Bevezetés a számítástechnikába (jegyzet)

Bérci Norbert, Uhlár László, Tuza Zoltán

2013/2014 őszi félév

Tartalomjegyzék

1. fejezet

Linux bevezető

1.1. Egy kis történelem

1.1.1. A kezdetek

A számítógépek az ötvenes évektől a nyolcvanas évek elejéig a kutatók eszközei voltak és a tömegek elől gyakorlatilag el voltak zárva. Több kutató használt egy nagy gépet, a fejlesztéseiket megosztották egymással, igazi kis közösségek jöttek így létre. A kor egyik legendás gépe volt a PDP10-es, ezen az ITS nevű operációs rendszer futott, aminek továbbfejlesztésén dolgozott Richard Stallman is (lásd később). A PDP10 fejlesztését azonban a gyártója a nyolcvanas évekre abbahagyta, az intézeteknek újabb gépek után kellett nézniük. Ezek már más operációs rendszert futtattak, melyek nem voltak szabadok, már ahhoz is titoktartási szerződést kellett aláírniuk, ha egy futtatható másolatot akartak. Azaz tilos lett egymásnak segíteni, az eddig együttműködő közösségek felbomlottak, nem oszthatták meg egymással fejlesztéseiket.

1.1.2. A GNU projekt

Richard Stallman ezt az új helyzetet nem tudta elfogadni, elhatározta egy új, teljesen nyílt operációs rendszernek a megírását: 1983 táján létrehozta a GNU projektet, hogy terveit megvalósítsa. (A GNU jelentése: GNU is not UNIX). Ekkor fogalmazta meg a GNU kiáltványt (melyet teljes terjedelmében például a http://gnu.hu oldalon olvashatunk el) és a szabad szoftverekkel kapcsolatos alapelveit:

- a program szabadon használható bármilyen célra,
- a programot bárki szabadon módosíthatja igényei szerint,
- a programot bárki továbbadhatja akár ingyen akár pénzért,
- a program módosított verzióinak meg kell felelnie ugyanezen feltételeknek.

Ezen elvek jogilag is megfelelő formába öntésének eredményeképpen jött létre egy különleges licenc, a GPL (General Public Licence) (teljes szövege magyarul szintén a http://gnu.hu oldalon olvasható) és a szabad szoftvereket támogató alapítvány, az FSF (Free Software Foundation) is. Ez utóbbi céljáról, működéséről részletesebb leírás olvasható többek között az fsf.hu oldalon.

A GNU projekt keretében számos programot kifejlesztettek, már csak egy valami hiányzott: egy olyan kernel (az operációs rendszer magja), melyen futtatni lehetne ezeket.

1.1.3. A Linux

1991-ben egy finn egyetemista, Linus Torvalds épp egy új projekten kezdett el dolgozni: egy új, szabad operációs rendszeren, melyben ki akarta javítani az oktatásra akkoriban előszeretettel használt MINIX operációs rendszer hibáit, hiányosságait. Azaz adott volt egy kernel (Linus

[0] $Revision : 50\ Date : 2013-10-0722:02:54+0200(Mon, 07Oct2013)$

munkája) alkalmazások nélkül és adott volt egy alkalmazás gyűjtemény (GNU) kernel nélkül. Nem kellett sok idő ahhoz, hogy egymásra találjanak, így született meg a Linux, aminek az előzőek miatt a legpontosabb elnevezése: GNU/Linux. Ma már a GNU programokon kívül más projektből, más licenceléssel terjesztett szoftverek is tartoz(hat)nak egy disztribúcióhoz, így ma a Linux alatt a teljes operációs rendszert értjük. Pár év, és megjelentek az első disztribúciók: a kernel és a rengeteg GNU alkalmazás közül néhány összeépítve egy jól használható rendszerré (pl.: Debian: 1993. augusztus 16.).

Egy Linux disztribúció alatt tehát egy gondosan összeválogatott, rendszermagból, felhasználói- és rendszerszintű programokból álló, szerteágazó vagy specifikus felhasználásra alkalmas operációs rendszert értünk. Egy-egy nagyobb disztribúcióban olyan sok program található, hogy nagyon ritkán van szükség külső forrásból származó programok telepítésére. Ennek az az előnye, hogy a szoftverkomponensek egymáshoz alakíthatók, az együttes installálásuk és alkalmazásuk a lehető legkevesebb mértékben vezet hibás működésre. Sőt, a szoftvercsomagok egymásra épülése is megadható, így egy szoftver installálásakor a szükséges komponensek automatikusan telepíthetők, illetve az opcionális komponensek telepítésére felhívhatja a felhasználó figyelmét. Mindezek miatt a csomagok szigorú verziószámozással vannak ellátva, és az installáláshoz *csomagkezelő*t használunk, ami a fenti függőségeket ellenőrzi és az installálás óta kiadott frissítéseket is nyomon követi.

Disztribúciók közötti leggyakoribb különbségek

Disztribúciókat legtöbbször az különbözteti meg, hogy milyen célközönségnek és milyen feladatra készítik őket, így mindenki megtalálhatja a neki leginkább megfelelőt. Így léteznek olyanok, melyek lehetőséget nyújtanak arra, hogy szinte az összes konfigurálási lehetőséget egy grafikus felületen végezzük el és vannak olyanok is, amelyek megkövetelik, hogy a felhasználó mindent a konfigurációs állományok szerkesztésével állítson be a saját ízlésének megfelelően. Egyes disztribúciók célja, hogy mindig a lehető legfrissebb szoftvereket szállítsa, míg mások jól kitesztelt, stabil, ám emiatt kissé elavult csomagokat szállítanak. A legtöbb disztró adott közönséget céloz meg: profi vagy kezdő felhasználókat, adminisztrátorokat, „buherátorokat”, kevés memóriával rendelkező vagy csak CD-t tartalmazó gépeket stb. Néhány disztró a grafikus környezetet, míg mások inkább a karakteres konzolt támogatják.

További fontos különbség, hogy milyen csomagkezelőt használnak az adott terjesztésben. A könyvtárstruktúra általában hasonló módon van felépítve, viszont kisebb különbségek adódhatnak e tekintetben is, extrém esetekben teljesen eltérő felépítést is alkalmaznak a disztribútorok (pl.: GoboLinux). A disztrók egyik fő jellemzője az egyes programcsomagok installálásának, eltávolításának és frissítésének megkönnyítése és támogatása. A csomagkezelők a rengeteg feltelepíthető program karbantartását, frissítését, telepítését, stb. teszik könnyebbé: például a GNU/Debian 7 esetében kb. 37000 különböző program közül válogathatunk, így szinte biztosan megtaláljuk a felmerült feladataink megoldásához szükséges szoftvereket e bőséges választékban. Az egyes programok, csomagok pontos verzió számmal vannak ellátva, egy-egy program megfelelő működéséhez szükség lehet más programokra is, azaz függőségei lehetnek. Ezen függőségek (lehetőleg automatikus) feltelepítését is a csomagkezelők végzik.

Hardvertámogatás terén is adódhatnak különbségek, viszont alapvetően mind ugyanazt a kernelt használják, így elviekben ha egy disztribúció alatt egy hardver működik, akkor az bármely más, az adott architektúrát támogató disztribúció alatt is működésre bírható. Vannak céldisztribúciók is, például kifejezetten tűzfal vagy router üzemeltetésére. Megkülönböztethetjük őket az alapján is, hogy server, desktop vagy embedded felhasználásra szánják.

A disztrók nagy részének készítői komolyan veszik a biztonsági problémákat, és az ismert hibák javításait rövid időn belül elérhetővé teszik disztrójuk csomagfrissítési módszerének segítségével.

Nagy eltérések vannak a disztrók kiadásai között eltelt időben; egyes disztrók fix ciklust alkalmaznak (például 6 hónaponként egy új kiadás), más disztróknál nincs kötött kiadási ciklus. Léteznek kereskedelmi terjesztések és vállalati és otthoni / kis irodai disztribúciók is.

Nem mindegyik disztró ugyanazt a kernel verziót használja, továbbá sok disztró saját igényeinek megfelelően módosítja a hivatalosan kiadott, ún. „vanilla kernelt.”

A nagyobb és ismertebb disztribúciók (a teljesség igénye nélkül):

- UHU-Linux, magyar Linux-disztribúció
- Debian GNU/Linux
- Ubuntu, Kubuntu, Xubuntu
- Mandriva
- Red Hat
- Fedora
- CentOS
- openSUSE
- Slackware
- Gentoo
- Arch Linux
- Knoppix, Damn Small Linux, Live CD-ként való futtatásra tervezve
- CrunchBang Linux

Egy kis érdekesség: http://futurist.se/gldt/wp-content/uploads/12.10/gldt1210.svg

1.2. Programok futtatása

Nagyon fontos, hogy a jegyzet hátralévő részében szereplő parancsokat kipróbáljuk! Erre több lehetőségünk adódik:

- egy már telepített linux verzió használatával,
- a putty programmal belépünk valamely egyetemi szerverre (users, turdus) és ott dolgozunk
- egy letöltött és futtatott live CD/DVD/pendrive használatával (pl.: http://live.debian.net)
- új linux telepítésével (pl.: http://debian.org)

Bármelyik módot is használjuk, a belépés után a következő(höz hasonló) promptot kell kapnunk (ha grafikus felületet használunk, a belépés után el kell indtani a `Terminal` vagy `xterm` vagy valamilyen hasonló nevű alkalmazást):

```
bercin@users:~$
```

Ez a prompt a parancsértelmező [shell] kész állapotát jelöli, kezdhetjük begépelni a parancsokat. Mi a jegyzetben a `bash` parancsértelmezőt tárgyaljuk. A legtöbb Linux disztribúcióban, illetve egyéb UNIX-okon is ezt a shellt (vagy ezzel az általunk tárgyaltak szempontjából túlnyomó többségében kompatibilis változatot) használnak.

Figyelem! A Linux (és a UNIX-ok) a kis- és nagybetűket különböző betűknek tekintik mind a fájl- és könyvtárnevek, mind a parancsok neveinek (sőt, a parancsok paramétereinek) megadásakor. Különösen ügyeljünk a helyes használatra!

1.2.1. Paraméterek

Az `echo` program feladata, hogy kiírja a paraméterként átadott sztringeket:

```
bercin@users:~$ echo ABCD
ABCD
bercin@users:~$ echo EFGH IJKL MNO
EFGH IJKL MNO
bercin@users:~$
```

Itt tehát az `echo` a futtatott program, ami az első esetben egyetlen paramétert kap: `ABCD`, míg a második esetben három átadott paraméter van, amik a következők: `EFGH`, `IJKL`, `MNO`. A paramétereket a legtöbb esetben szóköz karakter választja el egymástól.

A parancsokról bővebb információt a `man` (manual - kézikönyv) paranccsal lehet kérni. Például az `echo` parancsról így:

```
bercin@users:~$ man echo
```

A `man` parancs paramétere annak a parancsnak a neve, aminek a kézikönyvét meg akarjuk jeleníteni. A megjelenített kézikönyvben a kurzormozgató billentyűkkel tudunk navigálni, és a `q` billentyűvel tudunk kilépni.

A programok többségének van `man` oldala, amiket a későbbiekben tárgyalt parancsok esetében is érdemes megnézni, mert a jegyzetben a parancsok lehetőségeinek csak töredékét tárgyaljuk. Magáról a parancsértelmezőről például a következő módon lehet bővebb információt szerezni:

```
bercin@users:~$ man bash
```

1.3. Könyvtárak és elérési utak

Minden futtatott program valamilyen könyvtárban fut, amit a program *aktuális könyvtár*ának nevezünk. Ez a shell esetében is így van. Az aktuális könyvtárat a `cd` paranccsal változtathatjuk meg: paraméterként annak a könyvtárnak a nevét kell megadni, amibe be akarunk lépni.

A könyvtárak fa struktúrában ábrázolhatók, azaz egy könyvtárban több másik könyvtár (vagy fájl) lehet, amikben ismét lehetnek újabb könyvtárak (vagy fájlok). Az viszont biztos, hogy minden könyvtárnak egyetlen szülő könyvtára van. Azt az útvonalat (könyvtárak adott sorrendjét), amellyel egy adott könyvtárhoz vagy fájlhoz eljuthatunk, a könyvtár vagy fájl *elérési út*jának nevezzük. Az elérési útban a könyvtárakat `/` jel választja el egymástól[1].

Minden könyvtárban létezik a `.` könyvtár, ami az aktuális könyvtárat (azaz saját magát), és a `..` nevű könyvtár, ami az adott könyvtár szülő könyvtárát jelöli. Próbáljuk ki:

```
bercin@users:~$ cd .
bercin@users:~$
```

Nem szabad meglepődni, hogy nem történt semmi, mert az aktuális könyvtárból az aktuális könyvtárba lépni nyilván semmilyen változást nem okozhat. Ugyanakkor a `..` (azaz a szülő) könyvtárba lépés már nem haszontalan:

```
bercin@users:~$ cd ..
bercin@users:/home$
```

Ennek a parancsnak az eredménye alapján láthatóvá vált: a parancsértelmező eddig is kiírta, hogy éppen melyik könyvtárban vagyunk, csak erre eddig nem fordítottunk figyelmet: most, a dollárjel előtti `/home` azt jelzi, hogy az aktuális könyvtár a `/home`-ra változott. Az eddig ott szereplő `~` (hullámjel, tilde) rövidítés az alapértelmezett könyvtárunkat jelezte. Linux (UNIX) alatt alapértelmezett esetben a felhasználóknak van egy könyvtára, ahova a felhasználó írási joggal rendelkezik, és ebbe a könyvtárba kerül, amikor belép a szerverre. Ez a könyvtár a `/home/felhasználónév` (ahol `felhasználónév` a saját felhasználónk neve) amit a felhasználó

[1] Figyelem! A könyvtár elválasztó jel Linuxban (és UNIX-ban) „jobbra dőlő” perjel, nem az, amit a Windows használ (ahol „balra dőlő” perjel választja el a könyvtárakat).

*home könyvtárá*nak nevezünk. A `cd ..` parancs tehát a `/home/felhasználónév` könyvtárból ennek szülő könyvtárába, azaz a `/home` könyvtárba vitt, és a parancsértelmező ezt a változást jelezte a promptban.

Az aktuális könyvtár neve lekérdezhető a `pwd` parancs segítségével:

```
bercin@users:/home$ pwd
/home
bercin@users:/home$ cd ..
bercin@users:/$ pwd
/
bercin@users:/$
```

A `/` nevű könyvtár a fájlrendszer gyökér könyvtárát jelöli, azaz azt a könyvtárt, aminek a szülő könyvtára is saját maga, így ebből a könyvtárból feljebb már nem lehet lépni:

```
bercin@users:/$ pwd
/
bercin@users:/$ cd ..
bercin@users:/$ pwd
/
bercin@users:/$
```

Ha az elérési út `/` jellel kezdődik, akkor az elérési utat *teljes elérési út*nak [full path] nevezzük. A `pwd` parancs mindig az aktuális könyvtár teljes elérési útját írja ki:

```
bercin@users:/$ pwd
/
bercin@users:/$ cd home
bercin@users:/home$ pwd
/home
bercin@users:/home$ cd bercin
bercin@users:~$ pwd
/home/bercin
bercin@users:~$
```

Ha az elérési út nem `/` jellel kezdődik, akkor is elérési útról beszélünk, de ezt az elérési utat *relatív elérési út*nak [relative path] nevezzük. A relatív elérési út azt jelenti, hogy az elérési út nem a gyökér könyvtártól, hanem az aktuális könyvtártól indul.

```
bercin@users:~$ pwd
/home/bercin
bercin@users:~$ cd /usr
bercin@users:/usr$ pwd
/usr
bercin@users:/$ cd local/bin
bercin@users:/usr/local/bin$ pwd
/usr/local/bin
bercin@users:/usr/local/bin$
```

A paraméter nélküli `cd` parancs visszavisz minket a home könyvtárunkba (bármi is az aktuális könyvtár):

```
bercin@users:/usr/local/bin$ pwd
/usr/local/bin
bercin@users:/usr/local/bin$ cd
bercin@users:~$ pwd
/home/bercin
bercin@users:~$
```

Új könyvtárat az `mkdir` paranccsal hozhatunk létre, üres könyvtárat az `rmdir` paranccsal törölhetünk:

```
bercin@users:~$ mkdir teszt
bercin@users:~$ cd teszt
bercin@users:~/teszt$ pwd
/home/bercin/teszt
bercin@users:~/teszt$ cd ..
bercin@users:~$ rmdir teszt
bercin@users:~$
```

1.4. Környezeti változók

A programok számára többféleképpen adhatók át adatok, aminek az egyik módja az előző példákban is látható parancssori paraméterként átadás. Egy másik módja a környezeti változókon keresztül történik, amit a programok a futásuk során lekérdezhetnek (és módosíthatnak). Egy környezeti változónak úgy lehet értéket adni, hogy a változó neve után egy egyenlőség jelet majd a beállítani kívánt értéket írjuk. Például a `PLD` változónak az `abcd` értékül adása a következő módon történik:

```
bercin@users:~$ PLD=abcd
bercin@users:~$
```

Az aktuális értéket a változónév elé tett `$` jellel lehet lekérdezni oly módon, hogy a parancssorban szereplő `$változónév` szöveget a shell a `változónév` nevű változó aktuális értékére *cseréli ki*, majd a parancssort újra értelmezi. Például az `echo $PLD` értelmezése a következőképpen történik: A shell először kicseréli a `$PLD` sztringet a PLD változó értékére (ami jelen esetben `abcd`), így eredményül az `echo abcd` parancssort kapja, amit aztán újra értelmez és végrehajt:

```
bercin@users:~$ echo $PLD
abcd
bercinusers~$
```

Nagyon fontos ismételten hangsúlyozni, hogy a shell végzi a `$PLD` sztring kicserélését a változó tartalmára, nem pedig a futtatott program! A program már csak a kicserélt sztringet kapja meg paraméterként, mit sem sejtve arról, hogy az eredetileg mi volt: az előző példában az `echo` tehát már csak az `abcd` paramétert kapja meg (amit aztán kiír).

A shellben tárolt környezeti változók listáját aktuális értékeikkel együtt a `set` paranccsal kaphatjuk meg.

1.4.1. Összefűzés

Szükségünk lehet arra, hogy egy környezeti változó aktuális értéke elé és/vagy mögé egy másik sztringet is beszúrjunk. Írassuk ki két környezeti változó értékét egymáshoz fűzve:

```
bercin@users:~$ ELEJE=abcd
bercin@users:~$ VEGE=efgh
bercin@users:~$ echo $ELEJE$VEGE
abcdefgh
bercin@users:~$
```

A sikeren felbuzdulva megpróbálhatunk egy környezeti változó értékéhez közvetlenül hozzáfűzni egy sztringet:

```
bercin@users:~$ echo $ELEJEefgh

bercin@users:~$ echo $ELEJE
abcd
```

```
bercin@users:~$ echo efgh
efgh
bercin@users:~$
```

Az utasítás azért nem működik (pontosabban működik, csak nem azt az eredményt adja, amit vártunk), mert a shell nem tudja, hogy mi az `ELEJE` környezeti változó értékét és a `efgh` sztringet akartuk összefűzni, hanem az `ELEJEefgh` nevű környezeti változó értékét kérdezi le, ami üres. A megoldás, hogy a `{}` karakterek közé írva a környezeti változó nevét, a shell már pontosan meg fogja tudni határozni, hogy meddig tart a környezeti változó neve, és honnét kezdődik a parancssor többi része:

```
bercin@users:~$ echo ${ELEJE}efgh
abcdefgh
bercin@users:~$
```

Természetesen ez a jelölésmód az előző példákkal is használható:

```
bercin@users:~$ echo ${ELEJE}
abcd
bercin@users:~$ echo ${VEGE}
efgh
bercin@users:~$ echo ${ELEJE}${VEGE}
abcdefgh
bercin@users:~$
```

1.4.2. PATH

Magának a parancsértelmezőnek is szüksége van néhány beállításra a működéséhez, ilyen például a `PATH` nevű környezeti változó, ami megadja, hogy melyik könyvtárakban kell keresni a begépelt parancsokat. Írassuk ki a PATH változó aktuális értékét a fentebb ismertetett módon:

```
bercin@users:~$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/games
bercin@users:~$
```

Eredményül teljes elérési utak kettősponttal elválasztott sorozatát kapjuk, ami azt mutatja, hogy a begépelt parancsok keresése *kizárólag* mely könyvtárakban történik. Esetünkben ezek a `/usr/local/bin`, `/usr/bin`, `/bin` és a `/usr/games` könyvtárak (ebben a sorrendben!). Ha a begépelt parancsot ezen könyvtárak valamelyikében megtalálja a parancsértelmező, akkor azt lefuttatja. Ha nincs ilyen fájl, akkor hibajelzést ad:

```
bercin@users:~$ svnsjkvnasjlvn
-bash: svnsjkvnasjlvn: command not found
bercin@users:~$
```

A `which` paranccsal megtudhatjuk, hogy egy (létező) program pontosan melyik, PATH-ban lévő könyvtárban található. Például az `echo` program elérési útja:

```
bercin@users:~$ which echo
/bin/echo
bercin@users:~$
```

A `which` programot bármely másik programra meghívhatjuk, akár saját magára is (ekkor azt fogja kiírni, hogy ő maga hol található):

```
bercin@users:~$ which which
/usr/bin/which
bercin@users:~$
```

1.4.3. PS1

A `PS1` környezeti változóval beállíthatjuk a parancsértelmező promptját. Mielőtt ezt megváltoztatnánk, mentsük el a régi értéket például a `MENTES` környezeti változóba:

```
bercin@users:~$ MENTES=$PS1
bercin@users:~$
```

A környezeti változó lekérdezését nyilván nem csak az `echo` paranccsal használhatjuk, hanem bármelyik másik paranccsal is. Idézzük fel a környezeti változó értékének a lekérdezését: először a shell kicseréli a `$változónév` kifejezést a `változónév` értékére, majd az így kapott sort újra értelmezi. Az előző példában tehát a `$PS1` helyére behelyettesítődött a `PS1` értéke, amit aztán a `MENTES` változónak adtunk értékül.

A `PS1` átállítása után az új prompttal ugyanúgy használhatjuk a parancsértelmezőt, mint eddig (ne lepődjünk meg, hogy furcsán néz ki a sor, úgy gépeljük be a parancsainkat, mintha az előzőekben megszokott prompt lenne!):

```
bercin@users:~$ PS1=_ez_az_en_promptom_
_ez_az_en_promptom_echo szia
szia
_ez_az_en_promptom_which echo
/bin/echo
_ez_az_en_promptom_
```

Nézzük meg, hogyan állíthatjuk vissza az eredeti állapotot:

```
_ez_az_en_promptom_PS1=$MENTES
bercin@users:~$
```

1.4.3.1. feladat. Mi a `PS1` (vagy a `MENTES`) tartalma?

A `PS1` beállításáról a későbbiekben lesz még szó bővebben.

1.4.4. LANG

A `LANG` környezeti változó beállításával átállíthatjuk a területi beállításokat, így például megváltoztathatjuk a parancsok által kiírt üzenetek nyelvét. Értékül a használni kívánt nyelv kétbetűs kódját kisbetűkkel és az ország kétbetűs kódját nagybetűkkel kell megadni, aláhúzásjellel elválasztva:

```
bercin@users:~$ svnsjkvnasjlvn
-bash: svnsjkvnasjlvn: command not found
bercin@users:~$ LANG=hu_HU
bercin@users:~$ svnsjkvnasjlvn
-bash: svnsjkvnasjlvn: parancs nem található
bercin@users:~$
```

Ha a magyarul kiírt hibaüzenetben az ékezetes betűk helyén kérdőjel, vagy egyéb más „furcsa” karakter szerepel, akkor a `hu_HU` helyett próbáljuk ki a `hu_HU.UTF8` értéket. Az UTF-8 karakterkódolásról a későbbiekben lesz részletesen szó.

1.4.4.1. feladat. Hogyan tudjuk a beállítás előtt elmenteni és az átállítás után az eredetire visszaállítani a `LANG` környezeti változót?

1.4.4.2. feladat. Hogyan tudjuk átállítani a területi beállításokat a Németországban használt németre?

1.4.4.3. feladat. Hogyan tudjuk átállítani a területi beállításokat az USA (délnyugati részén gyakran) használt spanyolra?

1.5. Fájlok kezelése

Az `ls` paranccsal kilistázható az aktuális könyvtár tartalma (ha nem adunk meg paramétert) vagy bármely más könyvtár tartalma (ha megadunk egy elérési utat):

```
bercin@users:~$ ls
public_html  teszt  ZHk
bercin@users:~$ ls /bin
bash          dd             lessecho        nisdomainname  tar
bunzip2       df             lessfile        pidof          tempfile
bzcat         dir            lesskey         ping           touch
bzcmp         dmesg          lesspipe        ping6          true
bzdiff        dnsdomainname  ln              ps             umount
bzegrep       domainname     login           pwd            uname
bzexe         echo           ls              rbash          uncompress
bzfgrep       ed             lsmod           readlink       vdir
bzgrep        egrep          mkdir           rm             which
bzip2         false          mknod           rmdir          ypdomainname
bzip2recover  fgrep          mktemp          rnano          zcat
bzless        fuser          more            run-parts      zcmp
bzmore        getfacl        mount           rzsh           zdiff
cat           grep           mountpoint      sed            zegrep
chacl         gunzip         mt              setfacl        zfgrep
chgrp         gzexe          mt-gnu          sh             zforce
chmod         gzip           mv              sh.distrib     zgrep
chown         hostname       nano            sleep          zless
cp            ip             nc              stty           zmore
cpio          kill           nc.traditional  su             znew
dash          ksh            netcat          sync           zsh
date          less           netstat         tailf          zsh4
bercin@users:~$
```

Ebben a példában az aktuális könyvtár elemeit (első-második sor) és a `/bin` könyvtár elemeit listáztuk ki (harmadik sortól az utolsóig).

Egy újonnan létrehozott könyvtár is tartalmazza a `.` és `..` könyvtárakat, de ezeket az `ls` alapértelmezetten nem mutatja:

```
bercin@users:~$ mkdir teszt
bercin@users:~$ cd teszt
bercin@users:~/teszt$ ls
bercin@users:~/teszt$
```

Linuxban (és UNIX-ban) a ponttal kezdődő könyvtárnevek és fájlnevek rejtettek, azaz az `ls` alapértelmezetten nem listázza ki ezeket. Ha az `ls` parancsot a `-a` paraméterrel hívjuk meg, akkor már megmutatja a ponttal kezdődő könyvtárakat illetve állományokat is:

```
bercin@users:~/teszt$ ls -a
.  ..
bercin@users:~/teszt$
```

Természetesen a `.` és `..` könyvtár minden könyvtárban megtalálható, bármelyiket is listázzuk ki:

```
bercin@users:~/teszt$ cd
bercin@users:~$ ls
public_html  teszt  ZHk
bercin@users:~$ ls -a
.   .bash_history  .bashrc  .lesshst  .profile     .ssh   ZHk
..  .bash_logout   .gnupg   .mc       public_html  teszt
bercin@users:~$
```

Jól látható ezen a példán, hogy a `.` és `..` könyvtárak mellett egy felhasználó home könyvtára általában sok más, ponttal kezdődő fájlt is tartalmaz, amik az előzőek alapján szintén rejtettek.

Linux (UNIX) esetében a felhasználó által futtatott programok beállításait általában ponttal kezdődő fájlnevű állományban tároljuk. A programjaink config fájljai tehát azért a felhasználó home könyvtárában vannak, mert így lehetővé válik, hogy felhasználónként más és más beállítások legyenek. Szükségszerű is itt tárolni a config állományokat, hiszen ezek máshol nem biztos, hogy tárolhatók, mert általános esetben a felhasználónak csak a home könyvtárában van írási joga.

Fájlokat másolni a `cp` (copy) paranccsal lehet, aminek első paraméterként meg kell adni azt az elérési utat, amit másolni szeretnénk, második paraméterként pedig azt az elérési utat, ahova másolni szeretnénk. Például a már jól ismert `echo` programot másoljuk a saját home könyvtárunkba `masik_echo` néven:

```
bercin@users:~$ cp /bin/echo masik_echo
bercin@users:~$ ls
masik_echo  public_html  teszt  ZHk
bercin@users:~$
```

Hiába létezik az aktuális könyvtárban a `masik_echo` fájl, ha a megpróbáljuk lefuttatni, hibajelzést kapunk:

```
bercin@users:~$ masik_echo
-bash: masik_echo: command not found
bercin@users:~$
```

Ennek magyarázata, hogy a shell kizárólag a `PATH` környezeti változóban felsorolt könyvtárakban keres, és ezek között nem található meg az aktuális könyvtár:

```
bercin@users:~$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/games
bercin@users:~$
```

A problémát egyszerűen orvosolhatjuk: a parancsnév helyett adjuk meg a futtatandó parancs elérési útját, így a shellnek nem kell a `PATH`-ban lévő könyvtárakban keresnie, hanem közvetlenül a megadott fájlt próbálja meg futtatni. Elérési útként megadható teljes elérési út:

```
bercin@users:~$ /home/bercin/masik_echo szia
szia
bercin@users:~$
```

És megadható relatív elérési út is:

```
bercin@users:~$ ./masik_echo szia
szia
bercin@users:~$
```

A shell úgy tudja megkülönböztetni, hogy relatív elérési utat vagy elérési út nélküli parancsnevet adtunk meg, hogy ha a parancsban nincsen `/` karakter, akkor parancsnévről van szó (és a `PATH`-ban megadott könyvtárakban keresi a parancsot), de ha van benne `/` karakter, akkor elérési útról van szó (sőt, ha az elérési út `/` jellel kezdődik, akkor abszolút elérési út, azaz a gyökér könyvtártól indul, ha nem `/` jellel kezdődik, akkor relatív, azaz az aktuális könyvtártól indul).

Pontosan azért volt tehát szükség az előző példában a `./` szerepeltetésére, hogy a shell tudja, elérési úttal megadott parancsot akarunk futtatni, és a `.` könyvtár megadásával ez pontosan azt jelenti, hogy az aktuális könyvtárban lévő fájlról van szó.

Erre a problémára az előző megoldáson kívül másik két megoldás is létezik: az egyik, hogy az aktuális könyvtárat (tehát a `.` könyvtárat) is beillesztjük a `PATH` listába, a másik, hogy létrehozunk egy saját könyvtárat a home könyvtárunkban, ahol a saját futtatható állományainkat fogjuk tárolni. Az első megoldás biztonsági kockázatokat rejt (ezért nem is keres a shell az aktuális könyvtárban), így válasszuk a második megoldást. Hívjuk ezt a könyvtárat `bin`-nek:

```
bercin@users:~$ mkdir bin
bercin@users:~$ PATH=$PATH:~/bin
bercin@users:~$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/games:/home/bercin/bin
bercin@users:~$
```

Létrehoztuk tehát a `bin` könyvtárat a home könyvtárunkban, majd a `PATH` környezeti változó végéhez hozzáfűztük ennek az újonnan létrehozott könyvtárnak az elérési útját. Részletesebben itt két csere is történt: a `$PATH:~/bin` sztringet a shell értelmezte és a `$PATH` sztringet kicserélte a `PATH` környezeti változó aktuális értékére, majd a `~` jelet is kicserélte a felhasználó home könyvtárának elérési útjára, így eredményül (ebben az esetben) a

```
/usr/local/bin:/usr/bin:/bin:/usr/games:/home/bercin/bin
```

értéket kapta. A végrehajtási fázisban tehát az történt, mintha a

```
PATH=/usr/local/bin:/usr/bin:/bin:/usr/games:/home/bercin/bin
```

parancsot írtuk volna be, ezt bizonyítja a `PATH` környezeti változó értéke kiíratásának eredménye.

