izvajanja paralelnega algoritma na enem procesorju paralelnega računalnika s časom izvajanja na N procesorjih:

$$S_N(n, N) = \frac{\text{Paralelni algoritem na enem procesorju paralelnega računalnika}}{\text{Paralelni algoritem na N procesorjih paralelnega računalnika}}. \tag{4}$$

Pohitritev je odvisna od velikosti problema (n) in števila razpoložljivih procesorjev (N). Lahko bi nas zanimalo, kako se pohitritev za dano število procesorjev spreminja z velikostjo problema, ali kako je za konstantno velikost problema pohitritev odvisna od števila procesorjev. Na ta način bi lahko prišli do maksimalne možne pohitritve za izbran algoritem.
V pogledu pohitritve je merilo naklonjeno paralelnemu algoritmu. Zaradi dodatnega bremena, ki ga prinese paralelna zasnova problema, se lahko čas izvrševanja paralelnega algoritma na sekvenčen način v primerjavi s sekvenčnim algoritmom v splošnem tudi podaljša.

4.2.3 Absolutna pohitritev

Kot absolutno merilo vzamemo izvršitev najhitrejšega znanega sekvenčnega algoritma na najhitrejšem sekvenčnem računalniku in ga primerjamo z izvršitvijo izbranega paralelnega algoritma na danem paralelnem računalniku. To daje podlago za objektivno vrednotenje paralelnih programov. Ker največkrat nimamo na razpolago niti najhitrejšega sekvenčnega računalnika niti najboljšega algoritma, je računanje absolutne pohitritve v praksi težko izvedljivo.

4.3 Dejavniki, ki omejujejo pohitritev

Smisel paralelizacije je pohitritev. Jasno je, da poljubna pohitritev ni mogoča. Vprašajmo se, kakšna pohitritev je sploh možna oziroma kaj jo omejuje.
Načeloma omejuje pohitritev dvoje, in sicer:

- izgube (časa), ki jih prispevajo stranski učinki paralelizacije oziroma dekompozicije problema,
- sekvenčna narava problema: "vsak problem vsebuje določen delež, ki se ga ne da paralelizirati".

4.3.1 Stroški dekompozicije

Sama dekompozicija problema ni zastonj. Kaže se na več načinov:

- Prirastek števila ukazov - zvečanje programskega bremena, (ang. Software overhead). V splošnem je potrebno zaradi paralelne dekompozicije za rešitev problema večje število ukazov (inicializacija spremenljivk, indeksiranje, ...).
- Neenakomerna porazdelitev bremena (neenakomerna porazdelitev podatkov ali neuravnotežena zahtevnost podoperacij). Breme naj bo enakomerno porazdeljeno, kajti hitrost je na koncu odvisna od najpočasnejšega člena).
- Porast komunikacijskega bremena (problem sinhronizacije, nezmožnost prekrivanja komunikacije s procesiranjem).

Če se vrnemo k problemu seštevanja M števil, ugotovimo:

- potrebna je distribucija podatkov po procesorjih, kar vzame svoj čas,
- v kolikor podatke ne porazdelimo enakomerno po procesorjih, se čas, ko so izračunane vse delne vsote, podaljša,
- delne vsote je treba zbrati skupaj ali sešteti sekvenčno (v kritičnem delu) - potrebna je komunikacija/sinhronizacija.

Kot bi pričakovali, komunikacijsko breme narašča s stopnjo dekompozicije ali t.i. zrnatostjo (ang. granularity) dekompozicije. Z drobljenjem problema na podprobleme, je delež računanja, ki odpade na posamezen proces/procesor vse manjši, komunikacijske zahteve pa naraščajo. To postane še bolj kritično, če se računanja ne da časovno prekrivati s komunikacijo. Lahko se zgodi, da večina procesov ne more napredovati, ker je napredovanje pogojeno s komunikacijo (npr. čakanje na podatke, delne rezultate). Pri majhni stopnji dekompozicije komunikacijsko breme navadno ne pride do izraza.

4.3.2 Amdahlov zakon

Amdahl je opozoril, da je pohitritev, ki se jo da doseči s paralelnim računanjem, omejena z deležem sekvenčnosti (stopnjo sekvenčnosti) danega problema. Po Amdahlu vsak problem sestavljata dva dela:

- del, ki se ga da reševati sočasno (vzporedno), t.i. paralelni del, ki se ga da paralelizirati in
- sekvenčni del, ki se ga ne da paralelizirati. Poljubna pohitritev zato ni možna, ne glede na število razpoložljivih procesorjev.

Naj bo problem tak, da vzame izvršitev sekvenčnega dela čas t_s, izvršitev paralelnega dela pa čas t_p. Sekvenčna rešitev problema zahteva čas

$$T_S = t_s + t_p.$$

Paralelna rešitev problema na N procesorjih traja čas

$$T_{P_N} = t_s + t_p/N.$$

Pohitritev s paralelizacijo znaša

$$S_N = \frac{1}{F + \frac{1}{N}(1-F)}, \tag{5}$$

kjer je $F = t_s/T_S$ stopnja sekvenčnosti problema, $0 \leq F \leq 1$. V primeru, ko je $F = 0$ je pohitritev linearna funkcija števila procesorjev. V splošnem je pohitritev nižja. V skrajnem primeru, ko je $F = 1$, pohitritve ni ($S = 1$).
Enačba (5) pravi, da že pri razmeroma majhni stopnji sekvenčnosti (na primer 1 %) pohitritev z večanjem števila procesorjev sprva sicer narašča, nato pa z večanjem števila procesorjev kaj dosti ne pridobimo. Pohitritev je omejena s stopnjo sekvenčnosti[2]

$$S_N \leq \frac{1}{F}.$$

Naj bo $F = 0.02$ (2 %) in $N = 10$. Pohitritev je $S_{10} = 8.5$. Povečajmo število procesorjev desetkrat, $N = 100$. S tem dosežemo $S_{100} = 33.6$. Na račun 10 krat več procesorjev smo pohitrili rešitev za $33.6/8.5 \approx 4$ manj kot štirikrat.
Amdahlov zakon pravi, da se zaradi sekvenčne narave problemov ne splača graditi računalnikov z zelo velikim številom procesorjev. Trditev temelji na predpostavki, da z naraščanjem zahtevnosti problema (na primer števila podatkov) narašča sorazmerno tudi sekvenčni delež (stopnja sekvenčnosti je konstanta). Praksa kaže, da to redkokdaj drži. Vendarle je Amdahlov rezultat pomemben, ker je opozoril na ta problem.

[2] Bolj splošno: vsak problem vsebuje delež, za katerega se v danih okoliščinah reševanja ne da izboljšati.

4.3.3 Gustafsonov zakon

Interpretacija Amdahlovega zakona v kontekstu paralelnega računanja je bila v preteklosti deležna precej kritike. Praksa je namreč pokazala, da se da z masovnim paralelizmom (velikem številom procesorjev) bistveno pohitriti reševanje sicer zahtevnih problemov. Amdahlov zakon gradi na predpostavki, da je stopnja sekvenčnosti konstanta, ki se ne spremenja z velikostjo problema. V praksi nas zanima paralelno reševanje zelo zahtevnih problemov, za katere to redkokdaj drži. Dejansko z rastjo problema narašča tisti del, ki se ga da paralelizirati oziroma reševati paralelno, če imamo na razpolago zadosti procesorjev, medtem ko delež sekvenčnosti ne narašča. Naraščanje sekvenčnosti z rastjo problema je prej posledica napačnega pristopa, kot resnične sekvenčne narave problema. Povečevanje števila procesorjev je s tega stališča smiselno.
Skoraj linearna pohitritev, ki jo je dosegel z masovnim paralelizmom na 1024 procesorjih, je Gustafsona (1988) vodila do naslednje ugotovitve. Stopnjo sekvenčnosti je podal (normiral) relativno glede na paralelen algoritem za N procesorjev,

$$F_G = \frac{t_s}{T_{P_N}} = \frac{t_s}{t_s + t_p(N)} \tag{6}$$

Stopnja vzporednosti pa je potem $(1 - F_G)$,

$$1 - F_G = \frac{t_p(N)}{t_s + t_p(N)}.$$

Na enem procesorju bi morali vzporedni del obdelati sekvenčno, za kar bi rabili $t_p(N) \times N$ časa. Pohitritev potem je:

$$S_{N_G} = \frac{t_s + t_p(N)N}{t_s + t_p(N)} = F_G + (1 - F_G)N = N - (N-1)F_G. \tag{7}$$

Tej pohitritvi, ki je definirana na osnovi sekvenčnosti normirane glede na paralelni algoritem, se večkrat reče normirana oziroma "skalirana" pohitritev.

Primer:
Z izvršitvijo na $N = 10$ procesorjih smo ugotovili $F = 0.6$, namreč 40 % časa porabimo na paralelnem delu in 60 sekvenčnem. Po Gustafsonu je pohitritev $S_{N_G} = 10 - 5.4 = 4.6$. Po Amdahlu je pohitritev $S_{N_A} = 10/6.4 = 1.6$.
Izračun pohitritve po Amdahlu v zgornjem primeru je kajpak zavajajoč (napačen). Stopnja sekvenčnosti po Amdahlovi definiciji ni enaka tisti po Gustafsonu. Vprašanje pa je, in v tem je bistvo problema, kako ugotoviti (oceniti) stopnjo sekvenčnosti, kot jo razume Amdahl.
Kasneje (1996) je Shi pokazal, da sta ob pravilni interpretaciji stopnje sekvenčnosti oba zakona ekvivalentna.

Primer:
Z osebnim avtom potujete iz Ljubljane v Novo Gorico. Po Ljubljani in Novi Gorici vozite 50 Km/h. Po Ljubljani do avtoceste je 6 Km. Z avtoceste po Novi Gorici je 4 Km. Pred semaforji stojite 8 minut, skozi "Rebrnice" vozite 15 minut (10 Km pri 40 Km/h). Ostalo vozite po avtocesti. Avtocesta je dolga 100 Km. Imate naslednje možnosti:

1. po avtocesti vozite s 50 Km/h, t.j. z isto hitrostjo kot po mestu,
2. po avtocesti vozite 100 Km/h,
3. po avtocesti vozite 150 Km/h ?,
4. po avtocesti vozite 200 Km/h ! ?,
5. po avtocesti vozite 250 Km/h ! ! !

Kakšne so "pohitritve" v primerjavi s hitrostjo 50 Km/h tudi na avtocesti?
Kako bi problem razlagali z vidika Amdahlovega oziroma Gustafsonovega zakona?

4.4 Glavna literatura

1. Parallel Processing Course, Undergraduate Year 2, http://www.cm.cf.ac.uk/Parallel/Year2/
2. Parallel Processing Course, Undergraduate Year 3, http://www.cm.cf.ac.uk/Tltp/html/ThYeCo.html
3. Ian Foster, Designing and Building Parallel programs, Addison-Wesley, 1995.
4. J. L. Gustafson. Reevaluating Amdahl's Law. Communications of the ACM, 31(5):532–533, May 1988.
5. Yuan Shi, Reevaluating Amdahl's Law and Gustafson's Law, Computer and Information Sciences Department, Temple University (MS:38-24), October 1996.

5 Snovanje paralelnih algoritmov

V splošnem obstaja več možnosti za rešitev danega problema. Med možnimi rešitvami vse niso enako dobre. Cilj snovanja algoritmov je poiskati optimalno oziroma tako rešitev, ki je v danih okoliščinah najbolj primerna. Zato pa je potreben sistematičen pristop.

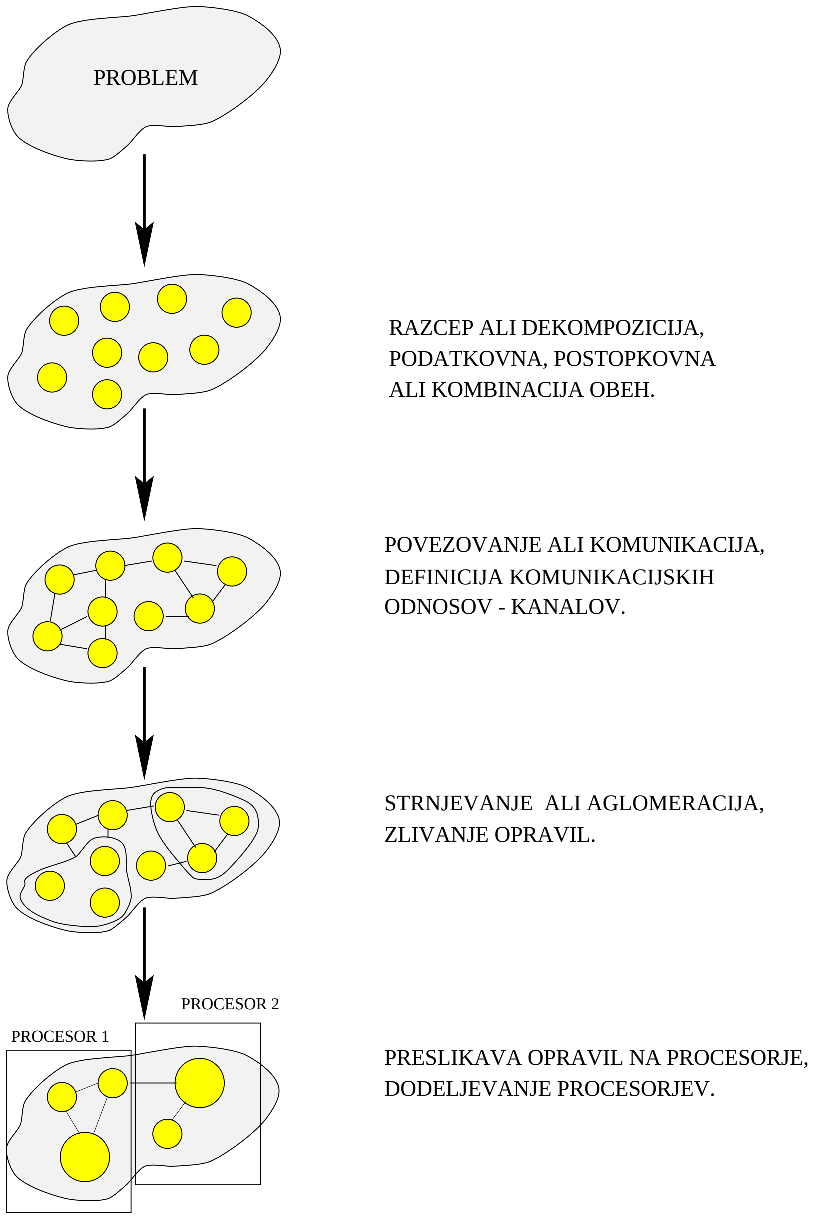
Vprašajmo se, kakšen naj bo dober paralelen algoritem? Najbrž tak, da nas pripelje čim hitreje do cilja - do rešitve problema. V primerjavi s sekvečnim algoritmom lahko rečemo, da bo temu tako, če bo odražal visoko stopnjo paralelnosti - dobro zajel paralalelizme, ki so prisotni v problemu. Algoritem in njegova realizacija (program) naj bo tudi kar se da učinkovit - dobro naj izkoristi arhitekturo razpoložljivega računalnika. Glede na to, da število procesorjev na računalnik s časom narašča, bi od dobrega algoritma pričakovali tudi, da bo izkoristil povečanje števila procesorjev. Program, ki je dober samo za dano število procesorjev oziroma ne zmore izkoristiti toliko procesorjev, kolikor jih je na razpolago, je s tega vidika slab program. Pa tudi tak program, ki s povečanjem zahtevnosti problema ne zmore izkoristiti večjega števila procesorjev, ni dober paralelen program. Algoritem naj bo sposoben "skaliranja" s problemom.
Komunikacije med dejavnostmi, ki so potrebne za rešitev problema in napredujejo sočasno, so sicer nujne, vendar nezaželene. Paralelen računalnik (računalniški sistem) je v bistvu sistem med seboj povezanih sekvenčnih procesorjev, od katerih ima vsak svoj (lokalni) pomnilnik, do katerege ima hiter dostop. Paralelen računalnik torej v sebi združuje določeno število med seboj povezanih von Neumannovih računalnikov. Ti računalniki med seboj komunicirajo bodisi preko sistema povezav (povezovalnega/ komunikacijskega omrežja) bodisi preko skupnega pomnilnika. Dostop do skupnega pomnilnika (beri/piši) ali prenos sporočil preko komunikacijskega omrežja (sprejmi/pošlji) je počasnejši kot dostop do lokalnega pomnilnika. Zato je želeno, da se večina referenc znotraj programa nanaša na lokalen pomnilnik, je lokalnega značaja. Dober algoritem naj torej upošteva (odraža) lokalnost računanja.
Tako smo prišli do treh osnovnih lastnosti, ki naj jih ima dober paralelen algoritem:

- vsebuje naj visoko stopnjo sočasnosti - paralelnosti,
- dopušča naj povečanje zahtevnosti problema in izkorišča povečanje števila prosesorjev - "skaliranje" s problemom,
- temelji naj na lokalnosti računanja - lokalnosti komuniciranja.

