

Bash gyakorlati segédlet

Készítette: Bolyki Balázs

Készült: 2023.09.04

1. óra

Háttérismeretek

Számítógépes Architektúrák gyakorlatokon shell scriptek írásával fogunk foglalkozni Unix alapú operációs rendszerekben. Mit tekintünk Unix alapú rendszernek, és mik is azok a shell scriptek?

Az eredeti AT&T Unix operációs rendszert még a hatvanas években fejlesztették, és sok későbbi operációs rendszernek szolgált alapjául. Ezeknek egy része tényleges Unix operációs rendszer (Solaris), mások pedig Unixhoz hasonló rendszerek: Mac OS X és a számtalan Linux disztribúció.

Az informatikus tanulmányok során érdemes megismerkedni legalább egy Linux operációs rendszerrel közelebbről (tehát érdemes saját otthoni gépre is feltelepíteni, akár virtuális gépen). A tanszéki gépeken ennek a jegyzetnek az írásakor Linux Mint van telepítve (a Mint egy Linux disztribúció). Ezen felül a következő Linux disztribúciókat érdemes megemlíteni:

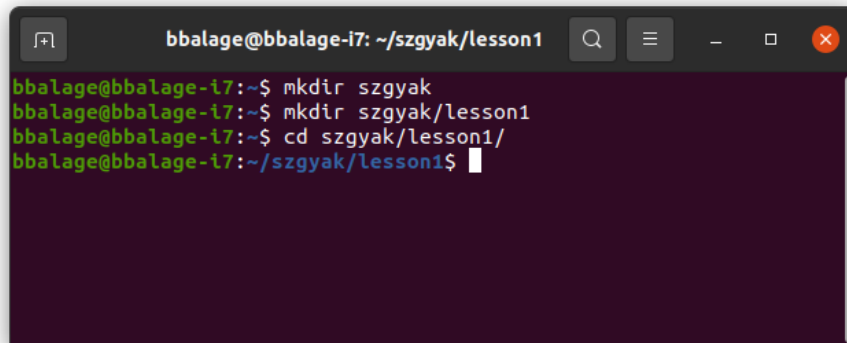
- **Ubuntu.** A Linux Mintnél mostanra elterjedtebbé váltak a különböző Ubuntu verziók (Ubuntu 18.04, 20.04, 22.04; a verziók a kiadásuk éve szerint vannak számozva). Könnyen installálható, felhasználóbarát. Otthoni gyakorláshoz tökéletes.
- **Arch Linux.** Arról híres, hogy annyira minimalista, hogy a felhasználónak gyakorlatilag mindent nem létfontosságú dolgot magának kell telepítenie.
- **Manjaro Linux.** Az Arch Linuxon alapul, csak igyekeztek “felhasználóbarátabbá” tenni.
- **Kali Linux.** Digitális analízálásra és sebezhetőség tesztelésre specializált Linux.

A shell kezelésének szempontjából nem lényeges, hogy melyik Linux operációs rendszert használjuk. De mi az a shell?

Az operációs rendszerek központi eleme a **kernel** (tehát mag). A *kernel* felelős az alapvető funkciók ellátásáért (például processzek kezelése, memória menedzselés, fájlrendszeri művelet; ezekről bővebben Operációs Rendszerek tárgyból). A *shell* (héj) egy parancsértelmező, ami a felhasználó szöveges utasításait fogja a kernel számára is értelmezhető utasításokká alakítani.

Tehát a shellt mi úgy fogjuk látni, mint egy terminálban futó parancsértelmezőt. Az alább látható képen egy terminál látható, amiben egy shell fut.

A fent látható terminálban a **Bash** parancsértelmező fut, a parancsok pedig

A terminal window with a dark purple background. The title bar shows 'bbalage@bbalage-i7: ~/szgyak/lesson1'. The terminal content shows the following commands and their outputs:

```
bbalage@bbalage-i7:~$ mkdir szgyak
bbalage@bbalage-i7:~$ mkdir szgyak/lesson1
bbalage@bbalage-i7:~$ cd szgyak/lesson1/
bbalage@bbalage-i7:~/szgyak/lesson1$
```

Figure 1: terminal screenshot

létrehoznak két mappát; SzArGyak (Számítógépes Architektúrák Gyakorlat) és benne a lesson1 mappát, majd belépnek a lesson1 mappába. Egy terminálban futhat másféle shell is, nem csak Bash (más shellek például: sh, zsh). A Windows operációs rendszereknek is van shellje, csak ott Powershellként és Command Line-ként találkozhatunk velük. Ezeknek más a szintaktikája, mint a Bash-nek. Mi a gyakorlatok keretében Bash scripteket fogunk írni.

Bash: az elnevezés egy szójátékból adódik. A mostani shell szabvány az eredeti AT&T Unixhoz tartozó shell szabványból indul ki, amelyet Stephen Bourne dolgozott ki. Bash = **B**ourne **A**gain **S**hell.

A Linux alapú rendszereknek az egyik nagy erőssége, hogy rengeteg előre elkészített “parancsal” rendelkeznek (kb. 700-1000). A parancsokat más néven “eszközöknek” (tool) nevezik, elvégre ezek többsége külső segédprogram. Néhány példát említve, hogy mire alkalmasak a parancsok:

- `cd` – egy másik mappába való belépésre szolgáló parancs
- `mkdir` – mappa létrehozására szolgáló parancs
- `wc` – szavak, sorok és karakterek számlálására szolgáló parancs
- `grep` – adott szöveges mintára illeszkedő sorok kiválasztása
- stb.

A későbbiekben feladatokon keresztül fogunk megismerkedni az ilyen parancsokkal, és ezeknek a használatával. A parancsok egymással kombinálhatók, és általuk komplex feladatok hajthatók végre.

Legelőször nyissunk egy terminált (Ctrl + Alt + T), amiben bash fut (feltételezően ez a default). A további feladatokat terminálban fogjuk csinálni.

Példák

1. példa

Hozzunk létre egy mappát, amit a gyakorlatok feladataihoz fogunk használni. A neve legyen **szgyak**. Ebben a mappában hozzunk létre egy másikat, aminek neve legyen **lesson1**. Lépünk bele ebbe a mappába!

A következő parancsokkal például ez megvalósítható:

```
mkdir szgyak
mkdir szgyak/lesson1
cd szgyak/lesson1
```

A fenti példában a következőket figyeljük meg: - 3 db parancsot adtunk ki, köztünk entert ütöttünk, tehát a parancsokat egyesével adtuk ki. A parancsokat a shell soronként értelmezte, tehát a shell egy interpreter. - A parancsoknál az első szó a **parancs neve**, a további szavak az **argumentumok**. Általánosan egy egyszerű parancs formája: **command_name command_argument_1 command_argument_2 command_argument_n**. Az argumentumokat szóközök választják el egymástól és a parancs nevéétől. Ez némileg értelmet ad annak a definíciónak az elméleti tananyagban, hogy *"a parancs fehér karakterekkel határolt szavak sora"*. - A parancsok nevei általában valami értelmes szópárosnak a rövidítései. **mkdir** = make directory = mappa (vagy szakmaiabban jegyzék) létrehozása. **cd** = change directory = jegyzék megváltoztatása, vagy *jegyzékváltás*. - A parancssorban látjuk, hogy milyen jegyzékben tartózkodunk éppen.

Ezzel kapcsolatban az is megfigyelhető, hogy a mappaneveket / (slash) jel választja el egymástól. Ez Unix alapú rendszereknél így van, míg Windowsban ez a jel \ (backslash).

2. példa

Írjuk ki, hogy hol tartózkodunk épp a mapparendszerben, az abszolút ösvénnyel!

Erre használhatjuk a **pwd** parancsot is (print working directory). Ez kiírja az aktuális mappát (working directory).

Üssük be a parancsot!

```
pwd
# valami ilyesmit kell kapnunk: /home/bbalage/szgyak/lesson1
# A hashmark (#) kommentet jelent bashben
# Tehát amit # után írsz, azt a parancssor nem fogja értelmezni
```

A kapott output eleje nem egyezik azzal, amit a terminálban olvashatunk. Ha megfigyeljük, akkor a ~ jel felcserélődött egy másik "ösvénnyel" (vagy *path*-szal). A ~ jel egy rövidítés a saját felhasználói gyökér mappánkra. Nálam a ~ jelentése /home/bbalage, míg más felhasználóknál ez más lesz.

Szintén megemlítendő, hogy Unix rendszerben a tényleges gyökérmappát a / jel azonosítja. **Ha a gyökérből kiindulva adjuk meg a path-t, akkor a “full path-t” adjuk meg.** Ellenben, ha a jelenlegi jegyzékből (working directory) indulunk ki, akkor a “relative path-t” (relatív ösvény).

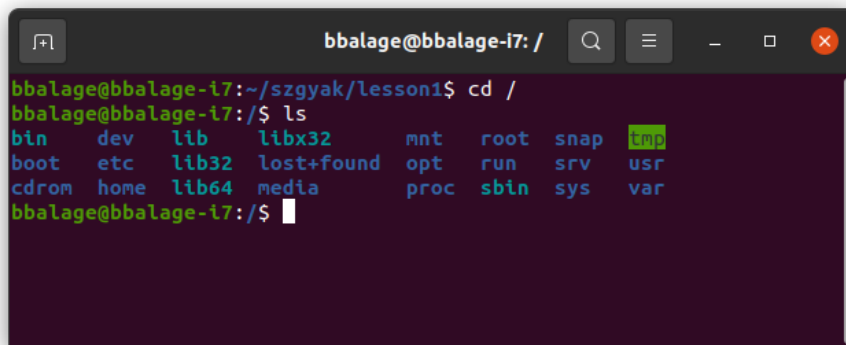
3. példa

Lépünk be a gyökér mappába, és írassuk ki a tartalmát!

Egy mappa tartalmának kiírására az `ls` parancsot tudjuk használni. Az `ls` parancsnak opcionálisan megadhatjuk, hogy melyik jegyzék tartalmát írja ki. Ha nem adunk meg semmit, akkor a working directory tartalmát írja ki.

```
cd /
ls
# Ha nem akarunk belépni a mappába, csak ki akarjuk írni
# a tartalmát, akkor # ezt az alábbi paranccsal egy lépésben
# is megtehetjük:
ls /
```

A kép szemlélteti, hogy mit fogunk kapni:



```
bbalage@bbalage-i7: /
bbalage@bbalage-i7:~/szgyak/lesson1$ cd /
bbalage@bbalage-i7:/$ ls
bin  dev  lib  libx32  mnt  root  snap  tmp
boot  etc  lib32  lost+found  opt  run  srv  usr
cdrom  home  lib64  media  proc  sbin  sys  var
bbalage@bbalage-i7:/$
```

Figure 2: terminal screenshot

Amiket látunk, azok rendszermappák. Csak néhányat kiemelve ezek közül:
- **bin**: a bináris állományok, tehát a programok találhatóak itt. A félévben használt programok közül sokat megtalálunk ebben a mappában. Az eddigiek közül a `mkdir`, a `pwd` és az `ls` parancs is itt van, de a `cd` parancs nincs, ugyanis a `cd` parancs magában a `bash` parancsértelmezőben van implementálva (része a nyelvnek), míg a többiek külső segédprogramok. Ezért mondjuk azt, hogy a `mkdir`, a `pwd` és az `ls` parancsok **külső** parancsok, a `cd` (és egyéb `bash`-ben implementált parancsok) **belső** parancsok. Ennek annyi a jelentősége, hogy külső parancsokat lehet külön telepíteni, mint egyfajta plugineket, és előfordulhat, hogy nem minden külső parancs elérhető egy adott rendszeren (mert nem

adtak a telepítőhöz, és a rendszergazda sem telepítette fel külön). - **dev**: a “devices” rövidítése, és a fizikai vagy logikai eszközök leíró fájljait tartalmazza (háttértár, cpu magok, terminálok, joystick input, billentyűzet, egér, stb.) - **root**: a rendszer felhasználó home mappája. - **home**: a többi felhasználó (és a saját felhasználónk) home mappáit tartalmazó mappa.

4. példa

Lépjünk be a **szgyak/lesson1** mappába, és hozzunk létre egy **whatever.txt** nevű fájlt!

```
cd ~/szgyak/lesson1
touch whatever.txt
```

A `cd ~/szgyak` parancsot mindegy, hogy hol adjuk ki, ugyanis a `~` jel a jelenleg belépett felhasználó home mappájának rövid jelölése, ezért a `~/szgyak` egy full path, nem pedig relative.

A `touch` parancs frissíti egy argumentumként megadott fájlt utolsó elérési és utolsó változtatási dátumát (last accessed, last modified). Ha nem létezik a fájlt, amit megadtunk, akkor létrehozza azt. Így a `touch` parancsot szoktuk fájlok létrehozására használni.

5. példa

Írassuk ki a **szgyak/lesson1** mappa teljes tartalmát a benne lévő fájlok tulajdonosaival együtt!

```
cd ~/szgyak/lesson1
ls -la
ls -la ~/szgyak/lesson1 # ha így adjuk ki a parancsot,
                        # akkor mindegy mi épp a working
                        # directory
```

Azt látjuk, hogy az `ls` parancsot némileg máshogy adtuk ki, mint eddig. Egy parancs működését lehet módosítani kapcsolók segítségével. Egy kapcsoló a parancs után jön, de az elhelyezkedése változhat parancsonként és kapcsolónként. Szinte minden parancsnak vannak kapcsolói, némelyiknek egészen rengeteg, és egészen bonyolultak. Ahhoz, hogy megnézzük, mik egy parancs kapcsolói, és mire valók, hogyan működnek, vagy az internetet, vagy a **manual page-et** hívjuk segítségül.

A következő parancssal megnézzük az `ls` parancs *manual entry-ét*:

```
man ls
```

A manual page-en lehet lefelé görgetni a nyilakkal, és ki lehet lépni a `q` karakter lenyomásával. Szinte minden parancshoz lesz manual entry, és ezeket használjuk is!

Próbáljuk ki az `ls` parancs kapcsolóit!

```
ls -l # kötőjellel adjuk meg a kapcsolókat
ls -a # ez egy másik kapcsoló
ls -la # ez a két előző kapcsoló kombinálva
      # (mintha mindkettőt kiadtuk volna)
```

A manualból megtudhatjuk ezekről a következőket:

- Az `l` kapcsoló bővebb adatokat jelenít meg a fájlokról és mappákról, nem csak a nevüket.
- Az `a` kapcsoló olyan mappákat is megjelenít, amik `.` jellel kezdődnek, vagy épp csak abból állnak. Ezek rejtett fájlok vagy mappák.
- Az `la` kapcsoló a kettő kombinációja: bővebb adat, és ponttal kezdődő nevek is.

A ponttal kezdődő nevek valamilyen kiegészítő dologhoz szoktak tartozni, amit nem szeretnénk, hogy az a felhasználó is lásson, aki csak kattintgat egy *file explorer* felületen. Ezek lehetnek parancssori fájlok, konfigurációk, de gyakorlatilag bármi más is, aminek úgy döntöttünk, hogy ponttal kezdődő nevet adunk.