A feladat megoldásához már csak egy lépés van hátra, az aktuális könyvtárban lévő `masik_echo` fájlt át kell mozgatni a `bin` könyvtárba, amit a `mv` (move) parancs segítségével tehetünk meg:

```
bercin@users:~$ ls
masik_echo  public_html  teszt  ZHk
bercin@users:~$ ls -a bin
.  ..
bercin@users:~$ mv masik_echo bin
bercin@users:~$ ls -a bin
.  ..  masik_echo
bercin@users:~$ ls
bin  public_html  teszt  ZHk
bercin@users:~$
```

Próbáljuk ki, hogy most már működik-e a `masik_echo` program indítása, holott az nincs is az aktuális könyvtárban:

```
bercin@users:~$ ls
bin  public_html  teszt  ZHk
bercin@users:~$ masik_echo szia
szia
bercin@users:~$
```

Az `mv` parancs az átmozgatás mellett átnevezésre is használható: gondoljunk bele, ha átmozgatok egy fájlt az aktuális könyvtárból az aktuális könyvtárba csak másik néven, akkor valójában átnevezést hajtottam végre. Emiatt nincs külön parancs átnevezésre a Linuxban (és a UNIX-okban).

Ha a `cp` vagy `mv` parancsnak második paraméterként (tehát a cél elérési útként) könyvtárat adunk meg, akkor abba a könyvtárba ugyanolyan néven fog másolódni (`cp` esetében) illetve átmozgatódni (`mv` esetében) a forrásként megadott fájl. Természetesen több fájlt is másolhatunk egy paranccsal, ha felsoroljuk az összes másolandó fájlt, majd az utolsó paraméterként megadjuk azt a könyvtárat, ahova azokat másolni szeretnénk. Ekkor az utolsó paraméter csak könyvtár lehet!

```
bercin@users:~$ ls bin
masik_echo
bercin@users:~$ cp /bin/mv /bin/cp /bin/mkdir /bin/rmdir bin
bercin@users:~$ ls bin
cp  masik_echo  mkdir  mv  rmdir
bercin@users:~$
```

Fájlok törlésére az `rm` parancs szolgál, paraméterekként meg kell adni a törlendő fájlok elérési útjait. Például az imént a `bin` könyvtárba másolt állományok közül néhány törlése:

```
bercin@users:~$ ls bin
cp  masik_echo  mkdir  mv  rmdir
bercin@users:~$ rm bin/mv bin/mkdir bin/cp
bercin@users:~$ ls bin
masik_echo  rmdir
bercin@users:~$
```

Természetesen ha egy könyvtár összes fájlját törölni szeretnénk, nem kell azokat egyesével felsorolni, hanem megadhatjuk a könyvtárnevet is az `rm` parancsnak:

```
bercin@users:~$ rm bin
rm: cannot remove ‘bin’: Is a directory
bercin@users:~$
```

Mivel alapértelmezetten az `rm` parancs fájlok törlésére szolgál, külön paraméter megadása szükséges, hogy könyvtárat töröljünk: `-r` (rekurzív törlés)

```
bercin@users:~$ rm -r bin
bercin@users:~$
```

Figyelem! Linuxban (és UNIX-ban) a törlés általában visszafordíthatatlan művelet! Nincs „Trash” vagy „Kuka”, amibe ideiglenesen átkerülnek a törölt állományok! Az `rm` parancs kiadásakor azok azonnal törlődnek.

1.5.1. Mintaillesztés

A shell nem csak a környezeti változók értékének lekérdezésére használt `$` jelet értelmezi speciálisan, hanem lehetőség van a fájlok neveinek mintaillesztésére, azaz a létező fájlnevek közül adott mintának megfelelők kiválasztására. A shell által erre a célra használt speciális karakterek és jelentéseik az 1.1. táblázatban láthatóak.

illesztő karakter	karakterek, amelyek megfelelnek a mintának
*	bármely karakter akárhányszor (nulla alkalommal is!)
?	bármely karakter pontosan egy alkalommal
[karakterek]	karakterek közül bármelyik pontosan egy alkalommal

1.1. táblázat. A shell mintaillesztő karakterei

A mintaillesztést a `/bin` könyvtárban fogjuk szemléltetni. Álljon itt egy lista a benne található fájlokról annak érdekében, hogy a lentebbi példákat a szerverre való belépés nélkül is meg lehessen érteni.

```
bercin@users:~$ cd /bin
bercin@users:/bin$ ls
bash          dd            lessecho      nisdomainname  tar
bunzip2       df            lessfile      pidof          tempfile
bzcat         dir           lesskey       ping           touch
bzcmp         dmesg         lesspipe      ping6          true
bzdiff        dnsdomainname ln            ps             umount
bzegrep       domainname    login         pwd            uname
bzexe         echo          ls            rbash          uncompress
bzfgrep       ed            lsmod         readlink       vdir
bzgrep        egrep         mkdir         rm             which
bzip2         false         mknod         rmdir          ypdomainname
bzip2recover  fgrep         mktemp        rnano          zcat
bzless        fuser         more          run-parts      zcmp
```

```
bzmore        getfacl       mount           rzsh          zdiff
cat           grep          mountpoint      sed           zegrep
chacl         gunzip        mt              setfacl       zfgrep
chgrp         gzexe         mt-gnu          sh            zforce
chmod         gzip          mv              sh.distrib    zgrep
chown         hostname      nano            sleep         zless
cp            ip            nc              stty          zmore
cpio          kill          nc.traditional  su            znew
dash          ksh           netcat          sync          zsh
date          less          netstat         tailf         zsh4
bercin@users:/bin$
```

Írassuk ki a `c` karakterrel kezdődő fájlokat:

```
bercin@users:/bin$ ls c*
cat  chacl  chgrp  chmod  chown  cp  cpio
bercin@users:/bin$
```

A `c*` mintának tehát megfelel minden olyan fájl, ami `c` karakterrel kezdődik és utána bármely karakterből bárhány szerepel. A `*` karakter azonban nem csak a minta végén lehet, sőt, nem csak egy szerepelhet belőle egy mintában:

```
bercin@users:/bin$ ls *l*
bzless  getfacl  lessecho  lesspipe  ls              readlink  tailf
chacl   kill     lessfile  ln        lsmod           setfacl   tempfile
false   less     lesskey   login     nc.traditional  sleep     zless
bercin@users:/bin$
```

A `*l*` mintának megfelel az összes olyan fájl, amiben legalább egy `l` betű szerepel. Hasonlóan egyszerű kilistázni azokat a fájlokat, amelyekben legalább egy `l` és utána legalább egy `s` betű szerepel:

```
bercin@users:/bin$ ls *l*s*
bzless  false  less  lessecho  lessfile  lesskey  lesspipe  ls  lsmod  zless
bercin@users:/bin$
```

Figyelem! A karakterek sorrendje fontos! Az előző minta nem ugyanaz, mint a `*s*l*`:

```
bercin@users:/bin$ ls *s*l*
lessfile  setfacl  sleep
bercin@users:/bin$
```

1.5.1.1. példa. Listázzuk ki azokat a fájlokat, amelyek második karaktere `e`:

```
bercin@users:/bin$ ls ?e*
getfacl  lessecho  lesskey   netcat   readlink  setfacl   zegrep
less     lessfile  lesspipe  netstat  sed       tempfile
bercin@users:/bin$
```

1.5.1.2. példa. Listázzuk ki azokat a fájlokat, amelyek második karaktere `e` és a hatodik karaktere `c`:

```
bercin@users:/bin$ ls ?e???c*
getfacl  lessecho  setfacl
bercin@users:/bin$
```

1.5.1.3. példa. Listázzuk ki azokat a fájlokat, amelyek pontosan három karakterből állnak:

```
bercin@users:/bin$ ls ???
cat  dir  ksh  pwd  sed  tar  zsh
bercin@users:/bin$
```

1.5.1.4. példa. Listázzuk ki azokat a fájlokat, amelyek az `abcd` karakterek közül valamelyikkel kezdődnek és utolsó karakterük `e`:

```
bercin@users:/bin$ ls [acbd]*e
bzexe  bzmore  date  dnsdomainname  domainname
bercin@users:/bin$
```

A mintaillesztés természetesen nem csak a fájlok kilistázásánál, hanem másolásnál (`cp`), átmozgatásnál (`mv`), törlésnél (`rm`) is hasznos, sőt – mivel a behelyettesítést a shell végzi – bármely más program is felhasználhatja ezt a funkciót, ha fájlok neveit várja paraméterként.

1.5.2. Egyéb speciális karakterek

Eddig elhallgattuk, de a szemfüles olvasó már találkozhatott azzal a problémával, hogy (például könyvtár létrehozásánál) a fentiek alapján nem tud szóköz karaktert tartalmazó nevet használni, hiszen a szóköz karakter paraméter elválasztó karakter, így ha megpróbálunk egy `Kedves Hallgatok` nevű könyvtárat létrehozni, akkor azt két külön könyvtárként fogja a shell létrehozni:

```
bercin@users:~$ mkdir Kedves Hallgatok
bercin@users:~$ ls
bin  Hallgatok  Kedves  public_html  teszt  ZHk
bercin@users:~$
```

Hasonló a probléma a mintaillesztő karakterek használatával is:

```
bercin@users:~$ mkdir ***K***
mkdir: cannot create directory ‘Kedves’: File exists
bercin@users:~$ ls
bin  Hallgatok  Kedves  public_html  teszt  ZHk
bercin@users:~$
```

A hibaüzenet magyarázata, hogy a shell a `***K***` mintát behelyettesíti az aktuális könyvtárban található `Kedves` fájlnévvel (ez létező név, hiszen éppen az előbb hoztuk létre ezt a könyvtárat), így az `mkdir` parancs a `Kedves` paramétert kapja, ami nyilván hibához vezet, hiszen nem hozható létre egy már meglévő névvel új könyvtár.

A problémát úgy orvosolhatjuk, hogy a speciális karakterek speciális jelentését kikapcsoljuk, azaz a kikapcsolás után a karakterek már önmagukat jelentik: a `$` karaktert nem próbálja meg a shell változó értékére cserélni, a mintaillesztő karaktereket nem próbálja meg fájlnevekre illeszteni, stb. A speciális jelentés kikapcsolására a `\` karakter használandó. Figyelem! Itt „balra dőlő” perjelről van szó, nem a könyvtárakat elválasztó „ jobbra dőlő” perjelről!

Egy szóközt tartalmazó könyvtár létrehozására használható tehát a következő módszer:

```
bercin@users:~$ mkdir Ez\ itt\ egy\ teljes\ mondat
bercin@users:~$ ls
bin  Ez itt egy teljes mondat  Hallgatok  Kedves  public_html  teszt  ZHk
bercin@users:~$
```

Egy kicsit problémás minden speciális karakter elé beszúrni a `\` jelet, de ez a probléma is meg van oldva: ha több karakter speciális jelentését akarjuk kikapcsolni, akkor a sztringet `'` jelek közé kell tenni:

```
bercin@users:~$ mkdir 'Ez itt egy masik hosszu nevu konyvtar'
bercin@users:~$ ls
bin                                 Hallgatok    teszt
Ez itt egy masik hosszu nevu konyvtar  Kedves       ZHk
Ez itt egy teljes mondat            public_html
bercin@users:~$
```

Néha azonban mégis szükség lenne arra, hogy környezeti változókat is tudjunk idézőjelek között megadott sztringekben szerepeltetni:

```
bercin@users:~$ szamlaszam=123456789
bercin@users:~$ mkdir 'A bankszamlam szama: $szamlaszam'
bercin@users:~$ ls
A bankszamlam szama: $szamlaszam       Ez itt egy teljes mondat  public_html
bin                                    Hallgatok                 teszt
Ez itt egy masik hosszu nevu konyvtar  Kedves                    ZHk
bercin@users:~$
```

A könyvtárnévben nem helyettesítődött be a `szamlaszam` környezeti változó értéke. Megoldás: ha a sztinget " jelek közé tesszük, akkor a shell a $ értelmezését továbbra is meg fogja tenni:

```
bercin@users:~$ mkdir "A bankszamlam szama: $szamlaszam"
bercin@users:~$ ls
A bankszamlam szama: 123456789         Ez itt egy teljes mondat  teszt
A bankszamlam szama: $szamlaszam       Hallgatok                 ZHk
bin                                    Kedves
Ez itt egy masik hosszu nevu konyvtar  public_html
bercin@users:~$
```

Természetesen magának a környezeti változónak is adható a fenti módon szóközt (vagy egyéb más speciális karaktert) tartalmazó érték:

```
bercin@users:~$ telefonszam="+36 12 345 6789"
bercin@users:~$ mkdir "A telefonszamom: $telefonszam"
bercin@users:~$ ls
A bankszamlam szama: 123456789         Hallgatok
A bankszamlam szama: $szamlaszam       Kedves
A telefonszamom: +36 30 123 4567       public_html
bin                                    teszt
Ez itt egy masik hosszu nevu konyvtar  ZHk
Ez itt egy teljes mondat
bercin@users:~$
```

A promptot most már átállíthatjuk úgy, hogy az sokkal szebben nézzen ki, mint az első példában (lásd az 1.4.3. részt):

```
bercin@users:~$ PS1="Ez az en promptom: "
Ez az en promptom: echo szia
szia
Ez az en promptom: ls
A bankszamlam szama: 123456789         Hallgatok
A bankszamlam szama: $szamlaszam       Kedves
A telefonszamom: +36 30 123 4567       public_html
bin                                    teszt
Ez itt egy masik hosszu nevu konyvtar  ZHk
Ez itt egy teljes mondat
Ez az en promptom:
```

A prompt beállításakor használhatunk speciális értékeket is, amiket a shell értelmezni fog, és a megfelelő értékre cserél. Ezek közül a leggyakrabban használtak az 1.2. táblázatban láthatóak.

Ahogy a táblázatból látszik, a \ karakter egyes esetekben speciális értelmezésű karakterek speciális értelmezésének kikapcsolására, más esetekben normál értelmezésű karakterek speciális értelmezésének bekapcsolására szolgál. Figyelem! A táblázatban szereplő karakterek a prompt beállításában értelmezettek csak a táblázat szerint!

1.5.2.1. példa. A prompt `felhasználó: <felhasználó> gép: <gép> könyvtár: <könyvtár> $ -` re beállítása a következő módon végezhető el:

```
bercin@users:~$ PS1="felhasználó: \u gép: \h könyvtár: \w $ "
felhasználó: bercin gép: www-users könyvtár: ~ $ echo szia
szia
felhasználó: bercin gép: www-users könyvtár: ~ $
```

speciális karakter	karakterek, amelyek megfelelnek a mintának
\u	felhasználónév
\h	a gép neve, amire be vagyunk jelentkezve
\H	a gép teljes neve, amire be vagyunk jelentkezve
\w	az aktuális könyvtár teljes elérési úttal
\W	az aktuális könyvtár elérési útjának utolsó eleme
\$	egyszerű felhasználó esetében $ jel, rendszergazda esetében # jel

1.2. táblázat. Prompt beállításához használható legfontosabb speciális karakterek

1.6. Átirányítás

Linuxban (UNIX-ban) minden elindított programnak az induláskor három különböző ki-bemeneti (I/O) csatornája létezik: a 0 számmal, a C nyelvben stdin-nel, a C++ nyelvben cin-nel jelölt *sztenderd bemenet*, az 1 számmal, a C nyelvben stdout-tal, a C++ nyelvben cout-tal jelölt *sztenderd kimenet* és a 2 számmal, a C nyelvben stderr-rel, a C++ nyelvben cerr-rel jelölt *hiba kimenet*. Ezeket a csatornákat át lehet irányítani. A sztenderd kimenetet átirányíthatjuk egy fájlba a parancs után > jelet végül a fájl nevét megadva:

```
bercin@users:~$ ls /bin/c*
/bin/cat  /bin/chacl  /bin/chgrp  /bin/chmod  /bin/chown  /bin/cp  /bin/cpio
bercin@users:~$ ls /bin/c* > kimenet
bercin@users:~$
```

A > speciális karakter az `ls` parancs kimenetét a `kimenet` fájlba irányította át (létrehozva a fájlt, ha az addig nem létezett), emiatt nem látszik a parancs eredménye a képernyőn. Egy fájl tartalmát a `cat` paranccsal írathatjuk ki. Írassuk ki a `kimenet` fájl tartalmát:

```
bercin@users:~$ cat kimenet
/bin/cat
/bin/chacl
/bin/chgrp
/bin/chmod
/bin/chown
/bin/cp
/bin/cpio
bercin@users:~$
```

Egy parancs nem csak a sztenderd kimenetére írhat ki üzeneteket, hanem a hiba kimenetén is megjeleníthet szöveget: ehhez a következő példában a c-vel kezdődő fájlokat és a `asadadads` nevű fájlt is megpróbáljuk kilistázni, de az utóbbi nem létezik:

```
bercin@users:~$ ls asadadads /bin/c*
ls: cannot access asadadads: No such file or directory
/bin/cat  /bin/chacl  /bin/chgrp  /bin/chmod  /bin/chown  /bin/cp  /bin/cpio
bercin@users:~$
```

A hiba kimenet átirányítására a 2> karaktereket kell használnunk:

```
bercin@users:~$ ls asadadads /bin/c* 2> kimenet
/bin/cat  /bin/chacl  /bin/chgrp  /bin/chmod  /bin/chown  /bin/cp  /bin/cpio
bercin@users:~$ cat kimenet
ls: cannot access asadadads: No such file or directory
bercin@users:~$
```

Jól látható, hogy a sztenderd kimenet (átirányítás hiányában) továbbra is megjelent a képernyőn, de a hibaüzenetet már a fájlban tároltuk el. Ha mind a hiba, mind a sztenderd kimenetet át akarjuk irányítani, akkor az előzőek természetesen egymás után is alkalmazhatók:

```
bercin@users:~$ ls asadadads /bin/c* > kimenet.sima 2> kimenet.hiba
bercin@users:~$ cat kimenet.sima
/bin/cat
/bin/chacl
/bin/chgrp
/bin/chmod
/bin/chown
/bin/cp
/bin/cpio
bercin@users:~$ cat kimenet.hiba
ls: cannot access asadadads: No such file or directory
bercin@users:~$
```

Ha a két kimenetet ugyanabba a fájlba akarjuk átirányítani akkor arra a következő módszert kell használni:

```
bercin@users:~$ ls asadadads /bin/c* > kimenet 2>&1
bercin@users:~$ cat kimenet
ls: cannot access asadadads: No such file or directory
/bin/cat
/bin/chacl
/bin/chgrp
/bin/chmod
/bin/chown
/bin/cp
/bin/cpio
bercin@users:~$
```

A `2>&1` azt jelenti, hogy a sztenderd error kimenetet (`2`) ugyanoda akarjuk átirányítani, ahova a sztenderd kimenet (`1`) aktuálisan irányítva van a feldolgozás pillanatában.

1.6.0.2. feladat. Mi a különbség az alábbiak között?
`parancs 2>&1 > fajlnev`
`parancs > fajlnev 2>&1`
Próbáljuk ki, és értelmezzük az eredményt!

Az előzőekben a kimenetet úgy irányítottuk át, hogy ha a fájl nem létezett, akkor azt a shell létrehozta, ha létezett, akkor törölte annak tartalmát, és ezután írta bele az átirányítás eredményét. Ha arra van szükségünk, hogy a fájl tartalmát ne törölje, hanem a már meglévő fájl végéhez írja hozzá az átirányítás tartalmát, akkor a `>>` karaktereket kell használni:

```
bercin@users:~$ ls /bin/cp* > kimenet
bercin@users:~$ cat kimenet
/bin/cp
/bin/cpio
bercin@users:~$ ls /bin/ch* > kimenet
bercin@users:~$ cat kimenet
/bin/chacl
/bin/chgrp
/bin/chmod
/bin/chown
bercin@users:~$ ls /bin/ca* >> kimenet
bercin@users:~$ cat kimenet
/bin/chacl
/bin/chgrp
/bin/chmod
/bin/chown
/bin/cat
bercin@users:~$
```

2. fejezet

Számok és karakterek ábrázolása

2.1. Számrendszerek

A számrendszer [numeral system - nem numeric system!] a szám (mint matematikai fogalom) írott formában történő megjelenítésére alkalmas módszer. Ebben a részben a helyiértéken (pozíción) alapuló számrendszereket tárgyaljuk. Léteznek nem pozíción alapuló számrendszerek is, ilyenek például a sorrendiségen alapuló római számok, de ezekkel a továbbiakban nem foglalkozunk.

2.1.1. A számrendszer alapja és a számjegyek

A helyiértéken alapuló számrendszerek két legfontosabb paramétere *a számrendszer alapja* [base, radix] és *az egyes pozíciókba írható számjegyek* [digit]. Ezek nem függetlenek: a számrendszer alapja meghatározza az egyes pozíciókba írható számjegyek maximumát: ha a számrendszer A alapú, akkor a legkisebb felhasználható számjegy a 0, a legnagyobb az $A-1$.

2.1.1.1. példa. A tízes számrendszerben a $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$ számjegyek szerepelhetnek, a nyolcas számrendszerben a $0, 1, 2, 3, 4, 5, 6, 7$ számjegyek közül választhatunk, míg a kettesben a $0, 1$ a két lehetséges számjegy.

Tíznél nagyobb alapú számrendszerek esetében a számjegyek halmazát 9 után az ABC betűivel egészítjük ki. A kis és nagybetűk között általában nem teszünk különbséget, bár egyes nagy alapú számrendszereknél erre mégis szükség lehet.

2.1.1.2. példa. A tizenhatos számrendszerben használható „számjegyek”: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f (vagy 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F).

Ha az a szövegkörnyezetből nem egyértelmű, a számrendszer alapját szögletes zárójelben a jobb alsó indexbe téve jelölhetjük. Például: $5221_{[10]}$, $726_{[8]}$ vagy $80_{[16]}$.

A jól ismert tízes alapú *decimális* számrendszeren kívül az informatikában a leggyakrabban használtak a következők: a kettes alapú *bináris*, a nyolcas alapú *oktális* és a tizenhatos alapú *hexadecimális*. Az előzőekben említett, indexben történő számrendszer megadás mellett bináris számrendszer jelölésére használatos a b postfix, oktális esetben egy kezdő 0 szerepeltetése, hexadecimális számok esetén a 0x, 0X prefixek vagy a h postfix. Az informatikában ezeket a jelöléseket használjuk a leginkább. Például: 100b (bináris), 065 (oktális), 0x243 (hexadecimális), 0X331 (hexadecimális), 22h (hexadecimális). Ha sem a szám előtt, sem utána, sem az indexében nincs jelölve, akkor decimális számrendszerben értelmezzük a leírtakat.

2.1.2. Alaki- és helyiérték

Egy adott számrendszerben leírt szám esetében egy *számjegy értéke* egyenlő a számjegy *alaki értéké*nek és *helyiérték*ének szorzatával. A számjegy alaki értéke a számjegyhez tartozó érték, a helyiérték pedig a számrendszer alapjának a pozíció szerinti hatványa. A $0, 1, \ldots, 9$ esetében az alaki érték egyértelmű, a betűkkel kiegészített esetben ezek: a=10, b=11, c=12, d=13 stb.

[0] $Revision : 46$ ($Date : 2013-09-1923:12:40+0200(Thu, 19Sep2013)$)

2.1.2.1. példa. A tízes számrendszerben felírt 32 szám esetében a 3 helyiértéke $10^1 = 10$, mivel az jobbról a második pozíción szerepel (és a helyiértékeket a nulladik hatványtól indítjuk), így ebben a példában a 3 számjegy értéke: $3 \cdot 10^1 = 3 \cdot 10 = 30$.

2.1.2.2. példa. A tízes számrendszerben felírt 32 szám esetében a 2 helyiértéke $10^0 = 1$, mivel az jobbról az első pozíción szerepel (és a helyiértékeket a nulladik hatványtól indítjuk), így ebben a példában a 2 számjegy értéke: $2 \cdot 10^0 = 2 \cdot 1 = 2$.

2.1.3. Egész számok leírása

Egész számokat általános esetben az $a_n a_{n-1} \dots a_1 a_0$ alakban írhatunk fel, és az így felírt szám értéke (A alapú számrendszert feltételezve):

$$(a_n \cdot A^n) + (a_{n-1} \cdot A^{n-1}) + \dots + (a_1 \cdot A^1) + (a_0 \cdot A^0)$$

ami nem más, mint a leírt számjegyek (az előzőekben megismert módon kiszámolt) értékeinek összege.

2.1.3.1. példa. Triviális példa: $405_{[10]} = 4 \cdot 10^2 + 0 \cdot 10^1 + 5 \cdot 10^0 = 400 + 5$

2.1.3.2. példa. $405_{[8]} = 4 \cdot 8^2 + 0 \cdot 8^1 + 5 \cdot 8^0 = 256 + 5 = 261$

2.1.3.3. példa. $1001101_{[2]} = 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 64 + 8 + 4 + 1 = 77$

2.1.3.4. példa. $0\text{xA3} = 10 \cdot 16^1 + 3 \cdot 16^0 = 10 \cdot 16 + 3 \cdot 1 = 163$

A negatív egész számokat úgy írjuk le, hogy abszolút értéküket az előző módon felírjuk valamely számrendszerben, majd elé $-$ jelet teszünk (bár ezt a jelölést a tízes számrendszeren kívül a gyakorlatban nem alkalmazzuk).

2.1.4. Nem egész számok leírása

Az egész számoknál megismert felírási módszert kiterjeszthetjük úgy, hogy a helyiértékek megadásánál nem állunk meg a nulladik hatványnál, hanem folytatjuk azt a negatív hatványokra is, így lehetőségünk adódik nem egész számok leírására. Általános esetben tehát ennek alakja: $a_n a_{n-1} \dots a_1 a_0 a_{-1} \dots a_{-k}$, és az így felírt szám értéke (A alapú számrendszert feltételezve):

$$a_n \cdot A^n + a_{n-1} \cdot A^{n-1} + \dots + a_1 \cdot A^1 + a_0 \cdot A^0 + a_{-1} \cdot A^{-1} + \dots + a_{-k} \cdot A^{-k}$$

Annak érdekében, hogy a mindkét végén (egész- illetve tört rész) tetszőlegesen bővíthető felírás egyértelmű legyen, ennek a két résznek a határát jelöljük tizedesvesszővel. Mi a magyar helyesírással ellentétben, a nem egész számok felsorolásának könnyebb olvashatósága érdekében a továbbiakban a tizedespontos[1] jelölést fogjuk alkalmazni. (Pl. 1,6, 2,4, 5,9 helyett $1.6, 2.4, 5.9$)

2.1.4.1. példa. Triviális példa: $405.23_{[10]} = 4 \cdot 10^2 + 0 \cdot 10^1 + 5 \cdot 10^0 + 2 \cdot 10^{-1} + 3 \cdot 10^{-2} = 4 \cdot 100 + 5 \cdot 1 + 2 \cdot \frac{1}{10} + 3 \cdot \frac{1}{100}$

2.1.4.2. példa. $405.23_{[8]} = 4 \cdot 8^2 + 0 \cdot 8^1 + 5 \cdot 8^0 + 2 \cdot 8^{-1} + 3 \cdot 8^{-2} = 4 \cdot 64 + 5 \cdot 1 + 2 \cdot \frac{1}{8} + 3 \cdot \frac{1}{8^2} = 256 + 5 + \frac{2}{8} + \frac{3}{64} = 261\frac{19}{64} = 261.296875$

2.1.4.3. példa. $1001101.01_{[2]} = 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 64 + 8 + 4 + 1 + \frac{1}{4} = 77.25$

Negatív nem egész számok leírása a negatív egész számok leírásához hasonlóan a $-$ jel szám elé írásával történik (amit szintén csak a tízes számrendszer esetében használunk).

[1] Ha nagyon pontosak akarunk lenni, akkor tizedespontról csak a tízes számrendszer használata esetén beszélhetnénk, bináris esetben inkább bináris pontról van szó (és hasonlóan oktális, hexadecimális stb. esetben).

2.1.5. Átváltás számrendszerek között

Az adott számrendszerből tízes számrendszerbe váltást a 2.1.3 és a 2.1.4 részek példáiban hallgatólagosan már bemutattuk. A fordított átváltásra nem térünk ki (a módszer könnyen kitalálható, lásd 2.1.6.5. feladat).

Az átváltás nagymértékben egyszerűsödik, ha binárisból oktális vagy hexadecimális számrendszerbe kell átváltani: egyszerűen hármasával (oktális esetben) vagy négyesével (hexadecimális esetben) kell a bináris számjegyeket csoportosítani, és az így képzett csoportokat átváltani:

2.1.5.1. példa. $1010111001_{[2]} = 001\ 010\ 111\ 001_{[2]} = 1271_{[8]}$

Az átváltás fordított irányban is hasonlóan egyszerű: az egyes oktális vagy hexadecimális számjegyeket kell átváltani és az így kapott hármas illetve négyes bináris csoportokat egymás után írni:

2.1.5.2. példa. $2b9_{[16]} = 0010\ 1011\ 1001_{[2]} = 1010111001_{[2]}$

Oktálisból hexadecimálisba vagy decimálisból hexadecimálisba illetve fordítva a bináris számrendszert közbeiktatva is átválthatunk ezzel a módszerrel:

2.1.5.3. példa. $2b9_{[16]} = 0010\ 1011\ 1001_{[2]} = 1010111001_{[2]} = 001\ 010\ 111\ 001_{[2]} = 1271_{[8]}$

2.1.6. Feladatok

2.1.6.1. feladat. $1010111001_{[2]} = ?_{[8]} = ?_{[16]}$

2.1.6.2. feladat. $54_{[8]} = ?_{[16]}$

2.1.6.3. feladat. $962_{[10]} = ?_{[8]} = ?_{[16]}$

2.1.6.4. feladat. $9a2d_{[16]} = ?_{[2]} = ?_{[8]} = ?_{[16]}$

2.1.6.5. feladat. Adjunk algoritmust (módszert) decimálisból a) oktális-, b) hexadecimális számrendszerbe történő közvetlen (tehát nem a bináris számrendszer közbeiktatásával történő) átváltásra!

2.1.6.6. feladat. Minden racionális szám (tört) leírható bármilyen alapú számrendszerben véges számjegy felhasználásával?

A http://www.digitconvert.com/ oldalon kipróbálhatók, ellenőrizhetők az átváltások.