Pri snovanju paralelnih algoritmov je to potrebno upoštevati.

5.1 Potek snovanja

Snovanje paralelnih algoritmov/programov zahteva sistematičen pristop. Snovanje (povzeto po Fosterju 95) načeloma poteka takole (Slika 11):



Slika 11: Potek snovanja paralelnega programa.

1. Razcep problema ali dekompozicija. To vključuje podatkovno dekompozicijo, postopkovno dekompozicijo ali kombinacijo obeh. Običajno je ena od obeh oblik paralelnosti v problemu bolje izražena, bolj očitna. Dekompozicija bi to morala upoštevati. V tej fazi snovanja nas lastnosti razpoložljive računalniške arhitekture ne zanimajo. Stopnja dekompozicije naj bo raje prevelika kot premajhna.

2. Povezovanje. V tej fazi definiramo potrebne komunikacijske relacije, strukture in komunikacijske algoritme (kdo s kom komunicira in na kakšen način).

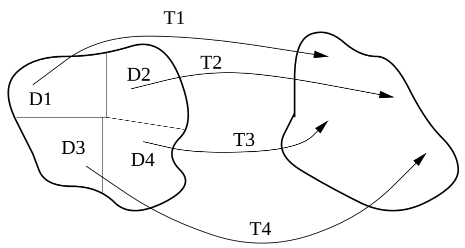
3. Združevanje ali aglomeracija. V tej fazi snovanja proučimo možnosti združevanja manjših opravil v večja opravila. To sicer zmanjšanje stopnjo vzporednosti vendar z namenom, da bi zmanjšali stroške komunikacije - cilj je poiskati primerno stopnjo zrnatosti.

4. Preslikava na dano računalniško arhitekturo. To vklučuje prilagoditev in namestitev programa na dotični računalniški sistem, to je porazdelitev opravil po procesorjih (razvrščanje) oziroma dodelitev procesorojev procesom (opravilom).

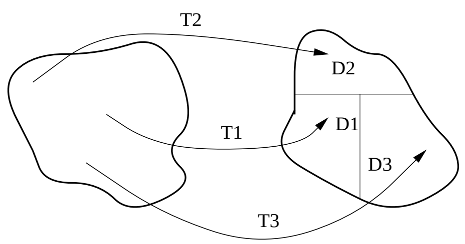
Posamezne faze snovanja med sabo niso neodvisne, kajti rezultat ene vpliva na potek druge. Zato je v splošnem potrebno razvojni cikel tudi večkrat ponoviti. Želimo pa si, da bi bila ta odvisnost čim manj izražena.

5.1.1 Razcep ali dekompozicija

V fazi razcepa skušamo problem razdeliti na take sestavne dele, ki bi se potencialno lahko reševali sočasno. Pri tem nas število razpoložljivih procesorjev ne zanima. Velja praktičen nasvet: število opravil naj bo v tej fazi vsaj za velikostni razred večje od pričakovanega števila procesorjev. Pri razcepu problema lahko začnemo s podatkovno dekompozicijo. Domeno računanja razdelimo na manjše poddomene, t.j. skupine podatkov približno enake obsežnosti. V naslednjem koraku podatkom pridružimo operacije. Rezultat je določeno število, običajno razmeroma majhnih opravil. Nato nadaljujemo z razcepom operacij na podoperacije.
Pri podatkovni dekompoziciji problema komunikacijske potrebe največkrat niso takoj razvidne - so slabo izražene. Posledica je veliko komunikacijsko breme. Možna je tudi obratna pot. Najprej razcepimo postopek na podoperacije, tem priredimo podatke oziroma skupine podatkov.
Največkrat je najbolj očitna možnost dekompozicije vhodnih podatkov (vhodna podatkovna dekompozicija), slika 12, vendar je možna in včasih primernejša dekompozicija na podlagi izhodnih rezultatov (slika 13).



Slika 12: Ponazoritev vhodne podatkovne dekompozicije. Opravila dostopajo do skupne izhodne podatkovne strukture, zaradi česar je potrebna sinhronizacija.



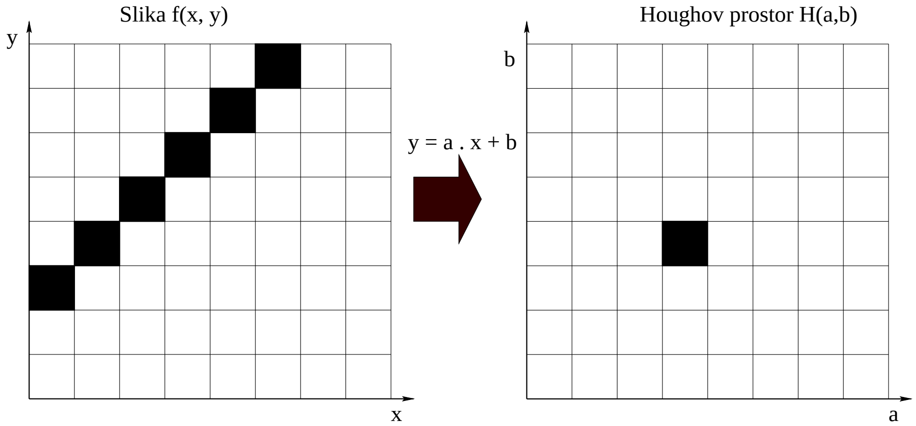
Slika 13: Ponazoritev izhodne podatkovne dekompozicije. Opravila dostopajo do skupne vhodne podatkovne strukture, zaradi česar je potrebna sinhronizacija.

Primer: Houghov transform

Dobro znana slikovna operacija pri obdelavi digitalnih slik je Houghova transformacija. Pride nam prav pri iskanju ravnih struktur v sliki, posplošena Houghova transformacija pa je primerna za iskanje poljubnih oblik.
Houghova transformacija preslika sliko $f(x,y)$ v parametrični prostor $H(a,b)$ po pravilu (enačbi premice):

$$y = a.x + b$$

Po tej enačbi dani premici (linearni strukturi) v sliki pripada v Houghovem prostoru točka (a,b) (Slika 14). Oziroma, kolinearne točke v sliki $f(x,y)$ se preslikajo v eno samo točko $H(a,b)$. Velja tudi obratno. Dani točki (x,y) v sliki pripada premica $b = -x.a+y$. Povedano drugače, šopu premic skozi točko (x,y) ustreza v Houghovem prostoru premica $b = -x.a + y$. Ali, šopom premic skozi kolinearne točke v sliki pripada šop premic skozi točko $H(a,b)$. Sečišče premic v transformiranem prostoru je torej pokazatelj ravne strukture v sliki. Zadnje dejstvo je podlaga za izračun transforma.



Slika 14: Ponazoritev Houghove transformacije.

Tipičen primer uporabe Houghove transformacije je naslednji:

- v originalni sliki poiščemo robne točke,
- robne točke preslikamo v parametrični prostor,
- v parametričnem prostoru poiščemo točke (zaradi šuma gruče točk), ki pripadajo - opisujejo ravne strukture v prvotni sliki.

Računanje transformacije same pa poteka običajno na naslednji način. Vhod je digitalna slika $f(i,j)$ robnih točk. Izhod je diskretiziran parametričen prostor $H(k,l)$.

- Elementu ali "celici" $H(k,l)$ rečemo akumulator. Vrednost vseh akumulatorjev je sprva nič.

- Skozi vsako točko $f(i,j)$ v sliki (v mislih) povlečemo šop premic, t.j. načrtamo premico v Houghovem prostoru tako, da povečamo za ena vrednost vseh akumulatorjev, ki zadoščajo pogoju $l = -k.i + j$.
- Sečiščem premic v Houghovem prostoru ustrezajo akumulatorji z veliko vrednostjo, ti pripadajo kolinearnim točkam (ravnim strukturam) slike.

Houghova transformacija je računsko zahtevna in je reda $O(K \times L \times E$, kjer je $K \times L)$ velikost (število akumulatorjev) Houghovega prostora in E je število robnih točk slike (Lahko rečemo, da narašča s tretjo potenco). Na podlagi podatkovne dekompozicije je možnih več načinov paralelizacije računanja. V primeru vhodne dekompozicije imamo:

- Vhodno sliko delimo na podslike približno enake velikosti, v skrajnem primeru na posamezne slikovne elemente,
- podslikam priredimo postopek - računanje premic.
- rezultat računanja je Houghov prostor.

Za tako dekompozicijo lahko povemo naslednje:

- breme računanja je neenakomerno porazdeljeno in naprej neznano, odvisno je od števila robnih točk v podsliki.
- komunikacijsko breme je veliko - vsa opravila dostopajo do skupnega polja akumulatorjev, spreminjajo vrednosti akumulatorjev, kar zahteva sinhronizacijo.

Možna bi bila izhodna podatkovna dekompozicija:

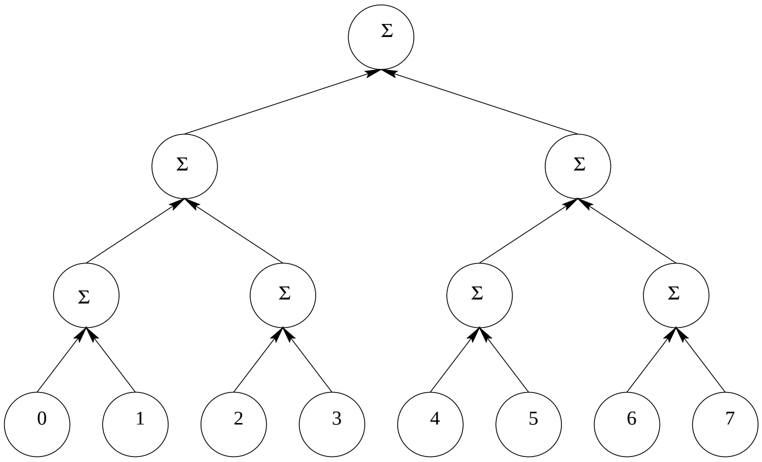
- Houghov prostor delimo na manjše dele po stolpcih (za dani l računanje za vse vrednosti k), po vrsticah (za dani k računanje za vse vrednosti l), na podmatrike, v skrajnem primeru na toliko delov kot je akumulatorjev,
- le-tem priredimo postopek, dobimo opravila za računanje premic za izbrane vrednosti parametrov k in l.

Ugotovimo,

- redundanca računanja je velika, breme računanja pa je enakomerno porazdeljeno.
- opravila (sočasno) dostopajo do skupne slike z operacijo branja, kar je manj kritično od operacije pisanja.

5.1.2 Deli in vladaj

Velikokrat se da odkriti sočasnot z razcepom po metodi "deli in vladaj". Rezultat operacije, kot je na primer seštevanje, dobimo na podlagi določenega števila vhodnih podatkov. Po metodi deli in vladaj postopoma delimo množico podatkov na (približno) enako velike podmnožice dokler se to da. Rezultat je razmeroma veliko število majhnih opravil, ki se bodo predvidoma odvijala sočasno. Slika 15 prikazuje rezultat postopne delitve na primeru seštevanja osmih števil. Končno vsoto generiramo na podlagi dveh delnih vsot štirih števil, ki se nadalje izračunata na podlagi dveh delnih vsot dveh števil. Tako namesto v osmih (N korakih) pridemo do rezultata v treh ($\log N$ korakih).



Slika 15: Razcep po metodi deli in vladaj.

5.1.3 Splošna načela dekompozicije

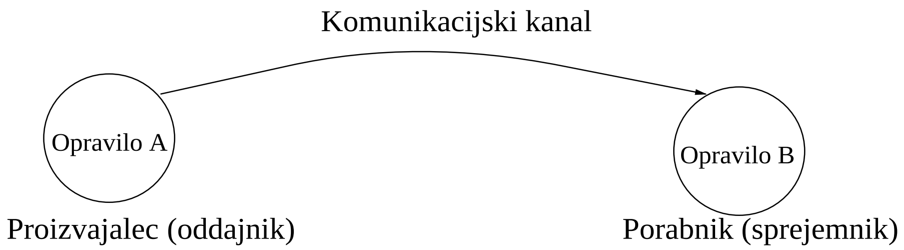
Pri dekompoziciji problema se velja držati naslednjih načel:

- dekompozicija naj bo fino zrnata - precej več opravil kot procesorjev.
- Opravila naj bodo približno enako velika.
- Z večanjem zahtevnosti problema naj se raje poveča število opravil kot velikost opravil.
- Dobro je proučiti alternativne možnosti razcepa, ker se le-te v kasnejših fazah snovanja lahko izkažejo za bolj ugodne.