6. példa

Hozzunk létre egy `tmp` mappát a `szgyak/lesson1` jegyzéken belül, és írassuk ki a tartalmát! Mi a mappa tartalma?

```
mkdir ~/szgyak/lesson1/tmp
# ha a working directory a ~/szgyak/lesson1 akkor elég ennyi is:
# mkdir tmp
ls -a ~/szgyak/lesson1/tmp # full path
ls -a tmp # relative path; függ a working directory-től
      # (tehát attól, hogy "hol vagyunk")
```

Azt látjuk, hogy még az üres jegyzéknek is van két bejegyzése: `.` és `..`. Az egy pontból álló bejegyzés magára a jegyzékre mutat, míg a két pontból álló bejegyzés a szülő jegyzékre (amiben az aktuális jegyzék van). Tehát a következő paranccsal visszalépünk a `szgyak` jegyzékbe:

```
cd ..
```

Az ilyen visszalépő directory neveket lehet láncolni is. Az alábbival kettőt lépünk "vissza":

```
cd ../../
```

És így tovább. Az alábbival nem csináltunk semmit, ugyanis `.` arra a mappára utal, amiben található.

```
cd .
```

7. példa

Töröljük a `tmp` directory-t!

```
cd ~/szgyak/lesson1
rmdir tmp
# ha kihagyjuk a cd-t, akkor természetesen megtehetjük ezt is:
# rmdir ~/szgyak/lesson1/tmp
# ugyanaz történik, csak egyik esetben full path, másik esetben
# relative path a hivatkozás típusa
```

rmdir = remove directory; kellően beszédes név, hogy ne kelljen magyarázni.

8. példa

Töröljük a `whatever.txt` fájlt!

```
rm ~/szgyak/lesson1/whatever.txt
```

rm = remove; szintén beszédes. Annyit viszont érdemes megemlíteni, hogy ez a parancs nem a kukába rakja a fájlokat, hanem ténylegesen törli őket. Nem fogjuk tudni a kukából visszaszerezni azt, amitől így szabadultunk meg.

9. példa

Hozzuk létre a következő mappaszerkezetet a `~/szgyak/lesson1` mappán belül (minden fájlt hagyjunk üresen):

```
|
|-src (mappa)
  |- main.c (fájl)
  |- util.h (fájl)
  |- util.c (fájl)
|-assets (mappa)
  |-textures (mappa)
    |-xy.png (fájl)
    |-wz.png (fájl)
    |-readme.txt (fájl)
  |-maps (mappa)
|-build (mappa)
  |-release (mappa)
  |-debug (mappa)
```

A jegyzékeket könnyen létrehozhatjuk az alábbi módon:

```
cd ~/szgyak/lesson1
mkdir src
mkdir assets
mkdir assets/textures
mkdir assets/maps
mkdir build
mkdir build/release
mkdir build/debug
```

Viszont ezzel nem tanultunk semmi újat, és kettővel több parancsot adtunk ki, mint szükséges!

Ha elolvassuk a `mkdir` parancs manual entry-ét (`man mkdir`), akkor megtudjuk, hogy van neki egy `-p` kapcsolója, ami a szülő mappa létrehozására is utasít. Például az alábbi parancs alaptól hibával elbukik:

```
mkdir build/release
```

Ez azért van, mert nincs `build` mappa, amiben létre lehet hozni a `release` mappát. A `-p` kapcsoló annyiban módosítja a működést, hogy az ehhez hasonló esetekben a szülő mappát is létrehozza a parancs (`-p` mint *parent*). Így kiadva a parancsokat:

```
cd ~/szgyak/lesson1
mkdir src
mkdir -p assets/textures
mkdir assets/maps
mkdir -p build/release
mkdir build/debug
```

A fájlokat egyszerűen hozzuk létre a `touch` paranccsal!

```
touch src/main.c
touch src/util.c
touch src/util.h
touch assets/textures/xy.png
touch assets/textures/wz.png
touch assets/textures/readme.txt
```

10. példa

Írassuk ki a mappák tartalmát, karaktergrafikusan rendezett formában!

```
tree
# igen, ennyi a parancs. Adjuk ki a lesson1 mappában!
```

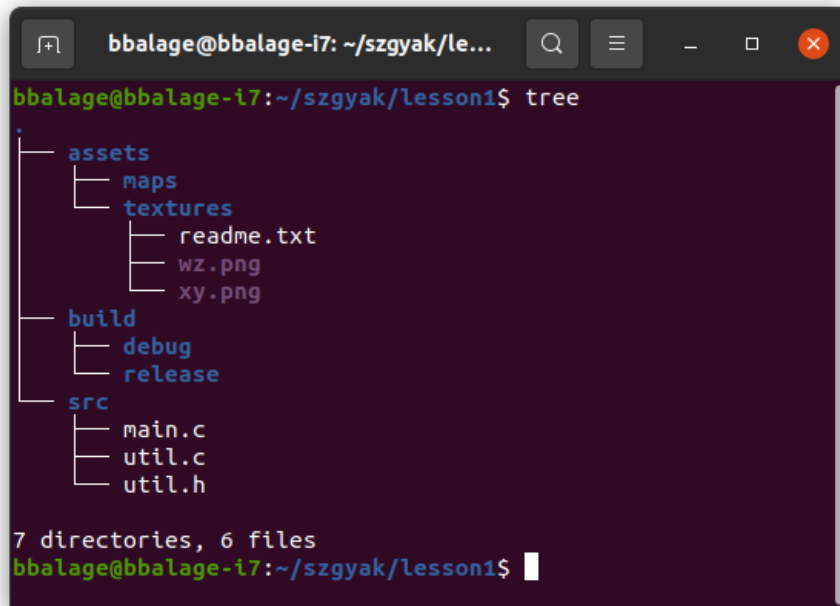
A `tree` parancs nem minden rendszerre van telepítve, viszont pont azt csinálja, amire szükségünk van. Ellenőrizzük, hogy a kapott mappaszerkezet olyan-e, mint az alább látható screenshoton. Ha nem olyan, akkor valamit elrontottunk.

11. példa

Töröljük az összes png fájlt a `textures` mappában!

```
rm assets/textures/*.png
```

A csillag minden szövegre illeszkedni fog (a reguláris kifejezésekről később majd bővebben). Ellenőrizzük a `tree` paranccsal, hogy a png fájlok (és csak azok) eltűntek-e. Erre az ellenőrzésre használhatjuk az `ls assets/textures` parancsot is (amelyik szimpatikus).



```
bbalage@bbalage-i7: ~/szgyak/le...
bbalage@bbalage-i7:~/szgyak/lesson1$ tree
.
├── assets
│   ├── maps
│   └── textures
│       ├── readme.txt
│       ├── wz.png
│       └── xy.png
├── build
│   ├── debug
│   └── release
└── src
    ├── main.c
    ├── util.c
    └── util.h

7 directories, 6 files
bbalage@bbalage-i7:~/szgyak/lesson1$
```

Figure 3: terminal screenshot

12. példa

Hozzunk létre egy `utils` mappát az `src` mappán belül, és helyezzük át bele az `util.c` és az `util.h` fájlokat!

```
mkdir src/utils
mv src/util.h src/utils/util.h
mv src/util.c src/utils/util.c
```

`mv` = move. A működése meglehetősen egyszerű. Első argumentuma annak a fájlnek vagy mappának az elérési útja (path), amit át akarunk helyezni, második argumentuma pedig az új helyen érvényes elérési út.

Tehát az a fájl, amit eddig `src/util.h` ösvényen értünk el, azt a parancs kiadása után `src/utils/util.h` elérési úton fogjuk elérni.

13. példa

Nevezzük át a `textures` mappában található `readme.txt` fájlt `readme_sprites.txt` fájlra.

```
mv assets/textures/readme.txt assets/textures/readme_sprites.txt
```

Átnevezésre az `mv` parancsot szoktuk használni a föntiek szerint.

14. példa

Másoljuk át a `readme_sprites.txt` fájlt a `maps` mappába `readme_maps.txt` néven!

```
cp assets/textures/readme_sprites.txt assets/maps/readme_maps.txt
```

`cp` = copy. A működése ugyanolyan, mint az `mv` parancsé, csak itt másolunk, nem áthelyezünk.

15. példa

Töröljük a `maps` és a `build` mappát!

```
rm -r assets/maps
rm -r build
```

Az `-r` kapcsoló jelentése *recursive*, hatása pedig az, hogy amennyiben az `rm` parancsnak egy mappát adunk meg, akkor rekurzívan törli a mappa teljes tartalmát (és minden a mappában lévő mappa tartalmát), mielőtt törli magát a mappát.

Összefoglalás

A következő parancsokat tanultuk: - `cd` megváltoztatja a `working directory-t` (konyhanyelven belép egy másik mappába). - `pwd` kiírja a jelenlegi `working directory-t` (konyhanyelven azt a `directory-t`, amiben vagyunk). - `ls` kiírja egy

directory tartalmát. - **mkdir** létrehoz egy új directory-t. - **touch** frissíti egy fájl legutóbbi elérési dátumát és legutóbbi módosítási dátumát, és ha még a fájl nem létezik, akkor létrehozza azt. - **rmdir** directory törlése. - **rm** fájlok, mappák törlése. - **tree** karaktergrafikusan kirajzolja a mappaszerkezetet. - **cp** fájlok, mappák másolása. - **mv** fájlok, mappák elérési útjának módosítása (tehát áthelyezése és/vagy átnevezése). - **man** manual entry megnyitása egy parancshoz.

Feladatok

Önálló gyakorló feladatok.

1. feladat

Induljunk felderítő útra a fájlrendszerben! Nézzünk bele más felhasználók home mappáiba! Keressük meg a legvalószínűbb helyet, ahova az értékes beadandóikat pakolják majd! Ha a fájljaik mindenki számára olvashatóak, akkor meg is nézhetjük őket. Ha írhatók, akkor törölhetjük is őket. Tanulság: ügyeljünk a jogosultságokra (következő órán megnézzük őket).

2. feladat

Módosítsuk az `util.h` utolsó változtatási dátumát, de csak a változtatási dátumát! Figyeljünk arra, hogy a parancs alpból mind az utolsó elérési, mind az utolsó módosítási dátumot módosítja! A feladat megoldásához olvassuk ki a megfelelő kapcsolót a `touch` parancs manual entry-éből (önálló utánanézés).

3. feladat

Hozzunk létre egy `include` mappát a `lesson1` mappán belül. Másoljuk bele az `util.h` fájlt, de adjuk ki úgy a parancsot, hogy csak akkor történjen másolás, ha a célmappában lévő azonos nevű fájl nem létezik, vagy elavult! (csak akkor másolj, ha szükséges) Keressük ki a megfelelő kapcsolót a `cp` parancs manual entry-éből!

1. Kérdések

Ellenőrző kérdések az első gyakorlathoz.

1. kérdés

Az alábbi `path` relatív vagy abszolút?

`/home/bbalage/Pictures/unicord.png`

2. kérdés

Az alábbi elérési útvonal relatív vagy abszolút?

runnables/run

3. kérdés

Az alábbi elérési útvonal relatív vagy abszolút?

.runnables/run

4. kérdés

Az alábbi elérési útvonal relatív vagy abszolút?

~/Documents/hallgatok.txt

5. kérdés

Hány bejegyzés van egy “üres” mappában?

6. kérdés

Mire utal a . (pont) nevű bejegyzés egy mappában?

7. kérdés

Mire utal a .. (pont-pont) nevű bejegyzés egy mappában?

8. kérdés

Van-e olyan mappa, amiben . vagy a .. bejegyzés különböző jelentéssel bír, mint a többi mappában? (ha igen, melyik az)

9. kérdés

Mit jelent a ~ egy elérési útvonalban Unix alapú rendszereknél?

10. kérdés

Igaz vagy hamis az alábbi állítás? Indokolja a választát!

A mkdir paranccsal egyszerre csak egy mappát lehet létrehozni.

11. kérdés

Igaz vagy hamis az alábbi állítás? Indokolja a választát!

Az rm paranccsal mappát nem lehet törölni, csak fájlt.

12. kérdés

Igaz vagy hamis az alábbi állítás? Indokolja a választát!

A cp paranccsal mappát vagy fájlt másolunk ugyanazon a néven, másik pozícióba a fájlrendszeren.

13. kérdés

Melyik írja le legpontosabban az mv parancs működését?

- a. Az mv paranccsal átmozgatunk egy fájlt egyik mappából a másikba.
- b. Az mv paranccsal átmozgatunk egy fájlt egyik mappából a másikba, és opcionálisan át is nevezük út közben.
- c. Az mv paranccsal adatokat mozgatunk a fájlrendszer egyik pontjából a másikba.
- d. Az mv paranccsal megváltoztatjuk egy fájl elérési útját.

14. kérdés

Melyik írja le legpontosabban a touch parancs működését?

- a. A touch parancs frissíti egy fájl utolsó módosítási időpontját és utolsó elérési időpontját.
- b. A touch parancs frissíti egy fájl létrehozási időpontját, utolsó módosítási időpontját, elérési időpontját, és létrehozza a fájlt, ha az nem létezik.
- c. A touch parancs frissíti egy fájl utolsó módosítási időpontját és/vagy utolsó elérési időpontját, és létrehozza a fájlt, ha az nem létezik.
- d. A touch parancs létrehoz egy fájlt, és beállítja az utolsó elérési időpontját.

15. kérdés

Igaz vagy hamis az alábbi állítás?

Az Unix egy Linux disztribúció.

16. kérdés

Melyik paranccsal törölhetünk üres mappát?

17. kérdés

Az alábbiak közül melyik írja le legpontosabban a pwd parancs működését?

- a. Kiírja a jelenlegi munkakönyvtár abszolút elérési útját.
- b. Kiírja a jelenlegi munkakönyvtár relatív elérési útját.
- c. Megváltoztatja a jelenleg bejelentkezett felhasználó jelszavát.
- d. Megváltoztatja a paraméterként megadott felhasználó jelszavát.

2. óra

Előkészület: Hozzunk létre egy külön mappát ennek a gyakorlatnak a feladataihoz, és lépünk bele (a mappába)! Hozzunk létre benne tetszőleges text fájlokat.

```
cd szgyak
mkdir lesson2
cd lesson2
touch 1.txt
touch 2.txt
```

Ezek után elkezdhetjük az új anyagot!

Jogosultságok

Az előző órán megtanultuk, hogy hogyan lehet terminálban navigálni a fájlrendszerbe. Nem néztük azonban meg, hogy milyen jogosultságok vannak, hogyan lehet ezeket megtekinteni, hogyan tárolja őket a Linux, és hogyan tudjuk őket módosítani.

A jogosultságok megtekintése meglehetősen egyszerű, és ennek a módját már tanultuk. Adjuk ki a következő parancsot a `lesson2` mappában.

```
ls -l
```

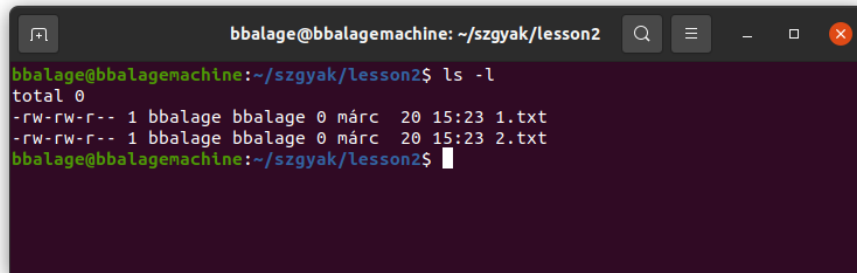
Eddig nem néztük meg jobban, hogy mit is ír ki pontosan az `ls` parancs az `-l` kapcsolóval kombinálva. Az alábbi képen látható egy lehetséges output:

Ilyenkor a következő oszlopok jelennek meg. 1. Jogosultságok. Rövidesen tisztázzuk az értelmezését. 2. Tulajdonos. Általában egyezik azzal a felhasználóval, aki létrehozta a fájlt. 3. Csoport. Rövidesen erre is kitérünk bővebben. 4. Méret. Bájtokban megadva látjuk. 5. Utolsó módosítás ideje. 6. Név.

Térjünk rá ezekből a jogosultságok oszlopra!

Jogosultságok egyszerű fájlokra

Unix alapú rendszeren háromféle jogosultságot különböztetünk meg: - olvasási (`read`), - írási (`write`), - futtatási (`execute`).