2.1.7. Számrendszerek pontossága

Fontos kiemelni, hogy nem egész számok felírása esetén nem biztos, hogy a szám pontosan leírható véges számjeggyel! Sőt, egy konkrét nem egész szám ábrázolásának pontossága függ a számrendszer alapjától: például az $\frac{1}{3}$ tízes számrendszerben nem írható fel véges számjeggyel, ugyanakkor hármas számrendszerben pontosan felírható: $\frac{1}{3} = 0.1_{[3]} = 0.33333\ldots_{[10]}$

2.1.7.1. feladat. Adjunk meg néhány példát arra, amikor az egyik számrendszerben véges számjeggyel felírható szám a másik számrendszerben nem írható fel véges számjeggyel!

2.1.7.2. feladat. a) Adjunk meg néhány példát olyan számra, ami egyetlen számrendszerben sem írható fel véges számjeggyel! b) Felírhatók ezek a számok tört alakban? c) Milyen számhalmazt alkotnak ezek a számok?

2.1.7.3. feladat. Kiválasztható olyan alapú számrendszer, amiben minden racionális szám pontosan ábrázolható véges hosszú karaktersorozattal? Indokoljuk meg!

2.2. Mértékegységek

Az informatikában használatos legkisebb egység a bit [bit] (sok esetben b-vel rövidítik, de a legfrissebb szabvány[2] a rövidítés nélküli formát ajánlja, ami kézenfekvő a számrendszerek részben tárgyaltak miatt, hiszen a b postfix a bináris számrendszert jelöli). Értéke 0 vagy 1 lehet. Használhatjuk tárolókapacitás vagy információmennyiség jelölésére. Az utóbbi egy felsőbb éves tárgy, az *Információ és kódelmélet* témája, mi itt csak a tárolási vonatkozásával foglalkozunk.

A bájt [byte] az informatika másik legfontosabb egysége, jele: B. Mi az általánosan elfogadott, a gyakorlatban majdnem kizárólagosan használt 1 B = 8 bit átváltást használjuk, bár egyes (egzotikus) architektúrák esetében ennél több vagy kevesebb bit is alkothat egy bájtot.

Az SI mértékegységrendszerben használatos k (kilo), M (mega), G (giga), T (tera), P (peta) stb. prefixek mellett a bit és a bájt esetében használatosak a Ki (kibi), Mi (mebi), Gi (gibi), Ti (tebi), Pi (pebi) stb. *bináris prefixek* is (lásd a 2.1. ábrán).

Fontos kiemelni, hogy az egyre nagyobb prefixek esetében egyre nagyobb a különbség az SI és a bináris prefixek között. Például a G (1000^3) és Gi (1024^3) között a különbség kb. 7%, a T (1000^4) és Ti (1024^4) között már kb. 10%.[3]

A kapcsolat a prefixek és a számrendszerek között ott fedezhető fel, hogy a használt prefixek mindig a számrendszer alapja valamely hatványának hatványai. Az SI esetben ez a tíz harmadik hatványa (illetve ennek további hatványai), de ugyanez igaz a bináris prefixekre is, amikor is ez a kettő tizedik hatványa (illetve ennek további hatványai).

SI		bináris	
prefix	szorzó	prefix	szorzó
k (kilo)	1000	Ki (kibi)	1024
M (mega)	1000^2	Mi (mebi)	1024^2
G (giga)	1000^3	Gi (gibi)	1024^3
T (tera)	1000^4	Ti (tebi)	1024^4
P (peta)	1000^5	Pi (pebi)	1024^5

2.1. ábra. SI és bináris prefixek

2.3. Gépi számábrázolás

A gépi számábrázolás a számok (számító)gépek memóriájában vagy egyéb egységében történő tárolását vagy valamely adathálózaton történő továbbítás formátumát adja meg.

2.3.1. Nem negatív egész számok ábrázolása

Egy nem negatív (előjel nélküli) egész szám [unsigned integer] ábrázolása megegyezik a bináris számrendszernél megismert leírással, azaz egy nem negatív egész számot a kettes számrendszerbe átváltott formájában tárolunk. A tömörebb írásmód miatt ugyanakkor ezt legtöbbször nem bináris, hanem hexadecimális formában írjuk le. (Ne feledjük, hogy a binárisból hexadecimálisba váltás nem más, mint négy bitesével csoportosítás, ahogy azt az előzőekben láthattuk.)

A kapott értékeket általában valamilyen fix hosszon tároljuk (a nem használt helyiértékekre nullát írunk), ami a gyakorlatban kizárólag egész byte méretű ábrázolást jelent. Így az előjel nélküli egészek is legtöbbször 1, 2, 4, 8, ... byte (8, 16, 32, 64, ... bit) hosszúak lehetnek. Így is hívjuk ezeket: 8 bites előjel nélküli egész, 16 bites előjel nélküli egész stb.

2.3.1.1. példa. A $46_{[10]}$ számot a memóriában a következőképpen tároljuk 1 bájton: 00101110 (=0x2E).

[2] ISO/IEC 80000, Part 13 - Information science and technology

[3] Különösen fontos ez a háttértárak esetében, ahol a gyártók inkább az SI prefixeket használják, mert így egy 1000000000000 B méretű lemezegység esetében 1 TB-ot tüntethetnek fel, míg ugyanez a bináris prefixekkel csupán 0.9 TiB

2.3.1.2. feladat. Az összeadás művelet hogyan végezhető el az előjel nélküli egész számok bináris tárolása esetén? Adjunk erre módszert (algoritmust)!

2.3.1.3. feladat. Hogyan dönthető el két előjel nélküli egész számról, hogy melyik a nagyobb? Adjunk rá algoritmust!

2.3.2. Negatív egész számok ábrázolása

Ebben a részben a negatív egészek ábrázolásának változatait tekintjük át.

Előjelbites ábrázolás

A legegyszerűbb módszer az előjeles egészek ábrázolására, ha az előjel nélküli egészek ábrázolásához egy előjelet jelentő bitet adunk (ami 0, ha pozitív az előjel és 1, ha negatív az előjel) és az ábrázolásból fennmaradó többi biten tároljuk a szám abszolút értékét az előzőekben tárgyaltak szerint.

2.3.2.1. példa. A -32 előjelbites ábrázolása 8 biten (1 bit előjel + 7 bit érték): 10100000

2.3.2.2. példa. A 18 előjelbites ábrázolása 8 biten (1 bit előjel + 7 bit érték): 00010010

Ez a megoldás sok szempontból nem megfelelő: a legkézenfekvőbb probléma, hogy ezzel a módszerrel lehetséges a $+0$ és a -0 ábrázolása is (8 biten ezek a következők: $+0$ = 00000000, -0 = 10000000), ami zavarhoz vezet (például a „nulla-e” vizsgálatot így két különböző értékre kell megtenni), továbbá az ilyen módon felírt számokkal végzett műveletek bonyolultabbak, mint amennyire az feltétlenül szükséges lenne.

2.3.2.3. feladat. A 2.3.1.2. feladatban kitalált összeadás művelet elvégezhető-e *módosítás nélkül* az előjelbites számábrázolási módszer használatával? Adjunk meg egy példát!

2.3.2.4. feladat. Módosítsuk a 2.3.1.3. feladatban kitalált algoritmust, hogy az két előjeles szám közül is ki tudja választani a nagyobbikat!

Kettes komplemens ábrázolás

Sokkal jobb eredményre vezet a *kettes komplemens* ábrázolás: ahelyett, hogy egy előjelbittel jelölnénk az előjelet, a következő módon járunk el: a negatív számhoz egyet hozzáadunk, az eredmény abszolút értékét binárisan ábrázoljuk a megadott számú biten (az előzőekben tárgyaltak szerint, mivel ez nem negatív), végül az így kapott számjegyeket invertáljuk. Ebből a számítási módból következik az ábrázolás neve: kettes komplemens.

A kettes komplemens számábrázolási módszert *előjeles egész* [signed integer] számábrázolásnak nevezzük.

2.3.2.5. példa. A -2 kettes komplemens ábrázolása 8 biten: $-2+1 = -1$ ennek abszolút értéke: 1, ábrázolva: 00000001, invertálva: 11111110.

2.3.2.6. példa. A -19 kettes komplemens ábrázolása 8 biten: $-19 + 1 = -18$ ennek abszolút értéke: 18, ábrázolva: 00010010, invertálva: 11101101.

Fontos tudnivalók:

- Kettes komplemens ábrázolásban is lehetséges nem negatív számok ábrázolása, aminek módja megegyezik az előjel nélküli egészek tárolási módjával. (Azaz ebben az esetben nem kell az előzőekben ismertetett műveleteket elvégezni.)
- A kettes komplemens ábrázolásban már csak egyetlen ábrázolási módja van a nullának.
- Az esetek túlnyomó többségében a gépi számábrázolás során az előjeles egészek ábrázolására a kettes komplemens ábrázolást használjuk.

2.3.2.7. feladat. Adjuk meg a 0 kettes komplemens ábrázolását 8, 16, 32, 64 biten!

2.3.2.8. feladat. Adjuk meg a -1 kettes komplemens ábrázolását 8, 16, 32, 64 biten!

2.3.2.9. feladat. Adjuk meg az 1 kettes komplemens ábrázolását 8 biten!

2.3.2.10. feladat. A 2.3.1.2. feladatban kitalált összeadás művelet elvégezhető-e *módosítás nélkül* a kettes komplemens számábrázolási módszer használatával? Adjuk össze az előző két feladatban kiszámolt, 8 bites -1 és 1 értéket, és ellenőrizzük, hogy nullát kaptunk-e!

2.3.2.11. feladat. Két, kettes komplemens módon ábrázolt számról hogyan dönthető el, hogy melyik a nagyobb? Alkalmazható *módosítás nélkül* ugyanaz az algoritmus, mint a 2.3.1.3 feladatban?

Eltolt ábrázolás

Soroljuk fel egy listában az n biten történő előjel nélküli számábrázolással felírható értékeket növekvő sorrendben. Az *eltolt* [excess] számábrázolási módszer ezeket az eltolás mértékében lefelé tolja úgy, hogy az újonnan belépő elemek az érték szerint csökkenő negatív számok legyenek (lásd 2.2. ábra).

tárolt adat	adat értelmezése		
3 biten	előjel nélküli egész	excess-2	excess-4
000	0	-2	-4
001	1	-1	-3
010	2	0	-2
011	3	1	-1
100	4	2	0
101	5	3	1
110	6	4	2
111	7	5	3

2.2. ábra. A 3 biten tárolható értékek előjel nélküli egész és excess-2 illetve excess-4 szerinti értelmezése

2.3.2.12. feladat. Létezik olyan excess ábrázolás, ami a negatív számok esetében megegyezik a kettes komplemens ábrázolással?

A lista eltolása helyett az ábrázolandó értékeket úgy is megkaphatjuk, hogy az ábrázolandó számhoz hozzáadjuk az eltolás mértékét, és az eredményül kapott számot ábrázoljuk az előjel nélküli egész számábrázolási módszere szerint.

2.3.2.13. feladat. Mi biztosítja, hogy az előző módszer működik? (Mi garantálja, hogy nem negatív számot kapunk, ha az ábrázolandó számhoz hozzáadjuk az eltolás mértékét?) Ha mégsem működik, az mit jelent?

2.3.2.14. példa. A -2 excess-2 ábrázolása 3 biten: $-2 + 2 = 0$, tehát ábrázolandó a 0 a nem negatív egészek ábrázolása szerint: 000 (lásd 2.2. ábra első sora).

2.3.2.15. példa. A -3 excess-4 ábrázolása 3 biten: $-3 + 4 = 1$, tehát ábrázolandó az 1 a nem negatív egészek ábrázolása szerint: 001 (lásd 2.2. ábra második sora).

2.3.2.16. példa. Az 5 excess-2 ábrázolása 3 biten: $5 + 2 = 7$, tehát ábrázolandó a 7 a nem negatív egészek ábrázolása szerint: 111 (lásd 2.2. ábra utolsó sora).

2.3.3. Egész számok adatábrázolásainak összehasonlítása

2.3.3.1. feladat. Hasonlítsuk össze az előzőekben ismertetett, negatív számok ábrázolására is alkalmas módszereket az alábbi szempontok alapján:

- Az összeadás művelet elvégezhető ugyanúgy, mint a nem negatív egészek ábrázolásánál?
- Két ábrázolt szám esetében a kisebb/nagyobb eldöntése (rendezés) elvégezhető ugyanúgy, mint a nem negatív egészeknél?
- Hányféleképpen ábrázolható a nulla?
- Hogyan végezhető el az invertálás (diszkrét matematikai nyelven az additív inverz számítása)?
- Hogyan végezhető el a kivonás művelet?

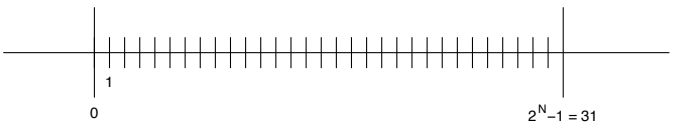
2.3.3.2. feladat. Hasonlítsuk össze a 8 bites számábrázolások esetén az előjel nélküli egész, az előjelbites egész, a kettes komplemens, a 127-tel eltolt, a 255-tel eltolt és a 256-tal eltolt számábrázolásokat! (Táblázatosan foglaljuk össze: egy sor legyen a tárolt 8 bit, az oszlopok legyenek a vizsgált ábrázolási módok, egy adott mezőbe írjuk be a mező sorának megfelelő bitsorozat értelmezését az oszlopnak megfelelő számábrázolás esetében, hasonlóan a 2.2. ábrához!)

2.3.4. Egész számok ábrázolási határai és pontossága

Előjel nélküli egész tárolás ábrázolási határai és pontossága

Az N biten történő, előjel nélküli egész számábrázolás esetén a tárolható legkisebb érték: 0, a tárolható legnagyobb érték: $2^N - 1$.

Előjel nélküli egész számábrázolás esetében a tárolás pontos, hiszen csak egész számokat kell tárolni, és a határokon belül minden egész szám pontosan tárolható. Ebből adódóan az ábrázolási intervallumot az ábrázolható számok egyenletesen töltik ki (lásd a 2.3. ábrán).



2.3. ábra. 5 bites előjel nélküli egész számábrázolás esetén az ábrázolási intervallum és az ezen belül ábrázolható számok.

2.3.4.1. példa. Ha 8 bites előjel nélküli egész ábrázolást használunk, akkor a legkisebb ábrázolható szám a 00000000 (értéke 0), a legnagyobb ábrázolható szám az 11111111 (értéke 255).

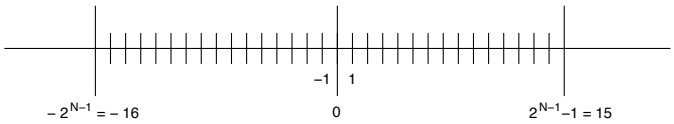
2.3.4.2. feladat. Mennyi a legnagyobb tárolható érték 8, 16, 32, 64 bites előjel nélküli egész esetében?

2.3.4.3. feladat. Összesen hány különböző érték tárolható 8, 16, 32, 64 biten, előjel nélküli egész számábrázolás esetében?

Kettes komplemens tárolás ábrázolási határai és pontossága

Ha kettes komplemens módon ábrázolunk egy egész számot és ehhez N bit áll rendelkezésre, akkor a tárolható legkisebb érték: -2^{N-1}, a tárolható legnagyobb érték: $2^{N-1} - 1$. Kettes komplemens számábrázolás esetében a tárolás pontos, hiszen csak egész számokat kell tárolni, és a határokon belül minden egész szám pontosan tárolható. Ebből adódóan az ábrázolási intervallumot az ábrázolható számok egyenletesen töltik ki (lásd a 2.4. ábrán).

2.3.4.4. példa. Ha 8 bites kettes komplemens ábrázolást használunk, akkor a legkisebb ábrázolható szám az 10000000 (értéke -128), a legnagyobb ábrázolható szám a 01111111 (értéke 127).



2.4. ábra. 5 bites kettes komplemens számábrázolás esetén
az ábrázolási intervallum és az ezen belül ábrázolható számok.

2.3.4.5. feladat. Kettes komplemens ábrázolás esetén miért nem ugyanannyi szám tárolható a pozitív és a negatív tartományban? (Azaz miért nem -127 és 127 vagy -128 és 128 a két határ?)

2.3.4.6. feladat. Mennyi az értéke a kettes komplemens ábrázolással, 8 biten tárolt 11111111 illetve a 00000000 számoknak?

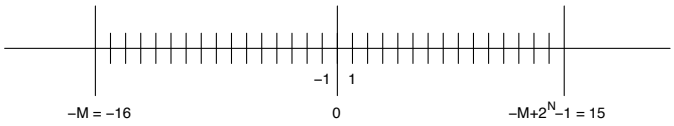
2.3.4.7. feladat. Eldönthető egyszerűen (ránézésre) egy kettes komplemens módon ábrázolt számról, hogy az negatív vagy pozitív?

2.3.4.8. feladat. Összesen hány különböző érték tárolható 8, 16, 32, 64 biten, kettes komplemens számábrázolás esetében?

2.3.4.9. feladat. Mi a kapcsolat a 2.3.4.3. feladat és a 2.3.4.8. feladatban kapott eredmények között?

Eltolt tárolás ábrázolási határai és pontossága

Az N biten történő eltolt-M ábrázolás esetén a legkisebb ábrázolható szám a $-M$, a legnagyobb ábrázolható szám a $-M + 2^N - 1$. Eltolt számábrázolás esetében a tárolás pontos, hiszen csak egész számokat kell tárolni, és a határokon belül minden egész szám pontosan tárolható. Ebből adódóan az ábrázolási intervallumot az ábrázolható számok egyenletesen töltik ki (lásd a 2.5. ábrán).



2.5. ábra. 6 bites excess-16 számábrázolás esetén
az ábrázolási intervallum és az ezen belül ábrázolható számok.

Túlcsordulás

Az egész számok véges biten történő ábrázolása miatt mindig van legkisebb és legnagyobb ábrázolható szám. Amikor műveletet végzünk, elképzelhető, hogy a művelet eredménye már nem ábrázolható az operandusokkal megegyező méretben. Ezt a jelenséget túlcsordulásnak [overflow] nevezzük. Túlcsordulás tehát lehetséges pozitív és negatív irányban is! Figyelem, az alulcsordulás (lásd a 2.3.5. részt) *nem* a negatív irányban történő túlcsordulást jelenti! Könnyebben megjegyezhető, ha úgy tekintünk a túlcsordulásra, hogy a szám abszolút értéke túl nagy és emiatt nem ábrázolható.

Túlcsordulás esetén – megvalósítástól függően – lehetséges

- *levágás*: a túlcsordult eredmény még ábrázolható részét tároljuk, a nem ábrázolható részt egyszerűen „elfelejtjük”. [4] A legtöbb architektúra így működik.
- *szaturáció*: a túlcsordult eredmény helyett a legnagyobb illetve legkisebb ábrázolható értéket tároljuk.

[4] Például mechanikus gázóránál vagy régebbi autók kilométer számlálójánál figyelhető meg ilyen jelenség, mert fix számú helyiértéken történik a mérés. A kilométer számlálók tekintetében ezt a tulajdonságot kihasználva tekerik körbe egyes nepperek az órát, hogy a kocsi kevesebbet futottnak tűnjön.

2.3.4.10. példa. Túlcsordulás pozitív irányban: ha 8 bites előjel nélküli egészekkel dolgozunk, a 156+172=328 összeget már nem tudjuk 8 biten tárolni (mert a legnagyobb tárolható érték a 255).

2.3.4.11. példa. Túlcsordulás negatív irányban: ha 8 bites előjeles egészekkel dolgozunk, a -84+(-79)=-163 összeget már nem tudjuk 8 biten tárolni (mert a legkisebb tárolható érték a -127).

2.3.4.12. példa. Levágás: ha 8 bites előjel nélküli egészekkel dolgozunk, a $156_{[10]} = 10011100_{[2]}$ és a $172_{[10]} = 10101100_{[2]}$ valódi összege ($328_{[10]} = 101001000_{[2]}$) helyett annak a 8 utolsó bitjét tároljuk: 01001000.

2.3.4.13. példa. Szaturáció: ha 8 bites előjel nélküli egészekkel dolgozunk, a $156_{[10]} = 10011100_{[2]}$ és a $172_{[10]} = 10101100_{[2]}$ valódi összege ($328_{[10]} = 101001000_{[2]}$) helyett az ábrázolható legnagyobb számot tároljuk: 11111111.

2.3.4.14. feladat. Mi (volt) az Y2K probléma? Mi a kapcsolat a túlcsordulás és az Y2K probléma között?

2.3.5. A lebegőpontos számábrázolás

Nem egész számok gépi ábrázolására a lebegőpontos [floating point] ábrázolást használjuk: a számot először átalakítjuk normalizált alakba, és az így kapott alak különböző részeit különkülön tároljuk.

Normalizált alak

Egy szám normalizált alakján olyan szorzatra bontását értjük (lásd 2.6. ábra), ahol a második tag a számrendszer alapjának valamely hatványa (amit a szám nagyságrendjének is nevezünk), az első tag értéke pedig annyi, hogy a második taggal megszorozva az eredeti számot kapjuk. További feltétel, hogy az első tag egyetlen nem nulla számjegyet tartalmazzon a tizedespont előtt, ami garantálja, hogy a normalizált alakban történő felírás egyértelmű legyen.

2.3.5.1. példa. Tízes számrendszerben a 380 normalizált alakja: $3.8 \cdot 10^2$, a 3.875 normalizált alakja $3.875 \cdot 10^0$, a 0.00000651 normalizált alakja $6.51 \cdot 10^{-6}$, a -53.75 normalizált alakja: $-5.375 \cdot 10^1$.

Az előzőekben definiált első tagot a szám *mantisszájának* [mantissa, significand, coefficient], a hatványkitevőt (a nagyságrendet) a szám *karakterisztikájának* vagy *exponens*ének [exponent] nevezzük. Negatív számok tárolásához szükség van még az előjelre is (lásd 2.6. ábra).

A lebegőpontos elnevezés abból adódik, hogy az ábrázolható számok nem fix helyiértékű tizedesjegyekkel kerülnek tárolásra[5], hanem az exponens alapján a mantissza tizedespontja változik („lebeg").

2.6. ábra. A -37.75 normalizált alakja tízes számrendszerben és ennek elemei (bekarikázva).

2.3.5.2. feladat. Adjunk arra példát, hogy az „első tag egyetlen nem nulla számjegyet tartalmazzon a tizedespont előtt" feltétel hiányában egy számot többféleképpen is fel lehet írni normalizált alakban!

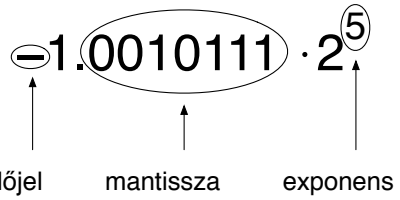
2.3.5.3. feladat. Adjuk meg a nulla normalizált alakját!

[5] ezt a módot fixpontos ábrázolásnak nevezzük

Bináris normalizált alak

Kettes számrendszerben ábrázolva a számot, a normalizált alak tovább egyszerűsödik, hiszen a mantissza tizedespontja előtt mindig 1 áll („az első tag egyetlen nem nulla számjegyet tartalmazzon a tizedespont előtt" feltétel miatt), amit így nem kell eltárolni (lásd 2.7. ábra), és ezt a megtakarított bitet a mantissza pontosabb tárolására lehet fordítani.

Fontos, hogy a lebegőpontos szám értelmezésekor ezt az el nem tárolt számjegyet is figyelembe vegyük, továbbá, hogy az első (nem tárolt) egyes után már fontosak az azt követő nullák, azaz például a 0010111 mantissza helyett nem tárolhatjuk el az 10111 értéket!



2.7. ábra. A -37.75 normalizált alakja kettes számrendszerben és ennek *tárolandó* elemei (bekarikázva).

Az exponens A normalizálásból adódóan az exponens is lehet pozitív vagy negatív. Ennek tárolásához az eltolt tárolási módszert használjuk.

Az előjel Az előjelet 1 biten tároljuk; ha értéke 1: a szám negatív, ha 0: a szám pozitív.

2.3.5.4. példa. A $380_{[10]} = 101111100_{[2]}$ normalizált alakja $1.011111_{[2]} \cdot 2^8$, azaz tárolandó a 0 előjelbit, a 011111 mantissza és a 8 karakterisztika (a megfelelő eltolással).

2.3.5.5. példa. A $-3.375_{[10]} = -11.011_{[2]}$ normalizált alakja $-1.1011_{[2]} \cdot 2^1$, azaz tárolandó az 1 előjelbit, a 1011 mantissza és az 1 karakterisztika (a megfelelő eltolással).

A lebegőpontos szám elemeinek tárolási mérete

Az előjelet mindig egy biten, a mantisszát és az exponenst megadott számú biten tároljuk. Ha az előzőekben kiszámolt mantissza vagy exponens mérete nem egyezik meg a tárolási mérettel, akkor az exponenst balról, a mantisszát jobbról egészíthetjük ki nullákkal (ha szükséges)!

2.3.5.6. példa. A mantissza 10 biten, az exponens 5 biten történő excess-15 ábrázolása esetén a tízezer ábrázolása: $10000 = 10011100010000_{[2]} = 1.001110001_{[2]} \cdot 2^{13}$, azaz tárolandó a 0 előjel, a 0011100010 mantissza (figyelem, kiegészítettük 10 bites hosszra nullákkal jobbról!) és a 13 exponens, utóbbi az excess-15 ábrázolás miatt 11100 formában.

2.3.5.7. példa. A mantissza 10 biten, az exponens 5 biten történő excess-15 ábrázolása esetén a -0.078125 ábrázolása: $-0.078125 = -0.000101_{[2]} = -1.01_{[2]} \cdot 2^{-4}$, azaz tárolandó az 1 előjel, a 0100000000 mantissza (figyelem, kiegészítettük 10 bites hosszra nullákkal jobbról!) és a -4 exponens, utóbbi az excess-15 ábrázolás miatt 01011 formában (figyelem, kiegészítettük 5 bites hosszra egy nullával balról!).

Lebegőpontos számábrázolás határai és pontossága

Az előjeles vagy előjel nélküli egész és a lebegőpontos számábrázolás esetében is csak fix értékek tárolhatók, de míg ezek az egészek esetében pontosan megegyeznek a tárolni kívánt egész számokkal, a lebegőpontos számábrázolás esetében ez nincs így, mivel bárhogy válasszuk is meg a számábrázolás határait, az ezek között lévő végtelen sok valós szám nyilván nem ábrázolható véges helyen. Ezt úgy is értelmezhetjük, hogy a lebegőpontos tárolás során kényszerű kerekítés történik. Mindezek miatt az ábrázolási határok mellett a pontosság is jellemez egy-egy konkrét lebegőpontos számábrázolást, ami megadja, hogy egy adott szám tárolása esetén a tárolni kívánt

és a tárolt szám értéke legfeljebb milyen távol lehet egymástól. A lebegőpontos számok normalizált alakú tárolásából következik, hogy a pontosságot a mantissza tárolási mérete határozza meg, az ábrázolási határok pedig elsődlegesen a karakterisztika ábrázolási méretéből adódnak. Fontos azt is kiemelni, hogy a pontosság az ábrázolási tartományban abszolút értelemben nem egyenletes, azaz függ az ábrázolni kívánt számtól (lásd a 2.8. ábra):

2.8. ábra. Lebegőpontos számábrázolás határai és a pontosan ábrázolható számok (2 bites mantissza, 3 bites exponens excess-4 módon tárolva).

2.3.5.8. példa. A mantissza 10 biten, az exponens 5 biten történő excess-15 ábrázolása esetén

- a tízezernél nagyobb, pontosan ábrázolható számok közül a legkisebb (lásd a 2.3.5.6. feladatot a 10000 ábrázolásához) a 0011100011 tárolt mantisszájú és a 11100 tárolt exponensű szám: $1.0011100011_{[2]} \cdot 2^{13} = 1.2216796875 \cdot 8192 = 10008$
- a tízezernél kisebb, pontosan ábrázolható számok közül a legnagyobb a 0011100001 tárolt mantisszájú és a 11100 tárolt exponensű szám: $1.0011100001_{[2]} \cdot 2^{13} = 1.2197265625 \cdot 8192 = 9992$

Az előzőekből adódik, hogy tízezer körül a hiba 8 (ami a tízezer 0.08%-a).

2.3.5.9. példa. A mantissza 10 biten, az exponens 5 biten történő excess-15 ábrázolása esetén

- az egy tízezrednél kisebb, pontosan ábrázolható számok közül a legnagyobb a 1010001101 tárolt mantisszájú és 00001 tárolt exponensű szám: $1.1010001101 \cdot 2^{-14} = 1.6376953125 \cdot 0.00006103515625 = 0.00009995698929$
- a tízezrednél nagyobb, pontosan ábrázolható számok közül a legkisebb a 0 előjelű, a 1010001110 tárolt mantisszájú és a 00001 tárolt exponensű szám: $1.1010001110 \cdot 2^{-14} = 1.638671875 \cdot 0.00006103515625 = 0.000100016593933$

Az előzőekből adódik, hogy egy tízezred körül a hiba kevesebb, mint egy tízmilliomod (ami az egy tízezred 0.1%-a).

2.3.5.10. feladat. A mantissza 10 biten, az exponens 5 biten történő excess-15 ábrázolása esetén adjuk meg (az előző két példához hasonló módon) az ezernél, a száznál, a tíznél, az egy tizednél, az egy századnál és az egy ezrednél nagyobb számok közül a legkisebb ábrázolhatót illetve a kisebb számok közül a legnagyobb ábrázolhatót, és számítsuk ki az abszolút, relatív hibát. Hogyan változik a relatív hiba az eltárolt szám függvényében?

A nem pontos tárolás akkor is látható, ha meggondoljuk, hogy a bináris normalizált alak felírásakor nem minden számjegy tárolható el, így azt kénytelenek vagyunk a tárolási méretre csökkenteni.

2.3.5.11. példa. A mantissza 10 biten, az exponens 5 biten történő excess-15 ábrázolása esetén a 10000.125 ábrázolása (lásd a 2.3.5.6. feladatot a 10000 ábrázolásához): $10000.125 = 10011100010000.001 = 1.0011100010000001_{[2]} \cdot 2^{13}$, azaz tárolni kellene a 0 előjelet, a 00111000 10000001 mantisszát és a 13 exponenst, utóbbi az excess-15 ábrázolás miatt 11100 formában. Jól látszik azonban, hogy a mantissza 10 biten történő tárolása miatt a 0011100010000001 első tíz karaktere tárolható csak el, így kényszerű kerekítés történik: a 10000.125 helyett a 10000 kerül tárolásra. (Ezen nem csodálkozhatunk, hiszen a 2.3.5.8 példában láthattuk, hogy 10000 és 10008 között nem tárolható el más szám.)

2.3.5.12. feladat. Mi történik a lebegőpontos szám ábrázolási határaival ill. pontosságával, ha a

- mantissza méretét 1 bittel növelem?
- exponens méretét 1 bittel növelem?