5.1.4 Povezovanje - komunikacija

Od opravil, ki so rezultat razcepa, se pričakuje, da bodo napredovala paralelno, ne pa tudi neodvisno. Dejavnosti enega opravila vsaj občasno potrebujejo podatke/rezultate drugih opravil, brez katerih ne morejo napredovati. Potrebne so komunikacije. Pretok podatkov določimo v naslednji fazi snovanja. To pomeni, da določimo:

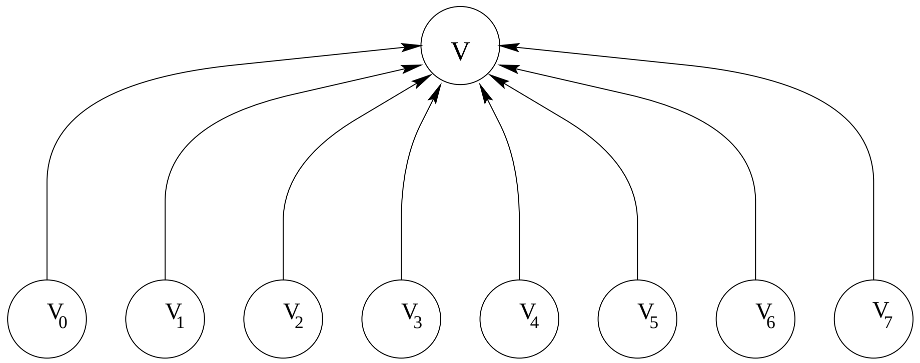
- komunikacijske kanale - kdo komunicira, kdo sprejema (uporablja) in kdo oddaja (proizvaja).
- algoritem oziroma protokole za komunikacijo.



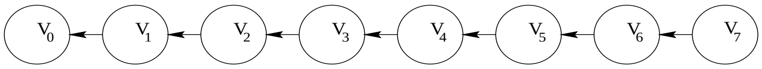
Slika 16: Komunikacija med opravili.

5.1.5 Komunikacijski vzorci

Oblike komuniciranja so lahko zelo raznolike, vsaka pa zahteva v nečem specifičen pristop. S to problematiko se je zato lažje ukvarjati, če identificiramo karakteristike posamezne oblike komuniciranja ter izdelamo metodologijo, ki je bolj primerna za dan tip - obliko komuniciranja.
Komunikacije so lahko lokalne ali globalne narave. Za *lokalne* komunikacije je značilno, da se odvijajo med razmeroma majhnim številom opravil. Določeno opravilo komunicira z razmeroma majhnim številom drugih opravil, ki so s tega vidika sosednja opravila. Sledi, da imajo opravila majhno število sosedov. V nasprotju z loklanimi komunikacijami pa v primeru globalnih komunikacij eno opravilo (ali vsa opravila) komunicira z velikim številom drugih opravil (lahko z vsemi opravili). Take komunikacije so s stališča paralelnih sistemov nezaželene. Slika 17 prikazuje primer globalne komunikacije za izračun vsote N števil ($N = 8$). Stopnja paralelnosti je majhna. Niti komunikacije niti računanje ni enakomerno porazdeljeno po opravilih. Slika 18 prikazuje primer lokalnih komunikacij za izračun vsote. Vsako opravilo k delni vsoti ($V = V + V_i$), ki jo dobi od desnega soseda prišteje svojo vrednost in rezultat pošlje levemu sosedu. Na koncu skrajno levo opravilo ima rezultat. Čeprav so računanje in komunikacije porazdeljene, je stopnja paralelnosti nizka (razen če opravila ne delujejo kot cevovod).
Nadalje so komunikacije lahko strukturirane ali nestrukturirane. V primeru *strukturiranih* komunikacij, se komunikacijskih vzorec regularno ponavlja. Rezultat je regularna komunikacijska struktura, na primer celično polje, drevo, piramida, metulj, hiperkocka. *Nestrukturirane* komunikacije so tiste, pri katerih skupnega komunikacijskega vzorca ni moč najti.



Slika 17: Primer globalnih komunikacij.

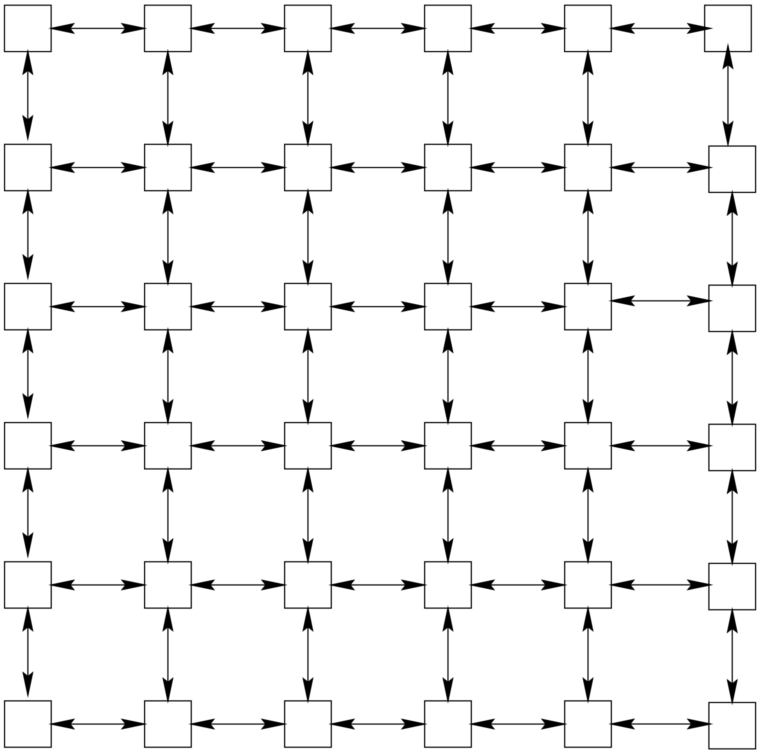


Slika 18: Primer porazdelitve komunikacij.

Komunikacije so lahko sinhronega ali asinhronega tipa. V primeru sinhronih komunikacij pride do komunikacije med proizvajalcem in porabnikom na podlagi obojestranske pripravljenosti na komuniciranje. V primeru asinhronih komunikacij pa teče oddajanje tudi brez eksplicitne pripravljenosti porabnika na sprejem.

Primer: Metode končnih diferenc

Strukturirana oblika lokalnih komunikacij nastane z diskretizacijo diferencialnih enačb po metodi končnih diferenc, glej sliko 19.



Slika 19: Komunikacijska shema na osnovi diskretizacije po metodi končnih diferenc.

Denimo, da nas zanima numerična rešitev Poissonove enačbe

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = f(x,y)$$

na kvadratu z danimi vrednostmi $u(x,y)$ na robu kvadrata (robnimi pogoji). Kvadrat razdelimo na majhne kvadratke velikosti $h \times h$. Dobimo kvadratno mrežo točk in iščemo rešitev v točkah $(x_i, y_j) = (i.h, j.h) = (i,j)$ tako, da enačbo "diskretiziramo" - aproksimiramo parcialne odvode s končnimi diferencami:

$$\begin{aligned}
\frac{\partial^2 u(x,y)}{\partial x^2} &\approx \frac{1}{h^2}(u(i+1,j) - 2u(i,j) + u(i-1,j)),\\
\frac{\partial^2 u(x,y)}{\partial y^2} &\approx \frac{1}{h^2}(u(i,j+1) - 2u(i,j) + u(i,j-1)).
\end{aligned}$$

Tako dobimo za vsako točko na mreži enačbo, ki podaja rešitev v dani točki v odvisnosti od rešitve v okoliških točkah ($h = 1$):

$$u(i,j) = \frac{1}{4}(u(i+1,j) + u(i-1,j) + u(i,j+1) + u(i,j-1) + f(i,j)).$$

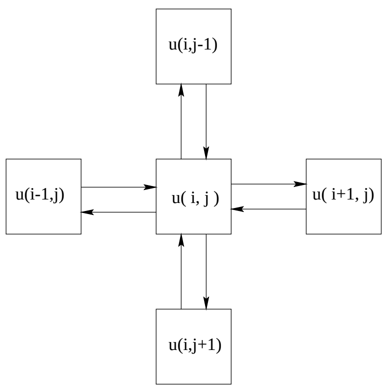
Eden od možnih pristopov za reševanje dobljenega sistema enačb je Jakobijeva iteracijska shema,

$$u^{t+1}(i,j) = \frac{1}{4}(u^t(i+1,j)+u^t(i-1,j)+u^t(i,j+1)+u^t(i,j-1)+f(i,j)), \quad (t=0,1,\ldots), \tag{8}$$

s katero iščemo boljši približek k pravi rešitvi na podlagi rešitve v prejšnji iteraciji (t).
Jacobijeva shema daje direktno podlago paralelnemu pristopu: vsaki točki priredimo opravilo $T(i,j)$, ki izvršuje naslednje zaporedje ukazov:

Opravilo $T(i,j)$

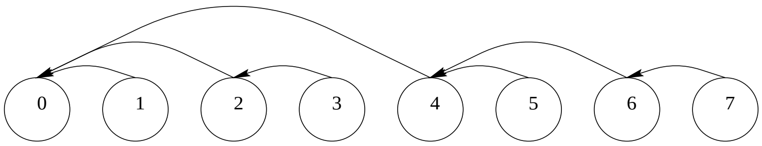
Za $t = 0, 1, \ldots$
  pošlji svojo rešitev $u^t(i,j)$ sosedom
  sprejmi rešitve od sosedov, $u^t(i+1,j), u^t(i-1,j), u^t(i,j+1), u^t)i,j-1)$
  reši enačbo 8



Slika 20: Opravilo $T(i,j)$ in okoliška opravila.

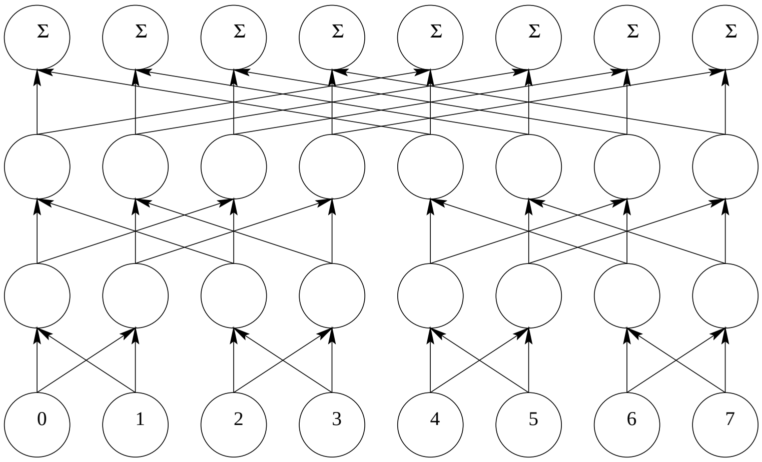
5.1.6 Strnjevanje ali aglomeracija

Rezultat razcepa in povezovanja je veliko število opravil z definiranimi komunikacijskimi relacijami. V tej fazi razvoja algoritma se lahko pokaže, da je komunikacijsko breme v primerjavi s paralelnostjo računanja preveliko. V fazi aglomeracije hočemo to razmerje popraviti z združevanjem majhnih opravil v večja opravila in hkrati približati rešitev konkretnemu računalniku. Prvo pravilo združevanja glasi: združi tista opravila, ki ne morejo napredovati sočasno.
V primeru računanja vsote na podlagi drevesne strukture ugotovimo, da morajo biti za nadaljevanje računanja prej izračunane delne vsote. Opravilo, ki računa vsoto štirih števil se ne more izvajati sočasno z opravili, ki računajo pare števil. Opravilo za računanje vsote štirih torej združimo z enim od opravil za računanje vsote para. Rezultat aglomeracije prikazuje slika 21.



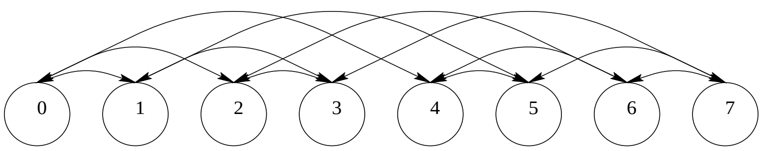
Slika 21: Združitev opravil, ki ne morejo napredovati sočasno.

Kadar stroški komunikacije presežejo stroške računanja, se splača razmisliti o podvajanju računanja in podatkov. Tudi na ta se da popraviti razmerje komunikacije/računanje v prid računanju. Denimo, da vsoto števil potrebujejo vsa opravila. Namesto distribucije vsote, ki jo izračuna eno opravilo, podvojimo računanje, glej sliko 22. Tako v $\log N$ korakih vsa opravila razpolagajo s končno vsoto.



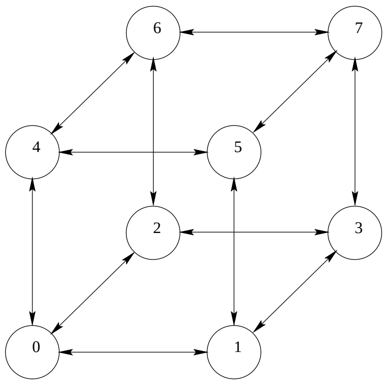
Slika 22: Podvajanje računanja.

Shemo opravil po aglomeraciji prikazuje slika 23. Vidimo, da je rezultat komunikacijska shema, v kateri je vsako opravilo povezano s tremi sosednimi opravili, stopnja povezanosti je 3. Komunikacijsko strukturo take vrste imenujemo hiper kocka stopnje N, kjer N pomeni število povezav med (sosednimi) opravili. Za naš primer je kocka skicirana na sliki 24.



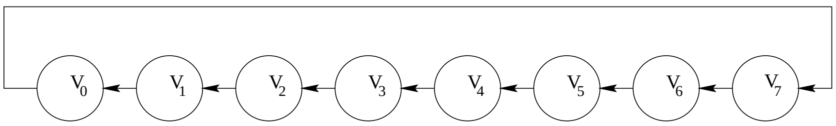
Slika 23: Podvajanje računanja in aglomeracija.

V primeru, da imamo linearno povezana opravila (Slika 18, končni rezultat pa potrebujejo vsa opravila, je učinkovita krožna komunikacijska shema (obroč), glej sliko 25. V obroču vsa opravila sočasno izvajajo isto operacijo prištevanja svoje vrednosti

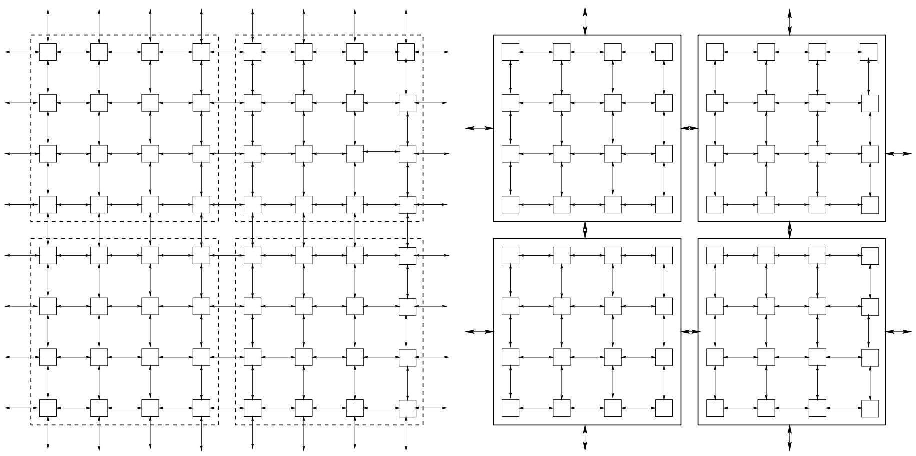


Slika 24: Metulj po aglomeraciji v obliki kocke.

k delni vsoti, to je $V_i = v_i + V_{i-1}$. Po N-1 korakih vsa opravila razpolagajo s končno vsoto - distribucija rezultata ni potrebna.