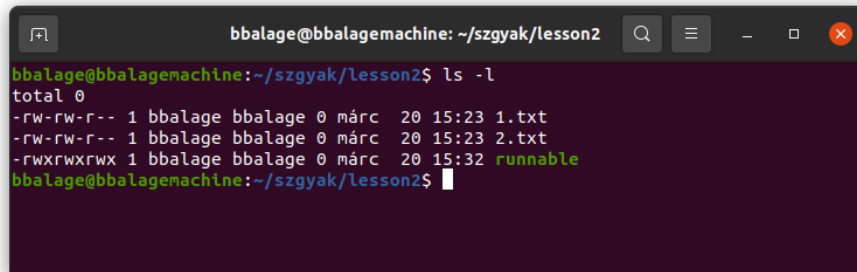


```
bbalage@bbalagemachine: ~/szgyak/lesson2
bbalage@bbalagemachine:~/szgyak/lesson2$ ls -l
total 0
-rw-rw-r-- 1 bbalage bbalage 0 márc 20 15:23 1.txt
-rw-rw-r-- 1 bbalage bbalage 0 márc 20 15:23 2.txt
bbalage@bbalagemachine:~/szgyak/lesson2$
```

Figure 4: terminal screenshot

A jogosultságokat minden elemnél háromféle felhasználóra értelmezzük. Ezek a felhasználók: - a tulajdonos, - a tulajdonosi csoportban lévő felhasználók, - mindenki más.

Ennek megfelelően egy fájlra 3x3 jog létezik, tehát 9. Tekintsük az alábbi képernyőképet:



```
bbalage@bbalagemachine: ~/szgyak/lesson2
bbalage@bbalagemachine:~/szgyak/lesson2$ ls -l
total 0
-rw-rw-r-- 1 bbalage bbalage 0 márc 20 15:23 1.txt
-rw-rw-r-- 1 bbalage bbalage 0 márc 20 15:23 2.txt
-rwxrwxrwx 1 bbalage bbalage 0 márc 20 15:32 runnable
bbalage@bbalagemachine:~/szgyak/lesson2$
```

Figure 5: terminal screenshot

A `runnable` nevű fájl mindenkinek joga van írni, olvasni és futtatni. Ezt jelöli, hogy látható az `rwxrwxrwx` a fájl előtt. Ezzel szemben a két text fájl csak a tulajdonosnak és a tulajdonosi csoporttagoknak tartozóknak van joguk írni, ellenben mindenkinek joga van olvasni. Futtatni senkinek sincs joga a text fájlokat (nem is lenne értelme futtatni egy text fájlt).

Látható, hogy azok a jogok, amik nincsenek megadva '-' karakterrel (kötőjel) vannak jelölve. Ami meg van adva, annak a megfelelő karaktere kiírásra kerül. Az első hármas `rwx` csoport a tulajdonos jogait jelöli, a második a tulajdonos csoportjában lévőket, a harmadik mindenki mást.

Megjegyzendő, hogy a fájl törléséhez is írási jog kell.

Jogosultságok számokként

Ahhoz, hogy meg tudjuk változtatni ezeket a jogosultságokat, először nézzük meg, hogy rendszer hogyan reprezentálja őket!

A jogosultságokat az Unix rendszerekben hagyományosan számokkal jelölik. A számok képzése a következő: - $r = 4$ - $w = 2$ - $x = 1$

A számok láthatóan kettő hatványai (második, első és nulladik hatványa). Ha ezeknek az összegét képezzük, akkor megkapjuk, hogy milyen jogok élnek a fájlra. Például, ha az `1.txt` fájl jogaiból indulunk ki, akkor a tulajdonos jogait a 6-os szám jelöli, mert $r + w$ joga van, ami számokkal $4 + 2 = 6$. Ezeket a számokat is ismételjük, tehát az adott fájl jogai 664, ugyanis a tulajdonos és a csoporttagoknak jogát a hatos szám jelöli, míg a többiekét a 4-es, mert ők csak olvashatják a fájlt.

Néhány példa kombináció: - 755 = `rw-r--r--`: A tulajdonos mindent megtehet a fájllal, de a többiek nem írhatják. - 640 = `rw-r-----`: A tulajdonos írhat és olvashat, a csoportba tartozók csak olvashatnak, mindenki más nem tehet semmit a fájllal. - 711 = `rw-x--x--x`: A tulajdonos mindent megtehet, de mindenki egyéb csak futtathatja a fájlt.

A tulajdonosnak nem kell több jogosultsággal rendelkeznie, mint mindenki másnak, de így logikus.

Most, hogy tudjuk a jelöléseket, nézzük a változtatásokat!

Jogosultságok megváltoztatása

A jogosultságok megváltoztatására a `chmod` parancsot használhatjuk. A parancs első paramétere az új jogosultságok számokkal, a második a fájl, amire meg akarjuk változtatni a jogosultságokat. Például:

```
chmod 640 1.txt
# ezután nézzük meg:
ls -l
```

A fenti parancs után az `1.txt` jogosultságai a következők lesznek: `rw-r-----`. Természetesen a parancsnak megadható bármilyen háromjegyű szám, amelyben nem szerepel 8-as vagy 9-es számjegy, de persze némelyik logikailag nincs sok értelme. Például miért adnánk meg 507-et? Ezzel a fájllal bárki bármit tehet, de a tulajdonosa nem írhatja, és a csoport is teljesen le van tiltva róla. Ha átrendezzük, akkor már több értelme van: 750. Ettől függetlenül az 507 is megadható, nem tiltja semmi.

A `chmod` parancsnak adhatunk a következőképpen is paramétert:

```
touch script.sh
# Edit the script
chmod +x script.sh
```


A fönti parancsoknál létrehoztunk egy `script.sh` nevű fájlt. Alapértelmezetten a `script.sh` fájlban sincs futtatási jog, viszont mi később tudni fogjuk, hogy az `.sh` kiterjesztésű fájlokban shell scripteket szoktunk tárolni, amelyeket esélyesen futtatni szeretnénk. Ehhez előbb írunk is kell a fájlba egy programot (amelyet a fönti példában nem tettünk meg, csak kommentben), viszont jogunk attól még nincs a futtatásra. A `chmod +x script.sh` parancs mindenkinek megadja a futtatási jogot a fájlhoz. Ha ennél finomabban akarjuk megadni, akkor vissza kell térnünk a számos jelöléshez.

Mappa jogosultságok

Mappák esetében is értelmezettek a read-write-execute jogok, de mást jelentenek, mint fájlok esetén. - **read:** Aki rendelkezik olvasási joggal, az listázhatja a mappa bejegyzéseit (a benne lévő fájlokat és mappákat). - **write:** Aki rendelkezik ezzel a joggal, az módosíthatja a mappa tartalmát. - **execute:** Aki rendelkezik ezzel a joggal, az beléphet a mappába. Megjegyzendő, hogy ha nem léphetünk be a mappába, akkor nem tudunk módosítani benne és nem tudjuk megfelelően listázni a bejegyzéseit sem. Tehát a read és write jog az execute jog nélkül nem sok hasznót jelent nekünk.

Csoportok

Minden fájlnak van egy tulajdonosi csoportja, amelyre a fönt olvasottaknak megfelelően kitüntetett jogosultságok vonatkozhatnak. Az `ls -l` parancs kilistázza a fájlokat a tulajdonossal és a csoporttal együtt.

A következő módon megnézhetjük, hogy az adott user melyik csoportokba tartozik:

```
groups username
# a username nevű felhasználó csoportjai kiírásra kerülnek
```

Minden fájlnak egy tulajdonosi csoportja van, de egy felhasználó több felhasználói csoportban lehet. Az Unixban vannak parancsok különböző csoportkezelési műveletekre. Mivel ez a tárgy nem az *Unix Rendszergazda* tárgy, ezért ezekbe a parancsokba nem fogunk itt mélyebben belemenni. Az alábbi parancsokat felsoroljuk, de legfeljebb minimális szükségünk lesz rájuk: - `useradd` új felhasználót ad a rendszerhez. - `groupadd` új csoportot ad a rendszerhez. - `usermod -a -G groupname username` egy meglévő usert (username) egy meglévő csoporthoz (groupname) hozzáad.

A föntiek inkább rendszergazdai feladatok, de ritkán személyes gépen is hasznosak lehetnek kényelmi okokból.

Lényegesen gyakrabban kell használni a `chown` parancsot, amely egy fájl tulajdonjogának megváltoztatására szolgál. Egyszerre lehet használni a tulajdonos felhasználó és a tulajdonos csoport megváltoztatására. Ha csak a tulajdonost akarjuk megváltoztatni, akkor a következőképpen tehetjük:

```
chown newusername filename
```

Amennyiben a tulajdonosi csoportot is meg akarjuk változtatni, akkor az alábbi módon adjuk ki a parancsot:

```
chown username:groupname filename
```

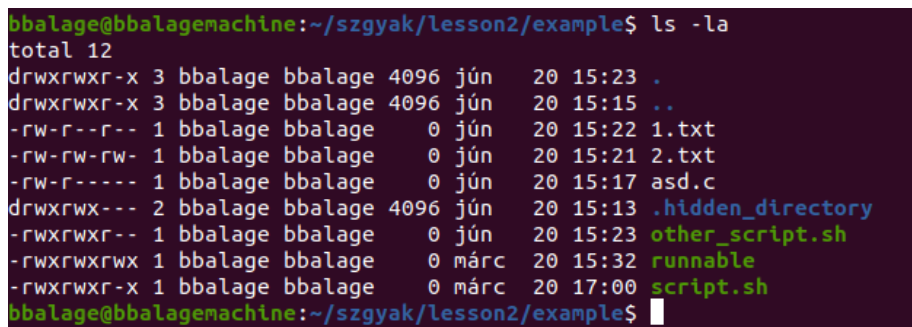
Megjegyzendő, hogy a `chown` parancshoz szükséges a rendszergazdai jogosultság, így ezt nem tudjuk kipróbálni, csak otthoni gépen, ahol rendelkezünk a rendszergazda jelszavával.

Példák

Az alábbi példán keresztül gyakoroljuk a jogosultságok beállítását!

1. példa

Hozzunk létre a `lesson2` mappán belül egy `example` mappát! Ezen a mappán belül hozzuk létre a megfelelő fájlokat a megfelelő jogosultságokkal, hogy az `ls -l` parancs a következő kimenetet generálja (a tulajdonosokkal ne törődjünk, mérettel, elérési, módosítási idővel ne törődjünk):



```
bbalage@bbalagemachine:~/szgyak/lesson2/example$ ls -la
total 12
drwxrwxr-x 3 bbalage bbalage 4096 jún  20 15:23 .
drwxrwxr-x 3 bbalage bbalage 4096 jún  20 15:15 ..
-rw-r--r-- 1 bbalage bbalage    0 jún  20 15:22 1.txt
-rw-rw-rw- 1 bbalage bbalage    0 jún  20 15:21 2.txt
-rw-r----- 1 bbalage bbalage    0 jún  20 15:17 asd.c
drwxrwx--- 2 bbalage bbalage 4096 jún  20 15:13 .hidden_directory
-rwxrwxr-- 1 bbalage bbalage    0 jún  20 15:23 other_script.sh
-rwxrwxrwx 1 bbalage bbalage    0 márc 20 15:32 runnable
-rwxrwxr-x 1 bbalage bbalage    0 márc 20 17:00 script.sh
bbalage@bbalagemachine:~/szgyak/lesson2/example$
```

Figure 6: terminal screenshot

A következő kóddal mindez kivitelezhető:

```
# Create folder and cd into it
mkdir example
cd example

# Create files and directories
touch 1.txt
touch 2.txt
touch asd.c
touch other_script.sh
touch runnable
touch script.sh
```

```
mkdir .hidden_directory

# Set the permissions
chmod 644 1.txt
chmod 666 2.txt
chmod 640 asd.c
chmod 770 .hidden_directory
chmod 774 other_script.sh
chmod 777 runnable
chmod 775 script.sh
```

Összefoglalás

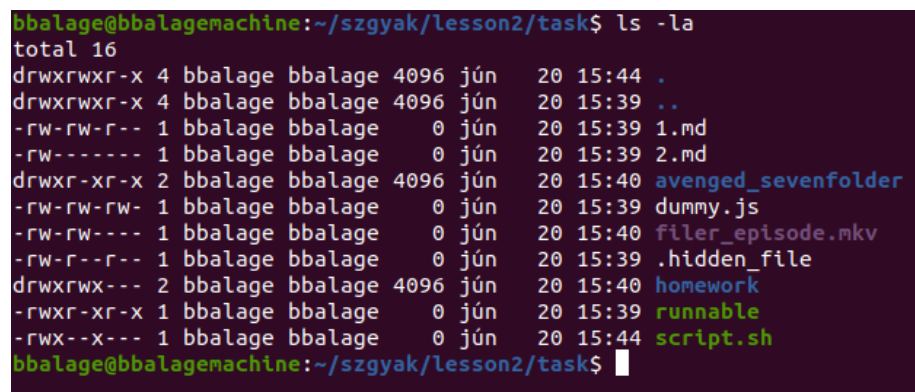
Az alábbi parancsokat tanultuk (csak az elsőt, a `chmod`-ot használtuk érdemben): - `chmod` módosítja a fájlokra adott jogosultságokat - `chown` módosítja egy fájl tulajdonosát (rendszergazdai jog kell a kiadásához) - `groups` kiírja egy felhasználó mely csoportokba tartozik - `useradd` új felhasználót ad a rendszerhez - `groupadd` új csoportot ad a rendszerhez

Feladatok

Önálló gyakorló feladatok.

1. feladat

Hozzon létre egy mappát a `szgyak/lesson2` mappában `task` néven! A mappában belül hozza létre a megfelelő mappákat és fájlokat a megfelelő jogosultságokkal, hogy a mappán belül kiadott `ls -la` parancs az alábbi screenshoton látható eredményt hozza! (ignorálja a fájloknak és mappáknak a jogosultsághoz és névhez nem kötődő tulajdonságait, tehát módosítási dátumot, tulajdonost, stb.)



```
bbalage@bbalagenachine:~/szgyak/lesson2/task$ ls -la
total 16
drwxrwxr-x 4 bbalage bbalage 4096 jún  20 15:44 .
drwxrwxr-x 4 bbalage bbalage 4096 jún  20 15:39 ..
-rw-rw-r-- 1 bbalage bbalage    0 jún  20 15:39 1.md
-rw----- 1 bbalage bbalage    0 jún  20 15:39 2.md
drwxr-xr-x 2 bbalage bbalage 4096 jún  20 15:40 avenged_sevenfolder
-rw-rw-rw- 1 bbalage bbalage    0 jún  20 15:39 dummy.js
-rw-rw---- 1 bbalage bbalage    0 jún  20 15:40 filer_episode.mkv
-rw-r--r-- 1 bbalage bbalage    0 jún  20 15:39 .hidden_file
drwxrwx--- 2 bbalage bbalage 4096 jún  20 15:40 homework
-rwxr-xr-x 1 bbalage bbalage    0 jún  20 15:39 runnable
-rwx--x--- 1 bbalage bbalage    0 jún  20 15:44 script.sh
bbalage@bbalagenachine:~/szgyak/lesson2/task$
```

Figure 7: terminal screenshot

2. Kérdések

Ellenőrző kérdések a második gyakorlathoz.

Adott az `ls -l` parancs kimenetének az alábbi részlete:

```
-rw-rw-r-- 1 robert ruby 241 szept 26 2022 README.md
```

Válaszoljon az alábbi kérdésekre a fenti kimenet alapján!

1. kérdés

Ki a tulajdonosa a `README.md` fájlnek?

2. kérdés

Melyik csoport a tulajdonosa a `README.md` fájlnek?

3. kérdés

Igaz vagy hamis a következő állítás? “A `README.md` fájl üres.”