Alulcsordulás

A túlcsorduláshoz (ami pozitív vagy negatív irányban túl nagy szám ábrázolásának kísérletét jelenti, azaz a számábrázolási intervallumból lépünk ki) hasonló az alulcsordulás [underflow]: olyan kis abszolút értékű számot akarunk ábrázolni, ami már nem ábrázolható.

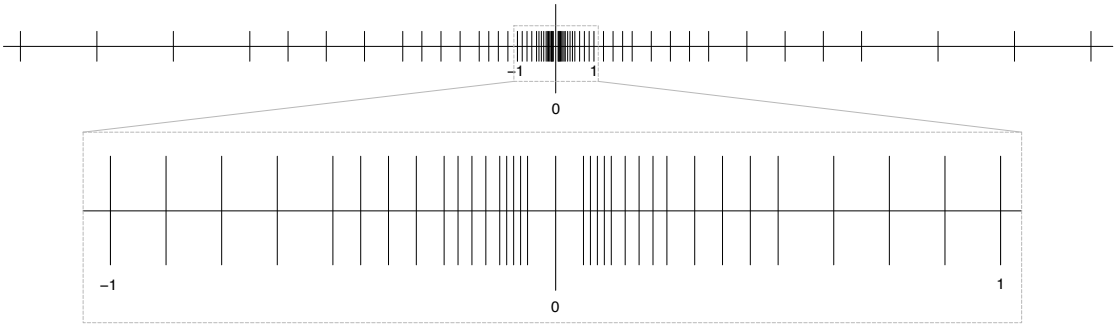
A kettes számrendszerben történő normalizált ábrázolásnak köszönhető megtakarítás (azaz hogy a mantissza első számjegye fixen 1, így a mantisszának csak az 1-től jobbra lévő részét tároljuk el, ezzel 1 bitet megtakarítva) hátrányos hatása itt jelentkezik: a legkisebb tárolható mantissza $1.0\ldots0$, a legkisebb karakterisztika $A^{K_{min}}$ (ahol A jelenti a számrendszer alapját, K_{min} pedig a legkisebb ábrázolható karakterisztikát), a legkisebb tárolható pozitív szám ezek szorzata: $1.0\ldots0 \cdot A^{K_{min}} = A^{K_{min}}$.

2.3.5.13. példa. 10 bites mantissza és 5 bites excess-15 exponens tárolás esetén a legkisebb ábrázolható pozitív szám: $1.0\ldots0 \cdot 2^{-15} = \frac{1}{2^{15}} = 0.000030517578125$.

2.3.5.14. példa. 10 bites mantissza és 5 bites excess-15 exponens tárolás esetén a legnagyobb ábrázolható negatív szám: $-1.0\ldots0 \cdot 2^{-15} = \frac{1}{2^{15}} = -0.000030517578125$.

Az előző két példa eredményéből következik, hogy a nullát sem tudjuk ábrázolni!

Sajnos a legkisebb pozitív és a legnagyobb negatív érték között a szomszédos távolságokhoz képest nagy ábrázolhatatlan tartomány húzódik, amit *alulcsordulási rés*nek nevezünk (lásd a 2.9. ábrán). A kis számok ábrázolhatatlansága mellett sokkal nagyobb probléma az, hogy ha



2.9. ábra. Lebegőpontos számábrázolás esetén
a nulla körüli alulcsordulási rés [underflow gap]
(2 bites mantissza, 3 bites exponens excess-4 módon tárolva).

az eddigiek alapján járnánk el, akkor bármely két ábrázolható lebegőpontos szám különbsége nem biztos, hogy ábrázolható maradna, azaz nullával helyettesítődne. Ez óriási probléma a kis abszolút értékű számokkal dolgozó algoritmusok esetében: ha egy kivonás után nem garantálható, hogy az eredmény ábrázolható (azaz nem nulla), akkor hogyan lehetnénk abban biztosak, hogy a következő számítás nem fog hibához vezetni? Példák: Ha egy számot elosztunk másik két, nem nulla szám különbségével, lehet, hogy nullával fogunk osztani? Ha egy nem nulla számból kivonunk egy másik nem nulla számot, majd később ugyanezt hozzáadjuk, akkor nem fogjuk visszakapni az eredeti számunkat?

2.3.5.15. példa. Számítsuk ki 10 bites mantissza és 5 bites exponens excess-15 tárolása esetében a második legkisebb pozitív számot: $1.0000000001 \cdot 2^{-15} = 2^{-15} + 2^{-25}$. Ha ebből kivonjuk a legkisebb ábrázolható számot (lásd A 2.3.5.13. példát) az eredmény 2^{-25} lesz, ami nyilván nem ábrázolható 10 bites mantisszán és 5 bites exponensen excess-15 formában (az eddig ismertetettek szerint).

2.3.6. Az IEEE 754 lebegőpontos számábrázolás

A lebegőpontos számok ábrázolásának a gyakorlatban is alkalmazott nemzetközi szabványa a az IEEE 754 = IEC 599 = ISO/IEC 60559. Az ebben definiált konkrét bináris lebegőpontos ábrázolások közül néhány látható a 2.10. ábrán. A szabvány a tárgyaltakon kívül még sok más tulajdonságot, funkciót is definiál, például a kerekítés szabályait, műveleteket, kivételkezelést, sőt tízes alapú lebegőpontos számábrázolást is, de ezekkel jelen tárgy keretében nem foglalkozunk.

elnevezés	mantissza méret [bit]	karakterisztika méret [bit]	karakterisztika eltolás
binary16	10	5	15
binary32	23	8	127
binary64	52	11	1023
binary128	112	15	16383

2.10. ábra. IEEE 754 = IEC 599 = ISO/IEC 60559
szabványos bináris lebegőpontos típusok jellemzői

Tárolási sorrend

A lebegőpontos szám tárolási sorrendje a következő: előjel, exponens, mantissza.

2.3.6.1. példa. A 2.3.5.6. példa esetében (binary16 formátum) tárolandó: 0 11100 0011100010.

2.3.6.2. példa. A 2.3.5.7. példa esetében (binary16 formátum) tárolandó: 1 01011 0100000000.

2.3.6.3. példa. A binary16 formátumban tárolt 0000 0100 0000 0000 bitminta értelmezése: előjel: 0, exponens: 00001, exponens értéke: $1 - 15 = -14$, mantissza: 0000000000, kiegészítve a nem tárolt bittel: 1.0000000000, azaz a tárolt szám: $1.0000000000 \cdot 2^{-14} = 2^{-14}$.

2.3.6.4. példa. A binary16 formátumban tárolt 0000 0100 0000 0001 bitminta értelmezése: előjel: 0, exponens: 00001, exponens értéke: $1 - 15 = -14$, mantissza: 0000000001, kiegészítve a nem tárolt bittel: 1.0000000001 azaz a tárolt szám: $1.0000000001 \cdot 2^{-14} = 2^{-14} + 2^{-24}$.

2.3.6.5. feladat. Mondjuk példát olyan műveletre vagy relációra, amelyet könnyebb elvégezni, ha az eltolt számábrázolást használjuk az exponens tárolására és a tárolási sorrend: exponens, mantissza (azaz nem a normalizált alak sorrendje: mantissza, exponens)

Subnormált ábrázolás

Az alulcsordulási hiba megszüntetéséhez – az IEEE 754 szabványnak megfelelve – az előzőekben tárgyaltakkal ellentétben a nullaként tárolt exponens értéket speciálisan kell kezelni: ebben az esetben a mantissza legnagyobb helyiértékű bitjét az előzőekben megismert fix 1 helyett nullának kell értelmezni, és az exponens eltolását is eggyel csökkenteni kell (azaz például excess-15-ről excess-14-re):

2.3.6.6. példa. Az IEEE binary16 formátumban tárolt 0000 0000 0000 0001 bitminta értelmezése: előjel: 0, exponens: 00000, exponens értéke: mivel az exponens tárolt értéke 0, az eredeti (excess-15) értelmezés ($0 - 15 = -15$) helyett a módosított számítási mód (excess-14) szerint: $0 - 14 = -14$, mantissza: 0000000001, kiegészítve a nem tárolt bittel: 0.0000000001, azaz a tárolt szám: $0.0000000001 \cdot 2^{-14} = 2^{-24}$. Ez az IEEE binary16 formátumban tárolható legkisebb érték.

2.3.6.7. példa. Az IEEE binary16 formátumban tárolt 0000 0000 0000 0010 bitminta értelmezése: előjel: 0, exponens: 00000, exponens értéke: mivel az exponens tárolt értéke 0, az eredeti (excess-15) értelmezés ($0 - 15 = -15$) helyett a módosított számítási mód (excess-14) szerint: $0 - 14 = -14$, mantissza: 0000000010, kiegészítve a nem tárolt bittel: 0.0000000010, azaz a tárolt szám: $0.0000000010 \cdot 2^{-14} = 2^{-23} = 2 \cdot 2^{-24}$.

2.3.6.8. példa. Az IEEE binary16 formátumban tárolt 0000 0000 0000 0011 bitminta értelmezése: előjel: 0, exponens: 00000, exponens értéke: mivel az exponens tárolt értéke 0, az eredeti (excess-15) értelmezés ($0 - 15 = -15$) helyett a módosított számítási mód (excess-14) szerint: $0 - 14 = -14$, mantissza: 0000000011, kiegészítve a nem tárolt bittel: 0.0000000011, azaz a tárolt szám: $0.0000000011 \cdot 2^{-14} = 3 \cdot 2^{-24}$.

Ha az IEEE binary16 formátumban tárolható második legkisebb pozitív számból (lásd a 2.3.6.7. feladatot) kivonjuk az ábrázolható legkisebb pozitív számot (lásd a 2.3.6.6. feladatot), akkor eredményül 2^{-24}-et kapunk, ami szintén ábrázolható. Így az IEEE 754 számábrázolási módszerrel bármely két nem nulla ábrázolható szám különbsége kizárólag akkor nulla, ha a két szám megegyezik. Ez egy nagyon fontos numerikus tulajdonság!

2.3.6.9. példa. Az IEEE binary16 formátumban tárolt 0000 0000 0000 0000 bitminta értelmezése: előjel: 0, exponens: 00000, exponens értéke: mivel az exponens tárolt értéke 0, az eredeti (excess-15) értelmezés ($0 - 15 = -15$) helyett a módosított számítási mód (excess-14) szerint: $0-14 = -14$, mantissza: 0000000000, kiegészítve a nem tárolt bittel: 0.0000000000, azaz a tárolt szám: $0.0000000000 \cdot 2^{-14} = 0$.

2.3.6.10. példa. Az IEEE binary16 formátumban tárolt 1000 0000 0000 0000 bitminta értelmezése: előjel: 1, exponens: 00000, exponens értéke: mivel az exponens tárolt értéke 0, az eredeti (excess-15) értelmezés ($0 - 15 = -15$) helyett a módosított számítási mód (excess-14) szerint: $0-14 = -14$, mantissza: 0000000000, kiegészítve a nem tárolt bittel: 0.0000000000, azaz a tárolt szám: $-0.0000000000 \cdot 2^{-14} = -0$.

A nulla kétfajta tárolási módjának az a jelentősége, hogy jelezhető, hogy az alulcsordult szám milyen irányból közelítette meg a nullát. (Hasonlóan jelezzük például a határérték számításnál, hogy a nullát milyen irányból közelítjük meg: $\lim_{x\to 0-} f(x)$ illetve $\lim_{x\to 0+} f(x)$)

A http://babbage.cs.qc.cuny.edu/IEEE-754/index.xhtml oldalon kipróbálhatók, ellenőrizhetők az átváltások.

Végtelenek és a NaN

Az IEEE 754 lebegőpontos számábrázolások a valós számokon kívül képesek tárolni a ∞-t és $-\infty$-t, továbbá a speciális NaN (Not a Number) értéket. Ez utóbbit kapjuk eredményül (többek között) akkor, ha nullát nullával osztunk vagy ha negatív számból vonunk négyzetgyököt.

Ha az exponens minden bitje 1 és a mantissza nulla, akkor az (előjeltől függően) $\pm\infty$ az ábrázolt érték, ha a mantissza nem nulla, akkor NaN az ábrázolt érték.

2.3.6.11. feladat. Adjuk meg a legnagyobb binary16-ban ábrázolható számot!

2.3.7. Numerikus matematika

A numerikus matematika foglalkozik már meglévő számítási algoritmusok vizsgálatával illetve olyan új algoritmusok tervezésével, amelyek figyelembe veszik, hogy a számítógépen végrehajtott számítás során az előzőekben ismertetett hibák az egymás után végzett műveletek során ne nőjenek olyan nagyra, hogy magának az eredménynek a használhatóságát veszélyeztetnék.

2.4. Karakterek és kódolásuk

2.4.1. Karakterek és karakterkészletek

A *karakter*t [character] (a számhoz hasonlóan) fogalomnak tekintjük, amit meg kell tudni jeleníteni írott formában, illetve el kell tudni tárolni a memóriában vagy bármely más tárolóeszközön, fájlban, illetve továbbítani kell tudni egy informatikai hálózaton. Karakter lehet az ABC egy betűje, egy szám, egy írásjel (a szóközt is beleértve) vagy egyéb más írásrendszerben használt jel illetve vezérlő karakter (például soremelés) is.

Karakterkészlet [character set, charset] alatt karakterek kiválasztott csoportját értjük (a kiválasztás lehet tetszőleges, de általában valamely ország, nemzetiség, nyelv, régió alapján történik).

2.4.2. Karakterek kódolása

Karakterek kódolása [character encoding, coded character set] alatt a karakterekhez valamilyen érték (kód) rendelését értjük. Ilyen például a Morze-kód, ami karakterekhez hosszabb és rövidebb impulzusokból álló kódokat rendel (amiket aztán könnyen lehet továbbítani például egy rádió adó-vevő segtségével), vagy a Braille-kód, ami karakterekhez 3D objektumokat rendel (amit aztán megfelelő technológiával „kinyomtatva” látásukban sérült emberek is képesek elolvasni). Az informatikában a karakterkódolás általában a karakterhez egy szám rendelését jelenti (amit aztán valamilyen módszerrel tárolunk). Pl: a 65 jelentse az „A” betűt.

Karakterek tárolási formáján [characer encoding form, character encoding scheme] a karakter kódok (számok) konkrét tárolási módját értjük. Ez lehet triviális, például hogy egy megfelelő

hosszúságú, egészek tárolására használt ábrázolást alkalmazunk, vagy lehet szofisztikáltabb, például egy tömörebb – de bonyolultabb – változó kódhosszúságú kódolás esetében. Pl. a 65-öt egy bájtos előjel nélküli egészként ábrázolva a 01000001 jelenti az „A” betűt.

Egyes kódolási módszerek egyszerre meghatározzák a karakterek kódolását és a kódok tárolási formáját. Tipikusan ilyen kódolási módszerek a klasszikus kódtáblák [code page, character map, charmap], amelyek előjel nélküli egészként, egy bájton ábrázolnak egy karaktert.

2.4.3. Klasszikus kódtáblák

Az ASCII kódtábla

Az American Standard Code for Information Interchange (ASCII) 7-bites kódtábla látható a 2.11. ábrán. Az egyes karakterek kódja a karakter sorának és oszlopának fejlécéből adódik: például a @ kódja 0x40, a W kódja 0x57.

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

2.11. ábra. Az ASCII 7-bites kódtábla (forrás: wikipedia.org/wiki/ASCII)

Nagyon fontos kiemelni, hogy az ASCII kódtábla 7-bites, azaz 128 karakter kódolására alkalmas. Ha 8-biten tároljuk vagy továbbítjuk, a legnagyobb helyiértékű bitet nullára állítjuk.

2.4.3.1. feladat. A home1.paulschou.net/tools/xlate/ honlapon ellenőrizhetők az ASCII karakterek bináris átváltásai.

Az ISO 8859-X kódtáblák

Az ISO/IEC 8859-X kódtáblák az ISO és az IEC szabványosító testületek közös kódtáblái, céljuk, hogy minél több regionális karaktert tartalmazzanak. Az ASCII kódtábla az ISO 8859-X kódtáblák része, azaz minden ASCII kód egyben érvényes ISO 8859-X kód is. Mi a leggyakrabban az ISO 8859-1 (nyugat-európai) és az ISO 8859-2 (közép-európai) kódtáblákkal találkozhatunk, ezeket szokás latin-1 és latin-2 kódtábláknak is nevezni. Az ISO 8859-1 kódtáblát ISO 8859-15 (latin-9) néven frissítették (többek között belekerült az euro karakter), és hasonló történt az ISO 8859-2-vel is: ISO 8859-16 (latin-10) néven frissítették.

A Windows kódtáblák

Az ASCII kiterjesztésével a Microsoft megalkotta saját 8-bites kódtábláit, külön-külön egyes régiókra. Mi a leggyakrabban a windows-1250 (közép-európai) és a windows-1252 (nyugat-európai) kódkészletekkel találkozhatunk. Ezeket szokás windows latin-2 illetve windows latin-1 kódtábláknak is nevezni.

Szintén fontos tulajdonsága ezeknek a kódoknak is, hogy az ASCII-t módosítás nélkül tartalmazzák, azaz annak csak kiegészítései. Az ISO 8859-X kódtáblák és a windows-xxxx kódtáblák nagy részben egyeznek (nem kizárólag a közös ASCII részek), de nem teljes mértékben kompatibilisek egymással.

Feladatok

2.4.3.2. feladat. Megváltozik-e egy adott ASCII karaktert tároló bájt, ha előjeles vagy előjel nélküli egészként tároljuk?

2.4.3.3. feladat. Keressük meg az ISO 8859-1, az ISO 8859-2, a windows-1250 és a windows-1252 kódtáblák kiosztásait!

2.4.3.4. feladat. Helyes eredményt kapok, ha egy ASCII kódolt karakterekből álló fájlt a.) windows-1250 b.) windows-1252 kódtáblák alapján próbálom értelmezni?

2.4.3.5. feladat. Mi lehet a magyarázata annak, hogy régebben (sajnos sokszor még ma is) az ő illetve Ő betűk helyett (hibásan) õ vagy Õ szerepelt?

2.4.3.6. feladat. Tudunk-e több, különböző kódkészlethez tartozó karaktert egyetlen szövegfájlban tárolni (és azokat helyesen megjeleníteni olvasáskor)?

2.4.3.7. feladat. Adjunk példát olyan szövegfájlra, ami nem csak ASCII karaktereket tartalmaz, de mégis azonos módon értelmezhető a.) ISO 8859-2 és windows-1250 b.) ISO 8859-1 és windows-1252 kódtáblákkal!

2.4.4. A Unicode

Az előzőekben ismertetett kódtáblák közös problémája, hogy önmagukban nem képesek több nyelvű szövegek tárolására (sőt, egyes esetekben még egyetlen nyelv esetében sem, például: kínai, japán), mivel a – 8 bites tárolásból adódó – lehetséges 256 különböző karakter nyilván nem elegendő a világ összes nyelvében használt betű és írásjel ábrázolására. Ennek a problémának a megoldására jött létre a Unicode konzorcium, ami létrehozta és karbantartja a Unicode ajánlást. A jelenlegi változat (Unicode 6.2 - 2012 szeptember) 93 írásrendszert és több, mint százezer karaktert tartalmaz, amik között élő és holt nyelvek mellett megtalálhatók matematikai és egyéb szimbólumok is.

2.4.4.1. feladat. A www.unicode.org/charts/ oldalon nézzük meg a Unicode által támogatott karaktereket!

A Unicode azonos az ISO/IEC 10646 szabvánnyal. A Unicode egy karakter kódolási szabvány (figyelem, önmagában nem kódtábla!), ami meghatározza, hogy egy konkrét karakternek mennyi a Unicode értéke [code point]. Általában U+hexa_kód formában jelöljük: például az euro jel kódja: U+20AC. Ezt a Unicode értéket különböző módokon lehet tárolni.

Az UTF-8

Az UTF-8 a Unicode egy tárolási formája (Unicode Transformation Format) ami a Unicode kódokat változó hosszon, 1-4 bájton tárolja, a kód értékétől függően. Legfontosabb tulajdonságai:

- ASCII kompatibilis, azaz minden ASCII szöveg egyben helyes UTF-8 szöveg is,
- önszinkronizáló, azaz nem kell az UTF-8 bájtsorozat elejéről kezdeni az olvasást, hogy pontosan el lehessen határolni az egyes karaktereket reprezentáló UTF-8 byte csoportokat.

Az UTF-16

Az UTF-16 tárolási forma változó hosszon, 2-4 bájton tárolja a Unicode kódokat. Nem ASCII kompatibilis.

Az UTF-32

Az UTF-32 tárolási forma fix hosszon, 4 bájton tárolja a Unicode kódokat. A kódolás nagyon egyszerű: az Unicode kódokat kell 4 bájtos egészekként tárolni. Nem ASCII kompatibilis.

Feladatok

2.4.4.2. feladat. Hogyan befolyásolják a tárolási méretet a használt karakterek? Mikor érdemes az UTF-8 és mikor az UTF-16 kódolási formát választanunk?

2.4.4.3. feladat. Az UTF-8 önszinkronizáló tulajdonságának milyen szerepe van a

- véletlenül kiválasztott pozícióból történő megjelenítésre,
- a hibás átvitelből adódó értelmezési problémákra?

2.4.4.4. feladat. Az UTF-8, UTF-16, UTF-32 kódolások közül melyek alkalmasak szövegek véletlenszerű elérésére (azaz például ha az x. karakterhez akarok közvetlenül ugrani)?

2.4.5. Szövegfájlok

Ha egy fájlban csak szöveget akarunk tárolni (pontosabban csak olyan karaktereket, amelyek mindegyike megtalálható egy kiválasztott karakterkészletben), akkor nincs más dolgunk, mint a karakterek (kódtáblája vagy valamely Unicode tárolási formája szerinti) bájtjait egymás után írni, és ezt eltárolni. Valójában is ez történik, ezeket a fájlokat nevezzük *egyszerű szövegfájl*oknak [plain text file], kiterjesztésük (általában): TXT.

2.4.6. Feladatok

2.4.6.1. feladat. Hasonlítsuk össze a 7 tárolási formáit: a.) előjel nélküli egészként, b.) kettes komplemens ábrázolású előjeles egészként, c.) ASCII kódolással d.) UTF-8 kódolással!

2.4.6.2. feladat. Honnét tudjuk, hogy egy szövegfájl beolvasásakor a kódokat melyik kódtábla szerint kell értelmeznünk?

2.4.6.3. feladat. Pusztán a szövegfájlba történő beleolvasással eldönthető-e általános esetben, hogy az milyen kódtábla szerint értelmezendő?

2.4.6.4. feladat. Töltsünk be a böngészőnkbe egy szövegfájlt, majd módosítsuk a kódtáblát! Kódoljuk át a szövegfájlt a iconv parancs segítségével, és nézzük meg az eredményt a böngészővel illetve az od programmal!

2.4.6.5. feladat. Töltsük be a www.itk.ppke.hu/oktatas oldalt, és a böngészőnkben állítsuk át a karakterkészletet! Mi történik?

2.4.6.6. feladat. Mi a mojibake? (Keressünk rá!)

2.4.6.7. feladat. Ha ékezetes karaktereket használunk egy SMS-ben, akkor van-e különbség az egy SMS-ben elküldhető karakterek száma között, ha magyar (jobbra dőlő) ékezetekkel rendelkező karaktereket vagy csak balra dőlő ékezetekkel rendelkező karaktereket használunk? Indokoljuk meg!

3. fejezet

Memóriakezelés és folyamatok

3.1. Bevezetés

A fejezet célja, hogy programozói szempontból bemutassa az elsődlegesen használt illékony [volatile] memóriát, azok szervezését, felépítését, továbbá a buszrendszereket, amin keresztül (többek között) a CPU és a memória kommunikál.

A fejezetben előkerülő témák mindegyikéről rengeteg irodalmat lehet találni, ezért sok esetben egyszerűsítünk, elhanyagolunk részleteket. Ezek felsőbb éves tantárgyakban hangsúlyosan fognak előkerülni (Digitális rendszerek és számítógép architektúrák, Operációs rendszerek, Áramkörök elmélete és számítása), itt a célunk az alapozás.

3.2. A busz

Általánosan a busz három fő részből áll: a *címbusz*ból, amin keresztül a cím továbbítódik, az *adatbusz*ból, amin keresztül az adat továbbítódik, és a *vezérlőbusz*ból, amin keresztül a működési módokhoz szükséges vezérlő információkat továbbítjuk (például, hogy írást vagy olvasást végzünk).

Ha a címbusz N bit széles, akkor ezen keresztül 2^N különböző cím továbbítására van lehetőség. Ha az adatbusz w bit széles, akkor ezen keresztül egy átvitellel [transfer] w bit továbbítható.

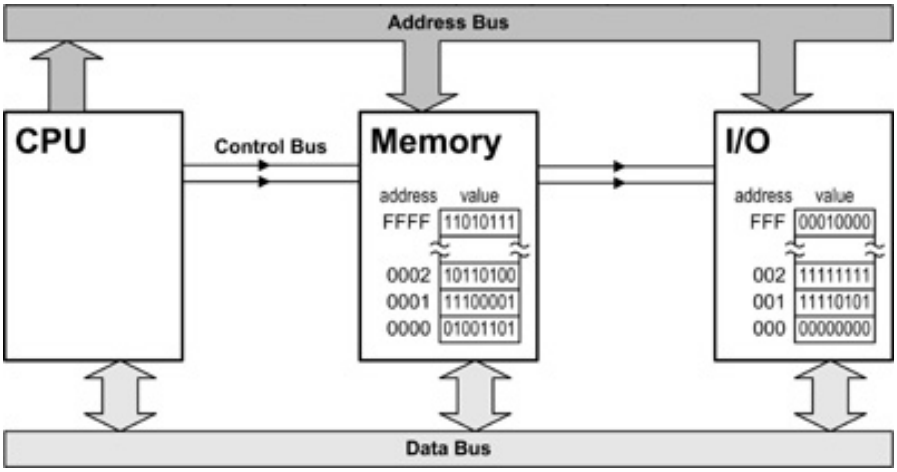
A 3.1. ábrán látható az adat- és címbusz madártávlati képe. A működési mód a következő: ha a processzornak egy adott címen lévő memória tartalmára van szüksége, akkor a memória címet a címbuszra, az olvasás utasítást a vezérlőbuszra írva megkezdődik az olvasási műveletet, melyre válaszul kis idő elteltével (ez az adat hozzáférés ideje) az adatbuszon megjelenik a kért adat, amit a processzor beolvass. Adat írásnál a processzor a cím mellett az adatot is elküldi a cím- illetve adatbuszokon.

3.2.0.8. feladat. Miért nem lehetett a régi 32 bites processzorokkal 4GiB-nál nagyobb memóriát használni?

3.2.1. Az órajel

A buszokon a kommunikáció periodikusan, az *órajel* alapján történik, ami egy négyszögjel. Az egyes konkrét busz megvalósítások pontosan előírják, hogy az adat és címbuszon megjelenő adatok mely időpontban tekinthetők érvényesnek (olvasás esetén), illetve mely időpontban kell azoknak stabilan megjelenniük (írás). Ez a pillanat általában az órajel felfutó vagy lefutó éle. Mivel a gyakorlatban nincs tökéletes négyszögjel sem alak szempontjából (a jel inkább trapéz alakú), sem időbeli pontosság szempontjából (a felfutó és lefutó élek nem pontosan az elvárt időpontban történnek [jitter]), ráadásul ezek a különbségek változnak is, a digitális kommunikáció során ezekre figyelemmel kell lennünk.

[0] $Revision : 52$ ($Date : 2013 - 10 - 0722 : 05 : 15 + 0200(Mon, 07Oct2013)$)



3.1. ábra. A CPU, memória és az I/O vezérlő blokkdiagramja. Forrás: http://www.talktoanit.com/A+/aplus-website/lessons.html

3.2.2. Bájtsorrend

Amennyiben rendelkezünk egy adott N címhosszúságú és w szóhosszúságú memóriával, úgy a következő fontos kérdés, hogy hogyan tudunk olyan adatokat ábrázolni, melyek hossza meghaladja a szóhosszúságot. Legyen a memória az egyszerűség kedvéért byte-szervezésű (8 bites szóhossz), tároljunk ebben a memóriában egy 32 bites előjel nélküli egész számot. Az egyik lehetőség, hogy a legkisebb (1 byte szélességű) memória címen a tárolandó szám legmagasabb helyiértékű bitjeit tároljuk, majd így haladunk sorban a többi bájt esetében, helyiérték szerint csökkenő sorrendben. Ez a *big-endian* tárolási mód. A fordított sorrendet - azaz amikor a tárolandó szám legkisebb helyiértékeit tartalmazó bájtot tároljuk az első memória címen - nevezzük *little-endian*-nak. A big- és little-endian fogalmak az adatkommunikáció esetében nem a memória cím szerinti sorrendet, hanem a továbbítás szerinti sorrendet határozzák meg.

Little-endian szervezésűek az Intel, AMD processzorai, míg big-endianok a régebbi RISC processzorok: PowerPC, SPARC, MIPS. Egyes architektúrák már képesek mindkét működési módra is: ilyenek az újabb generációs RISC processzorok, és a manapság nagyon gyakran alkalmazott ARM processzor is.

Egy adat elemeinek tárolási sorrendje nem csak az informatikában, hanem a hétköznapokban is használatos: például németül a 23 szám esetében először a hármat mondják és írják, majd csak utána a kettőt (little-endian), ugyanakkor a magyarban a kétjegyű számokat big-endian módon írjuk és ejtjük (először a tízeseket majd utána az egyeseket). Ugyanilyen különbség figyelhető meg a dátumok elemeinek esetében is: magyarban a sorrend: év, hónap, nap, míg brit nyelvterületen a sorrend: nap, hó, év (sőt, ezt még kombinálják, mert például az USA-ban a sorrend: hónap, nap, év, amit mixed-endiannak nevezünk).

3.2.3. Igazítás

Másik fontos fogalom az igazítás [alignment]: mivel a CPU és a memória között szó-szélességű a kommunikáció, ezért a több szón ábrázolt adatokat több részletben kell átküldeni az adatbuszon (lásd buszok szakasz). Mivel ez a kommunikáció arra van hangolva, hogy bináris szavak egész számú többszörösét továbbítsák, ezért hasznos, ha a memóriában a különböző adatok egymás után azonos számú szavanként következnek, vagy másként megfogalmazva: szóhatáron kezdődjenek. A tárolandó adatok azonban nem mindig ugyanannyi bitből állnak (tehát nem ugyanannyi szót foglalnak el), ezért fordítási időben „felhízlalhatjuk” őket, hogy minden adat egyforma hosszú legyen. Például ha minden adat egyformán 4 szóból áll, akkor memóriaolvasásnál 4-esével lehet léptetni a memóriacímeket (hiszen csak annyinként kezdődik új adat). Ezzel javul a memória olvasás sebessége és a rendszer teljesítménye. (Fordítva is gondolkozhatunk: ha nem megfelelő igazítással dolgozunk, akkor a rendszer teljesítménye jelentősen csökkenhet.)