Slika 25: Podvajanje računanja namesto distribucije rezultata.



Slika 26: Zmanjšanje komunikacijskega bremena z aglomaracijo.

V primeru velikega števila lokalno povezanih opravil se splača preučiti razmerje "površina / prostornina", pri čemer je površina sorazmerna s komunikacijskim bremenom, prostornina z obsegom računanja. Z aglomeracijo se da razmerje komunikacija/računanje izboljšati v prid računanja, kot prikazuje slika 26. Slika prikazuje mrežo $8 \times 8 = 64$ lokalno povezanih opravil. Vsako opravilo komunicira s štirimi sosedi. Število podatkov, ki si jih opravila izmenjajo je $64 \times 4 = 256$. V primeru, da združimo $4 \times 4 = 16$ opravil v eno opravilo, dobimo 4 velika opravila, med katerimi je vsega skupaj 16 komunikacijskih kanalov, po katerih se opravila izmenjajo 4 podatkovne enote, ali v seštevku $4 \times (4 \times 4) = 64$ podatkov.

5.1.7 Preslikava algoritma na računalnik – dodelitev procesorjev

Po aglomeraciji dobimo določeno število opravil T_i, $(i = 0, 1, \ldots, N-1)$, ki morajo komunicirati med seboj. Denimo, da imamo računalnik z M procesorji P_j, $(j = 0, 1, \ldots, M-1)$, ki so na nek način povezani med seboj. Predpostavljamo, da je $N > M$ ali celo $N >> M$. Naša naloga je, da opravilom dodelimo procesorje oziroma opravila preslikamo na računalniški sistem.
Procesorje lahko dodelimo na več načinov. Med možnimi načini iščemo tako dodelitev, ki minimizira čas izvrševanja (od trenutka, ko prvo opravilo zaçne do trenutka, ko zadnje opravilo konča). Razen minimalnega časa lahko upoštevamo še druge kriterije, ki so posledica omejenih sredstev, na primer koliko časa si lahko privoščimo za iskanje optimalne dodelitve, kako se dodelitev obnaša pri večanju števila opravil (N) in/ali procesorjev (M), kolikšno je maksimalno število sočasnih komunikacijskih kanalov med procesorji, kakšno je maksimalno število opravil na procesor, ipd. Za problem dodelitve procesorjev procesom se smatra, da je za NP poln. Z drugimi besedami, verjetno ne obstaja algoritem, ki bi problem dodelitve rešil v polinomskem času (algoritem s polinomsko časovno zahtevnostjo). V praksi si pomagamo s hevristiko. Praktičen nasvet pri dodeljevanju procesorjev se glasi:

- dodeli opravila, ki lahko napredujejo sočasno, različnim procesorjem. To daje podlago sočasnemu napredovanju opravil.
- Dodeli opravila, ki intenzivno komunicirajo med seboj, istemu procesorju. To veča lokalnost komunikacij in računanja.

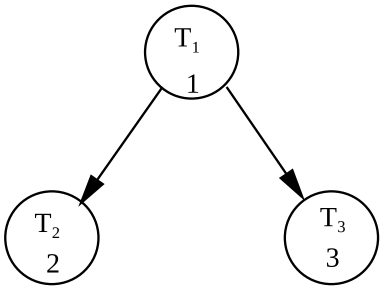
Velikokrat si zgornja kriterija nasprotujeta in potreben je kompromis.

Podatkovna dekompozicija pogosto privede do enako obsežnih opravil in strukturiranih lokalnih komunikacij. V takih primerih dodeljujemo procesorje tako, da zmanjšamo medprocesorsko komunikacijo. V primeru neenakomernih opravil in zahtevnih oblik komuniciranja se poslužujemo postopkov za uravnoteženje bremena - želimo, da bi bili procesorji približno enako obremenjeni.

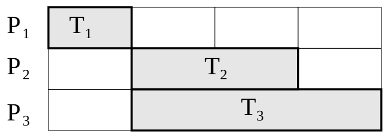
Rezultat postopkovne dekompozicije je pogosto veliko število kratkotrajnih opravil. To pomeni, da se breme se med reševanjem problema dinamično spreminja. V takih primerih se poslužujemo razvrščanja opravil, na primer: dodeli opravilo procesorju, ki je najmanj ali sploh ni obremenjen. Vedno pa se moramo zavedati, da tudi razvrščanje vzame svoj čas, ki ne sme izničiti pridobitve.

5.1.8 Uravnoteženje bremena in razvrščanje opravil

Kot rečeno določa paralelen algoritem množica med seboj odvisnih opravil. Medsebojno odvisnost opravil skupaj z zahtevnostjo (časom, ki je potreben za izvršitev) predstavimo z grafom opravil. Graf opravil je usmerjen graf, v katerem so opravila podana z vozlišči, medsebojna odvisnost pa s povezavami med vozlišči. Slika 27 prikauzje opravila T_1, T_2 in T_3, ki potrebujejo za izvršitev eno, dve in tri enote časa, pri čemer je začetek napredovanja opravil T_2 in T_3 pogojen s koncem izvrševanja oziroma rezultatom opravila T_1, sicer pa ti dve opravili napredujeta sočasno. Izvršitev algoritma na sekvenčnem računalniku bi trajala šet enot. V primeru izvršitve na triprocesorskem sistemu bi bila možna naslednja dodelitev procesorjev (slika 28). Izvrševanje programa traja 4 časovne enote, enako kot na dvoprocesroskem sistemu.

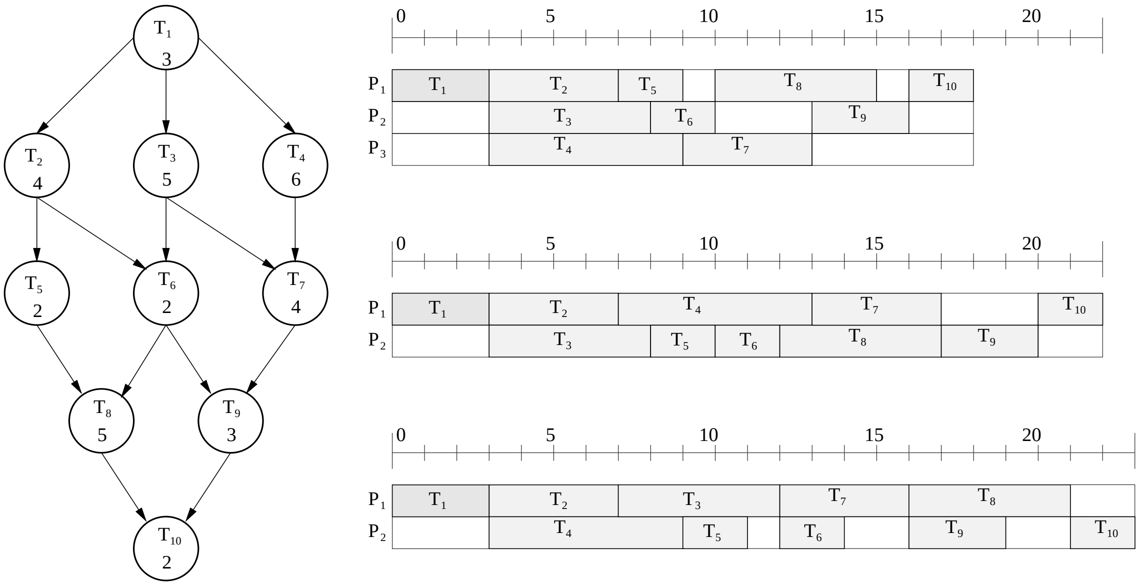


Slika 27: Graf opravil. Opravili T_2 in T_3 lahko napredujeta sočasno, vendar šele, ko opravilo T_1 konča.

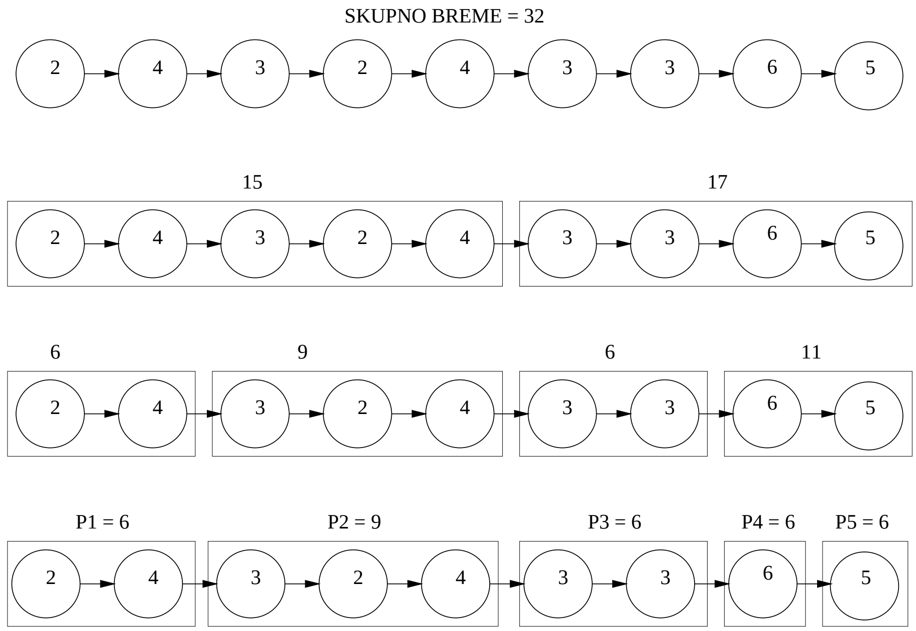


Slika 28: Ganttov diagram izvrševanja programa na triprocesorskem sistemu.

Nekoliko bolj zapletene odnose med opravili skupaj s potrebnimi časi za izvršitev prikazuje slika 29. Ugotovimo lahko, da bi za izvršitev na sekvenčnem računalniku potrebovali 36 enot časa. Skozi graf pelje več poti, vsaka pot predstavlja zaporedje opravil, ki se morajo izvršiti sekvenčno, določa njihov potek izvršitve. Z analizo poti skozi graf ugotovimo, da je "najtežja" (časovno najzahtevnejša) pot tista, katere izvršitev traja 18 enot. Algoritma se v krajšem času ne da izvršiti. Slika 29 prikazuje tudi možne načine dodelitve procesorjev v dvo in troprocesorskem sistemu. Optimalna pohitritev s tremi procesorji je $S_3 = 36/18 = 2$ in učinkovitost $E_3 = 2/3 = 0.67$. Pohitritev na dveh procesorjih je $S_2 = 36/22 \approx 1.6$, učinkovitost pa $E_2 \approx 1.6/2 = 0.8$.



Slika 29: Graf opravil in Ganttov diagram. Za sekvenčno izvršitev bi poterbovali 36 enot časa. Za paralelno izvršitev na treh procesorjih v najboljšem primeru potrebujemo 18 enot. Več procesorjev ne bi pomagalo. V primeru dveh procesorjev potrebujemo več časa.



Slika 30: Dodelitev procesorjev po metodi bisekcije.

V primeru velikega števila lokalno povezanih majhnih opravil dosežemo dodelitev procesorjev, ki je blizu optimalni, z uravnoteženjem bremena po metodi bisekcije. Metoda bisekcije zaporedoma deli množico opravil na dve manjši približno enako zahtevni podmnožici, dokler ni število podmnožic enako številu procesorjev. Slika 30 prikazuje eno od možnih dodelitev petih procesorjev devetim linearno povezanim opravilom po bisekcijski metodi.

5.2 Glavna literatura

1. I. Foster, Designing and Building Parallel Programs, Addison-Wesley, 1995 (dosegljiva tudi na spletnem naslovu http://www.mcs.anl.gov/dbpp).

2. Parallel Processing Courseware, Department of Computer Science, Cardiff University, http://www.cm.cf.ac.uk/Tltp (1999).

3. I. Pitas (Ed.), Parallel Algorithms for Digital Image Processing, Computer Vision and Neural Networks, Wiley 1993.

5.3 Glavna literatura

1. I. Foster, Designing and Building Parallel Programs, Addison-Wesley, 1995 (dosegljiva tudi na spletnem naslovu http://www.mcs.anl.gov/dbpp).

2. A.H. Roosta, Parallel Processing and Parallel Algorithms, Theory and Computation, Springer-Verlag New York, 2000.

3. Parallel Processing Courseware, Department of Computer Science, Cardiff University, http://www.cm.cf.ac.uk/Tltp (1999).

4. I. Pitas (Ed.), Parallel Algorithms for Digital Image Processing, Computer Vision and Neural Networks, Wiley 1993.

6 Uvod v Unix

V tem besedilu so na kratko opisani osnovni sistemski klici in funkcije sistema UNIX s primeri uporabe v programskem jeziku C. Deklaracije funkcij oziroma klicev in primeri uporabe so večinoma usklajeni z ANSI C in POSIX standardi.

Priročnik je namenjen študentom devetega semestra smeri Avtomatika (PA in IS) in sicer kot doplnilo k predavanjem Vporedni sistemi in pripomoček za opravljanje laboratorijskih vaj.

6.1 Uporabljeni viri

[1] M. J. Rochkind, *Advaned UNIX Programming*, Prentice-Hall, 1985.
[2] W.R. Stevens, *Advaned Programming in the UNIX Environment*, Addison-Wesley, 1992.
[3] —, *Real-Time Programming Manual, Series 800 HP-UX*, Hewlett Packard, 1986.
[4] S.A. Rao, *UNIX System V Network Programming*, Addison-Wesley, 1993.
[5] 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.

[6] M.J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, 1986.
[7] J.L. Petersson, A. Silberschatz, *Operating System Concepts*, 2-nd Ed., Addison Wesley, 1985.