4. kérdés

Igaz vagy hamis az alábbi állítás? “A `robert` csoportjába tartozó felhasználók szerkeszthetik a `README.md` fájlt?”

5. kérdés

Igaz vagy hamis az alábbi állítás? “A fájl egy szövegszerkesztőben akárki által megnyitható.”

6. kérdés

A következő parancs kiadása után a `README.md` fájl ki számára lesz olvasható?

```
chmod 600 README.md
```

7. kérdés

Adott az alábbi jogosultság egy fájlra: `rw-rw-rw-`. Fejezze ki számmal!

8. kérdés

Adott az alábbi jogosultság egy fájlra: `rw-r-----`. Fejezze ki számmal!

9. kérdés

Adott az alábbi jogosultság egy fájlra: `rw-r-xr-x`. Fejezze ki számmal!

10. kérdés

Adott az alábbi jogosultság egy fájlra: `rwrxrwxrwx`. Fejezze ki számmal!

11. kérdés

Adott az alábbi jogosultság egy fájlra: `rwX-----`. Fejezze ki számmal!

12. kérdés

Adott az alábbi jogosultság egy fájlra: `rwXr--r--`. Fejezze ki számmal!

13. kérdés

Az alábbi szám egy fájl jogosultság halmazt fejez ki: 644. Ki tudja a fájlt olvasni?

14. kérdés

Az alábbi szám egy fájl jogosultság halmazt fejez ki: 644. Ki tudja a fájlt olvasni?

15. kérdés

Az alábbi szám egy fájl jogosultság halmazt fejez ki: 711. Futtatható-e a fájl egy tetszőleges felhasználó számára?

16. kérdés

Az alábbi szám egy fájl jogosultság halmazt fejez ki: 640. Tudja-e a fájlt írni a birtokos csoport tagja?

17. kérdés

Az alábbi szám egy fájl jogosultság halmazt fejez ki: 600. Ki látja a fájlt a rendszeren?

18. kérdés

Az alábbi szám egy mappa jogosultság halmazt fejez ki: 600. Ki léphet be a mappába?

19. kérdés

Az alábbi szám egy mappa jogosultság halmazt fejez ki: 660. Ki látja a fájlokat a mappában?

20. kérdés

Az alábbi szám egy mappa jogosultság halmazt fejez ki: 744. Ki hozhat létre fájlt a mappában?

21. kérdés

Az alábbi szám egy mappa jogosultság halmazt fejez ki: 555. Ki törölhet fájlt a mappában?

3. óra

Ezen az órán új parancsokat fogunk tanulni, és megtanuljuk kombinálni őket. Mielőtt azonban belefogunk a parancsok kombinálásába, tanuljunk meg terminálban fájlokat szerkeszteni!

nano (terminálos szövegszerkesztő)

A nano egyike a tömérdek terminálos szövegszerkesztőknek, amik elérhetőek szoktak lenni egy Unix rendszerre (ismertek még: emacs, vim). A nano egy olyan szövegszerkesztő, ami teljesen a terminálban működik, és nem kell külön ablakot nyitnia, hogy használható legyen. Az egér is szinte teljesen haszontalan.

A nano segítségével tudunk fájlokat szerkeszteni anélkül, hogy új ablakot kéne nyitnunk. Ez abban az esetben jó, mikor csak pár sort be akarunk írni egy fájlba, vagy rendszergazdaként akarjuk szerkeszteni az adott fájlt.

Nyissunk egy terminált, és hozzunk létre az **szgyak** mappán belül egy **lesson3** mappát! Lépünk be ebbe a mappába, és nyissunk meg egy **tmp.txt** nevű fájlt a nano segítségével!

```
mkdir szgyak/lesson3
cd szgyak/lesson3
nano tmp.txt # command name and file path
```

Írjunk bele tetszőleges szöveget a fájlba! Például: “Blablabla; blablabla”. A fájlt el tudjuk menteni a Ctrl + O billentyűkombinációval (O, mint Out). Ezt követően ki tudunk lépni a Ctrl + X lenyomásával (X, mint eXit).

Ha ezután kiadjuk az **ls** parancsot, akkor látjuk, hogy a fájl létrejött. Ha a tartalmát is meg akarjuk nézni, akkor vagy megint megnyitjuk a nano szövegszerkesztővel, vagy pedig csak kiíratjuk a tartalmát a terminálra. Ezt a **cat** parancs segítségével tudjuk megtenni.

```
cat tmp.txt # kiírja a fájl tartalmát a terminálra
```

Parancsok kombinálása

Az Unix parancsok azt az elvet követik, hogy egy parancs **egyetlen** lényegi feladatot hajtson végre. Ennek megfelelően, ha komplexebb feladatot akarunk végrehajtani, akkor valószínűleg nem fogunk olyan parancsot találni, ami megoldja nekünk azt. Helyette össze kell kombinálni meglévő parancsokat valamilyen módon. A parancsok kombinálására első körben az alábbi módszereket tekintjük:

- `parancs1 && parancs2` -> ha `parancs1` sikeresen végrehajtott, akkor hajtsdjon végre `parancs2` is.
- `parancs1 || parancs2` -> ha `parancs1` elbukott, akkor hajtsdjon végre `parancs2`
- `parancs1 | parancs2` -> `parancs1` kimenetét `parancs2` bemenetére irányítja
- `parancs1; parancs2` -> `parancs1` után végrehajtja `parancs2`-t. Ez gyakorlatilag nem kombinálás, mert csak kiadtunk egymás után két parancsot, annyi különbséggel, hogy 2 sor helyett 1 sorban tettük meg.

Ezekon felül lehet még a parancs outputját valamilyen egyéb “folyamra” (stream) is irányítani. Ez akkor hasznos például, ha fájlba szeretnénk írni, vagy csak “kukázni” akarjuk az outputot. A következő “irányításokat” tekintjük meg:

- `parancs > streamname` -> a parancs outputját a `streamname` nevű streamre irányítja. Tipikus eset, hogy `streamname` egy fájl; ekkor az output felülírja a fájl tartalmát (az eredeti tartalom törlődik).
- `parancs >> streamname` -> ugyanaz, mint az előző, csak a parancs outputja hozzáfűződik a streamhez, nem pedig felülírja azt. Ez fájl esetén a látványos, ugyanis ilyenkor a fájl eredeti tartalma nem törlődik, mint az előző esetben.

Vegyük észre, hogy alapból is egy streamre írunk, a *standard output* streamre (*stdout*, 1-es azonosító). Ismert a *standard input* stream is (*stdin*, 0-s azonosító), továbbá a *standard error* stream (*stderr*, 2-es azonosító). Ezekkel majd azután foglalkozunk, hogy megtanultunk pár hasznos parancsot.

Az eddigiek szemléltetéséhez tanuljuk meg az `echo` parancsot (mint visszhang)! Ez mindössze annyit csinál, hogy kiírja, amit írunk neki:

```
echo "Hello World!"  
# kiírja, hogy Hello World! és rak egy új sor karaktert a végére
```

Tekintsük meg példákon keresztül a parancsok kombinálását!

Példák

1. példa

Próbáljunk meg belépni egy `tmp` nevű jegyzékbe, és ha nem sikerül, akkor írjunk ki hibát!

```
cd tmp || echo "Error"
```

A fenti példában az első parancs a `cd tmp`. A két vonal jelentése, hogy ha az első parancs elbukik, akkor hajtódjon végre a második parancs. A második parancs `echo "Error"`. Tehát ha `cd tmp` elbukik, akkor `echo "Error"` végrehajtódik, egyébként nem.

Hajtsuk végre a parancsot úgy, hogy a `tmp` mappa nem létezik, és nézzük meg az eredményt! Ezután hozzuk létre a mappát, és adjuk ki úgy is a parancsot!

2. példa

Hozzuk létre a `dir_1` nevű mappát, és ha sikerül, lépünk is bele (egyébként ne csináljunk semmit)!

```
mkdir dir_1 && cd dir_1
```

A fenti az előző példához nagyon hasonló, csak itt a második parancs akkor hajtódik végre, ha az első sikeres volt.

3. példa

Próbáljunk meg belépni egy mappába, és ha sikerül, akkor írjuk ki a working directory-t, egyébként hozzuk létre a mappát! (working directory kiírása: `pwd`)

```
cd dir_2 && pwd || mkdir dir_2
```

A fenti parancs rövid magyarázatra szorulhat. 3 részből áll: 1. `cd dir_2` – megpróbál belépni a mappába 2. `pwd` – kiírja a jelenlegi working directory-t 3. `mkdir dir_2` – létrehozza a mappát

Kétféleképpen történhet a végrehajtás, attól függően, hogy a mappába be lehet-e lépni. 1. **Az első parancs elbukik:** az `&&` jel utáni parancs nem hajtódik végre (mert az csak akkor hajtódik végre, ha az előtte lévő parancs sikeres), a `||` jel utáni viszont igen, mert azelőtt hiba lépett fel. Tehát létrejön a directory. 2. **Az első parancs sikeresen lefut:** az `&&` jel utáni parancs végrehajtódik, bizonyára sikeresen, mert a `pwd` parancsnak nincs itt érdemi lehetősége elbukni. Ekkor az `||` jel utáni parancs nem hajtódik végre, mert előtte nem volt hiba.

További lehetőségek lépnének fel, ha `pwd` elbukhatna.

Megjegyzendő, hogy alapesetben, ha kétszer egymás után adjuk ki a parancsot, akkor elsőre hiba lesz, másodjára nem.

Kérdés: mi lesz, ha a `dir_2` mappa már létezik, de nincs rajta execute jogosultságunk?

4. példa

Próbáljunk meg belépni a `dir_3` mappába, és ha nem sikerül, hozzuk létre azt, majd lépünk bele! Mindezt természetesen oldjuk meg egyetlen sor parancsban!

Megoldás:


```
cd dir_3 || mkdir dir_3 && cd dir_3
```

A fenti példákat tovább lehetne kombinálni, de nem érdemes. Már a három részből álló kombinált parancsok is ritkák, ugyanis ránézésre nem a legjobban érthetők. Gyakrabban használjuk a stream-ek (folyamok) átvezetését egyikből másikba. Nézzünk erre is néhány példát!

5. példa

Hozzunk létre egy fájlt `xyz.txt` néven, és írjuk bele `echo` segítségével a nevünket!

```
touch xyz.txt
echo "Johnny Bravo" > xyz.txt
```

Megjegyzendő, hogy a `touch` valójában felesleges, mert a második parancsnál, ha nem létezik a fájl, akkor a stream irányítás létrehozza.

```
echo "Johnny Bravo" > xyz.txt # Ha xyz.txt nem létezik,
                              # akkor itt létrejön
```

A fájl tartalmát megnézhetjük `nano` vagy `cat` segítségével is.

6. példa

Készítsünk egy `www.txt` nevű fájlt, és írjuk bele `echo` segítségével a nevünket és a neptun kódunkat! Ezután másoljuk be ennek a fájlnek a tartalmát az `xyz.txt` fájl végére!

```
echo "Donald Duck" > www.txt
echo "QUACK1" >> www.txt
cat www.txt >> xyz.txt
# Nézzük is meg az xyz.txt tartalmát:
cat xyz.txt
```

A fenti parancsoknál elismételendő, hogy `>` kitörli a fájl eredeti tartalmát, `>>` pedig hozzáír a fájl eddigi tartalmához.

7. példa

Kérdezzük le a jelenlegi mappa tartalmát, és írjuk bele a visszaadott szöveget az `content.txt` fájlba!

```
ls > content.txt
```

Tanulság: bármely parancs outputját bele lehet irányítani egy fájlba.

8. példa

Megesik, hogy installálunk valamit a gépre, és az installálás közben olyan szövegek íródnak ki, amikre nincs szükségünk, nem akarjuk látni őket. Szimuláljuk ezt az `ls -la` parancssal! Adjuk ki a parancsot, de ne jelenjen

meg az outputja! (Igen, ennek nincs sok értelme, de nem tanultunk még olyan parancsot, aminek az outputját van értelme ignorálni. Úgyhogy nézzük meg az output ignorálást, és felejtjük el, hogy erre a parancsra biztos nem használnánk!)

```
ls -la > /dev/null
```

Magyarázat: a `/dev/null` egy fájl a rendszeren (ami a gyökér mappán belül lévő `dev` mappában van). Ennek a fájlnek az a különlegessége, hogy amit beleírunk, az eltűnik. Konkrétan arra szolgál, hogy az ignorálandó streameket beleirányítsuk. Nem fogjuk használni semmire, de ha látjuk valahol tutorialban, akkor ne lepődjünk meg rajta! Ez egy speciális fájl, aminek nincs és nem is lesz tartalma.

9. példa

Próbáljuk meg ismét létrehozni `dir_3`-at, de úgy, hogy már létezik! A hibaüzenet ne jelenjen meg!

Azt vesszük észre, hogy a következő megoldás nem működik:

```
mkdir dir_3 > /dev/null
```

Itt jön a képbe az, hogy a rendszeren három standard csatornát különböztetünk meg: - **stdin**: *Standard input* rövidítése. A felhasználó által beírt szöveg megy rá. Azonosító száma: 0. - **stdout**: *Standard output* rövidítése. A programok által kiírt adatok mennek rá (`echo`, `ls`, stb.). Eddig többnyire ezt a csatornát láttuk a terminálunkon megjeleneni. - **stderr**: *Standard error* rövidítése. Ide mennek a programok által kiírt hibaüzenetek. Ezek is megjelennek a terminálunkon, ezért nem látjuk a különbséget közte és az `stdout` között.

A fenti megoldás azért nem működik, mert a `>` operátor csak az `stdout` csatornát irányítja bele a `/dev/null` fájlba, az `stderr` csatornát nem.

Megoldás:

```
mkdir dir_3 2> /dev/null
```

Ebben az esetben annyi történt, hogy odaírtuk a `>` elé annak a csatornának az azonosítóját, amit át szeretnénk irányítani. Itt megjegyzendő, hogy a következő két parancs teljesen ugyanazt csinálja:

```
ls > random_something.txt # stdout csatornát beleirányítja a fájlba
ls 1> random_something.txt # szintén az stdout-ot irányítja bele,
                             # csak ki is írtuk
```

Alapértelmezettként `>` és `>>` operátorok az 1-es (standard output) csatornát irányítják át.

A csatorna irányításokra később vissza fogunk térni, mikor mi magunk is írunk hibaüzeneteket.

grep parancs

A harmadik féle parancskombinálás a | jel (amit bash nyelvben “csővezeték operátor” néven is ismernek) által lehetséges. **Ezt használják leggyakrabban.** Mint azt korábban olvashattuk, ez az operátor **az egyik parancs kimenetét egy másik parancs bemenetére irányítja.** Emiatt kicsit olyan, mintha egy csővel összekötné a kettőt (ezért a “csővezeték operátor” elnevezés).

A leggyakrabban ezt az operátort a **grep** paranccsal kombinálva használják. Ez egy nagyon (nagyon-nagyon) hasznos parancs, úgyhogy sokat fogunk vele foglalkozni. Gyakorlatilag ennyit csinál: **kiválasztja a bemenetéből azokat a sorokat, amelyek illeszkednek egy megadott mintára.**

Nézzük meg példákon keresztül! Először hozunk létre egy fájlt `people.csv` néven. A tartalma a következő legyen:

```
Name;Birthdate;Phone;Skill
Robert Bob;1997.09.12.;06201975555;IT
Zsuber Driver;1988.10.11.;06304445555;Driving
Hatori Hanso;1966.01.11.;06301234555;Smithing
Rinaldo Orson;2001.05.24.;06709330000;IT
Travis Camel;1970.10.01.;06301717171;Horses
Dagobert McChips;1956.08.31.;06700001111;Cooking
Bumfolt Rupor;1967.09.11.;06201112233;Marketing
```

Ebben a fájlban fogunk keresgetni a **grep** parancs segítségével.