3.3. Memória

Memóriák esetében két nagy csoportot különböztetünk meg: az egyik az úgynevezett illékony memória (RAM), mely a tápfeszültség megszűnésével elveszti a benne tárolt információt, ilyenek például a számítógépek elsődleges memóriái. A másik csoportot a nem illékony memóriák adják, amik (többek között) lehetnek csak olvashatók (ROM), UV fénnyel törölhetők (EPROM), elektronikusan törölhetők (EEPROM), vagy írhatók-olvashatók (flash). A nem-illékony memóriák írási és olvasási sebessége jelentősen kisebb, mint illékony társaiké, ez az ára a tápfeszültségmentes tárolásnak. Mivel egy nagy teljesítményű processzort folyamatosan adatokkal kell ellátni (annak érdekében, hogy ne álljon tétlenül), gyorsan írható és olvasható illékony memóriákat használunk erre a célra. A továbbiakban csak az illékony memóriákkal foglalkozunk.

A memória alapegysége a cella, mely egy bit (logikai 0 vagy 1) tárolására alkalmas (gyakorlatilag egy térvezérlésű tranzisztor állapota határozza meg a tárolt bitet - részletesen az Áramkörök elmélete és számítása című felsőbbéves tárgyban).

3.3.1. SRAM, DRAM, SDRAM

Az alábbiakban összehasonlítunk két, félvezető technikával működő illékony memória megvalósítást. Közös ezekben a megvalósításokban, hogy tápfeszültség szükséges az adatok tárolásához, a különbség közöttük a megvalósítási módban (és az ebből következő tulajdonságaikban) van.

A statikus RAM cella [Static RAM, SRAM] esetében egyetlen bit tárolásához 6 tranzisztor szükséges és a tápfeszültség folytonos jelenléte esetén a tárolt bit nem változik meg. Ezzel szemben a dinamikus RAM cella [dynamic RAM, DRAM] egyetlen tranzisztorból és egy kondenzátorból áll, ezen kondenzátor töltöttsége határozza meg a cella által tárolt bit értékét. Az utóbbi megvalósítás hátránya, hogy a szivárgó áramok (sőt, minden egyes olvasás!) miatt a kondenzátor töltöttsége idővel csökken, és bizonyos idő után egy olyan szint alá esik, amikor már nem lehet biztonsággal megállapítani az eredetileg benne tárolt információ logikai értékét (mivel nem tudjuk, hogy a töltöttség azért olyan alacsony, mert eredetileg is csak ennyire töltöttük fel, vagy azért alacsony, mert az eredetileg magas töltöttség időközben elfogyott). A dinamikus RAM cella tehát - a statikussal szemben - a benne tárolt érték rendszeres frissítését igényli, ami alatt a memória írásra/olvasásra nem érhető el. Ezekből fakadóan a DRAM esetén különböző memóriaszervezési feladatokat kell megoldani: célunk, hogy a memória egy részét frissíteni tudjuk, míg a másik része addig elérhető maradjon írás-olvasás céljára.

A DRAM cellában az írás/olvasás a kondenzátor töltését/kisütését jelenti, ezért a művelet sokkal lassabb, mint az SRAM esetében, mivel a kondenzátor a kívánt töltését $1 - e^{-t}$ alakú függvény szerint éri el illetve e^{-t} alakú függvény szerint veszti el, ellentétben az SRAM cella szinte azonnali (tranzisztor kapcsolási idejű) írásához/olvasásához képest.

A DRAM esetében az egységnyi területen elérhető tároló kapacitás megközelítőleg háromszorosa a SRAM értékének, emiatt az egy cellára vetített előállítási költség DRAM esetében sokkal kevesebb. Többek között ennek is köszönhető, hogy a DRAM jelentősen elterjedt az SRAM rovására, függetlenül a bonyolult memóriafrissítési igényektől.

Az SDRAM a memória buszhoz szinkronizált DRAM-ot jelenti, azaz a memória írás/olvasás SDRAM esetében órajelre történik. (A DRAM esetében az olvasás/írás a lehető leghamarabb megtörtént, azaz az adatbuszon a lehető leghamarabb megjelentek az értékek.)

A mai hétköznapi számítógépekben (asztali, beágyazott, hordozható) a processzor mellett lévő cache-hez SRAM-ot, a fő memóriához SDRAM-ot használunk.

3.3.2. DDR SDRAM

A DRAM memória mátrixosan van elrendezve, így a címzése sem egyetlen lépcsőben (a cím címbuszon történő beállításával), hanem két lépcsőben történik: először a sor címét küldjük el a címbuszon (aktiváljuk a sort), majd az oszlop cím elküldésével történik meg a teljes címzés. Ebből adódóan a címvezetékek száma a fele, mint amennyire szükség lenne egy lépcsőben (lineárisan) történő címzés esetén.

A DDR [Double Data Rate] SDRAM kihasználja az előbbi tulajdonságot, és egy kiválasztott sor esetében egyetlen órajel alatt két oszlopcímnek megfelelő adat írására/olvasására ad lehe-

tőséget (innét a neve: double data rate): egyet az órajel felfutó élére, egyet a lefutó élére. A DDR2 már négy oszlopcímnek megfelelő adat, a DDR3 nyolc oszlopcímnek megfelelő adat írását/olvasását teszi lehetővé egy órajel periódus alatt. Fontos kiemelni, hogy mindez kizárólag az adatbuszra vonatkozik, azaz a sor és oszlop címből továbbra is órajelenként egy állítható be, csak az írt/olvasott adatmennyiség változik az adatbuszon. Az egyes generációk a használt feszültségszintet is csökkentették, így csökkentve a fogyasztást (disszipált teljesítményt).

A DDR memóriák megjelölésénél alkalmazott DDR-X esetében az X a másodpercenként írható/olvasható memóriacímeket jelenti egy chipre vonatkozólag [transfer rate], így ez megegyezik a memória adatbusz frekvenciájával. Például a DDR-200 jelölés 100 MHz-en működő memóriát jelent, órajelenként két adategység címzésének lehetőségével, azaz másodpercenként 200 M írás/olvasás művelet elvégzését, azaz 200 MHz-es adatbuszt. Hasonlóan, a DDR3-1600 jelölés egy 200 MHz-en működő memóriát jelöl, órajelenként 8 oszlopcím írásával/olvasásával, azaz másodpercenként 1600 M írás/olvasás művelet elvégzését. A DDR, DDR2, DDR3 chipek 100-266 MHz-en működnek, az adattöbbszörözés miatt a DDR esetében 100-200 MHz-es, a DDR2 chipek esetében 200-533 MHz-es, a DDR3 chipek esetében 400-1066 MHz-es adatbuszra csatlakozva.

Ha figyelembe vesszük a memória (azaz az adatbusz) szélességét, akkor az előző értékeket megszorozva a bájtban mért adatbusz szélességgel, megkapjuk a PC-Y jelölést, ami a másodpercenként (teoretikusan) átvihető bájtok számát jelenti. Például egy 64 bites (8 bájt) szélességű DDR3-1600 memória esetében a használt jelölés: PC3-12800, ami 12.8 GiB/s (teoretikus) átviteli sebességet jelöl.

Egyes memóriamodulok esetében találkozni lehet az L illetve az U jelölésekkel (pl. DDR3L-1600). Az előbbi alacsony feszültségű [low voltage], míg az utóbbi különösen alacsony feszültségű [ultra low voltage] memóriamodulokat jelöl.

3.4. Memóriakezelés

3.4.1. Közvetlen elérés

Közvetlen memória hozzáférés [flat model] esetén a programkód által használt cím közvetlenül kikerül a címbuszra (lásd az előző részt), és megcímzi a memória megfelelő rekeszét. A memóriacímek semmilyen módon nem módosulnak, a program által használt címek a valódi memóriacímeknek felelnek meg, ami miatt ez a leggyorsabb (és legegyszerűbb) memóriaelérési mód. Általában beágyazott, kis rendszereknél használjuk, amik a legtöbb esetben egyetlen program futtatására alkalmasak.

3.4.2. Virtuális memória

A közvetlen elérésű memória nem teljesíti a következő igényeket:

- Ideiglenesen a fizikai memóriánál több memória használata: A futó programoknak (a kezelt feladattól függően) jelentős mennyiségű memóriára lehet szükségük, ugyanakkor átlagosan ennél sokkal kevesebb memóriával is beérik. Az eszközeinkben lévő memória méretének a lehetséges legnagyobb foglaláshoz való igazítása nem gazdaságos, mert így a memória nagy része a működés során kihasználatlan maradna. (A szükségesnél kevesebb memória pedig nyilván meggátolja a program futását.)

- Biztonság: Ha több program fut egyszerre (például olyan rendszeren, ahova egyszerre több felhasználó is beléphet, vagy egy felhasználó egyszerre több programot is futtathat), szükséges, hogy a programok által használt memória területek elkülönültek legyenek, minden program csak a saját memóriaterületét érhesse el.

- Hatékony memóriakihasználás: Ha több program fut egyszerre, nem hatékony, ha a futás elején memóriát foglalunk számukra, mert az a futás során elfogyhat (több memóriára van szükség, mint gondoltuk az indításkor), illetve a lefoglalt memória kihasználatlan marad (ha többet foglaltunk, mint amire szüksége van). Futás közben történő memóriakezelésre van szükség.

- Osztott memória: hatékony lenne, ha több, azonos program futásakor a kód csak egyetlen példányban szerepelne a memóriában, míg az adatok minden program számára saját memóriában tárolódhatnának.

3.4.2.1. feladat. Mutassuk meg, hogy az előző igények miért nem teljesíthetők a közvetlen elérésű memóriával!

Megoldásként azt a módszert választottuk, hogy a program az operációs rendszertől kér (és kap) memóriát, amelyet sajátjaként használhat. Az operációs rendszernek így a fizikai memória megfelelő kezelésével lehetősége van a fenti igények kielégítésére.

Virtuális memória használata esetén a futó programok által kezelt memóriacímek *virtuális cím*ek, amelyek fizikai címre fordítását (és ha szükséges, háttértárról való beolvasását) az operációs rendszer végzi, de a megfelelő teljesítmény eléréséhez hardver támogatás is szükséges.

3.4.3. Swap

A tárhely problémára megoldást jelentene, ha egy olcsóbb tárat (háttértár) is memóriaként lehetne használni: a memória egy-egy részét ideiglenesen a háttértárra írjuk, majd onnét szükség esetén visszaolvassuk a memóriába. A módszer nyilvánvaló hátránya, hogy az ilyen módon megvalósított virtuális memória sokkal lassabb működést tesz csak lehetővé (a háttértárak elérése és átviteli sebessége nagyságrendekkel lassabb, mint a memória esetében), de a háttértárak egységnyi tárolókapacitásra vonatkoztatott ára sokkal kevesebb, mint a memória esetében, így – ha a rendszerrel szemben támasztott követelmények megengedik – ezt a módszert használjuk.

Nagyon fontos kiemelni, hogy ezt csak akkor célszerű használni, ha a memóriaigény csak ritkán lépi túl a berendezésben lévő fizikai memóriát. Ellenkező esetben jobb megoldás a fizikai memória bővítése (különben használhatatlanul lassú lesz a rendszer).

3.4.4. Szegmentálás

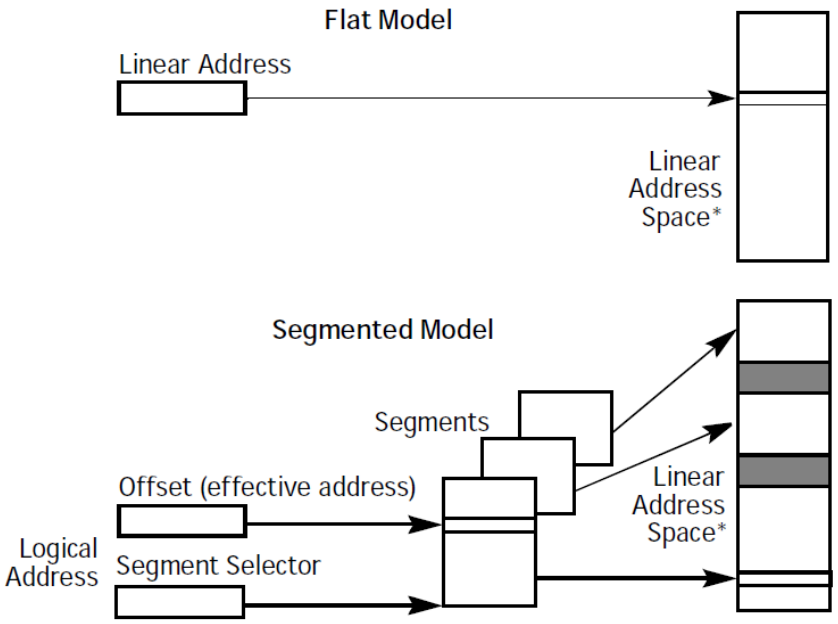
A memória szegmentálása nem más, mint a memória különböző, (nem feltétlenül egyforma méretű) részekre bontása. Egy memória rekesz megcímzéséhez két dologra van szükségünk: a szegmens kiválasztóra, ami megadja a szegmens elejét és az eltolásra [offset], amely segítségével a szegmensen belüli elhelyezkedést adjuk meg. A szegmens kiválasztó nem memóriacím, hanem a szegmensek leíróit tartalmazó listában választ ki egy elemet (konkrétan egy sorszám a listában). Ekkor egy cím fordítás (szegmens szelektor+eltolás) után alakul ki a fizikai cím.

A szegmentálás előnye, hogy különböző funkciójú szegmenseket tudunk kialakítani, amit a processzorok általában hardveresen is támogatnak. A szegmentálás célja a programok egymástól való elválasztása (biztonsági szempontból), továbbá a szegmensek átméretezhetőségének és memóriában történő átmozgathatóságának lehetősége. Sőt, általában egy programon belül is több szegmenst lehet létrehozni: a kód szegmenst, amiben a program futtatható kódjai vannak, és az adat szegmenst, amiben a futás során használt adatok találhatóak (egyéb szegmensek is létrehozhatók, de ezzel most nem foglalkozunk).

Megjegyzés: programozás során sokszor fogunk segmentation fault vagy access violation hibákat kapni. Ezek főként akkor fordulnak elő, amikor programozási hibából adódóan (pl. tömb határain kívüli elem elérésre irányuló próbálkozás) olyan memória területet szeretne elérni a program, amihez nincs joga, azaz szegmens határt sért a program.

A nagy tár probléma legegyszerűbb megoldása az lehetne, ha a memória szegmenseket szükség szerint kiírnánk a háttértárra. Ez azonban a következő újabb problémákhoz vezet:

- Mivel a szegmensek méretei különbözőek, a háttértáron külső töredezettség alakul ki: a szegmensek folyamatos kiírása és visszaolvasása (azaz a háttértárról való törlése) miatt a háttértáron lévő üres helyek egyre szétszórtabban, kisebb darabokban állnak rendelkezésre, ami megnehezíti újabb szegmensek kiírását, mivel a kisebb szabad területekre az újonnan kiírandó szegmensek nem biztos, hogy elférnek. Nyilvánvaló megoldás a háttértáron a virtuális memória számára fenntartott hely bővítése, de az inhatékonyságon ez nem javít.
- Az előzőnél sokkal súlyosabb probléma, hogy nincs lehetőség a szegmensek egyes részeinek kiírására, mert a szegmens+offszet címzésből adódóan a szegmenseknek egy darabban kell maradniuk, nem lehetnek bennük a háttértárra kiírt „lyukak”.



3.2. ábra. A közvetlen elérésű és a szegmentált memória kezelési modellek az Intel x86 architektúrákon. Forrás: IA-32 Architectures Software Developer Manuals

A szegmensek háttértárra írásával megvalósított virtuális memóriakezelést a gyakorlatban nem (vagy csak nagyon ritkán) alkalmazzuk.

3.4.5. Lapozás

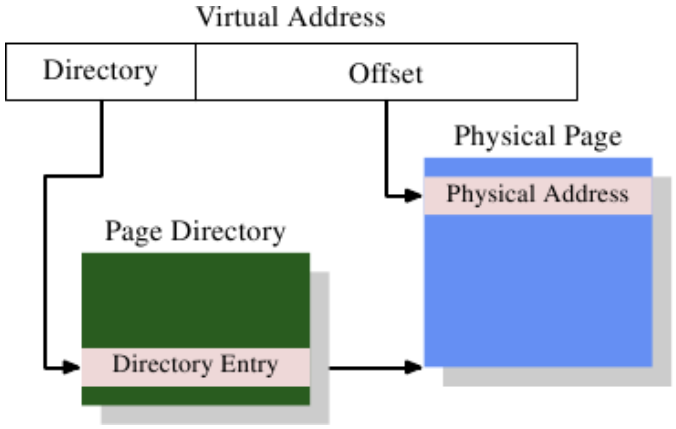
A szegmentálás előnyeinek megtartása mellett, de az előzőekben említett problémák megoldásaként a következőképpen járunk el: a szegmenseket kisebb, de fix méretű egységekre, *lap*okra osztjuk, és ezeket a lapokat írjuk ki illetve olvassuk vissza a háttértárról. A külső töredezettség megszűnik, mivel az ugyanolyan méretű lapok egymással problémamentesen kicserélhetőek, ugyanakkor egy újabb (de sokkal kisebb jelentőségű) problémával szembesülünk: a fix méretű lapok egyben azt is jelentik, hogy lefoglalható memória (és így a szegmens méret is) a lapméret egész számú többszöröse lehet csak, más szóval az utolsó lapon legtöbbször lesz valamekkora nem használt memória.

A lapozás a szegmentálástól függetlenül is alkalmazható, ekkor nem a szegmenseket, hanem a teljes virtuális memóriát osztjuk fel lapokra.

A lapozás szintén processzor szinten van támogatva és a processzor memória kezelő egysége végzi a szükséges adminisztrációt. A lapok tipikus mérete 4 KiB (vagy 64 KiB nagy memóriájú rendszereken), és a programok és adatok ezeken a lapokon vannak elhelyezve. Mivel a virtuális cím tartomány egyik része a fizikai memóriában, a másik része a háttértáron található, ezért a rendszer teljesítménye attól is függ, hogy a lapozást milyen módszerrel végezzük, azaz milyen algoritmus szerint írjuk ki, illetve olvassuk vissza a lapokat. Ez teremti meg a lapkezelési stratégia szükségességét, amely eldönti, hogy egy lap a fizikai memóriában vagy a merevlemezen legyen és ez a kiírás illetve visszaolvasás mikor történjen meg. Sőt, sokszor érdemes a ritkán használt lapokat háttértárra írni, és a memóriában olyan fájlokat tárolni, amelyeket gyakran használnak a folyamatok, így azok sokkal gyorsabban elérhetők (file(system) cache).

Nézzük meg a 3.3. ábra segítségével, hogy hogyan fordítunk le egy virtuális memóriabeli címet fizikai címre. A virtuális cím két részből áll:

- Laptár azonosítóból, ami alapján kikereshető a laptárból az a bejegyzés, ami az adott lappal kapcsolatos, és amiben (egyéb adminisztratív információk mellett) a lap fizikai memóriában lévő kezdő címe megtalálható.



3.3. ábra. A virtuális cím fizikai címre fordítása a lapozás segítségével.

- Eltolásból, ami a lapon belüli címet adja meg.

Amennyiben egy lapnak nincs fizikai memóriabeli címe, akkor page fault kivétel váltódik ki a virtuális memória kezelőben és elindul a lap beolvasása a fizikai memóriába, majd a táblázat adatai frissülnek, többek között a fizikai memóriabeli címével. A *lap könyvtár*beli bejegyzés a lap kezdő címeken felül egyéb információkat is tartalmaz a lapról, az egyik ilyen az, hogy az adott lap futtatható-e. Tehát megtudjuk mondani, hogy az adott lapon adat van és ezért a processzor ne próbálja meg azt utasításként értelmezni.

3.5. Az operációs rendszer

Az operációs rendszerek célja, hogy a rendelkezésre álló erőforrásokat (helyi hardver erőforrásokat: processzort, memóriát, adattároló egységeket, megjelenítő egységeket, egyéb perifériákat; illetve a távolról, hálózaton keresztül elérhető erőforrásokat) elérhetővé tegye a felhasználói programok (vagy más hálózati eszközök) számára, gondoskodjon ezek hatékony és biztonságos felhasználásáról.

Modern operációs rendszerekben, modern hardveren (látszólag vagy ténylegesen) egyidejűleg több felhasználói program is futhat, ebbe beleértendők a több, különböző felhasználó által futtatott programok is, azaz több felhasználó is használhatja egy időben az erőforrásokat. Egy felhasználós [single user] illetve egyszerre csak egy programot futtatni képes [single task] operációs rendszerekkel nem foglalkozunk, kizárólag több felhasználót kezelő [multi-user] és több programot (kvázi-)párhuzamosan futtatni képes [multi tasking] operációs rendszereket tárgyalunk. Jelen jegyzetben a GNU/Linux operációs rendszerrel foglalkozunk részletesebben, de az általános megállapítások nagy része, a specifikusak egy része más operációs rendszerekre is igaz. A tématerület az *Operációs rendszerek* tárgyban mélységében ismertetésre kerül.

3.5.1. A CPU ütemezés

Az operációs rendszer a processzor (számítási) erőforrásait felosztja, ütemezi [schedule] az éppen futó programok között. Ez általában úgy történik, hogy minden éppen futó program valamennyi ideig kizárólagos használatra megkapja a processzort, majd más programok futtatása következik. Ezt időosztásos [time sharing] működési módnak nevezik. Ha a programok számára kiosztott időszelet elegendően rövid, az ember számára úgy tűnhet, hogy a programok párhuzamosan futnak, akár még egy olyan (régebbi) gépen is, ami csak egyetlen processzort (és egyetlen processzor magot) tartalmazott, így a valódi párhuzamos futtatás ezen fizikailag lehetetlen volt. Napjaink több processzoros (otthoni környezetben inkább csak több magos) hardverei esetében itt inkább arra kell gondolni, hogy több programot akarunk futtatni, mint ahányat a hardver fizikailag párhuzamosan futtatni képes. A továbbiakban a párhuzamos futtatást mindig ilyen értelemben értjük.

A párhuzamosan futó programok között a váltás történhet fix időnként is, de általánosabb, hogy akkor történik meg a váltás, ha a program már kellően sokáig futott vagy olyan erőforráshoz akar hozzáférni, ami jelenleg nem elérhető: például be akar olvasni egy fájlt valamilyen háttértárról. Ekkor a processzornak várakozni kellene (mivel a háttértárak általában sokkal lassabban képesek az adatokat beolvasni, mint amit a CPU képes lenne feldolgozni), így az operációs rendszer úgy dönt (a hatékonyabb CPU kihasználás érdekében), hogy egy másik, I/O műveletre nem váró program kapja meg a vezérlést. *Preemptív* [preemptive] multitaskingnak nevezzük azt az eljárást, amikor az operációs rendszer a program hozzájárulása nélkül tudja a program futását ideiglenesen megszakítani (majd egy későbbi időpontban folytatni).

3.5.2. RTOS

Valós idejű az operációs rendszer [real-time operating system - RTOS], ha garanciákat tud adni arra, hogy egy program valamely esemény bekövetkezte után mennyi idővel kezdi meg a futását, illetve a programok képesek processzor időt előre lefoglalni, azaz adott időnként biztosan hozzáférni az erőforrásokhoz. Az előbbi tulajdonság számítógép vezérelt rendszerek esetében fontos, hiszen például egy ipari rendszerben az egyes fizikai paraméterek megváltozására rövid időn belül reagálni kell, ellenkező esetben akár visszafordíthatatlan folyamatok is lejátszódhatnak (például egy túlnyomás érzékelő jelére későn reagáló vezérlő rendszer már nem tudja időben kinyitni az elvezető szelepeket és robbanás következik be). Az utóbbira példa egy audiovizuális tartalom megjelenítő program (például film lejátszó), aminek az egyes képkockák megjelenítéséhez rendszeresen, adott időközönként hozzá kell férnie a processzorhoz (és a megjelenítő eszközhöz, adattárolókhoz, stb.) hogy folyamatos legyen a lejátszás. Ezek a problémák nyilván csak akkor jelentkeznek, ha az operációs rendszer több programot is futtat egyszerre, és azok az erőforrásokat jelentős mértékben használják is: ha egyetlen program fut, senki sem veheti el a processzort a programtól.

Az operációs rendszer *soft real-time*, ha az előző célokat próbálja elérni, és *hard real-time*, ha erre garanciát is vállal.

3.5.3. Rétegezett felépítés

Nagyon fontos következménye az időosztásos működésnek (is), hogy a programváltások miatt a programok nem használhatják a hardvert közvetlenül, kizárólag az operációs rendszeren keresztül. A legfelső réteg tehát az alkalmazói programok, alatta foglal helyet az operációs rendszer, és legalul a hardver.

A különböző perifériák, hardver elemek működését ún. eszközkezelők [device driver] biztosítják, amik az operációs rendszerbe beépülve lehetővé teszik, hogy azokat a felhasználói programok is elérhessék.

A programok az operációs rendszer funkcióit programozási interfészeken [application programming interface - API] keresztül érik el. Ez Linux, UNIX esetében SO (shared object), Windows esetben DLL-eken (dynamically linked library) keresztül valósul meg. Linuxban az `ldd` paranccsal megjeleníthető, hogy a parancssorban megadott bináris állományok melyik SO-kkal linkelődnek össze, ha elindítjuk őket:

3.5.3.1. példa.
```
bercin@users:~$ ldd /bin/bash
libncurses.so.5 => /lib/libncurses.so.5 (0xb76ba000)
libdl.so.2 => /lib/libdl.so.2 (0xb76b6000)
libc.so.6 => /lib/libc.so.6 (0xb7570000)
/lib/ld-linux.so.2 (0xb76fc000)
bercin@users:~$
```

3.6. Folyamatok és szálak

*Folyamat*nak [process] nevezzük valamely végrehajtható állomány operációs rendszer által futtatott példányát. Modern operációs rendszerekben minden folyamat saját memóriaterülettel ren-

delkezik, amit szabadon felhasználhat, de másik folyamat adatát alapértelmezetten nem érheti el.

Az aktuálisan futó folyamatokról a `ps` paranccsal tudunk információt kérni:

3.6.0.2. példa. `bercin@users:~$ ps`

```
  PID TTY          TIME CMD
26974 pts/5    00:00:00 bash
27684 pts/5    00:00:00 ps
bercin@users:~$
```

Ha a saját folyamatainkon kívül más folyamatokat is ki akarunk listázni, a `ps ax` parancsot használhatjuk:

3.6.0.3. példa. `bercin@users:~$ ps ax`

```
  PID TTY      STAT   TIME COMMAND
 5887 tty4     Ss+    0:00 /sbin/getty 38400 tty4
 5888 tty5     Ss+    0:00 /sbin/getty 38400 tty5
 5890 tty2     Ss+    0:00 /sbin/getty 38400 tty2
 5892 tty3     Ss+    0:00 /sbin/getty 38400 tty3
 5894 tty6     Ss+    0:00 /sbin/getty 38400 tty6
 5972 ?        Ss     0:00 /sbin/syslogd -u syslog
 6023 ?        S      0:00 /bin/dd bs 1 if /proc/kmsg of /var/run/klogd/kmsg
 6025 ?        Ss     0:00 /sbin/klogd -P /var/run/klogd/kmsg
 6046 ?        Ss     0:00 /usr/bin/dbus-daemon --system
 6060 ?        Ss     0:00 /usr/bin/system-tools-backends
 6118 ?        Ss     0:00 /usr/sbin/sshd
 6149 ?        Ss     0:00 avahi-daemon: running [kanape.local]
 6150 ?        Ss     0:00 avahi-daemon: chroot helper
 6195 ?        Ss     0:00 /usr/sbin/cupsd
 6230 ?        S      0:00 /bin/sh /usr/bin/mysqld_safe
 6273 ?        S      0:00 logger -p daemon.err -t mysqld_safe -i -t mysqld
 6367 ?        S      0:06 postgres: writer process
 6368 ?        S      0:00 postgres: stats buffer process
 6369 ?        S      0:01 postgres: stats collector process
 6405 ?        Ss     0:08 postgres: writer process
 6406 ?        Ss     0:01 postgres: stats collector process
 6463 ?        S      0:00 /usr/sbin/inetutils-inetd
bercin@users:~$
```

3.6.0.4. feladat. A `man ps` parancs segítségével keressük meg a dokumentációban, hogy mit jelent a STAT oszlopban az S, R, Z státusz!

3.6.0.5. feladat. A `ps axl` parancs kimenetében mit jelentenek a UID, PID, PPID oszlopok?

3.6.0.6. feladat. Mire szolgál a `top` parancs?

3.6.0.7. feladat. Hogyan lehet a `nice` és a `renice` parancsokkal egy folyamat nice level -ét megváltoztatni? (Ismét használjuk a `man` parancsot a dokumentáció elolvasásához!)

3.6.0.8. feladat. Mire szolgál a `ps axf` parancs?

4. fejezet

Adattárolás és fájlrendszerek

4.1. Bevezetés

Egy program futása során az elvégzett számításokból (rész)eredmények keletkeznek, amit az illékony[1] memória [volatile memory] tárol. Ha ezeket az eredményeket el akarjuk tárolni a számítógép két bekapcsolása között vagy egy program két futtatása között, akkor szükséges, hogy a tárolás elektromos áram nélkül is biztosítható legyen. Erre kezdetben nyomtatót, illetve lyukkártyákat használtak (ez utóbbi annyival volt szerencsésebb, hogy egy lyukkártya-olvasóval könnyen vissza lehet tölteni az információt a memóriába). Később megjelentek a szalagos tárolási módszerek, melyek segítségével nagyobb mennyiségű adatot tudtunk lineárisan elmenteni (emlékezzünk a magnókazettákra, ahol ha egy számot ki szerettünk volna hagyni, azt a szalag gyors tekerésével tudtuk csak megtenni).

Az áttörést a cserélhető lemezes olvasó jelentette, ahol egy mágnesezhető korongot forgattunk egy mozgatható mágneses olvasó/író fej előtt, szemben a mágneses szalaggal, ahol az a olvasó fej fixen volt tartva. Ezzel a fejet különböző, a korong más és más részén lévő adatsávok fölé tudtuk helyezni. Ezt a módszert - melyben a tárolón lévő adatok bármelyikét a többi adat érintése/átlépése nélkül érhetjük el - hívjuk véletlen hozzáférésnek [random access]. Ezen a ponton két irány indult el, az egyik mentén a cserélhető lemezeket fejlesztették, míg a másik vonalon létrejöttek fix- vagy merevlemezek.