6.2 Razvrstitev sistemskih funkcij/klicev

Storitve operacijskega sistema (sistemskega jedra) so dejavnosti, ki jih sistem nudi uporabniškim programom oziroma procesom. Do njih dostopajo programi/procesi s sistemskimi klici in funkcijami. Množica sistemskih klicev/funkcij definira vmesnik programov do sistemskega jedra. Po namenu uporabe razdelimo sistemske funkcije/klice v grobem na štiri skupine:

- funkcije za vhodno/izhodni sistem oziroma datotečni sistem,
- funkcije za upravljanje procesov,
- funkcije za komunikacijo med procesi.
- funkcije za vpogled v trenutno stanje sistema in administriranje.

Pod izrazom *sistemski klic* načeloma razumemo prenos izvajanja procesa iz uporabniškega v sistemski način oziroma prehod izvajanja procesa v sistemsko jedro. Ta prehod je zaradi spremembe načina izvajanja realiziran s *programsko prekinitvijo – izjemo*.

Pod izrazom *sistemska funkcija* načeloma razumemo podprogram – funkcijo, ki se izvede izključno v uporabniškem načinu ali pa vključuje tudi sistemski klic. 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 in klicev manj pomembno. V tem priročniku oba izraza nista uporabljana popolnoma dosledno. Razlog za to izhaja tudi iz dejstva, da so nekateri sistemi klici v novejših sistemih "prešli" v funkcije. Razen tega so lahko nekatere sistemske storitve, ki so v enem sistemu realizirane kot sistemski klici v drugih sistemih realizirane kot funkcije.

7 Datotečni sistem

Osnovni objekt datotečnega sistema je datoteka. Sistemski klici/funkcije, ki se nanašajo na delo z datotekami so `open, creat, read, write, lseek` in `close`. S temi funkcijami direktno dosegamo storitve (usluge) sistemskega jedra ("kernel-a"), ki skrbi za prenos podatkov med uporabnikovim programom in zunanjo napravo. Morebitno medpomnenje (izravnava) podatkov se opravlja izključno na nivoju jedra. Večini uporabnikov so bolj znane funkcije `fopen, fread, fwrite, fseek, fclose`, ki realizirajo medpomnjenje na nivoju uporabniškega programa/procesa, do klica sistema pa pride samo v primeru, da je to zares potrebno. To se odraža običajno v manjšem številu prehodov med uporabniškim in sistemskim načinom izvajanja in "hitrejšem" napredovanju procesa. Zato se uporaba funkcij `fopen, fread, fwrite, fseek, fclose` priporoča.

Razen naštetih osnovnih funkcij se pogosteje uporabljajo še `dup, dup2, fcntl, ioctl, stat`, `chmod, chown, link, unlink, remove, mkdir, rmdir` in tako dalje.

Vsaka datoteka, ki jo nek proces "odpre", je enolično določena z datotečno številko oziroma datotečnim `deskriptorjem`. Deskriptor je celo pozitivno število ali nič. V starejših sistemih je bilo število deskriptorjev na posamezen proces in s tem število sočasno odprtih datotek omejeno na 20 (števila 0,1,...,19), kasneje razširjeno na 64, sicer pa je to parameter, ki se v splošnem od sistema do sistema razlikuje. Po dogovoru so deskriptorji 0, 1 in 2 rezervirani za standardno vhodno in standardno izhodno datoteko ter datoteko za sporočila napak. Namesto dejanskih vrednosti deskriptorjev teh datotek se priporoča uporaba simboličnih konstant:
`STDIN_FILENO, STDOUT_FILENO in STDERR_FILENO`,
ki so definirane v *unistd.h*, imajo pa vrednost $0, 1$ in 2.

7.1 Funkcija open

```
#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. `Open` vrne prvi "prosti" deskriptor (najmanjše še prosto – nedodeljeno – število), a se na to nikoli ne zanašamo. Argument `flags` predpiše eno od treh osnovnih operacij z datoteko (*beri*, *piši*, *beri in piši*) in poleg ustrezne osnovne operacije še po potrebi neobvezna dodatna določila (*ustvari, podaljšaj*, i.t.d.). Simbolična imena in dejanske vrednosti konstant osnovnih operacij so:

```
O_RDONLY       (dejanska vrednost = 0)   odpri samo za branje
O_WRONLY       (dejanska vrednost = 1)   odpri samo za pisanje
O_RDWR         (dejanska vrednost = 2)   odpri za branje in pisanje
```

S funkcijo `open` je možno tudi ustvariti novo datoteko (konstanta `O_CREAT`). Tedaj je obvezen tretji argument `mode`, ki definira določila ("bite") dovoljenj novo nastale datoteke (kdo sme datoteko brati, spremeniti, in podobno), glej opis za `creat`. Za obširnejšo razlago glej priročnik **man read** (sistemski ukaz UNIX-a).

7.2 Funkcija creat

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

int creat( const char *path, mode_t mode );
```

Vrne deskriptor na novo ustvarjene datoteke ali -1 v primeru napake.

`Creat` ustvari novo ("prazno") datoteko z imenom `path` in vrne njen deskriptor. 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
```

Tipična vrednost argumenta je 0644 (pozor: osmiška konstanta), ki dovoli lastniku datoteke branje in pisanje (cifra 6), skupini iz katere je uporabnik branje (cifra 4), in vsem ostalim tudi branje (cifra 4), glej primer 7.9. Klic

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

je ekvivalenten klicu

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

7.3 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 postane deskriptor ponovno prost. Ker proces oziroma jedro z `exit` zapre vse odprte datoteke, se `close` često opušča. Na avtomatično "zapiranje" datotek se (vedno) ne gre zanašati.

7.4 Funkcija read

```
#include <unistd.h>

ssize_t read( int fd, void *buff, size_t nbytes );
```

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

Opomba: klasična definicija (oz. deklaracija) je:

```
int read( int fd, char *buff, unsigned nbytes)
```

Funkcija `read` prebere iz datoteke `fd` v pomnilnik kamor kaže kazalec `buff` največ `nbytes` bajtov ali manj, če jih toliko ni na razpolago. Vrne dejansko število prebranih bajtov, 0 v primeru branja s konca datoteke (prebranih "nič" podatkov je znak za konec datoteke), ali -1 v primeru napake. Pozor: argument `nbytes` vedno pomeni število bajtov, neodvisno od resničnega tipa branih podatkov.

7.4.1 Primer

Naslednji kos programa odpre za branje (`flags = O_RDONLY`) datoteko z imenom "`Podatki`" in iz nje prebere 16 podatkov tipa `float` v polje `mp`.

```
...
int   fi;
float mp[16];
if ((fi = open("Podatki", O_RDONLY)) == -1){
  printf("Napaka pri odpiranju datoteke Podatki\n");
  exit (1);
}
if (read(fi, mp, 16 * sizeof(float) ) != 16 * sizeof(float)){
  printf("Prebranih ni 16 podatkov\n");
  exit(1);
}
close (fi);
...
```

7.5 Funkcija write

```
#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` osembitnih podatkov iz pomnilnika kamor kaže `buff` na datoteko
small tt 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 medija – diska).

7.6 Funkcija lseek

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

off_t lseek( int fd, off_t offset, int whence );
```

Vrne nov odmik, -1 v primeru napake.

Vsaki odprti datoteki je pridružen relativen *odmik* od začetka datoteke do trenutnega mesta pisanja ali branja. Odmik se po vsaki operaciji branja ali pisanja poveča za prebrano število bajtov, lahko pa ga spremenimo z ukazom `lseek`. To omogoča direkten dostop do podatkov v datoteki, ki je sicer (po definiciji) sekvenčnega značaja. 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      (dejanska vrednost = 0)   glede na zacetek datoteke
SEEK_CUR      (dejanska vrednost = 1)   od trenutnega polozaja
SEEK_END      (dejanska vrednost = 2)   glede na konec datoteke
```

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

7.6.1 Primeri

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

7.7 Funkciji dup, dup2

```
#include <unistd.h>

int dup( int fd );

int dup2( int fd, int fdupd );
```

Vrneta nov deskriptor ali -1 v primeru napake.

Funkciji `dup` in `dup2` podvojita deskriptor `fd` že odprte datoteke. 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 uspešnem klicu (povratku) je ista datoteka dosegljiva z dvema različnima deskriptorjema. Tipični primeri uporabe funkcij `dup` so preusmeritev branja ali pisanja in cevi ("pipes").

7.7.1 Primer

Naslednji kos programa "preusmeri" izpis s standardnega izhoda v datoteko z imenom `tempfile`.

```
...
fi = creat( ''tempfile'', 0644 );
close( 1 ); dup( fi ); close( fi );
...
```

7.8 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, glej
man fcntl in **man ioctl**.

7.9 Primer

Naslednji program prepiše datoteko z imenom argv[1] na datoteko z imenom argv[2]. Glej pomen bitov dovoljenj.

```
/* ---------------------- POPR ------------------------------
   Program: prepis
        program za prepis datoteke.

   Uporaba: prepis stara nova
            stara: ime obstojece datoteke
            nova : ime nove datoteke
*/

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

```
#include <malloc.h>

int
main( int argc, char *argv[] )
{
  char *Pomnilnik;
  int  fi, fo;
  int  Npodatkov, Ppodatkov;

  if (argc != 3){
    printf("\nUPORABA: %s vhod izhod\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 pri odpiranju datoteke %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 pri ustvarjanju datoteke %s\n", argv[0], argv[2]);
    exit(3);
  }
/* Zagotovimo si pomnilnik za podatke, najvec Npodatkov    */
  Npodatkov = 512;

  if ((Pomnilnik = (char *) malloc( sizeof(char) * Npodatkov)) == NULL){
    printf("%s: napaka pri dodeljevanju pomnilnika\n", argv[0]);
    exit(4);
  }

/* prepisujemo podatke do konca datoteke                    */

  while ((Ppodatkov=read(fi,Pomnilnik,sizeof(char)*Npodatkov)) != 0){
    if (Ppodatkov == -1){
      printf("%s: napaka pri branju datoteke %s\n", argv[0], argv[1]);
      exit(5);
    }
    if( write(fo, Pomnilnik, Ppodatkov) != Ppodatkov){
      printf("%s: napaka pri zapisu datoteke %s\n", argv[0], argv[2]);
      exit(6);
    }
  }
  if ((close(fi) == -1) || (close(fo) == -1)){
    printf("%s: napaka pri zapiranju datotek\n",argv[0]);
    exit(7);
  }
  exit(0);

} /* Konec main() */
```

7.10 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) na nivoju uporabnikovega programa, sicer pa imajo funkcije enak/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" (ang. *stream*) z imenom `pathname` za operacijo določeno z argumentom (`type = ''r''` za read, `''w''` za write, i.t.d) in vrne datotečni kazalec na strukturo tipa `FILE`. Ta kazalec je nato referenca na podatkovni tok za vse kasnejše operacije (`fread, fwrite, ...`).
Funkcija `fread, fwrite` prebere/zapiše ustrezno število `nitems` podatkovnih enot velikosti `size` bajtov. Funkciji vrneta dejansko število prenešenih podatkov ali kodo napake (-1).

7.10.1 Primer

Naslednji program prepiše datoteko z imenom argv[1] na datoteko z imenom argv[2]. Preverjanje napak branja in pisanja je opuščeno.

```
#include <stdio.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 );
}
```

7.11 Druge vhodno izhodne funkcije

Funkciji `fread, fwrite` sta za branje in pisanje podatkov v "binarni" obliki – podatke prenašamo take kot so, brez pomenske interpretacije. Formatiran vhod/izhod opravljata funkciji `fscanf` in `fprintf` oziroma v primeru standardne vhodne ter izhodne datoteke funkciji `scanf` in `printf`,
glej **man fprintf** in **man fscanf**.

Za delo z znakovnimi nizi (branje, pisanje, prepisovanje, združevanje, analiziranje, i.t.d.) je na razpolago knjižnjica funkcij: `fgets, gets, fputs, puts` in funkcije, ki začenjajo s `str`, na primer: `strtok, strcat, strcmp, strcpy, strlen`,
glej **man gets**, **man puts**, **man strtok**, i.t.d..

7.11.1 Primer

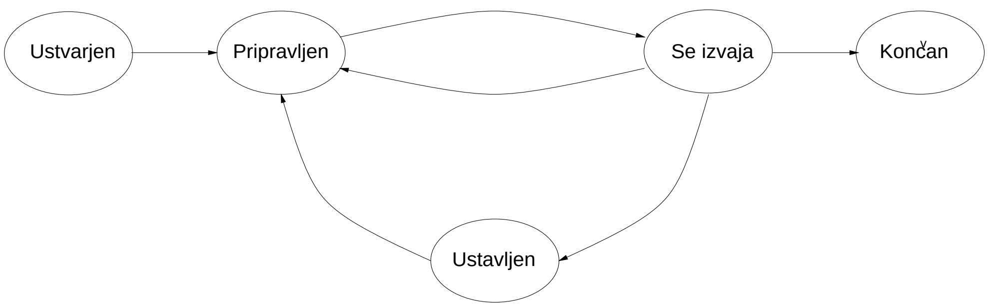
Naslednje programsko besedilo daje primer podprograma – funkcije getargs.c – za analizo "ukazne" vrstice, ki jo odtipkamo na terminalu. Vrstica – znakovni niz – se prebere s funkcijo `gets` in analizira (razčleni na besede) s funkcijo `strtok`. Funkcija `getargs` vrne polje kazalcev na podnize (besede) v odtipkani vrstici.

```
#include <stdio.h>
#include <string.h>
/* ----------------------------------------------------------------

   Funkcija za analizo niza - ukazne vrstice

   int getarg( argv, 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:

   main()
   {
     char *Argumenti[12];
     int  Status;

     while( (Status = getargs( Argumenti, 12 )) > 0){
      printf("OK\n");
    }
    if(Status == -2){
      exit(0);
    }
}
*/

int getargs( argv, maxarg )
char **argv;
int  maxarg;
{
  static char cmd[100];
  int    i;

  if( gets( cmd ) == NULL)
    return -2;
  cmdp = cmd;
  for( i = 0; i <= maxarg; i++){
    if( (argv[i] = strtok(cmdp," ")) == NULL)  /* glej man strtok */
      break;
    cmdp =  NULL;
  }
  if( i > maxarg ){
    printf("Prevec argumentov v ukazni vrstici\n");
    return - 1;
  }
  return i; /* stevilo argumentov */
```

```
}
```

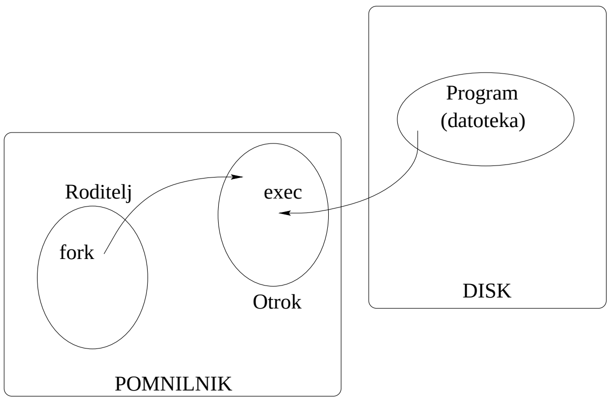
8 Upravljanje procesov

Proces je program v izvrševanju. Med izvrševanjem proces spreminja stanje. Ko nastane, gre najprej v stanje *pripravljen*. V stanju *pripravljen* ima proces zagotovljene vse pogoje, da bi se izvajal, če bi imel procesno enoto, vendar je procesna enota dodeljena drugemu procesu. Proces gre v stanje *se izvaja*, kadar se mu dodeli procesna enota. Menjavo procesov (pomenski preklop) opravi del operacijskega sistema, ki mu pravimo dispečer. Iz stanja izvajanja gre proces nazaj v stanje *pripravljen*, ko mu poteče dodeljeni čas, če prej ne konča. V stanje *ustavljen* gre proces tedaj, kadar nima več pogojev da bi se izvajal (na primer čaka da bo vhodno/izhodni prenos končan).



8.1 Sistemski klici/funkcije za upravljanje procesov

Osnovni sistemski klici za upravljanje procesov v sistemu UNIX so `fork`, `exec`, `wait`, `exit`. S `fork` nastane nov proces, z `exec` se proces inicializira iz programa in z `exit` se proces konča.



8.2 Funkcija fork

Funkcija `fork` daje edino možnost za nastanek novega procesa. Druge možnosti razen klica `fork` za nastanek procesa ni.

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

pid_t  fork( void );
```

Vrne 0 otroku, Pid otroka roditelju, -1 v primeru napake.