10. példa

Írassuk ki azokat a sorokat, ahol a személy skillje az IT!

```
cat people.csv | grep "IT"
```

A `cat` paranccsal kiolvassuk a teljes `people.csv` fájlt, és a | operátorral “belevezetjük” a `grep` parancsba. Emellett megadjuk, hogy mit keresünk az adott sorban: “IT”. A `grep` kiírja azokat a sorokat, amiben benne van a megadott szöveg.

11. példa

Írassuk ki azokat a sorokat, ahol a telefonszám 30-as.

```
cat people.csv | grep "0630"
```

12. példa

Írassuk ki azokat a sorokat, ahol a személy 2000-ben vagy utána született.

```
cat people.csv | grep ";2"
```

13. példa

Írassuk ki azokat a sorokat, ahol a *Skill* megnevezése *-ing* végződésű.

```
cat people.csv | grep ";*ing"
```

Ez már a következő óra anyaga (reguláris kifejezések), de a csillagról egyéb tanulmányainkból már tudhatjuk, hogy általában mindenre illeszkedik. Sajnos sokkal bonyolultabb kereséseket nem tudunk megvalósítani reguláris kifejezések nélkül. Ezeket már a következő órára hagyjuk.

14. példa

Készítsünk egy `IT.txt` nevű fájlt az alábbi tartalommal:

```
People at IT:  
***
```

Ahol `***` helyére rakjuk be a `people.csv`-ből az IT-s emberek sorait.

```
echo "People at IT:" > IT.txt  
cat people.csv | grep "IT" >> IT.txt
```

A fenti sorokban kombináltuk a stream átirányító operátort a csővezeték operátorral. Külön-külön már láttuk a működésüket, ami egyben sem bonyolultabb. *Mindössze balról jobbra kell olvasni, hogy mi történik, és akkor egyértelmű lesz.*

Feladatok

Az alábbi feladatokat csináljuk meg önállóan!

1. feladat

Gyűjtsük ki egy `voda.txt` nevű fájlba a `people.csv` fájlból azokat, akik 70-es telefonszámmal rendelkeznek.

2. feladat

Próbáljunk meg belépni a `dir_4` nevű mappába, és ha sikerül, hozzunk létre benne egy `im_tired.txt` nevű fájlt. Ha nem sikerül, hozzuk létre a `dir_4` mappát!

3. feladat

Próbáljunk meg kilistázni a `dir_5` nevű mappa tartalmát. Ha nem sikerül, ne írjunk hibát, hanem hozzuk létre a `dir_5` mappát!

4. feladat

Hozzuk létre az alábbi fájlokat egy `pictures` nevű directory-ban:

```
1.jpg
100.jpg
200.jpg
1.png
10.png
20.png
234.jpg
10.svg
```

Listázzuk ennek a mappának a tartalmát, és minden png fájlt írjunk bele egy `sprites.list` nevű fájlba! (Tipp: a korábbiakban `cat people.csv` parancsot most az `ls pictures` parancs fogja felváltani.)

Összefoglalás

A következő parancsokat tanultuk: - `cat` kiírja egy fájl tartalmát a standard outputra. - `echo` kiírja a standard outputra a paraméterként megkapott szöveget. - `grep` kiírja a megadott mintára illeszkedő sorokat.

3. Kérdések

Ellenőrző kérdések a harmadik gyakorlathoz.

1. kérdés

Mit ír ki az alábbi parancs, ha `dir_1` létezik, és be lehet lépni (az esetleges hibaüzenet nélkül)?

```
mkdir dir_1 || echo "OK"
```

2. kérdés

Mit ír ki az alábbi parancs, ha `dir_1` nem létezik (az esetleges hibaüzenet nélkül)?

```
mkdir dir_1 && echo "OK"
```

3. kérdés

Mit ír ki az alábbi parancs, ha `dir_1` létezik (az esetleges hibaüzenet nélkül)?

```
mkdir dir_1 || mkdir dir_1 && echo "OK"
```

4. kérdés

Mit ír ki az alábbi parancs, ha `dir_1` nem létezik (az esetleges hibaüzenet nélkül)?

```
cd dir_1 || echo "Making directory" && mkdir dir_1\  
&& echo "OK" || echo "Problem"
```

5. kérdés

Mit ír ki az alábbi parancs, ha `dir_1` létezik (az esetleges hibaüzenet nélkül)?

```
cd dir_1 || echo "Making directory" && mkdir dir_1\  
&& echo "OK" || echo "Problem"
```

6. kérdés

Mit ír ki az alábbi parancs, ha `dir_1` létezik, de nincs jogunk belépni (az esetleges hibaüzenet nélkül)?

```
cd dir_1 || echo "Making directory" && mkdir dir_1\  
&& echo "OK" || echo "Problem"
```

7. kérdés

Ír-e hibát az alábbi parancs, ha `dir_1` nem létezik?

```
cd dir_1 > /dev/null
```

8. kérdés

Ír-e hibát az alábbi parancs, ha `dir_1` létezik?

```
mkdir dir_1 3> /dev/null
```

9. kérdés

Ír-e hibát az alábbi parancs, ha `dir_1` nem létezik?

```
ls dir_1 2> /dev/null
```

10. kérdés

Üres lesz-e az `xyz.txt` fájl a következő parancs után, ha `dir_1` nem létezik?

```
cd dir_1 2> xyz.txt
```

11. kérdés

Üres lesz-e az `xyz.txt` fájl a következő parancs után, ha `dir_1` nem létezik?

```
cd dir_1 1> xyz.txt 2> zzz.txt
```

12. kérdés

Hány sort fog tartalmazni a `records.txt` fájl a következő parancsok után?

```
echo "John;29" > records.txt
echo "Alice;32" >> records.txt
echo "Marlon;50" >> records.txt
echo "Dracula;598" >> records.txt
cp records.txt records_2.txt
cat records_2.txt | grep "8" > records.txt
```

13. kérdés

Hány sort fog tartalmazni a `records.txt` fájl a következő parancsok után?

```
echo "John;29" > records.txt
echo "Alice;32" >> records.txt
echo "Marlon;50" >> records.txt
echo "Dracula;598" 2>> records.txt
cp records.txt records_2.txt
cat records_2.txt | grep "5" > records.txt
```

14. kérdés

Hány sort fog tartalmazni a `records.txt` fájl a következő parancsok után?

```
echo "John;29" > records.txt
echo "Alice;32" >> records.txt
echo "Marlon;50" >> records.txt
echo "Dracula;598" >> records.txt
cp records.txt records_2.txt
cat records_2.txt | grep "2" >> records.txt
```

4. óra

Ezen az órán kicsit bővebben megnézzük a “csővezetékelést” Bashben, és a reguláris kifejezésekbe is betekintünk.

Készítsük elő a munkánkat azzal, hogy létrehozunk egy `lesson4` nevű nappát, benne pedig a `people.csv` fájlt az előző óráról ismert tartalommal (legegyszerűbb tehát kopizni).

```
mkdir szgyak/lesson4
cd szgyak/lesson4
cp ../lesson3/people.csv ./ # bemásolja a fájlt az eredeti
                             # néven ebbe a directoryba
```

Csővezeték operátor láncolása

Hasonlóan a `&&` és `||` operátorokhoz, a `|` (pipeline) operátor is láncolható. Általánosan:

```
parancs1 | parancs2 | parancs3
```

Ha a `people.csv` fájlból ki akarjuk olvasni azokat az embereket, akiknek 70-es a telefonszáma és az 1900-as években születtek, akkor azt megtehetjük például így:

```
cat people.csv | grep ";0670" | grep ";19"
```

Egy ábrával szemléltetve ilyenkor a következő történik:

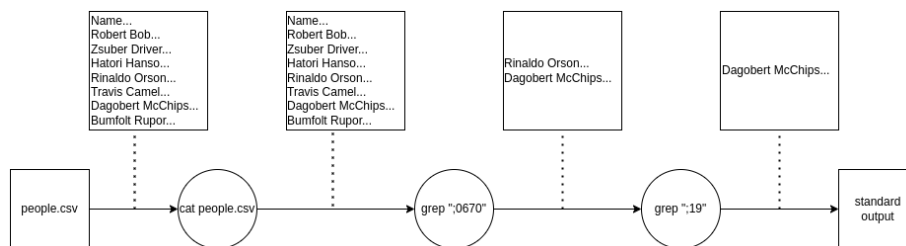


Figure 8: filter_commands

Tehát a `grep` parancs “megszűrte” a bemenetét, mielőtt a kimenetére engedte. Azokat a parancsokat, amelyek így viselkednek, “filter”-eknek (vagy szűrő parancsoknak) nevezzük. **Filter tehát az a parancs, ami a bemenetről olvas, csinál valamit, majd kiírja az eredményt.** Ezekből néhány: - `head` az első `x` sort írja ki a kimenetre. - `tail` az utolsó `x` sort írja ki a kimenetre. - `sort` rendezi a bemenetet, és kiírja a kimenetre. - `grep` ismerjük. - `wc` megszámolja a szavak, karakterek, sorok számát, és kiírja a kimenetre. - `rev` soronként fordítva írja ki a bemenetet. - `uniq` rendezett adatokban vizsgál számosságot és egyediséget (például megnézi, hogy egy sor hányszor szerepel). - `cut` oszlopokat vág ki az adatból. - `tr` karaktereket cserél ki az adatban.

Nézzünk meg néhányat példákon keresztül!

1. példa

Írjuk ki a `people.csv` fájl utolsó három sorát!

```
cat people.csv | tail --lines=3
# A következő parancs ugyanezt csinálja
# (csak nem kell kiírni, hogy '--lines')
cat people.csv | tail -3
```

A `tail` parancs ugyanúgy szűrőként működik, mint a `grep`, csak más műveletet hajt végre. A parancs neve után megadott számnak megfelelő mennyiségű sort

írja ki a fájl végéről.

2. példa

Írjuk ki a `people.csv` fájl első három sorát, fejléccel együtt!

```
cat people.csv | head --lines=3
```

A `head` ugyanúgy működik, mint a `tail`, csak nem az utolsó, hanem az első sorokat írja ki.

3. példa

Írjuk ki a `people.csv` fájl utolsó olyan sorát, ahol a telefonszám harmincas!

```
cat people.csv | grep ";0630" | tail -1
```

Vagy:

```
cat people.csv | grep ";0630" | tail --lines=1
```

A fenti parancs az eddigiek alapján könnyen érthető, ha tudjuk, hogy a csővezeték sok parancson keresztül is láncolhatjuk.

4. példa

Írjuk ki a `people.csv` fájl összes sorát, kivéve az utolsó ötöt!

```
cat people.csv | head --lines=-4
```

Vagy a következő is ugyanezt csinálja

```
cat people.csv | head
```

Ha a `--lines` után negatív számot adunk meg, akkor nem az első n darabot írja ki, hanem az utolsó n darabot **nem** írja ki. Hasonló a `tail` működése is, egy kis csavarral, amit meglátunk a következő példában.

5. példa

Írjuk ki a `people.csv` fájl első három sorát, fejléc nélkül!

```
cat people.csv | tail --lines=+2 | head --lines=3
```

Vagy rövidebben:

```
cat people.csv | tail +2 | head -3
```

Ebben az esetben a `tail` az n -edik sornál kezdi a kiírást, ahol a számot `+` jellel adjuk meg (nem minusszal, mint a `head`-nél).

6. példa

Írjuk ki a `/dev` mappa első 10 fájlját vagy directory-ját, akkor is, ha azok rejtettek! Ne jelenjenek meg a `.` és `..` speciális jegyzékek!

```
ls -la /dev | tail +4 | head -10
```

Itt az `ls -la` parancs első három sorát vágtuk le (ami a *total number of disk blocks* sor és a két speciális directory sora), majd a maradékból meghagytuk az első 10-et.

Ez a példa közelebb áll ahhoz, amire gyakorlatban is használjuk a Bash nyelvet. A csv fájlön könnyű megmutatni ezeknek a parancsoknak a működését, de a gyakorlatban nem csak fájlok tartalmán, hanem parancsok kimenetén is dolgozunk. Ezek akkor szoktak előjönni, mikor valami command line (parancssori) eszközt használunk, ami listáz nekünk valamit. Ilyeneket jelenleg nem igazán ismerünk, és nem is fogunk, amíg valami valós vagy egyetemi projektben nem jönnek szembe velünk.

Ettől függetlenül megemlíteném, hogy például a `docker` eszközt. Ez virtuális gépeket kezel, és megfelelő kapcsolókkal kiírja nekünk a jelenleg működő virtuális gépeket, a nevükkel és az azonosítójukkal. Ekkor a `grep` parancs jól jön nekünk, hogy a sok virtuális gép közül megtaláljuk a megfelelő nevűt. Ha nagyon command line mágusok vagyunk, akkor a név mellé az id-t is kiszedhetjük bash segítségével, és akár a gépet is leállíthatjuk. Ez bajlódós keresés, kijelölés, kopizás és beillesztés lenne, de ha csinálunk rá magunknak egy scriptet, akkor egyetlen parancsból megoldható.

A bash egyik nagy előnye abban van, hogy gyakran ismételt command line parancsokat automatizálhatunk.

7. példa

Írjuk ki a `/dev` mappa utolsó 5 diskjét (most tekintsük azokat a fájlokat disknek, amiknek a tulajdonos csoportja `disk`!)

```
ls -la /dev | grep " disk " | tail -5
```

8. példa

Írjuk ki a `people.csv` fájlból csak az emberek neveit (a fejléc nélkül)!

```
cat people.csv | tail +2 | cut --field=1 --delimiter=';'
# Vagy ugyanez rövidebben
cat people.csv | tail +2 | cut -f 1 -d ';'
```

A fenti parancs utolsó eleme magyarázatra szorul, tehát a `cut` parancs. Ez egy oszlopokból álló szövegből kiolvassa a megfelelő oszlopokat. Egy oszlop lehet bájt vagy karakter méretű is, de nekünk most *mezőkre* van szükségünk. A `--fields=1` kapcsoló megmondja, hogy hányadik mezőt szeretnénk kiolvasni a bemeneti szövegből. A mezőket viszont a fájlön belül el kell választani valamivel, amit általában *delimiter* néven ismerünk a szaknyelvben. A `--delimiter=';'` kapcsoló megmondja, hogy milyen karakterek adják az oszlophatárokat a bemeneti szövegben (esetünkben a pontosvessző).

9. példa

Írjuk ki a `people.csv` fájlból csak a telefonszámokat, és azokból is csak az első 20-ast!

```
cat people.csv | grep ";0620" | cut -f 3 -d ';' | head -1
```

10. példa

Írjuk ki a `people.csv` fájlból az emberek neveit betűrendben (csak a neveket).

```
cat people.csv | tail +2 | cut -f 1 -d ';' | sort
```

11. példa

Írjuk ki a `people.csv` fájlból a születési hónapokat (csak a hónapokat)!

```
cat people.csv | cut -f 2 -d ';' | cut -f 2 -d '.'
```

Ezen a ponton kezdhethetjük érezni, hogy elég messzire lehet menni ezeknek a parancsoknak a kombinálásával. Sok parancs van, saját kapcsolókkal rendelkeznek, és az eddigiekhez hasonlóan működnek. További parancsok megismerése és bonyolítás helyett térjünk át a reguláris kifejezésekre, amelyek szintén hasznosak lesznek nekünk!

Reguláris kifejezések

Az egyetlen egyszer a következőt mondta nekem az egyik tanár: “Az informatikában a két legnehezebb dolog a caching és a reguláris kifejezések.”