Mivel ezek az eszközök nem közvetlenül vannak a számítógép alaplapjára építve, hanem valamilyen csatlakozón keresztül kapcsolódnak hozzá, így adattároló perifériák gyűjtőnévvel hivatkozunk rájuk. Például az egeret és a monitort is perifériának tekintjük, az egyiket adatbeviteli perifériának, míg a másikat adatmegjelenítő perifériának hívjuk. Egy adattároló eszköz legfontosabb három jellemzője a következő:

- (adat)hozzáférési idő [(data) latency]: Az adat megcímzése és az adat kiolvasása között eltelt idő. Mai merevlemezeknél ez néhány ms.
- adatátviteli sebesség [transfer rate]: Egy időegység alatt az eszközre írt vagy onnét kiolvasott adat mennyisége.
- tárolókapacitás [capacity]: az eszközön tárolható bitek/bájtok mennyisége. Fontos tisztázni, hogy ez egy bruttó érték: mivel nem a „nyers" merevlemezt használjuk a nettó érték ettől eltérő lehet hiszen a különböző logikai fájlrendszerek más-más módon építik fel a tárolási adatszerkezeteket - lásd a fájlrendszerek részt.

További fontos jellemző az adattárolás élettartalma, azaz meddig képes egy eszköz a ráírt információt megőrizni. A mágneses elven működő eszközök általában előbb szenvednek mechanikai hibából kifolyó adatvesztést, mint hogy a mágneses elven tárolt adat elveszne. További, külső tényezők miatt is sérülhet az adatintegritás, például mechanikai behatás vagy hőhatás miatt[2];

[0] $Revision : 51$ ($Date : 2013 - 10 - 0722 : 03 : 42 + 0200(Mon, 07Oct2013)$)

[1] tápfeszültség megszűnésével a benne tárolt adat elveszik

[2] Minden mágneses anyagnak létezik egy úgynevezett Curie-pontja, ezen hőmérséklet felett az anyag elveszti mágneses tulajdonságát

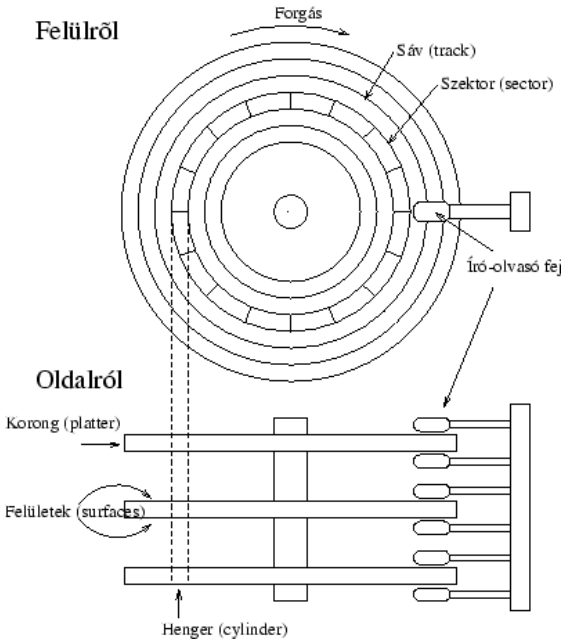
a Compact Diskek (CD) esetében például a felület elgombásodása jelent veszélyt az információra nézve. Az Információ- és kódelmélet című tárgyban részletesen tárgyalásra kerülnek az információ tömörítéséhez illetve az adatvesztéssel szembeni részleges rezisztenciához szükséges módszerek.

4.2. Adattároló perifériák

4.2.1. Merevlemez (Hard Disk Drive - HDD)

Ahogyan a cserélhető lemezes adattárolást, úgy a „fix" vagy „merevlemezes" adattárolást is az IBM mérnökei találták fel az 1950-es években[3]. Maga az adattárolás elve az elmúlt ötven évben nem sokat változott, egyedül a megbízhatóság és a tárolható adatmennyiség mértéke nőtt drasztikusan. A külső levegőtől elzárt vagy porszűrővel ellátott nyitott ház a következő részeket tartalmazza:

- mindkét oldalán mágnesezhető korongok
- olvasó/író fejek, amelyek a mágneses felület felett pár nanométerrel - légpárnán siklanak[4]
- vezérlő elektronika, amely pozicionálja a fejeket, olvasás esetén elvégzi az analóg mágneses mérés digitalizálását, illetve a csatoló felületnek megfelelő adatátviteli protokollt megvalósítja (IDE, SATA, stb.).
- forgatómotor, jellemzően 5400 illetve 7200 forulat/perc sebességgel forgatja a lemezeket, tehát az olvasó fejek - a sugárirányú pozíciótól függően - kb. 270 km/h-val száguldanak a lemezek felett.



4.1. ábra. A merevlemez vázlatos képe

A lemezeket a különböző kerületek mentén sávokra osztják, a sávokat pedig szektorokra, ez a legkisebb címezhető egység egy merevlemezen. Ezek mérete régebben 512 byte volt, jelenleg elérhető 4096 byte is. Mivel az olvasó fejek együtt mozognak, ezért a különböző lemezeken azonos sávon állnak minden pillanatban, ezeket a sávokat együttesen cilindernek nevezzük. Látható

[3] Az első merevlemezt IBM 350 RAMAC néven forgalmazták és 5 megabyte tárolókapacitással rendelkezett, ezt ötven darab 24"-os lemezzel érték el

[4] ebből következik, hogy nincs vákuum a merevlemez belsejében

tehát, hogy az összetartozó adatokat szomszédos szektorokra ill. azonos cilinderekre érdemes írni. Fontos fogalom még a klaszter, ami az azonos sávon egymás után elhelyezkedő szektorok gyűjtőneve. Lásd a 4.1. ábrán. A merevlemez hatékony felosztása és az adatok tárolása a merevlemezre telepített fájlrendszer feladata, melyet részletesebben tárgyalunk.

4.2.2. Compact Disk (CD)

A cserélhető lemezes fejlődési vonalat az utóbbi évtizedig a mágneses tárolási elven működő eszközök határozták meg, de ezek tárolókapacitása nem nőtt és/vagy hozzáférési ideje nem csökkent olyan mértékben, mint a Compact Diskeké, ezért a továbbiakban nem is foglalkozunk velük.

A CD-s adattárolás egy - a merevlemezes tárolástól eltérően - alapvetően optikai elven működő tárolási módszer. Egy lézerdióda által kibocsátott koherens fénysugár tapogatja le a CD lemez felületét, melyen apró gödrök és púpok váltják egymást, melyekről másként verődik vissza a lézersugár - ezzel reprezentálva a bitek értékeit. Gyártás során a CD ROM lemezen - mint a merevlemeznél - sávokat és szektorokat hoznak létre, amelyben a biteket a vájatok [pit] reprezentálják. Pl.: ha van mélyedés [pit] akkor az logikai egyes jelent, ha nincs [lane] akkor logikai nullát. Amikor az olvasó fej mindig azonos szögből megvilágítja a felületet, akkor a vájatba beeső lézer fény máshova verődik vissza, mint az lézer fény, mint ami nem esett bele a vájatba. Mivel a lemez fixen síkban forog és az olvasófej is meghatározott szögben világítja meg a felszínt, ezért a várható visszaverődési helyekre fényérzékeny szenzorokat helyeznek el. Értelemszerűen tehát a logikai egyes és nullás értékek más-más szenzorból váltanak ki jelet. Ebből a felépítésből látható, hogy minél fókuszáltabb a lézersugár, illetve minél kisebbek a vájatok a lemez felületén, annál nagyobb az elérhető adatsűrűség (természetesen további fontos paraméterek is vannak - anyagtechnikai jellemzők, a lézer hullámhossza, valamint több adattároló réteggel rendelkező lemezek is léteznek). A CD-ROM elnevezésben a ROM (Read Only Memory/Media) rövidítés arra utal, hogy ezeket az eszközöket csak egyszer lehet írni, utána már csak olvashatóak. Azóta kifejlesztették az újra írható CD lemezeket is, ahol az olvasástól eltérő tulajdonságú lézersugárral visszaállítják az eredeti felületet - természetesen ebben az esetben nem vájatokkal dolgozunk, hanem a felületnek változtatjuk meg a visszaverődési tulajdonságait az írás során.

4.2.3. Pendrive, Flashdrive, SSD

Mivel ezen eszközök működésének megértéséhez komoly elektronikai háttér tudásra van szükség, ezért a technikai részleteket nem tárgyaljuk. A működési elvről elegendő annyit megjegyeznünk, hogy ezek a tárolók olyan különleges áramkörök, amelyekben lévő tranzisztorok tápfeszültség jelenléte nélkül is képesek megtartani azt az állapotot, amit tápfeszültség jelenlétében beállítottunk. Fontos még megjegyezni, hogy ebből a technikai megvalósításból kifolyólag az ilyen típusú eszközök írási és olvasási sebessége jelentősen eltér egymástól.

A flash memóriák két fő típusát különböztetjük meg: a NAND és a NOR memóriacellából állókat, amik nevüket onnét kapták, hogy a megvalósításuk a logikai NAND (negált ÉS) és NOR (negált VAGY) kapukéra hasonlít. A legfontosabb különbség köztük, hogy a NAND cellák gyorsabban törölhetők és írhatók, kevesebb szilícium területet igényelnek, azaz olcsóbbak is, továbbá a tárolt adatokat csak blokkokban lehet elérni (a NOR flash bájt szintű elérést is lehetővé tesz). Mindebből adódóan napjaink flash memóriái majdnem kizárólag NAND alapúak, és elsődleges feladatuk a háttértárak helyettesítése (mivel azok szintén blokkonként címzettek).

A hardveres megvalósítás iránt érdeklődők a következő kulcsszavak mentén tudnak további információhoz jutni: Flash memory, Floating gate, EEPROM.

Wear levelling

A flash memóriák/tárolók egyik legfontosabb tulajdonsága a már említett blokkonként történő törlés, ami miatt egyetlen bájt megváltoztatása is a teljes blokk törlését követeli meg. Nagyon fontos tehát, hogy flash tárolók esetén több, egymáshoz közel lévő bájt írását egyetlen műveletben végezzük. Sajnos a blokk újraírások számának is van egy felső határa, ami napjainkban (2013) jellemzően százezres nagyságrendű, így a blokkokba szervezett írás sem elegendő, arra is szükség van, hogy a blokkokat lehetőség szerint ugyanannyira használjuk el (ugyanannyiszor töröljük), így megnövelve az eszköz élettartamát. (Ha ezt nem tennénk, akkor lennének olyan

blokkok az eszközön, amit olyan sokszor újraírtunk/töröltünk, hogy azok meghibásodnak, és ha ezek olyan helyen helyezkednek el, ami a fájlrendszer szempontjából kritikus, akkor akár a teljes tárolóeszközt is használhatatlanná tennék. A nem kritikus helyen lévő hibás blokkok is bosszúságot okoznak, hiszen ezek azt jelentik, hogy az eszközön tárolt fájl kiolvasott tartalma nem egyezik meg azzal, amit odaírtunk.)

Ez a meghibásodás-elkerülő technika a *wear levelling*, ami (a memóriakezelésnél megismert virtuális memóriához hasonlóan) egy közbülső réteget képez a flash memória fizikai blokkjai és a felsőbb rétegek (driver, operációs rendszer) által megcímzett logikai blokkok között. Így lehetősége nyílik arra, hogy egy logikailag ugyanarra a blokkra irányuló írás/törlés műveletet más fizikai blokkra irányítson, azaz a fizikai blokkok írásainak/törléseinek számát közel azonos szinten tartsa, így a tárolóeszköz élettartamát nagyságrendekkel meghosszabbítsa (azaz az élettartam ne a legtöbbet írt/törölt logikai blokktól függjön, hanem a teljes eszköz blokkjai gyakorlatilag közel egy időben romoljanak el, az írási/törlési műveletek szétterítésének, egyenletesebbé tételének köszönhetően).

A wear levelling technikában megkülönböztetünk dinamikus és statikus módszert: a dinamikus módszer csak a törlések/írások során végzi el a logikai és a fizikai blokkok összerendelésének megváltoztatását az adott blokk törlési számának figyelembe vételével, míg a statikus módszer a nem írt/törölt blokkokra is kiterjeszti ezt. A különbség tehát, hogy amíg a dinamikus esetben azok a logikai blokkok, amelyeket nagyon ritkán írunk, a helyükön maradnak (és jó állapotban vannak, hiszen csak ritkán írtuk felül), addig a gyakran írt blokkok gyakran cserélődnek, de egyre rosszabb állapotú blokkokon foglalnak helyet (bár ezen blokkok egyenletesen rosszak). A statikus esetben a nem írt/törölt blokkokat is fizikailag áthelyezi a flash vezérlő, így az összes blokk közel ugyanazon az elhasználtsági fokon van. Ha a teljes tárat tekintjük, ez nagy élettartam növekedéshez vezethet (nyilván attól függően, hogy felsőbb szintről a tár mekkora részét írjuk felül/töröljük: például ha egy flash tárolót úgy használunk, hogy a rá másolt adatokat azok felhasználása után töröljük és úgy írunk rá újabb adatot, akkor nincs számottevő különbség a statikus és a dinamikus wear levelling között, viszont ha nagy részén fixen ugyanaz az adat található, és csak kis részét írjuk újra-és-újra, akkor óriási a különbség a statikus és a dinamikus wear levelling által elért élettartam hosszabbodás között.) A statikus módszer egyben azt is jelenti, hogy akkor is törlést/írást kell végezni, amikor azokat nem a felsőbb rétegek kezdeményezik, emiatt a teljesítménye elvileg kevesebb, mint a dinamikus módszerrel működő háttértáré, ugyanakkor megfelelő ütemezéssel ez a különbség számottevően csökkenthető. Ugyanezen okok miatt a statikus wear levelling komplexebb algoritmust, így komplexebb hardveres implementációt igényel, azaz drágább.

Fontos kiemelni, hogy az ilyen típusú meghibásodások oka kizárólag az írás/törlés műveletek száma, nem pedig az utolsó írás óta eltelt idő, azaz egy csak olvasásra használt flash tároló élettartama nagyságrendekkel hosszabb, mint egy írásra is használté.

A meghibásodások elkerülése érdekében végzett wear levelling mellett szükséges, hogy a már meghibásodott blokkokat is nyilván tartsuk, hogy az arra történő írást elkerüljük. Ekkor viszont arra is lehetőség van, hogy a gyártás során eleve hibás blokkokat szintén megjelöljük, ami hatékonyabb, kevesebb selejttel történő (azaz olcsóbb) gyártást jelent, mivel a jelenlegi technológia nem garantálja a 100%-os hibamentességet.

A wear levellinget és a hibás blokkok nyilvántartását az USB (pen)drive-ok illetve az SSD-k hardveres megvalósításban tartalmazzák (*flash vezérlő*), ezeket az eszközöket úgy kell használni, mintha hagyományos háttértárak lennének, minden fentebb említett feladatot elvégez a hardver. Költség és funkcionalitás szempontjából a pendriveok általában dinamikus- míg az SSD háttértárak statikus wear levellinget tartalmaznak.

4.3. Partíciók

Lehetőségünk van arra, hogy a háttértáron lévő területet felosszuk és különböző méretű, de összetartozó területeket hozzunk létre. Egy ilyen területet partíciónak hívunk. Minden partíció külön kezelhető a többitől: törölhető, formázható, másolható, és saját fájlrendszerrel rendelkezik. Ezért fizikailag egy lemezen tárolhatunk különböző operációs rendszereket, anélkül, hogy zavarnák egymást.

A partíciók MS Windows alatt betűvel címkézve jelennek meg, pl. C:, D:. - fontos megjegyezni, hogy ha fizikailag másik lemezen van egy partíció az a meghajtó betű jeléből nem deríthető ki (pl. elképzelhető, hogy a C: meghajtó egy partíció, amely a teljes merevlemezt elfoglalja, míg a D: illetve E: meghajtók fizikailag a C: -től különböző, de ugyanazon a lemezen helyezkednek el: valamilyen arányban megosztják a lemez területét).

GNU/Linux alatt már tisztább a helyzet, a *dev* könyvtár tartalmazza a számítógéphez csatolt perifériák eszközfájljait, így a merevlemezekét is. Az IDE csatolóval rendelkező lemezeket `hda`, `hdb`, `hdc`, `hdd`, ... névvel találjuk a könyvtárban, míg a SATA vagy SCSI csatolóval rendelkezőek `sda`, `sdb`, `sdc` stb névvel érhetőek el. Általában két IDE csatlakozó van egy alaplapon, amelyre két-két eszközt lehet csatlakoztatni, ezért a `hda` az elsőleges IDE csatlakozó master eszköze, a `hdb` ugyanezen kábelen lévő slave eszköz. A `hdc`, `hdd` a másodlagos IDE csatlakozóra felfűzött eszközöket jelzik. Ha a `hda` lemez partíciókat tartalmaz, akkor az elsőleges partíció `hda0` néven, míg a második partíció `hda1` néven fog szerepelni a `/dev` könyvtárban. SATA/SCSI eszközök esetén az `sd` után következő `a,b,c,d` betűjelek és az 1,2,3,4 számok ugyanezt jelentik (például: `\dev\sda0` vagy `\dev\sda1`).

Természetesen az operációs rendszert informálni kell arról, hogy milyen partíciók léteznek az adott lemezen, amit a Master Boot Record (MBR) tartalmaz. Ebben találhatók a partíciók méretei, kezdő és vég értékeik valamint az, hogy melyik partíció tartalmaz operációs rendszer elindításához szükséges adatokat - ezt/ezeket a partíciókat hívjuk bootolható partíciónak.

A MBR-t az 1980-as években találták ki. Továbbfejlesztése a GUID Partion Table (GPT), amely számos kiterjesztést tartalmaz MBR-hez képest, például az MBR esetében a legnagyobb partíció mérete maximum 2 TiB lehet, míg a GPT esetén ez 8 ZiB (ZiB = Zeta Byte = 10^{21} Byte), továbbá GPT partícionálás esetén a partíciók száma sem korlátozott (ellenben az MBR-rel).

4.4. Fájlrendszerek

A fájlrendszer feladata, hogy az eltárolandó fájlokat és könyvtárakat a háttértár egy partícióján a megfelelő helyen elhelyezze, garantálja annak visszaolvashatóságát, valamint a változásokat adminisztrálja. Tehát a fájlrendszer funkciója kettős: egyrészt tárolja egy adott partíción lévő adatok (fájlok) helyét, másrészt kezeli az ezekhez kapcsolódó metaadatokat[5]. Minden, a fájlrendszerben tárolt adathoz (egy adott fájl fizikai elhelyezkedése a lemezen) tartozik egy metaadat bejegyzés is, ez tartalmazza a fájl vagy könyvtár nevét, a létrehozás, módosítás, utolsó hozzáférés dátumát, a tulajdonos adatait, valamint a hozzáférési jogosultságokat. A könyvtárak a fájlrendszer adatbázisában lévő bejegyzésként vannak tárolva (tehát amennyiben egy lemezről elveszítjük a fájlrendszert leíró adatbázist (a metaadatokat), úgy a nyers adatok (fájlok tartalma) visszanyerhetőek, de: 1) nem fogjuk tudni, hogy melyik fájl hol kezdődött, hogyan következnek egymás után a fájl tartalmát alkotó blokkok és hol van vége 2) nem fogjuk tudni rekonstruálni a könyvtárrendszert.) Látható, hogy a fájlrendszer helyes működése létfontosságú a tárolt adatok használhatóságának szempontjából, éppen ezért a fájlrendszer adatbázisa a lemezen általában több példányban, sokszorosítva kerül eltárolásra, csökkentve a megsérülés valószínűségét.

Mivel a számítógépek számos különböző feladatra használhatóak (családi személyi számítógép, bankszámlakezelő rendszer, egy tőzsdei kereskedőrendszer vagy egy milliós forgalmú webkiszolgáló), ezért az adatok tároláskor is különböző igények léphetnek fel (mind teljesítmény, mind biztonság tekintetében). Ezen igényekre kielégítésére rengeteg fájlrendszer megvalósítás létezik más-és-más tulajdonságokkal, teljesítménnyel, funkcionalitással. Ebben a jegyzetben részletesen a FAT, illetve az ext2/3/4 fájlrendszerrel fogunk megismerkedni.

Mivel fájlrendszer szükséges magának az operációs rendszernek az installálásához is, a fájlrendszereket fájlrendszereket az operációs rendszerrel együtt szállítják: például MS Windows operációs rendszerhez három fájlrendszer érhető el: a FAT, NTFS, exFAT. Linux alatt a legelterjedtebb az ext3/4 fájlrendszer, de számos egyedi igényt kielégítő megvalósítás is létezik (XFS, JFS, btrfs, Reiserfs, stb).

Léteznek hálózati fájlrendszerek is (Google Drive, NFS, sshfs, stb.), amelyek az operációs rendszer szempontjából átlagos partíciónak tűnnek, de fizikailag az adatok nem az adat számítógép háttértárán tárolódnak, hanem egy távoli szerveren. Például az egyetemen lévő tárhelyeinket

[5] adatok tulajdonságait leíró adatok

(turdus, users) is felcsatolhatjuk meghajtóként az általunk használt operációs rendszerben (sőt, valójában is ez történik, nézzük meg a `mount` paranccsal).

Fontos megjegyezni, hogy a fájlrendszerek fájl- és könyvtárelhelyezési implementációja nem azonos az általunk megszokott könyvtárstruktúrával! Tehát a felhasználók számára látható könyvtárstruktúra mögött a fizikai tárolási módja ettől teljesen eltérő, ahogy ezt látni fogjuk.

4.4.1. Fragmentáció

Egy fájlrendszerben a fájlok blokkokban tárolódnak. Ha ezek a blokkok fizikailag nem egymás utáni területen helyezkednek el a háttértáron, akkor *külső fragmentáció*ról beszélünk. Ez azoknál a háttértáraknál érdekes, amelyeknél az egymás utáni blokkok olvasása sokkal gyorsabb művelet, mint a távoli blokkok olvasása (ilyen háttértár a merevlemez, mivel a fej mozgatásához illetve a lemez megfelelő helyre forgásához idő kell). Annak érdekében, hogy a fájlok olvasását meggyorsítsuk, a blokkokat egymás után helyezhetjük, amit *defragmentáció*nak nevezünk.

Mivel a fájlokat blokkokban tároljuk, és blokknál kisebb egység lefoglalására nincs lehetőség[6], a fájlt tartalmazó blokkok közül az utolsó majdnem mindig tartalmaz szabad helyet. Ezt *belső töredezettség*nek nevezzük, mivel a blokkokon belül van kihasználatlan hely. Mindebből az is következik, hogy ha N darab 1 bájtot tartalmazó állományt hozunk létre, akkor a fájlrendszerben minden fájlhoz egy blokk lefoglalódik, azaz a lemezen lévő tárhely nem $N \cdot 1$ bájttal csökken, hanem $N \cdot$ blokkméret bájttal.

A blokkméret megválasztásával a belső fragmentáció csökkenthető (hiszen ekkor az utolsó blokk mérete is kisebb, így kevesebb kihasználatlan hely lehet benne), de a blokkok száma növekszik (hiszen kisebb blokkból több jön létre, mivel a teljes partíciót fel kell osztani blokkokra), ami nagyobb lehetőséget ad a külső fragmentációra. Az általánosan használt blokkméret 1-4-64 kiB között változik a partíció méretétől függően, mert ez vállalható kompormisszumot jelent a külső és belső fragmentáció között.

4.4.2. Fájlrendszer implementációk

Flash fájlrendszerek

A flash tárolókra optimalizált fájlrendszerek figyelembe veszik a flash tárolók fentebb tárgyalt sajátosságait, ugyanakkor nagyon fontos, hogy ezek a fájlrendszerek kizárólag közvetlenül a flash tárolókon használandók, azaz az USB (pen)drive, SSD háttértárak esetében szükségtelen ilyen fájlrendszereket használni, mert a flash speciális kezelését a hardver (a fentebb említett flash vezérlő) elvégzi. Flash fájlrendszerekre példa: JFFS(2), YAFFS.

A FAT fájlrendszer

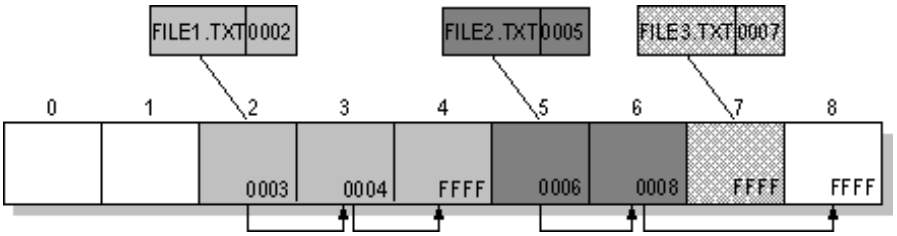
A File Allocation Table (FAT) fájlrendszert a Microsoft fejlesztette, és a Windows XP megjelenéséig ez volt a Windows operációs rendszerek által kizárólagosan használt fájlrendszer. A FAT fájlrendszer klasztereket tart számon, és a különböző FAT verziók főként abban különböznek, hogy hány biten tárolják a klaszterek sorszámait (FAT12, FAT16, FAT32). Ez bitszám határozza meg, hogy összesen mekkora lehet egy FAT partíció mérete. Az operációs rendszert is tartalmazó FAT partíciók tartalmaznak boot sectort is, ez a szektor kerül beolvasásra a memóriába az operációs rendszer bootolásának első lépéseként.

Az alábbiakban részletesen megnézzük, hogyan tárolja a fájlokat a FAT fájlrendszer. A 4.2 ábrán látható a partíció egy darabja: a tárolt fájlok, valamint a hozzájuk tartozó allokációs tábla-beli bejegyzések. Az allokációs tábla - többek között - tartalmazza a fájl nevét és annak a klaszternek a számát, ahol a fájl kezdődik. Minden klaszter végén található egy cím amely a fájl többi darabját tároló klaszterre mutat vagy egy 0xFFF(F...F) jelzés, ami azt jelenti, hogy ez az utolsó klaszter, amiben a fájl részlete volt eltárolva. Fontos észrevennünk, hogy a rendszer nem követeli meg, hogy egy nagyobb méretű fájl egymás utáni klaszterek sorozataként legyen a lemezen: az operációs rendszer utasításaitól függően akár rengeteg, a lemez különböző pontjain elhelyezkedő klaszterbe is kerülhet a fájl egy-egy darabja. A klaszterméret az esetek túlnyomó többségében 2-32 KiB közötti.

[6] az újabb fájlrendszerek már adnak erre lehetőséget, pontosan ezt a problémát megoldandó

FAT fájlrendszert használnak a pendrive-ok.

4.4.2.1. feladat. Egyes pendrive-ok esetében a háttértár első néhány kilobájtja speciálisan kialakított. Mi lehet ennek az oka?

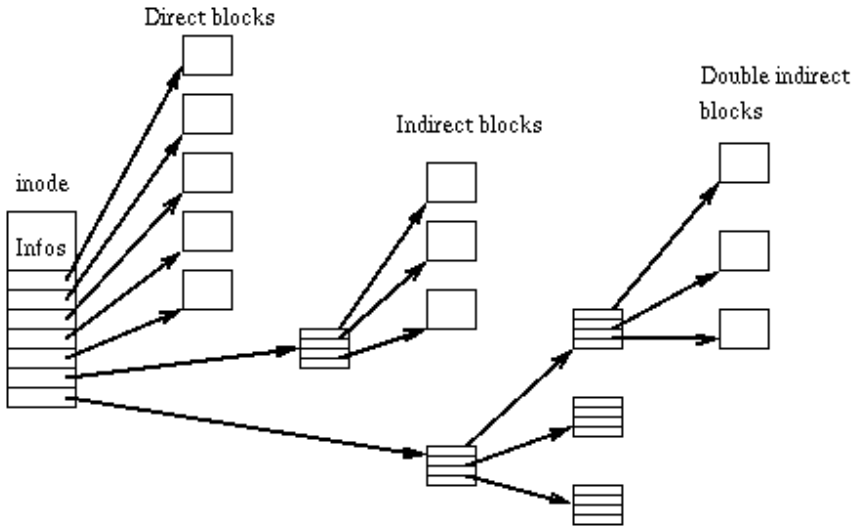


4.2. ábra. Egy FAT partíció darabja. Forrás:
http://www.ntfs.com/images/recover-FAT-structure.gif

Az Ext2 fájlrendszer

Az ext2 fájlrendszer alapegysége a blokk, amelynek mérete tipikusan 1-8 kiB-ig terjed (ez a fájlrendszer létrehozásakor beállítható érték, de később nem változtatható és minden blokk ekkora méretű lesz). Így ha létrehozunk egy fájlt amiben elhelyezünk 2 karaktert az minimum 1 kiB-ot (vagy épp 8 kiB-ot, ha akkora a blokkméret) fog elfoglalni a lemezen. A szuperblokk a partíció elején helyezkedik el és az operációs rendszer bootolásához szükséges adatokat, illetve magáról a fájlrendszerről egyéb információkat tartalmazz. A blokkokat csoportokban tárolják, ezzel is csökkentve a töredezettség mértékét. Ezeket a blokkokból álló csoportokat *extents*-nek nevezzük.

A ext2 fájlrendszerben minden fájlt és könyvtárat egy úgynevezett inode ír le. Az inode tartalmazza a fájllal vagy könyvtárral kapcsolatos adminisztratív információkat: fájlnevét, létrehozás-, módosítás dátumát, tulajdonost, jogosultságokat, stb. Az inode többi része 12-15 linket (blokk címeket) tartalmaz, amely egy csoportot címez meg, ezek a direkt blokkok (lásd a 4.3. ábrán). Amennyiben a fájl mérete meghaladja a direkt blokkokban tárolható adatmennyiséget, akkor az utolsó link helyére nem egy direkt blokk címet helyezünk az inode-ban, hanem egy másik csoportleírót, ami további blokkokra vagy csoportleírókra mutat. Ezzel a módszerrel a legnagyobb tárolható fájl mérete 1 kiB-os blokk méretnél 16 GiB, míg 8 kiB blokkméret esetén 2 TiB.



4.3. ábra. Az ext2 inode felépíése Forrás:
http://upload.wikimedia.org/wikipedia/commons/a/a2/Ext2-inode.gif

A fájlok és könyvtárak mellett létezik egy másik típusú inode bejegyzés-típus is, ez a link. A link nem más, mint egy bejegyzés a fájlrendszerben, amely egy másik fájlrendszer-beli bejegyzésre hivatkozik. Önmagában tehát nem tárol adatot, hanem az őt megnyitó programot továbbirányítja az általa mutatott fájlra (ennek egyszerűbb változata az MS Windows-beli parancsikon). Két típusát különböztetjük meg, az egyik a soft link a másik a hard link.

A hard link esetében a könyvtárbejegyzésben szereplő inode bejegyzés egy már létező i-node-ra mutat. A hard linkek tehát pontosan ugyanúgy néznek ki, mint az adott fájl első könyvtárbejegyzése, azaz a hard linkek egyenrangúak! Ezzel szemben a soft link egy speciális fájl, amely annak a fájlnak az elérési útját tartalmazza, amire mutat. Ebből következően a hard linknél a mutatott fájl vagy könyvtár addig nem törölhető, amíg létezik rá mutató link (ezt a link számlálóból tudja - lásd feladatok). Hard link létrehozására az `ln` parancs használható. Szintaktikája: `ln régi új` ahol `régi` jelenti azt a már meglévő fájlt, amire linket akarunk létrehozni, és `új` jelenti a létrehozandó linket. Az `ls -i` paranccsal kilistázhatók az i-node számok is, így ellenőrizhető, hogy a hard link valóban ugyanarra az i-node-ra mutat. Fontos, hogy hard link csak fájlra hozható létre (azaz könyvtárra nem)! Soft linket az `ln -s régi új` paranccsal hozhatunk létre.