S `fork` proces-roditelj ustvari nov proces-otroka, ki je skoraj identičen njemu samemu-roditelju. `Fork` "se vrne dvema", otroku in roditelju. Oba procesa se izvajata naprej asinhrono in sicer z ukazom, ki sledi klicu fork.

8.2.1 Primer

```
....
if( (pid = fork( )) == 0){
  printf("Otrok\n");  /* Fork vrne otroku vrednost 0        */
  /* tu sledi poljubno programsko zaporedje in obicajno exec */
}
else if( pid == -1){
  printf("Napaka, proces ni ustvarjen\n");
}
else{
  printf("Roditelj, ustvaril sem proces s pid = %d\n", pid);
}
...
```

8.3 Funkciji exit, _exit

Proces lahko končna na tri regularne in na dva neregularna načina:

- *Regularno*: z `return` iz main, ali od kjerkoli z `exit` ali z `_exit`.
- *Neregularno*: na lastno pobudo s klicem `abort` ali na zahtevo od zunaj (s sprejemom *signala*).

Klic `_exit` povzroči takojšni konec procesa in povratek k jedru. `Return` in `exit` sta si ekvivalentna in končata s klicem `_exit`. `Exit` ali `return` opravita pred povratkom z `_exit` še določene zaključne operacije, ki so včasih potrebne (npr. zapiranje datotek, čiščenje v/i medpomnilnikov). V primeru, da se eksplicitni klic ustrezne funkcije v main opusti, se privzame povratek `return`.

```
#include <stdio.h>
void exit( int exitstatus ); /* ekvivalentno return( exitstatus ) */

ali

#include <unistd.h>
void _exit( int exitstatus );
```

Na ta način lahko proces, ki konča z `exit`, javi procesu ki ga čaka, z argumentom `exitstatus` na kakšen način je končal (glej `wait`).

8.3.1 Primer

```
int main( )
{
  ...
  exit(0); /* ali return(0) --- povratek z 0, roditelj dobi 0 */
}
```

Vrednost nič je dogovorjena za regularen konec procesa (na nivoju aplikacije), pozitivne vrednosti $(1, 2, 3, \ldots)$ so znak napake.

8.4 Funkcija wait

Funkcija `wait` omogoča procesu-roditelju, ki ustvari nov proces-otrok, da čaka (se sinhronizira) na njegov konec.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait( int *statloc );
```

Vrne -1 in ni otroka za čakanje, ali PID dočakanega procesa.

Proces-roditelj z `wait` čaka na *prvi* proces-otrok, ki bo končal. `statloc` vsebuje 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.

8.5 Funkcije exec

`Exec` je funkcija, ki omogoča inicializacijo procesa iz programa in njegovo izvršitev. Pozor: z `exec` ne nastane nov proces. Na voljo je šest različic klica `exec`: `execl, execv, execle, execve, execlp, execvp`.

```
#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[]);
```

Vrne -1 v primeru napake in inicializacija ni uspela. V primeru da `exec` uspe do povratka ne pride.

Klici (in torej imena klicev) se razlikujejo glede na način podajanja argumentov, na način podajanja imena programa in glede na prenos (upoštevanje) procesovega "okolja".

- 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`.

8.5.1 Primer

```
...
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{
    while( wait( &s ) != p );      /* cakam, da otrok konca          */
    printf("Proces-otrok PID = %d koncan, exit status %d\n", p, s >> 8);
    exit( 0 );
  }
}
```

8.6 Funkcija sleep

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

Funkcija `sleep` ustavi izvajanje procesa za časovni interval `seconds` sekund. Proces gre v stanje ustavljen in v tem intervalu ne obremenjuje procesne enote.

8.7 Funkcija system

Funkcija `system` daje večini uporabnikov najprikladnejši način za izvršitev ukazne vrstice s sintakso školjke (lupine) znotraj programa. Realizira klic `fork, exec` in `wait`.

```
#include <stdio.h>

int system( const char *String );
```

8.7.1 Primer

```
...
system( "ls -l" );    /* izpis tekocega direktorija  */
...
system( "cp sss xxx"); /* prepis datoteke sss na xxx */
...
```

Razmislite o možni realizaciji funkcije `system`.

9 Komunikacija med procesi

Za komunikacijo med procesi obstaja v sistemu UNIX precej možnosti. Osnovne oblike medprocesnega komuniciranja (ang. Interprocess Communication – ali s kratico IPC) dajejo:

- cevi (ang. pipes): klic `pipe`, in poimenovane cevi (ang. named pipes): klic `mkfifo`,
- signali: klici `signal, kill, raise, alarm, pause`,
- sporočila (ang. messages): klici `msgget, msgctl, msgsnd, msgrcv`,
- semafori: klici `semget, semctl, semop`,
- skupen (deljen) pomnilnik (ang. shared memory): klici `shmget, shmctl, shmat, msgdt`,
- komunikacijske vtičnice (ang. sockets): klici `socket, bind, listen, accept, connect`.

Cevi in poimenovane cevi ter signali so v sistemu UNIX "standardna" oblika komuniciranja, pri čemer se signali praviloma uporabljajo za javljanje "kritičnih" oziroma izjemnih dogodkov.

Sistemi semaforjev, sporočil ter skupen pomnilnik predstavljajo razširitev. Ko govorimo o sistemu medprocesnega komuniciranja včasih mislimo prav na eno izmed teh treh oblik, pri čemer obravnavamo cevi kot poseben primer sporočil. Semaforji so predvideni za sinhronizacijo procesov; sinhronizacija procesov je pač ena od oblik medprocesne komunikacije.

Komunikacijske vtičnice niso omejene na medprocesno komunikacijo na enem računalniku (na eni postaji). Nastale so iz potrebe po komunikaciji med krajevno porazdeljenimi procesi v omrežju – procesi na različnih postajah, neodvisno od krajevne namestitve, materialne opreme in operacijskega sistema. Razumljivo, da dopuščajo komuniciranje med procesi na istem računalniku.

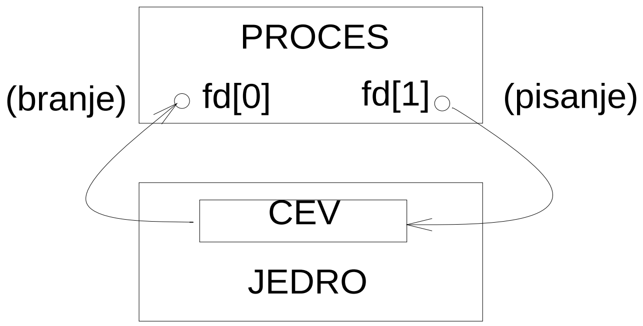
9.1 Cevi in sistemski klic pipe

Cev ("pipe") je *enosmerni* komunikacijski kanal, ki lahko obstaja med dvema procesoma v sorodstvu (otrok-otrok, roditelj-otrok). Za obojesmerno komunikacijo sta potrebni dve cevi.

```
#include <unistd.h>

int pipe( int fd[2] );
```

Vrne 0 in komunikacijski kanal je ustvarjen ali -1 v primeru napake.

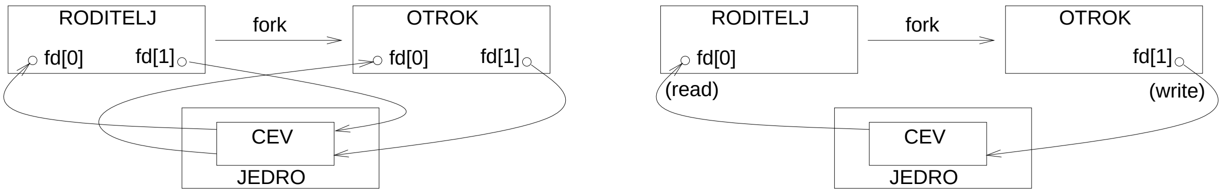
V argumentu `fd` vrne dva deskriptorja, ki dajeta dostop do obeh "koncev" cevi: do konca za branje iz cevi in do konca za pisanje v cev.



Cev z obema koncema dostopnima istemu procesu nima prave vrednosti. Cev "napeljemo" med dva procesa na primer takole:

- odpremo cev (klic pipe),
- ustvarimo nov proces (klic fork); nov proces podeduje deskriptorja odprte cevi,
- en konec cevi (na primer za pisanje) pustimo odprt v enem procesu, drug konec cevi (za branje) pustimo drugemu procesu.

V cev pišemo s funkcijo `write( fd[1], ...`, iz cevi beremo s funkcijo `read( fd[0], ...`, dokler je ne zapremo `close` (torej enako kot bi bila datoteka).



9.1.1 Primer

Naslednji program ustvari cev in dva procesa (otroka), ju inicializira iz programov `odd` in `spr` in med njima napelje cev. Za proces `odd` preusmeri standardni izhod (deskriptor 1) v oddajni konec cevi, za proces `spr` preusmeri standardni vhod (deskriptor 0) na sprejemni konec cevi. Rezultat preusmeritve je, da proces `odd` namesto na standardni izhod piše v cev. Podobno, proces `spr` namesto s standardnega vhoda bere iz cevi.

```
/* Enosmerna komunikacija dveh procesov s cevjo       */
/* Opusceno je preverjanje napak                      */
...
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("odd", "odd", (char *)0);
    exit( 2 );                /* 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("spr", "spr", (char *)0);
    exit( 3 );                /* za vsak slucaj, ce exec ne uspe   */
  }
  exit( 0 ); /* koncaj brez cakanja */
} /* end main */
```

9.2 Poimenovane cevi in FIFO

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* (ang. "named pipe"). Poimenovane cevi omogočajo komunikacijo med poljubnimi procesi, ne le procesi v sorodstvu.

```
#include <sys/stat.h>

int mkfifo(char *path, mode_t mode);
```

Vrne 0 in FIFO obstaja ali -1 v primeru napake.

`Mkfifo` je POSIX nadomestek za tvorjenje FIFO z `mknod` ("make node"). Argumenti k `mkfifo` imajo podoben pomen kot argumenti za `creat`. Klic `mkfifo` ustvari FIFO, ki ga zatem odpremo z `open`.

Tipičen primer uporabe FIFO so komunikacije po načelu odjemalec–strežnik (ang. client-server), pri čemer strežnik sprejema zahteve od večjega števila odjemalcev na vsem dobro znanem FIFO imenu (znano ime–pot).

9.3 Sistem semaforjev

9.3.1 Uvod - splošno o semaforjih

Semafor je celoštevilčna spremenljivka, ki si jo delita dva ali več procesov. Načeloma (po definiciji) sta nad semaforjem poleg inicializacije možni le dve operaciji: *čakaj* – oznaka P in *javi* – oznaka V. Druga imena za *čakaj* so še: zahtevaj, zakleni, postavi, ustavi, i.t.d. Druga imena za *javi* so: sprosti, odkleni, briši, obudi, i.t.d. *Binarni semafor* lahko zavzame eno od dveh možnih vrednosti. *Števni semafor* lahko zavzame poljubno pozitivno vrednost ali nič. Operacijski sistemi, ki realizirajo sistem semaforov, včasih dopuščajo "razširjen" nabor operacij (npr. preverjanje stanja semaforja).

Semafor se uporablja za sinhronizacijo procesov, tipično pri dostopanju dveh ali več procesov do skupnega sredstva, ki ne dopušča sočasnega dostopa. 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 nekdo drug, 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.

Sistem UNIX nudi procesom na nivoju sistemskega jedra množice binarnih in števnih semaforjev (ang. Semaphore Set). Sistemski klici za delo s semaforji so: `semget, semctl in semop`.

9.3.2 Funkcija semget

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

Vrne številko množice semaforjev ali -1 v primeru napake.

Funkcija `semget` ustvari množico semaforjev (v sistemskem jedru) in vrne njeno identifikacijsko številko, preko katere 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, kot pri datotekah. Argument `key` je:

```
IPC_PRIVATE za komunikacijo med procesi v sorodstvu,
ftok()      za komunikacijo med poljubnimi procesi, glej man ftok.
```

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 (klic `semget` z `IPC_PRIVATE`). Za procese izven sorodstva se ta problem najsplošneje reši s pomočjo funkcije `ftok` (ang. File To Key).

9.3.3 Funkcija semctl

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ... /* arg */ );
```

Funkcija `semctl` realizira zahtevano operacijo nad semaforom s številko `semnum` v množici `semid`; operacijo določa argument `cmd` kot sledi:

```
GETVAL       vrne vrednost semafora
SETVAL       postavi vrednost semafora na arg
GETALL       vrne vrednost vseh semaforov v mnozici (atomicno)
SETALL       postavi vrednost vseh semaforov v mnozici (atomicno)
```