A reguláris kifejezéseket a *text matching* feladatához használják. **Egy reguláris kifejezés egy minta, aminek ismeretében egy “reguláris kifejezés motor” egyértelműen meg tudja állapítani, hogy egy adott szövegre illeszkedik-e a kifejezés.** Példa alapján érthetőbb: az alábbiak reguláris kifejezések, mellettük pedig azok, amikre illeszkednek:

- `[0-9]{3}`: minden három hosszúságú számsorra illeszkedik.
- `^[A-Z][a-z]+`: minden nagybetűvel kezdődő szóra illeszkedik.

A továbbiakban megismerjük néhány építőelemét a reguláris kifejezéseknek. Egy rövid összefoglalót mutatok be ebben a jegyzetben, de ennél sokkal bőségesebb a téma, és az internet rengeteg információt tartalmaz vele kapcsolatban. A reguláris kifejezéseket széles körben használják, például egy weboldal formjának validálásakor (megmondják vele, hogy a szöveges mező milyen felépítésű szöveget tartalmazhat), vagy épp keresésekkor (megmondják vele, hogy milyen felépítésű szöveget keresnek). Hasznos, de bonyolult téma, és nem fogunk nagyon elmélyülni benne.

Továbbra is a `people.csv` fájl tartalmán fogunk gyakorolni, de javasolom, hogy most az egyszer hagyjuk itt egy időre a terminált, mert csak véget nem érő szenvedést fog okozni nekünk.

Egy utolsó dologra azonban még rávilágítanék a Bash parancsokkal kapcsolatban: láttuk, hogy az órán vegyesen használtam a kétféle idézőjelet, az egyszerű (') és a kétszerest ("). Ennek gyakorlati jelentősége is lesz innentől, ugyanis az egyszerű idézőjel **csak szöveget** tartalmaz, míg a kétszeres **tartalmazhat értelmezendő változókat**. Ezt a jövő órán bővebben látni fogjuk, egyelőre azonban elégedjünk meg azzal, hogy **a reguláris kifejezéseket érdemes egyszerűes idézőjelek közé írni!**

Adjunk új tartalmat a `people.csv` fájlunk, hogy bonyolultabb legyen a feladat, és indokolt legyen a reguláris kifejezések használata! Az új tartalom:

```
Name;Birthdate;Phone;Email;Skill
Robert Bob;1997.09.12.;06201975555;bob2bob@gmail.com;IT
Zsuber Driver;1988.10.11.;06404445555;uber@gmail.com;Driving
Hatori Hanso;1966.01.11.;063012345;jean'dark@gmail.com;Smithing
Rinaldo Orson;2001.05.24.;06709330000;D&D@gmail.com;IT,Weight lifting
Travis Rick Camel;1970.10.01.;06301717171;mummy@citromail.hu;Horses
Dagobert McChips;1956.08.31.;067001111;clap.clapgmail.com;Cooking,Catery
Bumfalt Rupor;1967.09.11.;06201112233;HP123@mg.mh.hu;Marketing
Mad Marx;1965.09.11.;06301442233;money.bad@gmail.com;Motorcycle Riding
Borland Ci Rattleneck;1980.11.02;+36304225555;teller@citromail.hu;Bottleneck Construction
Baileys Margareti;1999.12.09;06209876543;b1gdr1nk@HOND.hu;Drinking,Catering
Larry Lipstick;1988.07.22;+36206547382;baby@gmail.com;Hairstyling
Inigo Montoya;1977.04.11;+367065682;pr3pare@gmail.to;Swordfighting
```

A <https://regex101.com/> weboldalon van lehetőség interaktívan és viszonylag vizuálisan kipróbálni a reguláris kifejezéseket. Az alábbi screenshot illusztrálja, hogy mit kell látnunk a weboldalon. Másoljuk be a `people.csv` tartalmát oda, ahol a képen is látható (a legnagyobb textboxba középen).

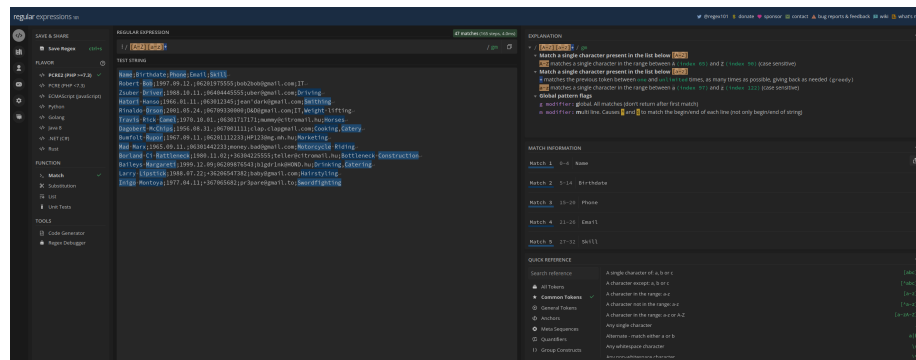


Figure 9: regex

A jobb jobb felső sarokban angol nyelvű magyarázatok láthatók a beírt kifejezések jelentésére. Középen felülre tudunk beírni reguláris kifejezéseket, amikkel keresünk a szövegben. A jobb alsó sarokban található pár rövid

emlékeztető, hogy mik a leggyakoribb reguláris kifejezések. Néhányat felsorolok itt, aztán példákon keresztül nézzük meg őket!

- A `.` bármely 1 db karakterre illeszkedik.
- A `?` karakter megmondja, hogy az előző karakter 1 vagy 0 alkalommal fordulhat elő (tehát `blabla?` kifejezés illeszkedik `blabla` vagy `blabl` szövegekre).
- A `*` karakter megmondja, hogy az előző karakter akárhányszor előfordulhat, akár 0-szor is (tehát `blabla*` kifejezés illeszkedik `blabla`, `blablaaaaaaa` vagy `blabl` szövegekre is).
- A `+` jel az előző karakter 1 vagy többszöri ismétlődését jelenti (tehát `blabla+` kifejezés illeszkedik `blabla` vagy `blablaaaaa` szövegekre is, de **nem** illeszkedik `blabl` szövegre).
- A `[]` szögletes zárójeleken belül megadott karakterekre (például `[abc]`) vagy karakterintervallumokra (például `[a-z]`) illeszkedik, de csak egyre.
- A `[^]` szögletes zárójeleken belül megadott karakterekre **nem** illeszkedik (az előző negálása), csak bármelyik másik egyetlen karakterre.
- A `{,}` kapcsos zárójeleken belül számszerűen megadható, hogy az előző karakterekből mennyi forduljon elő. Például `{2,4}` azt jelenti, hogy minimum 2, de maximum 4 karakterre fog illeszkedni. `{N,}` jel esetén N vagy több karakterre illeszkedik. `{N}` esetén pontosan N karakterre illeszkedik. Megjegyzendő, hogy a `?`, `*`, `+` karakterek megadhatók kapcsos zárójel intervallumokkal is, a következőképpen: `{0,1}` = `?`, `{0,}` = `*`, `{1,}` = `+`.
- A `^` karakter a sor elejére illeszkedik.
- A `$` jel a sor végére illeszkedik.

Ezek a leggyakoribb használt reguláris kifejezés építőelemek. Ebben a kurzusban nem lesz másra szükség.

A reguláris kifejezések a `grep` paranccsal fogjuk használni. Ahhoz, hogy a `grep` parancs reguláris kifejezéseket értelmezzen, meg kell neki adni az `-E` kapcsolót (mint `Expression`), majd a reguláris kifejezést javasoltan egyszeres idézőjelek között. Mielőtt beadjuk a `grep` parancsnak a kifejezést, győződjünk meg róla a fenti weboldalon, hogy helyesen adtuk meg a kifejezésünket!

12. példa

Írjuk ki a `people.csv` fájlból azokat az embereket, akiknek R-rel kezdődik a keresztnéve!

```
cat people.csv | grep -E '^R'
```

A `^` jel a sor elejét jelöli ki nekünk, tehát arra kerestünk rá, hogy melyik sor kezdődik R-rel.

13. példa

Írjuk ki a `people.csv` fájlból azokat az embereket, akiknek R-rel kezdődik a vezetéknéve!

```
cat people.csv | grep -E '^[a-zA-Z ]+R[a-z]+;'
```

A fenti kifejezés illeszkedik minden olyan nagy R betűre, ami az első pontosvessző előtti szó elején van, és csak betűk és szóközők előzik meg.

14. példa

Írjuk ki a `people.csv` fájlból azokat az embereket, akiknek legalább két skillje van!

```
cat people.csv | grep -E ',[a-zA-Z ]+$'
```

Arra kerestünk rá, hogy legyen vessző a sor végi szavak előtt.

15. példa

Írjuk ki a `people.csv` fájlból azokat az embereket, akiknek három neve van!

```
cat people.csv | grep -E '^[a-zA-Z]+ [a-zA-Z]+ [a-zA-Z]+;'
```

16. példa

Írjuk ki a `people.csv` fájlból azokat az embereket, akiknek valid az email címe! (valid email cím legyen most az, aminek a felhasználónevében csak betű, szám és `.` karakter van, amiket `@` követ, majd kisbetűkből álló csoportnév, amitől `.` választja el a záró, minimum 2, maximum 4 karakteres jelölőt; a felhasználónév minimum 3, maximum 100, a csoportnév minimum 3, maximum 20 karakter legyen)

```
cat people.csv | grep -E ';[0-zA-Z0-9.]{3,100}@[a-z]{3,20}\.[a-z]{2,4};'
```

A fenti szövegben felhívnám a figyelmet a *backslash* karakter (`\`) használatára a `.` karakter előtt. A `.` karakternek alpból jelentése van, ezért ha azt egyszerű karakterként akarjuk használni, be kell írunk elé egy *backslash* karaktert. Ugyanez igaz a többi speciális karakterre is (`? + { } \` stb.)

Megjegyzések

A korábbiakban a `cat` parancsot használtuk egy fájl kilistázására. Ez olyan szempontból jó volt, hogy megtanultuk a parancsot és a csővezetékkelést is szemléltette. A fenti példákban viszont valójában elhagyható. A `grep` parancsnak utolsó paraméterébe beírható a fájl neve, amiben keresni akarunk. Ugyanez igaz a többi megtekintett szűrő parancsra is. Például valid a következő:

```
head -1 people.csv
```

Ezzel a módszerrel viszont nem lehet például az `ls` parancs tartalmán dolgozni (bár a későbbiekben fogunk ciklust írni rá, hogy végigiteráljunk egy directory tartalmán). Tehát a következő nem fog működni:

```
head -1 ls
```

Feladatok

Tekintsünk néhány önálló gyakorló feladatot! A `people.csv` fájl tartalma az új, frissített tartalom legyen!

1. feladat

Írjuk ki a nevük szerint betűrendben azokat, akiknek egyetlen Skillje van!

2. feladat

Írjuk ki azoknak a nevét és a telefonszámát, akiknek az egyik neve o-val végződik!

3. feladat

Írjuk ki azoknak az embereknek a keresztnévét, akiknek a telefonszáma +36-tal kezdődik és valid (9 szám követi a +36-ot, és vagy 20-as, vagy 30-as, vagy 70-es)!

4. feladat

Írjuk ki azoknak az embereknek a vezetéknévét, akiknek a telefonszáma 06-tal kezdődik és valid!

5. feladat

Írjuk ki azoknak az embereknek a nevét, akik páros napon születtek!

4. Kérdések

Ellenőrző kérdések a harmadik gyakorlathoz.

1. kérdés

Mik azok a szűrő parancsok?

- Minden parancs előtt lefutó ellenőrző utasítások.
- Egy fájl tartalmán dolgozó módosító operátorok.
- Egy bizonyos inputot befogadó, majd annak egy módosított verzióját az outputra író parancsok.
- Adatbázis utasítások, amelyek csökkentik egy perzisztens fájlból kinyert adat számosságát.

2. kérdés

Mik azok a reguláris kifejezések?

- Egy minta, amellyel megadható, hogy a kifejezés milyen szövegre illeszkedik.
- Formális nyelvi leírásai a felhasználói követelményeknek egy Bash programmal szemben.

- c. Egy formális nyelvi leírás, amely megmondja, hogy a felhasználói input helyese.
- d. Egy keresőmotorok megadható szöveges kifejezés, amellyel pontosabban lehet keresni egy szöveges inputban.

Most már ismerjük a vizsgáló operátorokat. Nézzük, milyen szintaxissal tudjuk ténylegesen használni őket!

&& és || szerkezet

Ezt már láttuk korábban. Továbbra is ugyanazt csinálják, mint eddig, csak a `test` parancs segítségével feltételvizsgálatot társíthatunk hozzájuk. Például:

```
x=10; y=20
test $x -lt $y && echo "Smaller" # -lt means less-than
test $x -gt $y && echo "Greater" # -gt means greater-than
```

Ez a fajta feltételmegadás egysorosok esetén használható jól. Például a következő kód létrehoz egy fájlt, ha az nem létezik.

```
test -e some_file.txt || touch some_file.txt # -e means exists
```

Létezik két segédprogram is a rendszeren, a `true` és a `false`. Ezek a nevékhöz ilően úgy viselkednek, mintha egy mindig igazra (`true`) vagy egy mindig hamisra (`false`) kiértékelődő feltételt írtunk volna a helyükre. Például:

```
true && echo "This is written"
true || echo "This is NOT written"
false && echo "This is NOT written"
false || echo "This is written"
```

A `test` parancsoknak van egy *shortcut*-ja is, szögletes zárójelek formájában. A következő két sor parancs ugyanaz.

```
test 1 -ne 2 && echo "Thing"
[ 1 -ne 2 ] && echo "Same thing"
```

Ez az `if` szerkezetnél lesz intuitívabb. Szintén az `if` szerkezetnél fogjuk használni a `[[feltétel]]`, tehát a kétszeres szögletes zárójeles szerkezetet is.

```
[[ 1 -ne 2 ]] && echo "Almost the same thing"
```

Ez majdnem ugyanolyan, mint az egyszeres szögletes zárójel (és ennél fogva a `test` parancs), kivéve, hogy ez többféle szintaxist megenged. Például csak ebben tudjuk használni a reguláris kifejezés összehasonlító operátort `=~`.

A továbbiakban leginkább az `if-else` szerkezetet használjuk, mert ez rugalmasabb.

if-else szerkezet

Az if szerkezetek írásakor ugyanúgy használhatjuk a `test` operátort, mint eddig:

```
if test 1 -eq 1; then
    echo "true"
fi
```

A következőt azonban (főleg imperatív nyelvek ismeretében) intuitívabb olvasni:

```
if [ 1 -eq 1 ]; then
    echo "true"
fi
```

A két fenti kifejezés ugyanazt jelenti. A továbbiakban mi többnyire a lenti kifejezést használjuk majd, mert gazdagabb feltételmegfogalmazást biztosít nekünk:

```
if [[ 1 -eq 1 ]]; then
    echo "true"
fi
```

Ennél az egyszerű példánál természetesen mindhárom ugyanazt jelenti.

Az if szerkezet sajátossága Bashben a többi nyelvhez képest, hogy a lezáró parancsa a `fi`, ami az `if` fordítva. Ezt a logikát máshol is látjuk majd.

Az if szerkezethez rakhatunk `else` ágat is:

```
if [[ 1 -ne 1 ]]; then
    echo "true"
else
    echo "false"
fi
```

Az `else-if` szerkezetet a Bash a következőképpen oldja meg:

```
x=10
if [[ $x -eq 1 ]]; then
    echo "equal"
elif [[ $x -lt 1 ]]; then
    echo "less than"
else
    echo "greater than"
fi
```

Látható, hogy az `else-if` szavak összevonódtak az `elif` szóba.