A soft linknél a link az elérési utat tárolja, így a mutatott fájl vagy könyvtár nem tudja, hogy létezik olyan hivatkozás, amely reá mutat. Éppen ezért ha letöröljük a hivatkozott fájlt, a link „célpont" nélkül marad, és „törött" link [dangling / broken link] jön létre. Másik fontos különbség, hogy hard linket csak partíción belül lehet létrehozni, mivel az inode-ra mutat, aminek a számozása partíción belül egyedi. Ezzel szemben a soft link elérési utat tárol (ahogy azt már említettük egy partíció a könyvtárstruktúra tetszőleges pontjára becsatolható), így a soft link mutathat másik partíción elhelyezkedő fájlra is.

SWAP fájlrendszer

Egy adott pillanatban nem minden programot használunk, amit elindítottunk a számítógépünkön, illetve az adott programnak sem használjuk minden részét. Ebből kifolyólag a nem aktív programokat, valamint programrészeket az operációs rendszer nem a viszonylag szűkös memóriában tartja, hanem a háttértáron, az úgynevezett swap területen. A tárolás olyan formátumban történik, hogy a kiírt memórialapokat szükség esetén külön keresés-konvertálás nélkül a memóriába tudja visszatölteni. (Például: amikor a tálcára leteszünk egy programot és sokáig nem foglalkozunk vele, majd később elővesszük azt tapasztaljuk, hogy elég lassan reagál a kéréseinkre, és a háttértár nagy tempóban dolgozik: ekkor kerülnek vissza a swap területről a memóriába az adott programhoz tartozó adatok és program részletek). MS Windows alatt a fájlrendszerben egy fájlként jelenik meg a swap terület, amit Pagefile-nak hív a rendszer. GNU/Linux rendszereknél a swap tárterület fájl mellett egy linux-swap típusú fájlrendszerrel rendelkező külön partíció is lehet.

4.5. Könyvtárstruktúra és a fájlrendszer adminisztrációjának manipulációja

Linux alatt a BASH shell segítségével lehetőségünk van a parancssori utasítások segítségével fájlok és könyvtárak létrehozására, módosítására, valamint törlésére, tehát a könyvtárstruktúra módosítására. Továbbá lehetőségünk van a fájlrendszer adminisztrációs információk megjelenítésére, megfelelő jogosultság esetén módosítására.

4.5.1. Alapvető parancsok

A `cd`, `pwd`, `mkdir`, `rmdir`, `ls`, `rm`, `mv`, `cp` parancsokat az első óra anyaga tartalmazta. A `cat` paranccsal egy fájl tartalma jeleníthető meg, a `touch` paranccsal egy üres fájl hozható létre.

4.5.2. Jogosultságok

A linux disztribúciókban található egy kitüntetett felhasználó, a rendszergazda, ami a telepítés során jön létre, a neve: root. Neki mindenhez joga van, bármit törölhet, bármit megnyithat, létrehozhat felhasználót, stb. Az ő általa indított programok az ő jogaival futnak, egy szándékosan

vagy véletlenül megváltoztatott program a root jogaival futva komoly károkat tud okozni. Ezért a legtöbb disztribúcióban létre kell hozni már a telepítéskor egy korlátozott jogú felhasználót, akinek az adataival belépve korlátozott jogokkal tudunk dolgozni. Ez így biztonságos!

4.5.2.1. példa. Írjuk be: `cat /etc/passwd` A kapott hosszú lista első oszlopa a rendszerünkön lévő felhasználók neveit tartalmazza, a sajátunkat is ott kell látnunk. (Talán kiderült már: a cat utasítással szöveges fájlok tartalmát lehet kilistázni.)

A lentebb bemutatásra kerülő jogoknak igazi jelentősége a több felhasználó által használt rendszerek esetében van (pl.: users és turdus szerverek), ha egy gépet csak egyedül mi használunk, a jogosultságok állítgatása nem lesz annyira fontos.

Minden felhasználó valamilyen csoportnak is tagja (akár többnek is), mindenkinek van egy alapértelmezett csoportja (elsődleges csoport), ez Debian rendszeren megegyezik a felhasználó nevével, a felhasználó létrehozásakor jön létre, az új felhasználó egyből belekerül.

4.5.2.2. feladat. Adjuk ki a következő utasítást: `cat /etc/group` A kapott lista első oszlopa a rendszerünkön lévő csoportok neveit tartalmazza.

A Linux fájlrendszere tárolja a fájl tulajdonosának azonosítóját a fájlhoz tartozó csoportot és a hozzáférési jogosultságot is. A hozzáférési jogosultságok ábrázolásához egy három részből álló kódot használ, amit fájlmodnak nevezünk.

- Első rész a saját (user) jogot
- Második rész a csoport (group) jogot
- Harmadik rész mindenki más (others) jogait rögzíti

A saját jog alatt a fájl tulajdonosának jogait értjük, legtöbb esetben ő az adott fájl vagy könyvtár létrehozója is. Mindegyik rész a következő komponensekből áll:

- r (Read): olvasási jog (vagyis az adott fájl ezáltal olvasható)
- w (Write): írási jog (az adott fájl ezáltal válik írhatóvá)
- x (eXecute): végrehajtási jog (futtatási jog)

4.5.3. Jogosultságok megváltoztatása

Egy fájl tulajdonosi (hozzáférési) jogait csak a fájl tulajdonosa, vagy a rendszergazda tudja megváltoztatni a következő paranccsal: `chmod +|-<mód> <fájlnév>` Meg kell határozni az alábbiakat: adunk vagy elveszünk jogot (+: adunk, -: elveszünk), kinek/kitől (saját, csoport, mindenki más (ugo)), milyen jogot adunk (r w x / 4 2 1).

4.5.3.1. példa. Saját magunknak írási jog: `chmod u+w munka.tar.gz`

4.5.3.2. példa. Másoknak futtatási jog: `chmod o+x munka.tar.gz`

4.5.3.3. példa. Egyszerre több jogot is meg lehet változtatni: `chmod o+x,u+w munka.tar.gz`

4.5.3.4. példa. Mindenkinek minden jog: `chmod 777 munka.tar.gz` ugyanezt a funkciót valósítja meg a `chmod a+rwx munka.tar.gz`

4.5.3.5. példa. Csak nekem legyen jogom mindenhez: `chmod 700 munka.tar.gz` ugyanaz mint: `chmod u+rwx,g-rwx,o-rwx munka.tar.gz`

Fájlok esetében a végrehajtási jognak csak a futtatható fájloknál van jelentőségük (bináris állományok, scriptek). Könyvtárak esetén az olvasási jog azt jelenti, hogy elolvashatja a fájlok neveit az adott könyvtárban, az írási jog jelenti, hogy a könyvtárban állományt, könyvtárat hozhatunk létre, míg a futtatási jog megengedi a belépést a könyvtárba.

4.5.3.6. feladat. Hozzunk létre egy könyvtárat, és változtassuk meg a jogosultságait úgy, hogy aki ismeri a könyvtárban lévő fájlneveket, alkönyvtárakat, az el tudja ezeket olvasni, de más nem!

4.5.4. Alapértelmezett jogok

Amikor egy fájlt létrehozunk, akkor az a jogosultságok alapértelmezett értékével fog rendelkezni. Pl.: Létrehozunk egy üres fájlt:

```
$ touch akarmi
$ ls -la akarmi
-rw-r--r--  1 bnorbert  staff  0 Okt 31 06:14 akarmi
```

A létrehozáson kívül, alapértelmezés szerint írási és olvasási joggal, a csoportba tartozók és mindenki más pedig csak olvasási joggal rendelkeznek. Ennek az az oka, hogy az operációs rendszer a fájl létrehozásakor a 022 maszkot alkalmazza. Egy állomány létrehozásakor alapértelmezésben senki sem kap futtatási jogot. Az alapértelmezett maszk lekérdezhető a következő paranccsal:

```
$ umask
022
```

Könyvtárak létrehozása esetén a 777-ből vonódik ki a mask, azaz alapértelmezetten egy könyvtár 755 jogokkal jön létre. Fájloknál a 666-ból vonódik ki a mask, így 644 jogokkal jönnek létre a fájljaink.

4.5.5. Tulajdonos megváltoztatása

Az egyes bejegyzések (fájlok és könyvtárak) tulajdonosának megváltoztatása a chown parancs segítségével történik, valamint felhasználó csoport váltása a chgrp parancs segítségével lehetséges. A jogosultságokat is tartalmazó részletes listát a `ls -l` paranccsal kaphatunk (illetve kombinálhatjuk a már megismert `-i` kapcsolóval is, hogy az i-node azonosítók is láthatóak legyenek: `ls -li`

4.5.6. Fájlrendszerrel kapcsolatos parancsok

- Az fsck parancs segítségével lehet ellenőrizni, hogy a háttértár tartalma megegyezik-e az adminsztrációs fájlok által leírt állapottal, azaz a fájlrendszer koherens állapotban van-e. Ilyen például akkor fordulhat elő, amikor hirtelen kikapcsol a számítógép (pl. áramszünet esetén) és valamilyen lemezművelet félbeszakad. Szintén problematikus eset a fájlrendszer koherenciájának szempontjából, ha akkor távolítunk el egy cserélhető eszközt, amikor még nem fejeződött be a lemezre írás művelet.

- Fájlrendszer egy üres partícióra a mkfs parancs segítségével tudunk létrehozni, a parancs lefutása létrehozza az összes adminisztrációs állományt, ami szükséges a fájlrendszer menedzseléséhez. Hasonlóan, ha egy fájlrendszerrel rendelkező partíciót leformázunk, akkor a formázás létrehozza az üres adminisztrációs fájlokat (fontos, hogy ezzel még az előző fájlrendszerben tárolt adatok megmaradnak, csak nem tartozik hozzájuk adminisztrációs állomány).

- Az érvényes, hibamentes fájlrendszereket tartalmazó partíciókat használat előtt fel kell csatolnunk a könyvtárstruktúrába. Általában az mnt könyvtárban van egy - partícióhoz tartozó - üres könyvtár, ahová a mount paranccsal tudjuk becsatolni a partíciót.

4.5.7. Feladatok

- Mit csinál a df parancs? Keressük meg a man oldalán, hogy mit csinál a -T kapcsoló és futtassuk a df -T parancsot.

- Nézzük meg a stat parancs man oldalát, próbáljuk ki a következőekre: sima fájl, könyvtár, eszközfájl, soft link, hard link.

- Nézzük meg az ls -i parancsot, keressünk egy fájl a könyvtárból és nézzük meg, hogy az ls -i parancsban megadott inode szám egyezik-e a stat parancs kimenetével.

- Hozzunk létre soft és hard linkeket, fájlra, könyvtárra, figyelünk a link counter értékének változására. Töröljük azt a fájlt amire a link mutat mit tapasztalunk szoft illetve hard link esetén?
- nézzük meg a dumpe2fs parancsot és futtassuk az egyik partícióra. A grep parancs segítségével (grep -i superblock) nézzük meg hány példányban tárolódik a lemezen a szuperblokk.

4.6. Könyvtárszerkezet

Linux alatt fa gráfba van szervezve a teljes könyvtárszerkezet, (azaz ne számítsunk C, D, ... meghajtókra!) Mindennek az alapja a `/` jellel jelölt gyökérkönyvtár, más néven root. Ez minden fájlrendszer alapja, ebből ágaztatható le a teljes szerkezet.

4.6.0.1. példa. Adjuk ki a következő utasítást: `ls /` Hasonló listát kell látnunk:

```
bin
boot
cdrom
dev
etc
home
lib
lost+found
media
mnt
opt
proc
root
sbin
sys
tmp
usr
var
vmlinuz
```

Ezek a főkönyvtárak majdnem minden Linuxban változatlanul megvannak, leszámítva talán a /cdrom-ot és /media-t. A /media egy újabb „találmány”, ide kerülnek a cserélhető médiák. Nézzük, melyikben mi található:

bin, sbin: A bin könyvtárakban futtatható állományok vannak. Több bin könyvtár is található ezen kívül, például a /usr/bin és a /usr/sbin. Bár ez nem törvényszerű, de általában a bin könyvtárakban a minden felhasználó által elérhető programok kerülnek az sbin könyvtárakba pedig olyan rendszereszközök, melyeket általában rendszergazdák használnak. A /bin és /sbin az alaprendszerhez, a boot folyamathoz szükséges programokat tartalmazza, a felhasználói programok a /usr/bin /usr/sbin alá kerülnek.

boot: a boot könyvtárban találhatók a bootnál fontos fájlok: általában a rendszermag (kernel), illetve Grub rendszerbetöltő esetén annak konfigurációs állománya is.

cdrom:Ez alá csatolódik be a CD meghajtó egység.

dev: Linux alatt fájlokon keresztül érünk el mindent, a CD-vel kezdve, a hangon át, az egérig. Ezek a speciális eszközfájlok találhatók ebben a mappában.

etc: Az etc könyvtár a gyűjtőhelye a különböző programok globális konfigurációs fájljainak. Ellentétben a Windowsos registry megoldással, Linux alatt minden konfigurációs állomány egyszerű szövegfájlba van mentve, aminek nagy előnye, hogy az állományok akkor is egyszerűen elérhetők, ha a rendszer egyébként használhatatlan. Természetesen emellett az egyes programok felhasználó specifikus beállításokkal is rendelkeznek, ezeket a home könyvtárakban tárolja a rendszer, rejtett mappákban.

home: ez alatt a könyvtár alatt találhatók a felhasználói könyvtárak, az adott könyvtár alatt a felhasználónak teljes dúlási joga van, ezen az egy könyvtáron kívül azonban leginkább csak olvasási joga van alapból.

lib: a lib könyvtár alatt már a rendszer részei lapulnak: library fájlok, kernel modulok, stb.

lost+found: egy speciális könyvtár, jelen esetben egy ext3 típusú fájlrendszerrel szerelt partícióról van szó, ez a könyvtár nem is a Linux, mint inkább a fájlrendszer része: a fájlrendszer javításakor előkerült, névvel nem rendelkező fájl darabokat helyezi el itt a rendszer.

media: rendszerfüggő, általában a /media könyvtár alá kerülnek befűzésre a CD/DVD eszközök, pendrive illetve a floppy. Röviden: a cserélhető médiák.

mnt: a másik becsatolásra használt könyvtár. Ez alá a könyvtár alá kerülnek (általában) csatolásra a fix partíciók. Mivel ebben a könyvtárstruktúrában nincs kiemelt helye/neve egy egy meghajtónak, mint Windows alatt a C:, D:, stb., így egy-egy eszközt tetszőleges helyre befűzhetünk a fájlrendszerbe. Különösen praktikus ez például a home könyvtár estén: ha kinőjük az e célra fenntartott partíciót, és veszünk egy új vincsesztert, egyszerűen csak rámásoljuk anyagainkat, letöröljük az eredeti példányt, majd befűzzük a /home könyvtár alá az új adathordozót.

opt: a hivatalos leírás szerint külsős programok települnek ebbe a könyvtárba, de a rendszerek nagy részén üresen áll...

proc: Itt találhatóak az éppen futó folyamatokkal kapcsolatos metaadatok, illetve információk a rendszerről: processzorról, memóriáról, stb. Nagy mennyiségű hasznos információt talál itt az avatott kéz.

root: A rendszergazda (root) felhasználó home könyvtára

tmp: Az egyes programoknak szükségük van/lehet átmeneti fájlokra. Ezek kerülnek ide. Ez a másik olyan könyvtár, amely alapértelmezetten írható minden felhasználó számára.

usr: Ez alatt a könyvtár alatt található minden. Persze ez így kicsit túlzónak hat, de majdnem igaz: az usr könyvtár alatt található a telepített programok nagy része, hagyományból ide szoktunk forrásokat pakolni (/usr/src), és azt lefordítani. Itt találhatók a dokumentációk, itt találhatók az ikonok nagy része, stb...

var: Szintén számos szolgáltatás gyűjtőkönyvtára. Itt találhatók a naplófájlok, egyes programok hosszabb ideig tárolt, mégis átmeneti fájljai, alapértelmezetten a felhasználói postaládák, stb.

4.6.0.2. feladat. Nézzünk bele az egyes könyvtárakba: adjuk ki a következő utasítást (utánuk ENTER): `ls /bin` (aztán `ls /boot`, `ls /home`,...)

4.6.0.3. feladat. Gépeljük be, majd nyomjunk ENTER-t: `cat /proc/meminfo`

4.7. Egyéb parancsok

Néhány gyakran használt, fontosabb parancs:

date: kiírja az aktuális dátumot.

df: disk free, egy kis statisztikát jelenít meg az egyes partíciók foglaltságáról. pl.: `df -h`

du: disk usage, az egyes állományok, könyvtárak méretéről készít kis statisztikát. pl.: `du -hs ./` (a man alapján próbáljuk meg értelmezni az egyes kapcsolókat, paramétereket!)

ncal: calendar, egy kis naptár program. pl.: `ncal 2011`

Természetesen a listát még hosszasan lehetne sorolni, aki további parancsokkal szeretne megismerkedni, használja ki az internet lehetőségeit! Bármely kereső a „linux parancsok” kifejezésre több jól használható oldalt is ajánl.

4.8. Feladatok

4.8.0.4. feladat. Nézz utána, hogy mit csinál az ncal parancs!

4.8.0.5. feladat. A hét milyen napján születtél?

4.8.0.6. feladat. Mekkora helyet foglalsz a users.itk.ppke.hu szerveren?

4.8.0.7. feladat. Hozd létre a következő könyvtárstruktúrát a saját könyvtáradon belül!

```
./szulok/apa
./szulok/anya
```

4.8.0.8. feladat. Hozz létre egy fájlt (akár üreset is lehet) az apa alkönyvtáron belül! (touch, esetleg nano, esetleg cat,. . .)

4.8.0.9. feladat. Másold át az anya alkönytárba!

4.8.0.10. feladat. Írasd ki egy fájlba az elmúlt 10 percben módosított fálok neveit a munkakönyvtáradon belül! (find parancs)

4.8.0.11. feladat. Fűzd hozzá a fájl végéhez az aktuális dátumot! (date és átirányítás)

4.8.0.12. feladat. Módosítsd az előző fájl jogait, hogy neked csak írási jogod, másoknak (csoport, egyéb) pedig semmilyen joga ne legyen!

4.8.0.13. feladat. Próbáld meg a tartalmát kilistázni! (pl.: cat)

4.8.0.14. feladat. Szerezz információkat az od programról! (man, keresők,. . .)

4.8.0.15. feladat. Add ki a következő utasítást:
verb=echo ő | od -t x1
Értelmezd az eredményt!

5. fejezet

Hálózatok és protokollok

5.1. Egy kis történelem

5.1.1. A kezdetek

Az igény, hogy a számítógépek egymással valamiféle összeköttetésben legyenek, szinte egy idős az első elektronikus számítógépekkel. Kezdetben önálló, szinte egész termeket kitöltő számítógépeken dolgoztak az emberek. Korán megjelent az igény, hogy az egyik gépen megtalálható adat, program minél könnyebben átvihető legyen egy másik gépre anélkül, hogy ehhez külső adathordozót kelljen igénybe venni. A számítógépek méretének és árának csökkenésével egyre inkább elterjedt az a modell, hogy nem egy hatalmas gépen dolgoztak a felhasználók, hanem több kisebb számítógép volt például egy cég irodaházában. Mivel fizikailag egymáshoz közel voltak, jogos igény volt, hogy a viszonylag ritkán használt de drága perifériákból ne kelljen minden géphez külön-külön beszerezni egy példányt (pl.: nyomtató), hanem közösen használhassanak egy ilyen eszközt. Tehát a számítógépes hálózatok létrehozásának célja:

- Lehetővé teszi az erőforrások megosztását. A rendszerben levő erőforrások (háttértárak, nyomtatók, scannerek, egyéb perifériák) a jogosultságtól függően elérhetők bárki számára.
- Nagyobb megbízhatóságú működést eredményez, hogy az adatok egyszerre több helyen is tárolhatók, az egyik példány megsemmisülése nem okoz adatvesztést. Az azonos funkciójú elemek helyettesíthetik egymást. (Több nyomtató közül választhatunk.)
- Gazdaságosan növelhető a teljesítmény. A feladatok egy nagyszámítógép helyett megoszthatók több kisebb teljesítményű eszköz között. Sőt, egyes esetekben magát a nagy teljesítményű szervert is helyettesíthetik (cluster computing).
- Elérhetővé válnak a központi adatbázisok. Ezek az adatbázisok sok helyről lekérdezhetők, és sok helyről tölthetők. Csak így képzelhető el pl. egy valóban aktuális raktár vagy megrendelés állomány kezelés egy nagyvállalatnál.
- A hálózati rendszer kommunikációs közegként is használható (IP telefon, üzenetküldő szolgáltatások, email).

5.1.2. Az ARPA project

Az 1960-as évek közepén (dúl a hidegháború) az Amerikai Védelmi Minisztérium (U. S. Department of Defense) olyan parancsközlő hálózat kialakítását tűzte ki célul, mely átvészel egy esetleges atomcsapást. A fejlesztéseket a minisztérium ösztöndíjakkal támogatta. Az elméleti kutatások után olyan hálózat kialakítására írtak ki pályázatot, amely csomóponti gépekből áll, adathálózat köti ezeket össze és néhány csomópont megsemmisülése esetén is működőképes marad a hálózat többi része. A tenderre több cég is nevezett, a győztes 1969-ben állította üzembe az

[0] $Revision : 48$ ($Date : 2013 - 10 - 0707 : 32 : 38 + 0200(Mon, 07Oct2013)$)

első csomópontot, 1972-re 37-re nőtt a csomópontok száma. Ekkoriban kapta az ARPAnet nevet ez a hálózat (Advanced Research Project Agency). A 70-es évek végére összeköttetések épültek ki más helyi hálózatok és az ARPAnet között, mára ez a hálózat behálózza az egész Földet. A 80-as évektől nevezik a hálózatok ezen hálózatát internetnek.

5.2. Rétegezett felépítés

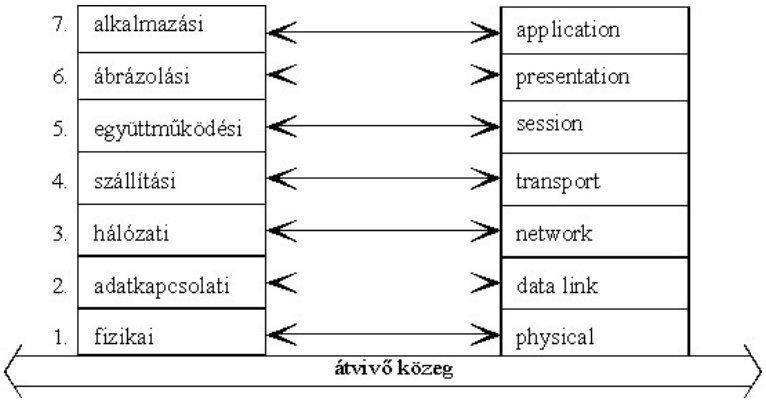
5.2.1. Okok és célok

Bizonyára el tudjuk képzelni, hogy a fentebb vázolt hálózatokon a kommunikáció meglehetősen összetett és bonyolult dolog. Nem lenne szerencsés, ha a programozónak olyan hálózati kommunikációra képes programokat kellene írnia, amely a teljes kommunikáció minden aspektusát megoldja. Ugyanannak a programnak kellene gondoskodnia a megfelelő feszültségszintek előállításától kezdve a megcímzett gép azonosításáig mindenről. Ha a hálózati működés valamely részén változtatnak, akkor az egész program módosítására szükség lehet. Ennek elkerülése érdekében a hálózati kommunikáció folyamatát logikailag több részre bontjuk: az egyes részek a folyamat egy jól meghatározott részéért felelnek, azt kell megvalósítaniuk. Csak arról kell gondoskodni, hogy az egyes részek (rétegek) egymást megértsék: egy jól definiált interfészt kell egymás felé mutatniuk.

Például a postai levelezést mint kommunikációs hálózatot tekintve, a postaládát kiürítő dolgozónak nem kell tudnia repülőt vezetni vagy hajót építeni és átszelni az óceánt, ha oda szól a levél, neki csak a ládát kell tudnia kiüríteni (de azt hiba nélkül) és el kell juttatnia a borítékokat a megfelelő helyre. Az ottani dolgozónak pedig nem kell tudnia, hogy a város mely részén vannak ürítendő postaládák és azokat hogyan kell kinyitni, neki csak a küldeményeket kell bizonyos szempontok szerint szétválogatnia, stb. Hasonló módon az egyes rétegek szolgáltatásait implementáló programozóknak sem kell az egész kommunikációs problémát egyben vizsgálniuk, nekik elég csak az adott rétegre koncentrálniuk. Feltéve persze, hogy azért van valaki, aki átlátja a teljes hálózati működést, és úgy tervezi meg az egyes rétegeket illetve a közöttük lévő interfészeket, hogy összességében a hálózat a kívánalmaknak (specifikációnak) megfelelően működjön.

5.2.2. ISO OSI

A Nemzetközi Szabványügyi Szervezet (International Organization for Standardization, ISO[1]) létrehozta az OSI (Open Systems Interconnection - nyílt rendszerek összekapcsolása) modellt (ISO/IEC 7498), ami hét rétegre bontja logikailag a számítógép hálózatok működését (lásd az 5.1. ábrán).



5.1. ábra. Az OSI rétegek

Minden egyes réteg az alatta lévő réteg szolgáltatásait veszi igénybe, és annak segítségével tud kommunikálni. A kommunikáció célja ugyanakkor, hogy egy másik hálózatba kötött eszköz

[1] a rövidítés nem a szervezet elnevezésének rövidítése, mivel az sok nyelven másképp hangzana, hanem a görög *isos* szóból származtatták (jelentése: egyenlő) lásd: http://www.iso.org/iso/about/about

ugyanilyen rétegével adatot cseréljen, aminek a specifikációját a kettejük közötti *protokoll*nak nevezzük. Az alsóbb réteg feladata, hogy ezt lehetővé tegye (ami ennek megvalósításához igénybe veheti az azalatti réteg szolgáltatásait, és így tovább). A két, egymás alatti réteg közötti kommunikáció az *interfész*en keresztül történik. A egyes hálózati eszközök közötti tényleges fizikai jeltovábbítás a fizikai réteg segítségével, az átviteli közegen zajlik.

Segítheti a megértést, ha két magas rangú államférfira gondolunk, akik tolmácsok segítségével kommunikálnak egymással: az egyik vezető a saját tolmácsának mondja, az elmondja a másik tolmácsnak, az lefordítja a saját főnökének. Kik kommunikálnak egymással? Bár a vezetők beszélni csak a saját tolmácsaikkal beszélnek (illetve a tolmácsok egymással), de mégis a két államférfi cserél eszmét a beszélgetés során.

5.3. Az egyes rétegek feladata

Ebben a részben a hétköznapokban manapság leggyakrabban használt technológiákat tekintjük át: az Ethernetet és a TCP/IP protokollcsaládot. Természetesen ettől eltérő protokollok használatára is lehetőség van, ezekről bővebben a Számítógép hálózatok felsőbbéves tárgyban lesz szó.

5.3.1. A fizikai réteg

Az adatokat (biteket) valamilyen fizikai jellé alakítva az adott átviteli közegen tudni kell továbbítani illetve fogadni. A fizikai réteg tehát meghatározza az átviteli közeget, annak elektromos-, egyéb jellemzőit, az esetleges csatlakozók méretét, formáját, a bekötés módját, a használható frekvenciákat, az alkalmazott kódolást, az esetleges ütközések érzékelésének módját, stb, azaz minden olyan paramétert, aminek specifikálása ahhoz szükséges, hogy a fizikai közegen biteket lehessen továbbítani két eszköz között. Két nagy csoportját különböztetjük meg a jeleknek:

analóg : az adott jellemző (megadott határok között) bármilyen értéket felvehet

digitális : az adott jellemző csak fix értékeket vehet fel

bináris : két lehetséges értéket vehet fel

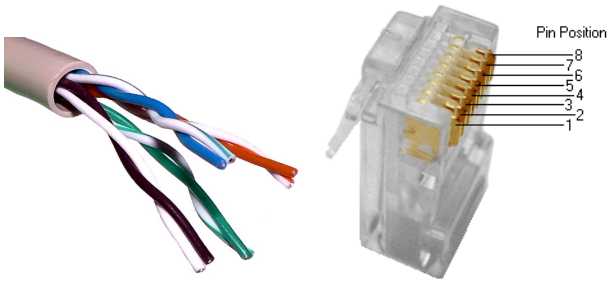
Gyakori, hogy a közeg analóg jelek továbbítására képes, emiatt meg kell oldani a digitális jelek analóg jellé konvertálását (DAC: digitális→analóg konverter), majd a vonal túlvégén az érkező analóg jeleket digitalizálni kell (ADC: analóg→digitális konverter). Ilyen konvertereket használunk például az audio CD-n tárolt digitális jelek analóggá konvertálásakor (DAC, például egy digitális bemenettel rendelkező rádióerősítő is tartalmaz ilyet), illetve a mikrofonnal érzékelt analóg jel digitálissá konvertálásakor (ADC).

Fontos fogalom az átviteli sebesség (sávszélesség), ami megadja az egy másodperc alatt átvitt bitek számát, mértékegysége a bps (bit per secundum): például 10 bps a sebesség, ha 10 bit adat továbbítódik egy másodperc alatt. Elterjedt prefixált mértékegységek még a kbps (ezer bit per másodperc), az Mbps illetve a Gbps hasonlóan megabit illetve gigabit egységekben. Nem összekeverendő a Bps-el, ami bájtban adja meg az átvitt adatmennyiséget.

UTP

UTP (Unshielded Twisted Pair: árnyékolatlan csavart érpár). Felhasználóként ezzel a kábellel találkozunk napjainkban a legtöbbet. 8 szál vezeték párosával egymás köré tekerve alkotja a kábelt, külön árnyékolás nélkül. A vezeték párok egymás köré tekerésével csökkentik a környezet zavaró jeleit, a megcsavarás számának változtatásával pedig a párok közötti áthallás csökkenthető (lásd az 5.2. ábrán).