9.3.4 Primer

Naslednji primer programa ustvari množico dveh semaforov za komunikacijo med procesi v sorodstvu (klic `semget` z `IPC_PRIVATE`), postavi prvi semafor na nič in drugi semafor na 1 (klic `semctl`). Nato sočasno–atomično preveri vrednost obeh semaforov (ukaz `GETALL`), vrednosti vrne v polju `SemArray`. Na koncu odstrani množico semaforov s klicem `semctl` in operacijo `IPC_RMID`.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEM_1    0    /* definiramo simbole - zgolj zaradi preglednosti */
#define SEM_2    1

int main( )
{
  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);
}
```

9.3.5 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 zahtevano operacijo (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 njegova vrednost pred tem večja od 1, se mu -1 enostavno prišteje oziroma 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 ("čakal bo na semaforju"), dokler:

- nek drug proces ne dvigne vrednosti semafora (to je regularen način, ki pomeni operacijo *javi*),
- nek drug proces odstrani semafor z `SEM_RMID` (v nasprotnem primeru preti zastoj),
- sprejme signal.

Razumljivo: tisti proces, ki prvi zmanjša vrednost semafora na nič, zahteva od drugih procesov, da na semaforu čakajo.

Operacijo *javi* realizira nasledji 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 semafora za ena in nadaljuje z izvajanjem. Vsi procesi, ki so morda čakali na tem semaforu (pomeni, da je bila prejšnja vrednost nič), dobijo možnost za napredovanje (jedro jih postavi v stanje *pripravljen*.

S komponento `sem_flg` strukture tipa `sembuf` je možno spremeniti osnovni pomen operacij na semaforu (glej **man semop**).

9.4 Deljen (skupen) pomnilnik

Deljen ali skupen pomnilnik (ang. Shared Memory) je ime za del fizičnega pomnilnika, ki je hkrati dostopen več kot enemu procesu. To pomeni, da je hkrati preslikan v logično naslovno področje večjega števila (tipično dveh) sočasnih procesov. Deljen pomnilnik ustvarja podlago za najhitrejši način medprocesne komunikacije. Do pretoka podatkov v klasičnem smislu sploh ne pride: en proces podatke vpiše ("odda") in drug proces jih bere ("sprejme"). Za komunikacijo procesov preko skupnega pomnilnika daje operacijski sistem "golo" podlago, vse ostalo – pravila komunikacije oziroma protokol, pa so prepuščena procesom samim (oziroma programerju).

Osnovni sistemski klici, ki se nanašajo na deljen pomnilnik, so: `shmget, shmctl, shmat` in `shmdt`.

9.4.1 Funkcija shmget

```
#include <sys/shm.h>

int shmget( key_t key, size_t size, int shmflg );
```

Vrne identifikacijsko številko 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. 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 vrne identifikacijsko številko `ShmId` preko katere je pomnilnik potem "znan" in ga je možno pripeti procesu s funkcijo `shmat`.

9.4.2 Funkcija shmctl

```
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Funkcija `shmctl` realizira nadzorno operacijo `cmd` nad pomnilnikom z identifikacijo `shmid`. Dovoljene so naslednje operacije:

- IPC_STAT v strukturi na katero kaže `buf` vrne vrednosti sistemske strukture pridružene 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.

9.4.3 Funkciji shmop: 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` k svojemu naslovnemu področju in vrne 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 z `shmdt` (ang. Detach), segment pa še obstaja, dokler se ga ne uniči s klicem `shmctl` z operacijo `IPC_RMID`.

9.4.4 Primer

Naslednje programsko besedilo realizira enostavno komunikacijo – brez sinhronizacije – preko deljenega pomnilnika med roditeljem in otrokom.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define  VELIKOST  0x10   /* 16 bajtov */

main( )
{
  int  i, Pid, ShmId;
  char *ShmAddr;

  if( (ShmId = shmget( IPC_PRIVATE, VELIKOST, 0600 )) == -1 ){
    printf("Napaka, shmget ni uspel\n");
  }
  if( (Pid = fork( )) == 0 ){   /* --- Otrok --- */
    if( (ShmAddr = shmat( ShmId, 0, 0 )) == (void *) -1 ){
      printf("Napaka, shmat za otroka ni uspel\n");
    }
    printf("Naslovi za ShmAddr od %x do %x (za otroka)\n",&ShmAddr[0],&ShmAddr[VELIKOST]);
    for( i = 0; i < VELIKOST; i++){ ShmAddr[ i ] = i; sleep( 1 ); }
```

```
    if( shmdt( ShmAddr ) == -1) printf("Napaka, shmdt ni uspel\n");
    exit( 0 );
  }
  else if( Pid > 0 ){  /* --- Roditelj --- */
    if( (ShmAddr = shmat( ShmId, 0, 0 )) == (void *) -1 ){
      printf("Napaka, shmat za roditelja ni uspel\n");
    }
    printf("Naslovi za ShmAddr od %x do %x (za roditelja)\n",&ShmAddr[0],&ShmAddr[VELIKOST]);
    for( i = 0; i < VELIKOST; i++){ sleep(1); printf("Shm[%3d] = %d\n",i,ShmAddr[i]); }
    if( shmctl( ShmId, IPC_RMID, 0 ) == -1)printf("Napaka, shmctl z SHM_RMID ni uspel\n");
    exit( 0 );
  }
  else{
    printf("Napaka fork()\n"); exit( 1 );
  }
} /* Konec main */
```

9.5 Signali

Signal je programska prekinitev, ki jo proces dobi od sistemskega jedra, od drugega procesa ali od samega sebe. Signal je možno poslati, presteči, prezreti ali obravnavati na "priporočen" oziroma "standarden" način. Signal se pojavi *asinhrono* z izvajanjem procesa, nanj pa je možno tudi čakati. Signali nosijo malo informacije in se uporaljajo predvsem za javljanje nenavadnih okoliščin ali nepravilnosti. Signali so bili v preteklosti na sistemih UNIX znani kot "nezanesljivi", ker se v nekaterih, sicer izjemnih okoliščinah, obnašajo drugače kot pričakujemo (smo predvideli v programu). Ta problem je v novejših sistemih v glavnem rešen.

Ostaja več vrst signalov. Število signalov je v SVR4 (System V Release 4) naraslo na 31. Vsak signal ima svoje ime oziroma makro definicijo konstante, ki ji je prirejeno pozitivno celo število (definicije so v datoteki *signal.h*). Imena signalov začenjajo s `SIG`, kot na primer `SIGKILL`. Signal `SIGKILL` ima številko 9. Nekatere izmed signalov pošlje sistemsko jedro "avtomatično", na primer deljenje z nič, neveljaven pomnilniški naslov, vendar lahko te signale pošilja tudi uporabniški proces. Nekateri signali so popolnoma v domeni uporabnika. Važnejši signali so:

```
SIGHUP   1   /* Hangup - prekinitev zveze s terminalom                     */
SIGINT   2   /* Prekinitev - pritisk prekinitvene tipkovne kombinacije (^C) */
SIGQUIT  3   /* Podobno kot SIGINT, a z drugo tipkovno kombinacija     (^\) */
SIGILL   4   /* Neveljaven ukaz                                            */
SIGABRT  6   /* Abort - ``abort'' signal poslan s sklicem abort            */
SIGFPE   8   /* Floating point exception - matematicna napaka              */
SIGKILL  9   /* kill - takoj koncal; se ne da prezreti ali presteci        */
SIGBUS  10   /* Bus error - napaka na vodilu                               */
SIGSEGV 11   /* Segmentation Violation - Pobeg iz naslovnega podrocja      */
SIGPIPE 13   /* vpis v cev brez odprtega konca za branje                   */
SIGALRM 14   /* Alarm - poslan, kadar se iztece casovnik nastavljen z alarm */
SIGTERM 15   /* Terminate - proces, ki dobi signal naj cim hitreje konca   */
SIGUSR1 16   /* Kot definira uporabnik                                     */
SIGUSR2 17   /* Kot definira uporabnik                                     */
SIGCHLD 18   /* Konec otroka - signal dobi roditelj, ko otrok konca        */
SIGPWR  19   /* Power Fail - v primeru izpada napajanja                    */
SIGVTALRM 20 /* Virtual timer alarm - signal poslan s settimer             */
SIGSTOP 24   /* Stop - ustavi proces; se ga ne da prezreti ali prestreci   */
SIGCONT 26   /* Continue - za nadaljevanje izvajanja ustavljenega procesa  */
```

Zaradi prenosljivosti programov se priporoča uporaba simboličnih namesto dejanskih vrednosti konstant.

9.5.1 Funkcija signal

```
#include <signal.h>

void (*signal(int sig, void (*action)(int)))(int);
```

Vrne prejšnjo vrednost naslova prestrezne funkcije ali `SIG_ERR` v primeru napake.

S funkcijo `signal` je možno procesu izbirati enega izmed treh načinov obravnavanja signala. Z argumentom `sig` se predpiše število (ime) signala in `action` določa način njegovega obravnavanja. Možen je eden od treh načinov obravnavanja signala:

- `SIG_DFL` - privzet (ang. default) način, ki je za večino signalov *konec* procesa.
- `SIG_IGN` - prezri signal. Proces signala v resnici niti ne dobi. Signalov `SIGKILL` in `SIGSTOP` se ne da prezreti.
- `address` - prestrezi signal. Naslov prestrezne funkcije je dan z `address`. Funkcijo definira programer sam. Po sprejemu (in streženju) signala je način obravnavanja signal (večioma) ponovno `SIG_DFL`. Če hočemo signal ponovno prestreči, je potreben ponoven klic `signal` znotraj prestrezne funkcije. Po povratku iz prestrezne funkcije (strežnika) se nadaljuje izvajanje procesa na mestu prekinitve (v trenutku sprejema signala). Prestrezna funkcija, ki jo definira programer, ima tri parametre:
  - sig - številka signala,
  - code - odvisno od strojne opreme,
  - scp - kazalec na strukturo - odvisno od strojne opreme.

9.5.2 Funkcija kill

```
#include <signal.h>

int kill(pid_t pid, int sig);
int raise(int sig);
```

Vrne 0 in signal je poslan ali -1 v primeru napake.

Funkcija `kill` pošlje *signal* procesu ali skupini procesov. Številko procesa podamo z argumentom `pid`:

- 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 (enakim "group ID"),

- če je `pid` < 0 se signal pošlje vsem procesom v skupini s številko pid (absolutno).

Argument `sig` je številka signala ali nič.

S funkcijo `raise` pošlje proces signal samemu sebi.

9.5.3 Funkciji alarm in pause

```
#include <signal.h>

unsigned int alarm(  unsigned int seconds );
```

Vrne 0 ali čas (v sekundah) do alarma.

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 samo eden) alarmni časovnik.

```
#include <signal.h>

int pause( void );
```

Ustavi proces, dokler ne dobi signala.

9.5.4 Primer

Naslednji program predstvalja možnost izvedbe sistemske funkcije `sleep`. Program si sam pošlje signal `SIGALRM` in ga prestreže.

```
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>

int main( )
{
  int spi( );
  static void PrestreziAlarm( );

  spi( 10 ); /* preiskusni klic */
  exit( 0 );
}
static void PrestreziAlarm( sig )
int sig;
{
  /* zgolj prestezemo signal */
  return;
}
int spi( sekund )
int sekund;
{
  signal( SIGALRM, PrestreziAlarm );
  alarm( sekund );      /* poslji signal   */
  pause( );
  return( alarm( 0 ) ); /* ugasni casovnik */
}
```

9.5.5 Primer

Kaj se zgodi če naslednjemu programu/procesu pošljemo signal `SIGINT`, bodisi z UNIX ukazom

kill -INT pid

ali s prekinitveno tipkovno kombinacijo (običajno ¡ctrl¿/c).

```
#include <stdio.h>
#include <signal.h>

void MojIntStreznik( );

main( )
{
  signal(SIGINT, MojIntStreznik );
  for( ; ; );
} /* End main */

void MojIntStreznik( )
{
  printf("To je moj SIGINT streznik\n");
  signal(SIGINT, MojIntStreznik );
}
```

9.6 Sistem sporočil

Sistem sporočil omogoča komunikacijo med procesi preko "sporočilnih vrst" (ang. 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 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.

9.6.1 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`.

9.6.2 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.

9.6.3 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.

9.6.4 Primer

Naslednja dva programa komunicirata preko sporočilne vrste; prvi odda sporočilo in drugi sporočila sprejema – do prekinitve. Procesa nista v sorodstvu.

Oddajnik sporočila

```
/* Msg Oddajnik */
#include <stdio.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( argc, argv )
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 ) == -1 ){ /* Oddaja      */
    printf("%s: Napaka, msgsnd ni uspel\n", argv[0]);
    exit( 2 );
  }
  exit( 0 );
}
```

Sprejemnik sporočila

Sprejmni proces v neskončni zanki čaka na sporočilo.

```
/* Msg Sprejemnik */
#include <stdio.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( argc, argv )
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 );
}
```

9.7 Sistem komunikacijskih vtičnic – socket

9.7.1 Uvod

Sistem komunikacijskih vtičnic (ang. socket) izvira iz BSD veje sistemov UNIX. Pojavil se je okrog leta 1982. Danes je v izpopolnjeni obliki prisoten tudi v veji sistemov UNIX System V.

Sistem vtičnic je in predstavlja razširitev zmožnosti medprocesnega komuniciranja. Omogoča enotno obravnavanje komunikacijske problemetike neodvisno od krajevne namestitve procesov. Vtičnice služijo za dvosmerno komunikacijo med procesi, ki so poljubno porazdeljeni po postajah v omrežju; seveda je na enak način možna komunikacija med procesi na istem računalniku.

Komunikacijska vtičnica je osnovni gradnik medprocesnega komuniciranja. Vtičnica (ang. socket) je ime za priključno mesto (dostopno točko) do enega od obeh koncev dvosmernega komunikacijskega kanala. 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. Osnovni problem komuniciranja procesov v omrežju je povezan z vprašanjem, kako med seboj "povezati" par vtičnic na oddaljenih postajah in na vtičnice "pripeti" želene procese. Sistem vtičnic reši tudi ta problem.

Vtičnice obstajajo znotraj vnaprej predpisanih komunikacijskih domen. *Komunikacijsko domeno* definira naslovna družina (naslovni prostor s specifikacijo pomena in formata naslovov) ter protokolovna družina (množica protokolov, ki izvira iz iste mrežne arhitekture). Znane naslovne, in s tem povezane protokolovne oziroma kar komunikacijske domene, so:

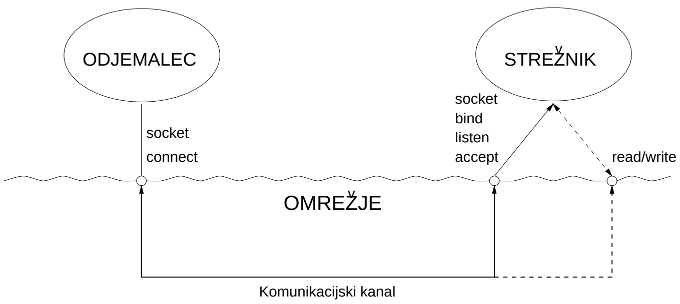
- `AF_UNIX` za naslovno domeno sistema UNIX.
- `AF_INET` za naslovno domeno Internet.
- `AF_CCITT` za naslovno domeno x.25.

Vtičnica ima svoj tip. *Tip* je povezan z lastnostmi komunikacijskega kanala/protokola oziroma storitvami, ki jih le-ta zagotavlja. Znani *tipi* vtičnic so:

- `SOCK_STREAM` za podatkovni tok. Ta tip zagotavlja povezan način komuniciranja: med obema koncema – vtičnicama – obstaja cel čas komuniciranja navidezna povezava; podatki gredo od enega do drugega konca zaporedno drug za drugim brez podvajanja ali izgubljanja.
- `SOCK_DGRAM` za *datagram*. Ta tip nudi nepovezan način komuniciranja: podatki gredo od konca do konca po različnih prenosnih poteh; obstaja možnost podvajanja in izgubljanja. Skratka, ni zagotovila, da bodo podatki pravilno dostavljeni.
- `SOCK_RAW` ali "goli" tip. Daje dostop do storitev nižjih slojev komunikacijske arhitekture; služi za razvojne namene.

9.7.2 Funkcije sistema vtičnic

S programskega stališča je *vtičnica* objekt, ki jo ustvari sistemski klic `socket`, vendar komunikacija lahko začne šele ko je vtičnica z `bind` "pripeta" na ime. Komunikacija med procesi je asimetrična po načelu "odjemalec-strežnik" (ang. client-server). Strežnik z `accept` na znanem naslovu vtičnice sprejema zahteve odjemalcev, ki s `connect` poskušajo priti do storitev strežnika – skušajo vzpostaviti zvezo z njim. Če zveza (accept-connect) uspe, strežnik "odpre" novo vtičnico, na kateri potem sam streže odjemalcu 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 z `send` in `recv`, dokler nekdo ne zapre vtičnice s `close`.



9.7.3 Funkcija socket

Funkcija `socket` ustvari en konec dvosmernega komunikacijskega kanala – vtičnico – in vrne njen deskriptor; na primer:

```
sd = socket( domena, tip, 0 );
```

usvari vtičnico v zahtevani naslovni *domeni* predvidenega *tipa* in s "standardnim" komunikacijskim protokolom.