A C-ből ismert `switch-case` szerkezet is megtalálható itt, a következő szintaxissal:

```
read -p "Which type of person are you, 0, 1, 2 or 3" ptype
case $ptype in
```

```

0)
  echo "You are chaotic evil.>";
1)
  echo "You are a mermaid.>";
2)
  echo "You are phylosopher.>";
3)
  echo "You are a Tauren Druid.>";
*)
  echo "That's not a person type, it's a default!>";
esac

```

A [[]] szerkezetben hasonlóan tudjuk használni az ÉS, VAGY, NEM logikai kifejezéseket, ahogy C-ben. Például a következő ellenőrzi, hogy x 10 és 20 között legyen:

```

if [[ $x -gt 10 && $x -lt 20 ]]; then
  echo "OK"
fi

```

Vagy hogy NE legyen szám:

```

reg_ex='^[0-9]+$'
# A felkiáltójel a következő kifejezés tagadása
if [[ ! $x =~ $reg_ex ]]; then
  echo "Not a number."
fi

```

Most már nagyjából mindent tudunk, amire a feltételes kódjainkhoz szükség lesz. Lássunk példákat!

Példák

A következő példákban gyakoriak lesznek az input ellenőrzések. Amennyiben egy input nem megfelelő, hibával lépünk ki a programból. A hiba jelzésére a program visszatérési értéke alkalmas. Ha a visszatérési érték nem nulla, az hibát jelent. Idő előtt kilépni az `exit` paranccsal tudunk.

Például:

```

#!/bin/bash

if [ 1 -ne 1 ]; # very unexpected error
  exit 1
fi

```

Ha nem írunk számot az `exit` után (vagy nem írunk `exit` parancsot a programba), akkor a visszatérési érték automatikusan nulla lesz (még akkor is, ha valamelyik parancs hibát adott a programon belül). Például a következő program nem tér vissza hibával, bár a `grep` maga hibára fut.

```
#!/bin/bash

grep "hello" nonexistent_file_blabla.txt
```

1. példa

Készítsünk egy shell scriptet, ami bemenetként egy téglalap két oldalának hosszát várja, és kiírja a síkidom területét! Valósítsuk meg csak egész számokkal! (Természetesen végezzünk ellenőrzéseket az inputon!)

```
#!/bin/bash

if [[ $# -ne 2 ]]; then
    echo "Two inputs are needed, $# is given." 1>&2
    exit 1
fi

reg_ex='^[0-9]+$'
if [[ ! $1 =~ $reg_ex || ! $2 =~ $reg_ex ]]; then
    echo "Both inputs must be positive integer numbers!" 1>&2
    exit 1
fi

echo "Area is: $((($1 * $2)))"
```

A `1>&2` kifejezés az `echo` után átirányítja a standard output streamet a standard error streamre.

Az első `if` kifejezés ellenőrzi, hogy pontosan két inputot adtunk-e meg.

A második `if` kifejezés biztosítja, hogy ha az első vagy a második kifejezés nem felel meg az egész számokra vonatkozó reguláris kifejezésünknek, akkor hibáüzenettel lépünk ki.

2. példa

Legyen adott egy fájl `name_id_pairs.txt` néven, ami id és név párosokat tartalmaz. Készítsünk egy shell scriptet, ami bekéri a nevet, és kiírja a hozzá tartozó id-t, vagy hibát ad, ha a név nem található a fájlban.

A fájl tartalma legyen a következő:

```
zsuzso:aef7421b
alfonz:eef4555b
barnabas:98726371
tom:aaaabbbb
jerry:cccdddee
hasselhoff:cafebabe
```

A következő script megvalósítja a feladatot:

```
#!/bin/bash

read -p "Please, type the name you are searching for! " name

row=$(cat name_id_pairs.txt | grep $name)

if [[ -z $row ]]; then
    echo "No name like that." 1>&2
    exit 1
fi

id=$(echo $row | cut -d ':' -f 2)
echo $id
```

3. példa

Az MVK Zrt. elérhetővé tesz egy szabványos GTFS adatbázist a fejlesztők számára, hogy a menetrendi adatokat a saját applikációikba tudják integrálni. Írjunk egy shell script fájlt, amely letölti ezt az adatbázist, és kilistázza belőle azokat az utakat, amelyek a Centrumból indulnak, vagy a Centrumba mennek!

Parancsok: wget, unzip (kitömörítésre), cat, grep

Szükséges ellenőrzések: - Ha a letöltendő fájl már egyszer le volt töltve, akkor az újbóli letöltés előtt töröljük az előző verziót! - Ha egy mappába már korábban ki lett tömörítve a letöltött állomány, akkor az újbóli kitömörítés előtt szabaduljunk meg ennek a mappának a tartalmától!

Megjegyzés: A tanszéki gépekről *certificate* problémák miatt a korábbiakban nem volt letölthető az adatbázis, ezért egy *github* repository-ba is feltettem. A hozzá tartozó parancs ki van kommentezve. Hibaüzenet esetén cseréljük le a jelenlegi url-t a következőre: <https://raw.githubusercontent.com/bbalage/Bash.Examples/master/assets/gtfs.zip>

```
#!/bin/bash

if [ -e gtfs.zip ]; then
    rm gtfs.zip
fi

if [ -d gtfs ]; then
    rm -r gtfs
fi

wget "https://gtfsapi.mvkzrt.hu/gtfs/gtfs.zip"
unzip gtfs.zip -d gtfs # próbáljuk ki -d kapcsoló nélkül is...
cat gtfs/routes.txt | grep "Centrum"
```

4. példa

Készítsünk egy shell scriptet, ami bekéri a felhasználó születési dátumát yyyy.mm.dd formátumban! Ellenőrizzük le a dátum helyességét, és írjuk ki, hogy a felhasználó hány éves!

Használjuk a `date` parancsot a jelenlegi dátum lekérésére!

```
#!/bin/bash

read -p "Please, type your birthdate in yyyy.mm.dd format! " bdate

reg_ex='^[0-9]{4}\.[0-9]{2}\.[0-9]{2}$'
if [[ ! $bdate =~ $reg_ex ]]; then
    echo "Date is not in proper format!" 1>&2
    exit 1
fi

byear=$(echo $bdate | cut -f 1 -d '.')
bmonth=$(echo $bdate | cut -f 2 -d '.')
bday=$(echo $bdate | cut -f 3 -d '.')

# A date parancs le tudja ellenőrizni egy szintaktikailag helyes
# dátum validitását. Például a hónap nem-e nagyobb 12-nél, stb.
date -d "$byear-$bmonth-$bday" > /dev/null || exit 1

bseconds=$(date -d "$byear-$bmonth-$bday" +%s)
cseconds=$(date +%s)

age_in_seconds=$((cseconds - bseconds))

if [[ $age_in_seconds -lt 0 ]]; then
    echo "You haven't been born yet!" 1>&2
    exit 1
fi

echo $(( age_in_seconds / 60 / 60 / 24 / 365 ))
```

Feladatok

1. feladat

Valósítsuk meg az 1. példa feladatát, de ezúttal lebegőpontos számokkal!

2. feladat

Valósítsuk meg a 2. példa feladatát, de ezúttal ne csak `name_id_pairs.txt` nevű fájlra működjön, hanem bármilyen nevű fájlra! A fájl nevét a script bemeneti

paraméterként fogadja! Ellenőrizzük, hogy a fájl létezik és olvasható-e, mielőtt a funkciók további részét megvalósítjuk!

3. feladat

Valósítsuk meg a 3. *példa* feladatát, de ezúttal a Centrum helyett bármelyik végállomást fogadjuk el, és bemeneti paraméterként adjuk át azt a scriptnek. Ha nincs ilyen végállomás, írjunk hibaüzenetet!

6. Kérdések

Ellenőrző kérdések a hatodik gyakorlathoz.

1. kérdés

Mit csinál az alábbi kód?

```
#!/bin/bash

reg_ex='^-?[0-9]+\.[0-9]*$'
if [ $1 =~ $reg_ex ]; then
    echo "OK"
else
    echo "Not OK"
    exit 1
fi
```

- Hibára fut.
- Ellenőrzi egy szövegről, hogy dátum-e.
- Ellenőrzi, hogy a bemeneti paraméter létezik-e.
- Ellenőrzi, hogy a bemeneti paraméter lebegőpontos szám-e.

2. kérdés

Mit csinál az alábbi kód?

```
#!/bin/bash

reg_ex='^-?[0-9]+\.[0-9]*$'
if [[ $1 =~ $reg_ex ]]; then
    echo "OK"
else
    echo "Not OK"
    exit 1
fi
```

- a. Hibára fut.
- b. Ellenőrzi egy szövegről, hogy dátum-e.
- c. Ellenőrzi, hogy a bemeneti paraméter létezik-e.
- d. Ellenőrzi, hogy a bemeneti paraméter lebegőpontos szám-e.

3. kérdés

Mit csinál az alábbi kód?

```
#!/bin/bash

if [[ ! -d build ]]; then
    mkdir build
fi
```

- a. Buildeli a Bash kódot.
- b. Ha a build directory nem létezik, akkor létrehozza azt.
- c. Ha a build fájl törölhető, akkor törli azt.
- d. Ha a build directory létezik, akkor létrehozza azt.

4. kérdés

Mit csinál az alábbi kód?

```
#!/bin/bash

if [[ "thing.txt" -nt "thing (1).txt" ]]; then
    rm "thing (1).txt"
else
    rm thing.txt
    mv "thing (1).txt" thing.txt
fi
```

- a. Törli a másodjára letöltött thing.txt fájlt.
- b. Ha thing.txt vagy thing (1).txt létezik, akkor törli azt.
- c. Törli a régebbit a thing.txt és a thing (1).txt fájlok közül.
- d. A thing.txt és thing (1).txt fájlok közül az újabbat hagyja meg thing.txt néven.

5. kérdés

Mit csinál az alábbi kód?

```
#!/bin/bash

if [[ -w thing.txt ]]; then
    rm thing.txt
    echo "OK"
elif [[ -e thing.txt ]]; then
    exit 1
    echo "Cannot delete." 1>&2
else
    echo "OK"
fi
```

- Megpróbálja törölni a thing.txt fájlt, és hibát ír, ha nem törölhető, de létezik.
- Törli a thing.txt fájlt, ha az létezik, és hibát ír, ha nem létezik.
- Törli a thing.txt fájlt, ha az írható, és hibával kilép, ha nem írható, de létezik.
- Hibára fut.

7. óra

A hetedik gyakorlaton a ciklusokkal és tömbökkel fogunk foglalkozni.

Ciklusok

Bashben léteznek a már máshonnan bizonyosan ismert `for` és `while` ciklusok, némileg más szintaktikával. Létezik ezen felül `until` ciklus is, ami nem sokban különbözik a `while` ciklustól, mindössze annyiban, hogy nem addig megy, amíg a feltétel igaz, hanem addig, amíg a feltétel hamis.

Nézzünk példákat a cikluskezelésre!

1. példa

Írjuk ki 0-tól 10-ig a számokat, `for`, `while` és `until` ciklussal is!

```
#!/bin/bash

# for ciklussal
for (( i = 0; i <= 10; i++)); do
    printf "$i "
```



```

done
echo

# while ciklussal
i=0
while [ $i -le 10 ]; do
    printf "$i "
    i=$((i + 1))
done
echo

# until ciklussal
i=0
until [ $i -gt 10 ]; do
    printf "$i "
    i=$((i + 1))
done
echo

```

2. példa

Generáljunk egy véletlen számot, és írjuk ki a nála kisebb négyzetszámokat!

A véletlenszám generálás Bashben a RANDOM változón keresztül elérhető. Alább néhány példa generálás látható, adott intervallumokban.

```

echo $((RANDOM % 100)) # [0-99]
echo $((RANDOM % 101)) # [0-100]
echo $((RANDOM % 100 + 1)) # [1-100]
echo $((RANDOM % 50)) # [0-49]
echo $((RANDOM % 51 + 50)) # [50-100]

```

Az alábbi script egy megoldása a feladatnak:

```

#!/bin/bash

rnd=$RANDOM

echo "The number is: $rnd"
echo "Lesser square numbers are: "

for (( i=1; i*i < $rnd; i++ )); do
    echo -n "$((i*i)) "
done

```

3. példa

Kérjünk be egy N értéket, és írjuk ki az első N Fibonacci számot, ahol nullát vesszük a nulladik Fibonacci számnak. Ellenőrizzük, hogy N egy legalább 2 értékű egész szám!

```
read -p "Please, give the value of N (at least 2):" N

reg_ex='^[0-9]+$'
if [[ ! $N =~ $reg_ex ]]; then
    echo "N must be a number!" 1>&2
    exit 1
fi

if [ N -lt 2 ]; then
    echo "N was too small." 1>&2
    exit 1
fi

f_before_prev=0
f_prev=1
echo "0: $f_before_prev"
echo "1: $f_prev"
for ((i = 2; i <= N; i++)); do
    f=$((f_prev + f_before_prev))
    echo "$i: $f"
    f_before_prev=$f_prev
    f_prev=$f
done
```

4. példa

Írjunk egy shell scriptet, ami bemeneti paraméterként egyetlen pozitív számot vár (hibát ír, ha nem ezt kap). A program feladata, hogy kiírja, hogy a kapott szám prím-e.

```
if [ $# -lt 1 ]; then
    echo "No number provided as input." 1>&2
    exit 1
fi

num=$1
re='^[0-9]+$'
if ! [[ $num =~ $re ]] ; then
    echo "error: Not a positive number." 1>&2;
    exit 1
fi
```

```

if [ $num -le 1 ]; then
    echo "Not prime."
    exit 0
fi

square_root=$(echo "scale=0; sqrt($num)" | bc)

for ((i=2;i<=square_root;++i)); do
    if [ $(($num % i)) -eq 0 ]; then
        echo "Not prime."
        exit 0
    fi
done

echo "Prime."

```

Tömbök

Bashben a tömbök használata kötetlenebb, mint egy C-szerű nyelvben. A tömb létrejöttekor nem kell tudnunk annak a hosszát. A gyakorlatban a tömbök inkább listaként üzemelnek, és ezt látjuk is majd a következő kódokban.

Egy tömböt az alábbi módon deklarálhatunk:

```
declare -a arr
```

A tömb elemeinek az alábbi módon adhatunk értéket. A sorrend nem kötött, és elemeket ki is hagyhatunk. Ilyenkor a kihagyott elemek üresek lesznek.

```
arr[0]=12
arr[1]=3
arr[5]=6
```

Ha el akarjuk érni a tömb egy elemét, akkor a következő szintaxissal tehetjük meg:

```
echo ${arr[1]} # a második elem
```

Egy tömböt deklarálhatunk és inicializálhatunk egyszerre. A következő kód létrehoz egy tömböt, és kiírja a tartalmát:

```
declare -a arr=(12 3 6)

# ${#arr[*]} megadja nekünk a tömbelemek számát
for ((i=0;i<${#arr[*]};++i)); do
    echo ${arr[i]}
done
```

A for ciklusnak akad egy másik, hasznos szintaxisa is, amit `for-each` ciklusként szoktak emlegetni, és a legtöbb imperatív nyelvben megtalálható.

```
declare -a arr=(12 3 6)
```

```
# ${arr[*]} a tömb minden elemét adja vissza
for element in ${arr[*]}; do
    echo $element
done
```

Érdekesség, hogy a program bemeneti paramétereinek listáját, és akár egy parancs kimenetét is megkaphatjuk tömbként. Így könnyű végigiterálni rajtuk `for-each` ciklussal. A további példák a tömbökkel foglalkoznak.