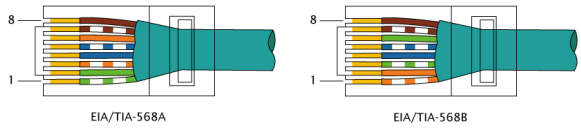
A csatlakozóban az egyes vezetékek sorrendje fontos, mivel azok kettesével vannak csavarva: nem elegendő az, hogy például a csatlakozó négyes pinje a kábel másik végén lévő csatlakozó négyes pinjéhez csatlakozik, illetve, hogy az ötös pin az ötös pinhez, hanem az is fontos, hogy a négyes és az ötös pinekhez csatlakozó vezetékek ugyanannak a vezetékpárnak a tagjai legyenek. Sőt, az sem mindegy, hogy az egyes érpárok közül melyek kerülnek egymás mellé, mivel az érpár



(a) UTP kábel (b) 8P8C csatlakozó

5.2. ábra. UTP kábel és csatlakozó

egyik fele a A leghasznosabb követni egy szabványos színkód alapú bekötési módot: ilyenből kétféle is létezik: ismertebb nevükön TIA/EIA-568A és TIA/EIA-568B, de mindkettő része a nemzetközi ISO/IEC 11801 szabványnak (lásd 5.3. ábrán).



(a) TIA/EIA 568A (b) TIA/EIA 568B

5.3. ábra. TIA/EIA 568 bekötési módok

Azért van belőlük kettő, mert a régebbi Ethernet szabványoknál ha két gépet közvetlenül akartunk összekötni, akkor a kábel egyik végére az egyik, a másik végére a másik változat szerinti kiosztás szerint kellett az érpárok bekötését elvégezni, amit *cross link* kábelnek nevezünk. Manapság is lehet ilyen kábeleket kapni hálózati termékeket forgalmazó szakboltokban, de a modern eszközök (újabb hálózati kártyák) automatikusan érzékelik, hogy milyen kábelt csatlakoztattunk, így használhatók cross link és straight-through kábellel is.

Az UTP kábeleket több kategóriába sorolják a paramétereik szerint (sintén az ISO/IEC 11801 illetve a vonatkozó TIA/EIA 568 szabványokban): CAT3: 16 MHz, CAT4: 20 MHz, CAT5: 100 MHz, CAT6: 250 MHz, CAT6A: 500 MHz, CAT7: 600 MHz, CAT7A: 1000 MHz frekvenciáig bevizsgált és minősített kábel (lásd az 5.4. ábrán).

5.4. ábra. Egy UTP kábelen feltüntetett jelölések (többek között a kategória)

5.3.2. Az adatkapcsolati réteg

Az adatkapcsolati réteg (Data Link Layer) alapvető feladata, hogy egy bitfolyam átvitelére képes fizikai rendszert egy olyan eszközzé alakítsa, ami adatintegritás ellenőrzött kommunikációt tesz lehetővé két, helyi hálózatba kötött eszköz között. Az adó oldal a bemenő adatokat meghatározott hosszúságú darabokra – *keret*ekké – tördeli, majd a protokollnak megfelelő kiegészítő információkkal egészíti ki (pl. a keretek előtt és mögött speciális bitmintákat helyez el a keret elejének és végének felismeréséhez; a hálózati eszközök címzéséhez szükséges adatokat fűz hozzá; adatintegritás ellenőrző kódokkal egészíti ki). A vevő oldal a fogadott adatkereteket a megfelelő részek értelmezése és levágása után továbbítja a felsőbb rétegnek.

Az ethernet

Környezetünkben legelterjedtebben az Ethernettel találkozhatunk (IEEE 802.3 szabványcsalád), ami egy lokális hálózati (LAN) protokoll, azaz Ethernettel csak ugyanabban az Ethernet hálózatban létesíthető kommunikáció egy másik eszközzel! Az Ethernet a MAC címmel biztosítja a hálózati eszközök címezhetőségét: minden egyes keret tartalmazza a küldő állomás MAC címét és a címzett MAC címét. A címzett MAC cím jelölhet egyetlen eszközt, de jelölhet több eszközt is (ebben az esetben *broadcast* címről beszélünk). A MAC cím az egész világon egyedi szám, 6 bájtból áll, 12 jegyű hexadecimális szám formájában szoktuk megadni. Pl.: 38:60:77:df:94:f3. Napjaink hálózati eszközei már lehetőséget biztosítanak arra, hogy a MAC címet megváltoztassuk, így az egyediség nem feltétlenül biztosított.

Az Ethernet manapság leggyakrabban használt verzióiban (100BASE-TX, 1000BASE-T) egy központi egységhez csatlakoznak az állomások egyesével, UTP kábellel (csillag topológiában). Ennek egyik előnye, hogy kábelhiba esetén csak az érintett gép esik ki a kommunikációból, továbbá amiatt, hogy a kábelhez (az átviteli közeghez) csupán két eszköz kapcsolódik, lehetőség van *duplex*[2] átvitelre: az átviteli közegen mindkét hálózati eszköz egyszerre tud kommunikálni úgy, hogy egymás adását nem zavarják, nem korlátozzák, mintha egy-egy párhuzamos csatornán zajlana a két irányú kommunikáció.

A különböző UTP kábelekből a következő, leggyakrabban használt Ethernet hálózatok építhetők:

- 10BASE-T: 10 Mb/s sebességű Ethernet 2 érpáron, standard telefon kábelen (IEEE Std 802.3 Clause 14.4.1)
- 100BASE-TX: 100 Mb/s sebességű Ethernet 2 érpáron, CAT5 kábelen (IEEE Std 802.3 Clause 25.4.9)
- 1000BASE-T: 1000 Mb/s (=1Gb/s) sebességű Ethernet 4 érpáron, CAT5 kábelen (IEEE Std 802.3 Clause 40.7.1)
- 10GBASE-T: 10Gb/s sebességű Ethernet 4 érpáron, CAT6, CAT6A, CAT7, CAT7A kábelen (IEEE Std 802.3 Clause 55.7.1)

A központi egység régebben hub volt, ma már legtöbbször switch. A két eszköz közötti különbség a következő: A hub az egyik csatlakozóján beérkező jelet az összes csatlakozón újra kiküldi, azaz a jelszintek és időzítések helyreállítása után egyszerűen csak megismétli azokat. Ennek egyik hátránya, hogy minden keret minden egységhez eljut, aminek biztonsági kockázata lehet (harmadik félhez is eljutnak két másik fél közötti adatok), a másik hátránya pedig, hogy a hálózat által biztosított sávszélességet nem hatékonyan használjuk fel, hiszen olyan keretek is közlekednek a kábelen, amelyek nem szólnak a kábelen lévő egyik egységnek sem, és nem is tőlük érkeznek. A switch erre megoldást kínál: a keretekben található fizikai cím (MAC) alapján eldönti, hogy mely csatlakozóján (portján) küldi tovább az adatot. A switch működése egy dinamikusan karbantartott táblázatra épül, amelyben a kapcsoló minden portjához feljegyzi az adott porton beérkező keretek küldőjének MAC címét. Ezzel a kapcsoló megismeri a hozzá kapcsolódó gépek helyzetét, tehát azt, hogy az egyes gépek a kapcsoló melyik interfészéhez kapcsolódnak. Egy beérkezett keret továbbításához csak meg kell vizsgálnia a táblázatot, hogy

[2] néhol full duplexnek hívják ezt a kommunikációs módot

a keretben szereplő cél MAC cím melyik interfészén keresztül érhető el. Egy interfészhez több gép MAC címe is feljegyezhető, ezért nincs akadálya hub-switch vagy switch-switch kapcsolatnak sem. A switch bekapcsolásakor kezdődő tanulási folyamat során fokozatosan alakul ki a kapcsoló táblázata, ezért normál jelenségnek tekinthető, ha egy olyan keretet kell továbbítania, amelynek a címzettje még a switch számára ismeretlen irányban van. Ekkor az ún. elárasztást alkalmazza, azaz a beérkezett keretet a fogadó port kivételével az összes többi portján kiküldi (és az erre érkező válaszból fogja megtanulni, hogy az az eszköz melyik portján érhető el).

Nagyon fontos kiemelni, hogy az Ethernet bármely más protokoll alkalmazása nélkül is lehetőséget ad a kommunikációra egy Ethernet hálózaton belül. Probléma akkor merül csak fel, ha több ilyen hálózatot kell összekapcsolni és közöttük is biztosítani kell az adatok továbbítását. Mivel az Ethernet hálózatok mérete korlátozott, erre előbb-utóbb szükség lesz, nem is beszélve arról, ha az internethez szeretnénk kapcsolni a helyi hálózatunkat.

5.3.3. A hálózati réteg

A hálózati réteg feladata a csomagok eljuttatása a forrástól a célig úgy, hogy azok akár több lokális hálózaton (LAN-on) is áthaladhatnak. Pontosan ez különbözteti meg az adatkapcsolati rétegbeli protokolloktól (pl. Ethernet), amik egy hálózaton belül képesek keretek célba juttatására. A célig egy csomag valószínűleg több csomópontot is érint, sőt, az is elképzelhető, hogy több párhuzamos útvonal is létezik. Az útvonal megválasztásához természetesen (részben) ismerni kell az átviteli hálózat felépítését, azaz a topológiáját, és ki kell tudni választani egy megfelelő útvonalat. A csomagoknak tartalmazniuk kell mind a forrás, mind a cél hálózati címet (ami általában különbözik az adatkapcsolati rétegben alkalmazott címtől!). A következőkben az IP-vel (Internet Protocol), mint hálózati rétegbeli protokollal foglalkozunk, a pontos protokoll specifikáció megtalálható a http://tools.ietf.org/html/rfc791 oldalon.

IP (v4) címek

Ahhoz, hogy a hálózati réteg megtalálja a csomagok címzettjét, minden gépnek rendelkeznie kell egy egyedi címmel, ez az IP cím. A jelenleg (még) legelterjedtebb IPv4 szerint ez a cím egy 32 jegyű bináris szám, amit a jobb olvashatóság miatt 8 bitenként decimális számmá alakítunk (pontozott négyes jelölés [dotted decimal notation]), például: 193.224.69.67.

A hálózati eszközöket fizikai vagy logikai szempontok alapján alhálózatba soroljuk. Az egy alhálózatban lévő gépek egymással közvetlenül (azaz hálózati rétegbeli útválasztók nélkül) tudnak kommunikálni. Annak eldöntése, hogy két gép egy alhálózatban található-e, az IP címükből eldönthető. Kezdetben a lehetséges címeket osztályokba sorolták, így beszélhetünk A, B, C, stb. osztályú címekről.

- A osztály: az első bit (a 32 -ből) 0, és 8 bit azonosítja a hálózatot (ebből a már említett első bit fix), a maradék 24 bit a hálózaton belül az egyes hálózati eszközöket. A legkisebb ilyen hálózat azonosító a 0 lehet, a legnagyobb a 127. Egy hálózaton belül kb. 2^{24} eszköz címezhető meg.
- B osztály: a cím első két bitje 10, és 16 bit azonosítja a hálózatot (ebből a már említett első két bit fix), a maradék 16 bit az egyes hálózati eszközöket. Így a legkisebb hálózatszám a 128.0 lehet, a legnagyobb a 191.255. Egy hálózaton belül kb. 2^{16} eszköz címezhető meg.
- C osztály: a cím első három bitje 110, és 24 bit azonosítja a hálózatot (ebből a már említett első három bit fix), a maradék 8 a hálózaton belül az egyes hálózati eszközöket. A legkisebb ilyen hálózatcím a 192.0.0, a legnagyobb a 223.255.255. Egy hálózaton belül kb. 2^8 eszköz címezhető meg.
- A többi osztállyal (a fennmaradó IP címekkel) jelen jegyzetben nem foglalkozunk, speciális célokra fenntartottak.

Minden hálózaton belül található két speciális cím: a *network address*, ami csupa 0 host részből áll, és a *broadcast address*, ami a csupa 1 host részből áll. Mivel ezeket a címeket nem lehet egyik gépnek sem kiosztani, minden egyes hálózatban az elméletinél kettővel kevesebb IP cím osztható ki hálózati eszközöknek.

5.3.3.1. példa. Az 19 -es hálózatban (A osztályú cím) a network address: 19.0.0.0, a broadcast address: 19.255.255.255, a kioszthető gépek száma: $2^{24} - 2$

5.3.3.2. példa. Az 155.13 -as hálózatban (B osztályú cím) a network address: 155.13.0.0, a broadcast address: 155.13.255.255, a kioszthető gépek száma: $2^{16} - 2$

5.3.3.3. példa. Az 210.15.9 -ás hálózatban (C osztályú cím) a network address: 210.15.9.0, a broadcast address: 210.15.9.255, a kioszthető gépek száma: $2^8 - 2$

Minden osztályban kijelöltek olyan címtartományt[3], amelyek a nyílt interneten nem használhatók (nem „publikusak"). Ezeket a privát címeket egy-egy belső alhálózatban lehet használni, de ilyen címek az internetre nem továbbíthatók, illetve nem is érkezhetnek onnét. Azaz ilyen privát IP címmel lehet, hogy egy időben több számítógép is rendelkezik a világon, de ebben a formában nem tudnak kommunikálni a külvilággal (és egymással sem). A tartományok a következők:

- A osztályban: 10
- B osztályban: 172.16–172.31
- C osztályban: 192.168.0–192.168.255

A publikus tartomány mellett két A osztályú speciális hálózatot[4] kell még megemlíteni: a 0 és a 127 hálózati azonosítójút: Az előbbi a „saját hálózatom" azonosítására szolgál, és csak abban a speciális esetekben továbbítható, ha az IP cím használata kötelező, de mégsem tudjuk azt a valós (érvényes) értékkel kitölteni. A 127-es hálózat a *loopback* hálózat, bármelyik címe pedig a „saját magam" IP címe, azaz olyan esetekben használjuk, amikor ugyanannak a gépnek akarunk küldeni valamit, amin éppen dolgozunk. Például fut egy program a saját gépünkön, és ki akarjuk próbálni, hogy működik-e: ekkor nem kell a gép valódi IP címét használni, hanem elegendő például azt, hogy 127.0.0.1. Másik példa: ha felinstalláltunk egy web szervert a gépünkre, akkor a 127.0.0.1 címet beírva a böngészőbe, kipróbálhatjuk, hogy működik-e, anélkül, hogy ehhez IP címet kellene a gépnek kiosztani.

A jelen jegyzetben tárgyalt speciális tartományok közül még egy fontos, a 169.254 számú hálózat[5]: ezt autokonfigurációs esetben használja az operációs rendszer: ha más módon a gép IP címe nem deríthető ki, de helyi hálózatban mégis használni szeretnénk a gépet mindenféle kézi beállítás nélkül.

A `ping` paranccsal megpróbálhatunk elérni egy IP címet: a megadott IP című gép általában válaszol a PING kérésre, így még a gép elérési idejéről is információt szerezhetünk (mennyi időbe kerül a PING csomagot a gépnek eljuttatni és onnét a választ megkapni [round trip time]):

```
bercin@users:~$ ping 173.194.44.55
PING 173.194.44.55 (173.194.44.55) 56(84) bytes of data.
64 bytes from 173.194.44.55: icmp_req=1 ttl=51 time=15.4 ms
64 bytes from 173.194.44.55: icmp_req=2 ttl=51 time=15.4 ms
64 bytes from 173.194.44.55: icmp_req=3 ttl=51 time=15.5 ms
^C
--- 173.194.44.55 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 15.487/15.501/15.522/0.102 ms
bercin@users:~$
```

A ping működik a már említett 127-es (loopback) hálózatra is:

```
bercin@users:~$ ping 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_req=1 ttl=64 time=0.038 ms
64 bytes from 127.0.0.1: icmp_req=2 ttl=64 time=0.030 ms
64 bytes from 127.0.0.1: icmp_req=3 ttl=64 time=0.028 ms
```

[3] RFC 1918 - http://tools.ietf.org/html/rfc1918
[4] RFC 1122 - http://tools.ietf.org/html/rfc1122
[5] RFC 3927 - http://tools.ietf.org/html/rfc3927

```
64 bytes from 127.0.0.1: icmp_req=4 ttl=64 time=0.039 ms
^C
--- 127.0.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2997ms
rtt min/avg/max/mdev = 0.028/0.033/0.039/0.008 ms
bercin@users:~$
```

Az elérési idő nagyságrendekkel kevesebb, ami abból következik, hogy a loopback IP címek nem hagyják el a gépet, azt az operációs rendszer kezeli.

Kezdetben az egyes hálózati eszközök tehát az IP címből meg tudták állapítani, hogy két gép egy alhálózaton van-e (a fenti módszerrel). Viszont már elég korán felismerték annak a veszélyét, hogy így nem túl gazdaságos a címek kiosztása, pl. ha valaki megszerzett egy A osztályú hálózatot, akkor $2^{24}-2$ darab különböző gépet helyezhetne el benne, de ennyire valószínűleg nincs szüksége. Felmerült az igény arra, hogy ezek helyett az alapértelmezett hálózatszámok helyett szabadabban lehessen gazdálkodni a még meglevő címekkel.

A CIDR hálózati osztályozás

A CIDR használatával már nem dönthető el pusztán a hálózat címéből, hogy hány bit alkotja a hálózati részt és hány bit a host részt (ahogy azt az osztály alapú címzésnél tettük). Az IP cím mellé meg kell adni egy újabb 32 jegyű bináris (32 bites) számot, amely az elején csupa 1-t tartalmaz, utána csupa 0-t, amit *netmask*nak nevezünk. Ezt szintén pontozott négyes jelöléssel írjuk le, például: 255.255.255.0.

A network addresst úgy kaphatjuk meg, hogy a netmaskkal és egy IP címmel bináris ÉS műveletet végzünk, míg a broadcast address a netmask negáltja és az IP cím bináris VAGY művelet végzésével számítható ki.

5.3.3.4. példa. Adott egy gép a 193.224.69.67 IP címmel és 255.255.255.0 netmaskkal. A network address kiszámítása:

IP	193.224.69.67	11000001111000000100010101000011
netmask	255.255.255.0	11111111111111111111111100000000
bináris ÉS művelet		
network address	193.224.69.0	11000001111000000100010100000000

5.3.3.5. példa. Adott egy gép a 193.224.69.67 IP címmel és 255.255.255.0 netmaskkal. A broadcast address kiszámítása:

IP	193.224.69.67	11000001111000000100010101000011
netmask negáltja	255.255.255.0	00000000000000000000000011111111
bináris VAGY művelet		
broadcast address	193.224.69.255	11000001111000000100010111111111

Sokkal kényelmesebb leírást, használhatóságot biztosít az úgynevezett CIDR (Classless Interdomain Routing) jelölés, amely esetén a hálózat pontozott négyes jelölése helyett az alhálózati maszk egyeseinek a számát adják meg. Például az előző példában szereplő 193.224.69.67 IP című és 255.255.255.192 netmaskkal rendelkező gép a rövidített CIDR jelöléssel: 193.224.69.67/26. Érdemes kipróbálni a következő oldalt: http://www.fport.hu/index.php?site=cidr

5.3.3.6. feladat. A CIDR kompatibilis a régi, osztály alapú címzéssel? Ha nem, miért? Ha igen, adjuk meg az egyes osztályokhoz tartozó netmaskokat!

5.3.3.7. feladat. Hány gép címezhető meg egy 255.255.255.128 netmaskú hálózatban?

5.3.3.8. feladat. Mennyi legyen a netmask, ha legalább 20, de legfeljebb 30 gép számára szeretnék IP címeket kiosztani?

Az ARP protokoll

Ha Ethernet hálózatról van szó, akkor az IP címen kívül az adatok tényleges elküldéséhez szükség van a címzett MAC címére is, mivel az adatkapcsolati réteg szintű kommunikáció Ethernet protokollal történik (emlékezzünk vissza a rétegezett felépítésre), így az Ethernet keretet ki kell töltenie, amiből a saját MAC címünket nyilván ismerjük, így csak a cél gép MAC címére van szükség.

Egy IP címhez tartozó MAC cím kiderítése az ARP (Address Resolution Protocol) protokoll feladata: Az A gép egy speciális Ethernet keretet küld a hálózatra, amely minden géphez eljut (broadcast), és az a gép, amelyiknek a keretben szereplő IP cím a saját IP címe (B gép), egy válasz keretet küld. Mivel a válasz is egy Ethernet csomag, és abban ki kell tölteni a forrás MAC címet (azaz B MAC címét), az A gép amikor megkapja a válasz csomagot, egyben megtudja az IP címhez tartozó MAC címet is. Mindezek után az eredetileg szándékozott adat elküldése már IP szinten is lehetségessé válik.

Az operációs rendszer az ARP válaszok fogadásakor egy belső táblázatot (ARP cache) tart karban, és ennek adatait használja az IP címhez tartozó MAC cím meghatározásakor, hogy ne kelljen minden egyes csomag küldésekor az ARP kérdés-válasz kommunikációt lejátszani. Ha a kérdéses MAC cím nem szerepel az ARP táblában, akkor az ARP kérdés-válasz lejátszódik; ha szerepel, akkor az Ethernet csomag egyéb kommunikáció nélkül kitölthető. Az ARP cache tartalmát lekérdezhetjük az arp paranccsal:

```
NorbiMBPr:trunk bnorbert$ arp -a
? (10.0.1.1) at b8:c7:5d:cf:59:2e on en0 ifscope [ethernet]
appletv (10.0.1.105) at b8:c7:5d:cf:59:2e on en0 ifscope [ethernet]
jcidev.vmnet (192.168.153.130) at 0:c:29:b0:55:d6 on vmnet8 ifscope [ethernet]
NorbiMBPr:trunk bnorbert$
```

A számunkra jelenleg érdekes részek a zárójelben lévő IP címek és az utánuk következő (hozzájuk rendelt) MAC címek.

Az útvonal meghatározása (routing)

Ha a forrás IP és a cél IP ugyanabban a hálózatban (network address) vannak, akkor a csomag a célgépnek közvetlenül elküldhető:

5.3.3.9. példa. Az A gép IP beállítása: 193.224.69.67/26, és szeretne kommunikálni a 193.224.69.121 című géppel (B gép). A saját hálózatunkban van? Először számítsuk ki az A gép network addressét:

IP A	11000001111000000100010101000011	193.224.69.67
netmask	11111111111111111111111111000000	255.255.255.192
network address	11000001111000000100010101000000	193.224.69.0

Majd számítsuk ki a B gép network addressét a saját hálózatunk netmaskját felhasználva (azért a saját hálózatunk netmaskját használjuk, mert azt akarjuk eldönteni, hogy a célgép ebben a hálózatban van-e):

IP B	11000001111000000100010101111001	193.224.69.121
netmask	11111111111111111111111111000000	255.255.255.192
network address	11000001111000000100010101000000	193.224.69.0

Mivel a két network address megegyezik, a két gép egy hálózatban (a saját hálózatunkban) van, így közvetlenül kommunikálhat vele az alsóbb rétegbeli protokoll segítségével (Ethernet).

Ha a cél IP cím nem a forrás IP címmel azonos hálózatban van, akkor a csomagot egy routernek (útvonal választónak) kell továbbítania, ennek a dolga valahogyan azokat a címzetthez eljuttatni. Ahhoz, hogy ez megtörténhessen, a router IP címét is ismerni kell.

5.3.3.10. példa. A kommunikáció kezdeményezője legyen megint a fenti példában szereplő 193.224.69.67/26 című gép. Kezdeményezzen kommunikációt a 193.224.69.51 IP című géppel.

Egy alhálózatban vannak? Ehhez a címzett címének veszi az első 26 jegyét: 11000001111000000100010100 (pontozott decimális alakban: 192.224.69.0). Látható, hogy különböznek, azaz nem egy alhálóban vannak, így a routernek kell küldeni a csomagot.

5.3.3.11. feladat. Mi az oka annak, hogy ha a saját hálózatunk netmaskját használjuk fel annak eldöntésére, hogy a célgép a saját hálózatunkban van-e, akkor nem követünk el hibát?

5.3.3.12. feladat. Lehetséges-e, hogy a router nem a saját alhálózatunkban van?

Lehetséges, hogy egy gép nem csak egy hálózati rétegbeli hálózathoz csatlakozik, ezért minden eszköznek van routing táblája, amely arról tartalmaz információkat, hogy adott alhálózatok esetén mely hálózati csatolón (azaz adatkapcsolati rétegbeli eszközön) küldje ki az csomagokat/kereteket. A routing tábla (többek között) a következő bejegyzéseket tartalmazza: hálózati cím, netmask, router címe, hálózati kártya. Linuxon ezt a `route` paranccsal listázhatjuk ki:

```
bercin@kanape:~$ /sbin/route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.140.2   0.0.0.0         255.255.255.255 UH    0      0        0 tun0
192.168.140.0   192.168.140.2   255.255.255.0   UG    0      0        0 tun0
10.3.18.0       0.0.0.0         255.255.254.0   U     0      0        0 eth0
0.0.0.0         10.3.19.254     0.0.0.0         UG    0      0        0 eth0
bercin@kanape:~$
```

Az első sor szerint a 192.168.140.2 IP-vel és 255.255.255.255 netmaskkal megadott hálózatba router nélkül lehet csomagokat küldeni (mert a gateway IP címe a 0.0.0.0) a tun0 hálózati csatolón keresztül (sor vége). A második sor szerint a 192.168.140.0 IP-vel és 255.255.255.0 netmaskkal megadott hálózatba a 192.168.140.2 routeren keresztül lehet csomagokat küldeni, és a router a tun0 hálózati csatolón érhető el.

Az operációs rendszer a lista sorrendjében próbálja végig a lehetséges útvonalakat, és az első egyezésnél szereplő adatokat használja fel.

5.3.3.13. példa. A 10.3.19.7 IP címre küldendő csomag esetében az első sor nem mutat egyezést (mivel 10.3.19.7 & 255.255.255.255 != 192.168.140.2), a második sor sem mutat egyezést (10.3.19.7 & 255.255.255.0 != 192.168.140.0) viszont a harmadik sor igen (mivel 10.3.19.7 & 255.255.254.0 = 10.3.18.0), így a csomagot a lokális hálózaton kell továbbítani (mivel ebben a sorban a gateway címe 0.0.0.0), az eth0 hálózati csatolón keresztül.

5.3.3.14. feladat. Milyen hálózatot jelöl a 255.255.255.255 netmask?

5.3.3.15. feladat. Milyen hálózatot jelöl a 0.0.0.0 netmask és a 0.0.0.0 network address? Milyen IP címekre fogja ezt a bejegyzést választani az operációs rendszer?

Fontos információkhoz juthatunk a `traceroute -n` parancs segítségével, amivel az adott géphez vezető útját, az egyes közbülső routereinek elérési idejeit láthatjuk:

```
NorbiMBPr:trunk bnorbert$ traceroute -n 217.20.130.97
traceroute to 217.20.130.97 (217.20.130.97), 64 hops max, 52 byte packets
 1  10.0.1.1  1.113 ms  0.753 ms  0.739 ms
 2  80.99.195.254  94.095 ms  22.981 ms  17.520 ms
 3  89.135.214.94  9.283 ms  10.238 ms  10.511 ms
 4  193.188.137.25  9.619 ms  9.788 ms  10.651 ms
 5  217.20.130.97  10.892 ms  8.904 ms  11.312 ms
NorbiMBPr:trunk bnorbert$
```

Az első oszlop a router sorszámát tartalmazza (hányadik router az útvonalon), a második oszlop a router IP címe. A többi oszlopban az adott router elérési ideje látható (bővebben lásd: `man traceroute`). Ez a parancs windowsban is megtalálható, de ott `tracert` -nek hívják.

A DNS

Igen ám, de ha mi beírunk a böngészőbe egy címet, az a legritkább esetben pontozott négyes jelölés, legtöbbször szövegesen megadott címek: például `www.index.hu` Hogyan lesznek a szöveges címekből IP címek? Erre szolgál a DNS (Domain Name Service).

Egy világméretű elosztott (azaz nem egy gépen tárolt, hanem részenként szétosztott) adatbázisban találhatóak a névhez IP címet rendelő információk. Ha a gépünknek szüksége van pl. a `turdus.itk.ppke.hu` gép címére, akkor egy kérdést küld a beállított DNS szervernek, ami általában tovább küldi a kérést: a .hu legfelső szintű (top level) domain valamely DNS szeréhez (több gépen vannak ugyanazok az adatok), hogy mely gép tárolja a ppke.hu tartomány címeit. Ekkor visszakapja a gépünk az egyik ilyen DNS szerver címét. Egy második körben már ehhez fordul egy újabb kéréssel, hogy megtudja, hogy mely gép tárolja az itk.ppke.hu tartomány címeit. Ekkor innét is visszakapja egy olyan DNS szerver címét, amelyhez fordulva kéréssel, már megkaphatja a turdus.itk.ppke.hu gép IP címét. (A kapott eredményt egy ideig megőrzi, legközelebb ne kelljen az egész utat újra bejárni.)

Linuxban a `host` paranccsal tudjuk ezt kipróbálni:

```
bercin@users:~$ host www.index.hu
www.index.hu has address 217.20.130.97
bercin@users:~$
```

Néhány esetben (a terheléselosztás miatt) több IP címet is visszakaphatunk:

```
bercin@users:~$ host www.google.hu
www.google.hu has address 173.194.44.55
www.google.hu has address 173.194.44.56
www.google.hu has address 173.194.44.63
www.google.hu has IPv6 address 2a00:1450:4016:803::101f
bercin@users:~$
```

A DHCP

A fentiekből látható, hogy minden eszköznek ki kell osztani egy IP címet, meg kell adni a hálózatához tartozó netmaskot, tudnia kell a router címét, és a DNS címet is (amennyiben szeretnénk szöveges címeket használni). Így elég bonyolulttá válik egy hálózatban az IP címek karbantartása: ügyelni kell, hogy mindegyik eszköz jó címeket kapjon, ne legyen ütközés (ugyanolyan IP címet ne kapjon két eszköz), stb. Ennek a problémának a megoldására dolgozták ki a DHCP-t. (Dynamic Host Configuration Protokol). Ez (többek között) az előző beállítások automatikus megoldását teszi lehetővé, ezzel segítve a rendszergazda, a felhasználó munkáját.

Ha a DHCP szolgáltatás nem elérhető, a gépek általában a fentebb említett 169.254 -es hálózatból automatikusan osztanak maguknak IP címet, amivel lehetővé válik, hogy a LAN-on belül kommunikáljanak.

Az IPv6

Napjainkra égető problémává vált, hogy a fentebb vázolt 32 bites IP címzés nem elegendő, elfogynak a kiosztható címek. Erre kínál megoldást az IPv6, amely esetén 128 bites címe van minden hálózati csatolónak. Bővebben: `http://en.wikipedia.org/wiki/IPv6` `http://ipv6forum.hu` `http://www.worldipv6launch.org` `http://www.telekom.hu/ipv6`

Az IP címek elfogyása annak is „köszönhető”, hogy az internet őskorában az A osztályú címeket is előszeretettel kiosztották az akkoriban jelentős vállalatoknak. Érdekes olvasmány ezek listája (mivel az akkoriban kapott tartomány a legtöbb esetben még mindig az adott cég birtokában van - ha a cég még létezik). Például: az USA kormányának különböző kutatási, technológiai, katonai részlegei; IBM; AT&T; Xerox; HP; DEC; Apple; MIT; Ford; UK Védelmi Minisztérium; UK Munka- és Nyugdíjügyi Minisztérium; duPont; US Postal Service; Bővebben például a `http://en.wikipedia.org/wiki/List_of_assigned_/8_IPv4_address_blocks` oldalon...