```
#include <sys/socket.h>

int socket(int af, int type, int protocol);
```

Vrne deskriptor vtičnice, preko katerega je vtičnica dosegljiva, -1 v primeru napake.

Argument `af` določa naslovno "družino" (address family):

```
AF_INET        (DARPA Internet addresses)
AF_UNIX        (path names on a local node)
AF_CCITT       (CCITT X.25 addresses)
```

Argument `type` predpiše *tip* vtičnice: `SOCK_STREAM, SOCK_DGRAM, SOCK_RAW`. Argument `protocol` predpiše podatkovni protokol, ki je odvisen od komunikacijske domene. Če je vrednost argumenta nič(običajen način), sistem izbere protokol sam (npr. TCP ali UDP za domeno `AF_INET`.
Tipično:

```
...
sd = socket( AF_INET, SOCK_STREAM, 0 ); /* za Internet domeno */
... (ali)
sd = socket( AF_UNIX, SOCK_STREAM, 0 ); /* za UNIX domeno     */
...
```

9.7.4 Funkcija bind

S klicem `socket` nastane vtičnica brez imena. Vtičnica sicer obstaja, vendar ji je potrebno pripeti še ime. Za to poskrbi funkcija `bind`, ki vtičnici z znanim deskriptorjem priduži ime. Na primer:

```
bind( sd, ime, sizeof(ime));
```

vtičnici z deskriptorjem *sd* pridruži *ime*. Vprašanje pa seveda ostane, kakšno je *ime*. Oblika – format imena je vsekakor odvisen od komunikacijske domene.

```
#include <sys/socket.h>
#include <netinet/in.h>     /* za AF_INET */
#include <sys/un.h>         /* za AF_UNIX */
#include <x25/x25addrstr.h> /* za AF_CCITT*/

int bind(int s, const void *addr, int addrlen);
```

Vrne 0, če bind uspe, sicer -1.

Ker je format adrese odvisen od domene, je tudi tip adresne strukture, ki nastopa kot drugi argument, odvisen od domene. Konkretno: za domeno `AF_INET` je struktura tipa `struct sockaddr_in`, za domeno `AF_INET` pa imamo `struct sockaddr_un`.

Na primer, za UNIX domeno:

```
...
struct sockaddr_un UnixAddress; /* ime je v bistvu pot */
...
UnixAddress.sun_family = AF_UNIX;
UnixAddress.sun_path   = "MojaPot";

bind( sd, &UnixAddress, sizeof( UnixAddress ) );
...
```

Definicija strukture `sockaddr_un` je v `/usr/include/sys/un.h`:

```
struct  sockaddr_un {
        short   sun_family;             /* AF_UNIX */
        char    sun_path[92];           /* path name (gag) */
};
```

Podobno, `sockaddr_in` je definirana v `/usr/include/netinet/in.h`.

```
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

9.7.5 Funkciji accept in listen

Potem, ko na vtičnico uspešno pripnemo veljavno ime, je za začetek komunikacije potrebna pripravljenost dveh, strežnika in odjemalca. Strežnik z `accept` čaka na `connect` odjemalca. Pred tem strežnik lahko z `listen` predpiše maksimalno število čakajočih zahtev za priključitev odjemalcev. Na primer, naj bo to število 5:

```
listen(sd,  5 );
```

Strežnik lahko od `accept` po želji pridobi naslov odjemalca, na primer:

```
sn = accept( sd, &NaslovOdjemalca, &DolzinaNaslova );
```

ali pa se zanj ne zanima:

```
sn = accept( sd, 0, 0 );
```

Seveda je oblika naslova in s tem v zvezi tip strukture drugega argumenta odvisen od domene.

```
#include <sys/socket.h>

int accept(int s, void *addr, int *addrlen);
```

Vrne nov deskriptor ali -1 v primeru napake.

Funkcija `accept` se uporablja skupaj z vtičnicami tipa `SOCK_STREAM`. Z `accept` strežnik čaka na `connect` odjemalca. Klic vrne nov deskriptor nove vtičnice, na kateri strežnik komunicira z odjemalcem.

9.7.6 Funkcija connect

```
#include <sys/socket.h>
#include <netinet/in.h>  /* samo za AF_INET */
#include <sys/un.h>      /* samo za AF_UNIX */
#include <x25/x25addrstr.h> /* za AF_CCITT  */

int connect(int s, const void *addr, int addrlen);
```

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.

9.7.7 Primer Odjemalca in strežnika v UNIX domeni

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( argc, argv )
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( argc, argv )
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));
  }
}
```

9.7.8 Internet Naslovi, imena, številke vrat

Vsako vozlišče/postaja v omrežju ima svoj naslov, po katerem je znana drugim postajam/vozliščem v omrežju. Naslov v Internet omrežju (Internet naslov ali na kratko IP naslov) je 32 bitno število (4 bajti). 32 bitni IP naslov se navadno podaja z desetiškim zapisom vrednosti posameznih bajtov. Vrednosti posameznih bajtov so nanizane druga za drugo in ločene s piko, na primer

```
193.2.72.140
```

je IP naslov ene od postaj na Fakulteti za elektrotehniko in računalništvo.

Naslovno področje Interneta (naslovi) je razdeljeno na razrede A, B, C, D in E. Razred A je za velika omrežja. Majhna omrežja spadajo v naslovni razred C. Razred se da hitro ugotoviti iz vrednosti zgornjega bajta naslova:

Razred	Naslovno območje	naslov omrežja	naslov vozlišča
A	0.0.0.0 do 127.255.255.255	7 bitov	24 bitov
B	128.0.0.0 do 191.255.255.255	14 bitov	16 bitov
C	192.0.0.0 do 223.255.255.255	21 bitov	8 bitov
D	224.0.0.0 do 239.255.255.255	-	28 bitov (množičen naslov)
E	240.0.0.0 do 247.255.255.255	-	27 bitov (rezervirano)

Naslovi postaj na FER spadajo v razred C. Naslove omrežij dodeljuje InerNIC, izbira naslovov postaj znotraj omrežja pa je poljubna, a mora biti enolična.

Vsaki postaji v omrežju je pridruženo ime – "Internet ime". Internet ime je sestavljeno iz imena postaje (ang. "host" name), domene, poddomene in podpoddomene. Na primer:

```
luz.fer.uni-lj.si
```

je polno Internet ime postaje z naslovom 193.2.72.140. Pri tem je `luz` ime postaje, `si` je ime Internet domene za Slovenijo, `uni-lj` je ime poddomene za Univerzo v Ljubljani, `fer` je ime podpoddomene FER.

Uporabniki največkrat poznajo (samo) imena postaj, čeprav je za delovanje omrežja pomembnejši IP naslov. V manjšem omrežju skrbi za preslikavo med imeni in naslovi kar vsaka posamezna postaja. V sistemu UNIX se preslikovalna tabela nahaja v datoteki `/etc/hosts`. Uporaba datoteke `hosts` se opušča. V večjih omrežjih (oziroma v novejšem času) omogoča dostop do postaj z imeni namesto z naslovi porazdeljena podatkovna baza DNS (ang. Domain Name System). Strežniki imen (ang. Name Servers), ki so postajam znani, na poizvedovanje za dano ime vrnejo iskani IP naslov.

Na posamezni postaji sočasno obstaja več procesov, ki želijo občasno komunicirati z drugimi procesi v omrežju. Za vzpostavitev zveze nekega procesa na eni postaji z enim od procesov na oddaljeni postaji naslov oddaljene postaje ni dovolj.

Potrebna je še številka *komunikacijskih vrat* (ang. communication port), na katerega se priključi proces. Za potrebe komunikacije je priključno mesto procesa popolnoma določeno s številko vrat in IP naslovom (ali imenom), npr. vrata 23 na postaji 192.2.72.140. Številka vrat je šestnajstbitno število. Števila 1 do 1024 so dodeljena s strani IANA (Internet Assigned Numbers Authority) in so rezervirana "za dobro znane" storitve (kot je denimo telnet, ftp,...). Na primer vrata 21 so za ftp strežnik, vrata 23 so za telnet. Števila do 5000 so rezervirana. Števila nad 5000 so na voljo za "privatne" potrebe.

9.7.9 Primer odjemalca in strežnika v INET naslovni domeni

Načelno sicer ni razlike med komunikacijo v UNIX in INET domeni. Vendar, ravno zaradi razlike v naslovih za različne domene obstaja s stališča programerja bistvena in včasih neprijetna razlika.
Naslednja programa vsebujeta osnovne elemente "strežnika" in "odjemalca". Strežnik na vratih 5500 čaka na vzpostavitev zveze odjemalca, dokler ne poteče z `alarm` nastavljeni čas (ena minuta). Ko se zveza z odjemalcem vzpostavi, ustvari (`fork`) nov proces, ki sprejema podatke od odjemalca, dokler odjemalec ne prekine zveze.
Odjemalec je preprost. Najprej ustvari dostop do vtičnice in nato s `connect` skuša vzpostaviti zveze z oddaljenim strežnikom. Po vzpostavitvi zveze bere vrstice besedila s tipkovnice in jih pošilja strežniku. Zveza se prekine s `close( sd )`.

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <fcntl.h>

extern int errno;

#define PORT_NUMBER 5500

int main( argc, argv )
int argc; char **argv;
{
  struct sockaddr_in sin_x;
  int    SizeSin_x;
  char   Buf[128];
  int    sd, sn, n;
  int    InitServer( );
  unsigned char *Ipa;

  if( argc != 2 ){
    printf("Uporaba: %s ImePostaje\n", argv[0]);
    exit( 1 );
```

```
  }
  if( (sd = InitServer( argv[1] )) < 0){
    perror("Napaka: "); exit( 1 );
  }
  listen( sd, 5 );
  alarm( 60 ); /* koncaj po eni minuti */
  while( 1 ){
    bzero((char *)&sin_x,sizeof(sin_x));
    SizeSin_x = sizeof(sin_x);
    if( (sn = accept(sd, (struct sockaddr_in *)&sin_x, &SizeSin_x)) < 0){
      perror("Napaka: "); exit( 2 );
    }
    /* zveza je vzpostavljena, ustvari strezni proces */
    if( fork() == 0 ){
      Ipa = (unsigned char *)&sin_x.sin_addr;
      printf("Zveza z %u.%u.%u.%u  je vzpostavljena\n", Ipa[0],Ipa[1],Ipa[2],Ipa[3]);
      while( (n = read( sn, Buf, sizeof( Buf ))) > 0 ){
        Buf[n] = 0;
        printf( "%s\n", Buf );
      }
      printf("Odjemalec je prekinil zvezo\n");
      exit( 0 );
    }
  }
} /* Konec main */

int InitServer( host )
char *host;
{
  struct sockaddr_in sin_x;
  struct hostent     *hp_x;
  int sd;
  unsigned char *Nas;

  if( (hp_x=gethostbyname(host)) == NULL) return( -1 );
  bzero((char *)&sin_x,sizeof(sin_x));
  bcopy(hp_x->h_addr,(char *)&sin_x.sin_addr,hp_x->h_length);
  sin_x.sin_family  = hp_x->h_addrtype;
  sin_x.sin_port    = PORT_NUMBER;

  printf("Ime: %s,  ", hp_x -> h_name);
  Nas = (unsigned char *)(hp_x -> h_addr);
  printf("Naslov: %u.%u.%u.%u\n", *Nas, *(Nas+1), *(Nas+2), *(Nas+3));

  if( (sd = socket(AF_INET,SOCK_STREAM,0)) < 0 ) return( -2 );
  if( bind(sd, (struct sockaddr_in *)&sin_x, sizeof( sin_x )) < 0) return( -3 );
  return( sd );
}

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

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <fcntl.h>

#define PORT_NUMBER 5500

extern  errno;

int main( argc, argv )
int argc; char **argv;
{
  int    i, sd, InitClient( );
  char   String[128];

  if( argc != 2 ){
    printf("Uporaba: %s ImePostaje\n", argv[0]);
    exit( 1 );
  }

  if( (sd = InitClient( argv[1] )) < 0 ){
    perror("Napaka: ");  exit( 1 );
  }
  else{
    printf("Tipkaj sporocilo, nato EOF\n");
    while( gets( String ) != NULL ){
      write(sd, String, strlen( String ) );
    }
    printf("Na svidenje\n");
    close( sd );
  }

  exit( 0 );

}

int InitClient( host )
char *host;
{
  struct sockaddr_in sin_x;
  struct hostent     *hp_x;
  unsigned char      *Addr;
  int    sd;

  if( (hp_x=gethostbyname(host)) == NULL) return( -1 );

  printf("Name   = %s, ", hp_x -> h_name);
  Addr = (unsigned char *) hp_x -> h_addr;
  printf("Address= %u.%u.%u.%u\n", *Addr, *(Addr+1), *(Addr+2), *(Addr+3));
  bzero((char *)&sin_x,sizeof(sin_x));
  bcopy(hp_x->h_addr,(char *)&sin_x.sin_addr,hp_x->h_length);
```

```
  sin_x.sin_family=hp_x->h_addrtype;
  sin_x.sin_port= PORT_NUMBER;

  if( (sd = socket(AF_INET,SOCK_STREAM,0)) < 0 ) return( -2 );
  if (connect(sd, (char *)&sin_x,sizeof(sin_x)) < 0) return( -3 );
  return( sd );
}
```