5. példa

Írjuk ki összes argumentumot, amit a program kapott!

```
for i in $*; do
    echo $i
done
```

6. példa

Listázzuk egy directory tartalmát, minden egyes fájlt és mappát számozottan! A directory elérési útját bemeneti paraméterként kapjuk meg. Ellenőrizzük, hogy kaptunk-e bemeneti paramétert, és hogy az tényleg egy létező directory-e!

```
#!/bin/bash

# Itt azt nézzük, hogy üres-e a változó
if [[ -z $1 ]]; then
    echo "No input given." 1>&2
    exit 1
fi

if [[ ! -d $1 ]]; then
    echo "Not an actual directory." 1>&2
    exit 1
fi

i=0
for name in $(ls $1); do
    echo "$i: $name"
    i=$((i+1))
done
```

7. példakód

Töltsünk fel egy 10 elemű tömböt véletlen számokkal, majd menjünk végig a számokon és növeljük meg őket 1-gyel!

```
#!/bin/bash

declare -a arr

for ((i=0; i<10; ++i)); do
    # random szám 0-tól 99-ig
    arr[$i]=$((RANDOM % 100))
    printf "${arr[$i]} "
done
echo

for ((i=0; i<10; ++i)); do
    arr[$i]=$((arr[i] + 1))
done

for element in ${arr[*]}; do
    printf "$element "
done
echo
```

8. példa

Hozzunk létre egy N elemű tömböt, ahol N-t `read` paranccsal kérjük be, és ellenőrizzük, hogy pozitív egész szám (hibával visszatérünk, ha nem az). Töltsük fel véletlen számokkal a tömböt, ahol a véletlen számok az [1-100] intervallumból kerülnek ki! Ezután végezzük el a következő műveleteket a tömbre: - Minimum elem kiírása (hányadik elem és mi az értéke). - Maximum elem kiírása (hányadik elem és mi az értéke). - Írjuk ki az elemek összegét. - Írjuk ki az elemek átlagát.

```
read -p "Please type the size of the array!" N

re='^[0-9]+$'
if ! [[ $N =~ $re ]] ; then
    echo "error: Not a positive number." 1>&2;
    exit 1
fi

if [ $N -eq 0 ]; then
    echo "error: Array size is 0." 1>&2
    exit 1
fi
```

```

declare -a arr

for ((i=0; i<N; ++i)); do
    # random szám 1-től 100-ig
    arr[$i]=$((RANDOM % 100 + 1))
    printf "${arr[$i]} "
done
echo

sum=arr[0]
mini=0
maxi=0
for ((i=1; i<N; ++i)); do
    if [ ${arr[$i]} -gt ${arr[$maxi]} ]; then
        maxi=$i
    fi
    if [ ${arr[$i]} -lt ${arr[$mini]} ]; then
        mini=$i
    fi
    sum=$((sum + arr[i]))
done

echo "Minimum: $mini - ${arr[$mini]}."
echo "Maximum: $maxi - ${arr[$maxi]}."
echo "Sum: $sum."
avg=$((echo "scale=4; $sum / $N" | bc))
echo "Average: $avg."

```

Feladatok

Önálló feladatok gyakorlásra.

1. feladat

Kérjünk be egy pozitív egész számot a felhasználótól, majd írjuk azokat a páratlan, pozitív egészeket, amik ettől a számtól kisebbek!

2. feladat

Valósítsuk meg a 6. példában írtakat, de ezúttal a scriptünk tetszőleges mennyiségű input directory-t fogadhat. Ellenőrizzük, hogy kaptunk legalább egy bemeneti paramétert, és hogy minden bemeneti paraméter directory-e. Ha igen, akkor írjuk ki mindegyik tartalmát! A számozást ne szakítsuk meg!

3. feladat

Írjunk egy scriptet, ami letölti a következő .zip fájlt az internetről:
<https://raw.githubusercontent.com/bbalage/BashExamples/master/assets/nums.zip>

A script tömörítse ki a mappát, majd másolja át az összes .txt fájlt belőle egy out_nums mappába, de úgy, hogy a .txt fájlok minden olyan sorát, ahol szám van, ossza el kettővel!

7. Kérdések

Ellenőrző kérdések a hetedik gyakorlathoz.

1. kérdés

Mit csinál az alábbi kód?

```
#!/bin/bash  
  
for i in $*; do  
    echo $i  
done
```

- Kiírja a bemeneti argumentumokat.
- Kiírja a bemeneti argumentumok számát.
- Megszámolja a bemeneti argumentumokat.
- Hibára fut.

2. kérdés

Mit ír ki az alábbi kód?

```
#!/bin/bash  
  
i=0  
until [ $i -lt 10 ]; do  
    i=$((i + 2))  
    echo $i  
done
```

- Kiírja a páros számokat 2-től 10-ig.
- Kiírja a páros számokat 0-től 8-ig.
- Nem ír ki semmit.

d. Hibát ír.

3. kérdés

Mit ír ki az alábbi kód?

```
#!/bin/bash

i=0
j=20
while [ $i -lt $j ]; do
    echo $((i + j))
    i=$((i + 1))
    j=$((j - 2))
done
```

- a. Hibára fut.
- b. A számokat 20-tól 14-ig.
- c. A számokat 20-tól 15-ig.
- d. A számokat 19-től 14-ig.

8. óra

A nyolcadik gyakorlaton a függvényekkel fogunk foglalkozni.

Függvények

Bashben a függvények nem egészen úgy működnek, mint C-ben, vagy más imperatív nyelvekben. Itt is szervezési egységek, amiket többször meghívhatunk a programunkban, vagy éppen olvashatóbbá tehetjük általuk a kódunkat. A különbségük a szintaktikán kívül a visszatérési értékek kezelésében van.

Legpraktikusabb úgy gondolni a Bash függvények, mintha maguk is scriptek vagy segédprogramok lennének, mint például `cut`, `grep`, `cat`, stb. Míg C-ben a visszatérési érték többnyire arra szolgál, hogy egy számított értéket visszaadjon nekünk, addig itt sikert vagy hibát jelez.

Legegyszerűbb megérteni példákon keresztül.

Bashben az alábbi módon tudunk függvényt definiálni, és meghívni:

```
f() {
    echo "Hello world!"
}

f
```


Az alábbi szintaxis is megengedett (tetszés kérdése):

```
function f {
    echo "Hello world!"
}
```

```
f
# Prints Hello world!
```

Gondoljunk úgy a függvényre, mintha egy kisebb parancs lenne! Argumentumokat is ennek megfelelően adunk neki:

```
function add_two_numbers {
    echo "$(($1 + $2))"
}
```

```
add_two_numbers 10 30
# Prints 40
```

Változóba tudjuk tenni a kimenetét, mintha parancs lenne.

```
function add_two_numbers {
    echo "$(($1 + $2))"
}
```

```
sum=$(add_two_numbers 10 30)
add_two_numbers $sum 15
# Prints 55
```

A visszatérési értéke nem arra szolgál, hogy az eredményt adja vissza, hanem hogy hibát jelezzon. Például:

```
function add_two_numbers {
    regex="^[0-9]+$"
    if [[ ! $1 =~ $regex || ! $2 =~ $regex ]]; then
        echo "Must be two numbers!" 1>&2
        return 1 # We use "return", not "exit".
                 # "exit" would quit the entire script
    fi
    echo "$(($1 + $2))"
    # The default return value is 0
}
```

```
add_two_numbers 10 20 && echo "OK" || echo "Not OK" # This succeeds
add_two_numbers 10 e0 && echo "OK" || echo "Not OK" # This fails
```

A visszatérési értéket megkaphatjuk a \$? kifejezéssel is, amely mindig a legutóbb kiadott parancs vagy meghívott függvény visszatérési értékét adja vissza:

```

function add_two_numbers {
    regex="^[0-9]+$"
    if [[ ! $1 =~ $regex || ! $2 =~ $regex ]]; then
        echo "Must be two numbers!" 1>&2
        return 1
    fi
    echo "$(($1 + $2))"
}

```

```

add_two_numbers 10 20
echo $? # Prints 0
add_two_numbers 10 e0
echo $? # Prints 1

```

Vegyük észre, hogy `return` utasítást használunk `exit` helyett, de ezen kívül minden ugyanolyan, mintha egy scriptet írnánk. Hivatkozhatunk tehát a korábbi órákon tanultakra.

FONTOS: a változók láthatóságában különbség van a scriptek és a függvények között. Az alábbi példa futtatásával ezt jól meg tudjuk figyelni:

```

x=10
f() {
    x=$((x+1))
    echo $x
}

```

```

f
f
f

```

A fenti kód 11, 12, 13-at ír ki. Ha C nyelvű kifejezésekkel akarunk élni, akkor az `x` változót globálisnak mondhatnánk. Ha erre nem számítunk, akkor váratlan hibáink lesznek.

Egy változót felüldefiniálhatunk a `declare` kulcsszóval:

```

x=10
f() {
    declare x
    x=10
    x=$((x+1))
    echo $x
}

```

```

f
echo $x
# Prints 11, then 10

```

Vigyáznunk kell azonban a változók létrehozásakor. Az alábbi esetben `y` nem fog létezni a függvényen kívül:

```
x=10
f() {
    declare y
    y=10
    y=$((y+1))
    echo $y
}

f
echo $x
echo $y
# y is unset by the end
```

Az alábbi esetben viszont már létezni fog:

```
x=10
f() {
    y=10
    y=$((y+1))
    echo $y
}

f
echo $x
echo $y
# y exists at the end, and it's 11
```

Trükkös, ugye? A következő kód jól szemlélteti, mi áll a háttérben:

```
x=10
f() {
    local y=11
    z=12
}

f
echo $x
echo $y
echo $z
# y exists at the end, and it's 11
```

Az `y` lokális, a `z` pedig nem, ezért `y` kívül nem látszik, `z` viszont igen. A `declare` kulcsszó a függvényen belül lokális változókat fog létrehozni. Az alábbi kódnál két `x` fog létezni, egy belső és egy külső (lokális és globális):

```
x=10
```

```
f() {
    local x=11
    echo $x
}
```

```
f
echo $x
# Prints 11 and 10
```

Most már láttuk, hogy a függvényeket hogyan lehet használni. Lássunk rájuk tényleges példákat!

Példák

1. példa

Készítsünk egy függvényt, ami összeadja az argumentumban megadott számokat. Ha nem kap argumentumot, akkor 0-t ír ki, egyébként az argumentumok összegét. Az argumentumokat nem szükséges ellenőrizni, hogy számok-e.

```
#!/bin/bash

sum() {
    local x=0
    for i in $*; do
        x=$((x + i))
    done
    echo ${x}
}
```

```
sum 10 20 30
sum
```

2. példa

Írjunk függvényt, ami legenerál N darab véletlen számot egy $[x-y]$ intervallumban. N , x és y értékeit paraméterekként kapja meg a függvény.

Ellenőrzések nem szükségesek a függvényen belül. Ha bármelyik paramétert nem adják meg, akkor a default értékek legyenek a következők: $N=5$, $x=1$, $y=90$.

Generáltassunk a függvénnyel 10 véletlen számot 800 és 900 között, majd 15 számot -10 és 10 között!

```
#!/bin/bash

generate() {
    local N=${1:-5}
```

```

local x=${2:-1}
local y=${3:-90}
range_size=$((y - x + 1))
declare -a arr
for ((i=0; i < N; ++i)); do
    rnd=$((RANDOM % range_size + x))
    arr[$i]=$rnd
done
echo ${arr[*]}
}

```

```

generate 10 800 900
generate 15 -10 10

```

Feladatok

Feladatok önálló megoldásra.

1. feladat

Egészítsük ki a második példát úgy, hogy a generált számok egyediek legyenek, és generáltassunk ötös lottó nyerőszámokat!

2. feladat

Készítsünk egy függvényt, ami egy vektort vár, és kiírja a vektor hosszát. A vektor dimenziója a bemeneti paraméterek száma. Ha nem kap bemeneti paramétert, akkor jelezzen hibát!

Írjunk egy scriptet, ami felhasználja ezt a függvényt! A felhasználótól kérjünk be egy vektort, és számítsuk ki az egységvektorát (az egy hosszúságú irányvektor, tehát a vektor összes komponense osztva a hosszával).

8. Kérdések

Ellenőrző kérdések a nyolcadik gyakorlathoz.

1. kérdés

Mit ír ki az alábbi kód?

```

function f {
    echo "Hello"
}
echo $f

```

a. Semmit.

- b. Hello
- c. Egy üres sort.
- d. f

2. kérdés

Mit ír ki az alábbi kód?

```
function f {  
    echo "Hello"  
    return 1  
}  
n=f  
echo $n
```

- a. Semmit.
- b. Hello
- c. 1
- d. f

3. kérdés

Mit ír ki az alábbi kód?

```
function f {  
    echo "Hello"  
    return 1  
}  
n=$(f)  
echo $n
```

- a. Semmit.
- b. Hello
- c. 1
- d. f

4. kérdés

Mit ír ki az alábbi kód?

```
function f {
    echo "Hello"
    return 1
}
echo $?
```

- a. Valószínűleg semmit.
- b. Hello
- c. 1
- d. f

5. kérdés

Mit ír ki az alábbi kód?

```
function f {
    echo -n "Hello "
}
f
echo $?
```

- a. Semmit.
- b. Hello
- c. Hello 1
- d. Hello 0

6. kérdés

Mit ír ki az alábbi kód?

```
x=10
function f {
    local x=1
    x=$((x + 1))
    if [ $# -lt 2 ]; then
        return 1
    fi
    echo=$((x + $1 + $2))
}
result1=$(f 12 10 || echo 0)
result2=$(f 10 || echo 0)
echo "$result1 : $result2 : $x"
```

- a. 24 : 0 : 10
- b. 23 : 0 : 10
- c. 23 : 0 : 12
- d. 23 : 12 : 10

9. óra

Ezen a gyakorlaton elengedjük a Bash nyelv használatával kapcsolatos technikai ismereteket. Ehelyett megtekintünk néhány használati esetet.

Alapvető megállapítás (triviális), hogy egy nyelv csak annyira lehet hasznos, amennyi szituációban hasznát tudjuk venni. A Bash nyelvet nem számítógépes játékok vagy komplex grafikus alkalmazások fejlesztésére találták ki. Legjobban rendszergazdai feladatokhoz, egyszerű parancssori automatizálásokhoz, installációs szkriptekhez működik. Mivel az ilyenek hosszúra nyúlhatnak, ezek helyett összegyűjtöttem pár olyan szituációt, amelyekben a Bash nyelv hasznomra vált az évek során. Ezek nagyrészt rövidek lesznek.

Az alábbiakban érdemi sorrend nélkül következnek a használati esetek.

Hasznos példák

Az `ssh` parancs

Talán a leghasznosabbal indítottam. `ssh` = Secure Shell. Gyakori, hogy a parancssort nem azon a gépen akarjuk használni, amin éppen vagyunk. Az `ssh` megengedi a távoli parancssori elérést.

```
ssh username@jerry.iit.uni-miskolc.hu
```

Az alábbi parancsot kiadhatjuk akár egy Linuxos, akár egy Windowsos gép termináljában, és segítségével rácsatlakozunk egy távoli gépre. A távoli gép jelenleg egy szerver, ami a `jerry.iit.uni-miskolc.hu` címen elérhető. A `@` elé beütjük a felhasználónevet, amivel be kívánunk jelentkezni. Ezután a parancs megkérdezi, hogy elfogadjuk-e az új `ssh` kapcsolat fingerprintjét (beírjuk, hogy *yes*). Ezt követően be kell írunk a jelszavunkat, és már kész is a távoli kapcsolat!

Innentől úgy kezelhetjük a terminált, mintha ott lennénk a távoli gépen. Ha le akarunk csatlakozni, csak adjuk ki a `logout` parancsot!

Hasznos ez a következő esetekben: - A szerver messze van, de dolgoznod kell rajta. Karbantartási, rendszergazda munka, devops feladatok. - A `jerry` szerverre kell csatlakoznod, mert valami beadandót kell rajta csinálnod. - A szerveren nincs grafikus felület (ez gyakori). Ilyenkor egyszerűbb `ssh` parancsot használni, mint levinni egy monitort és billentyűzetet a szerverterembe. - Kis készülékre, tipikusan Raspberry PI-re telepítettél Ubuntut vagy Raspbiant, és

a kis kutyü már benne van egy robotban. Csatlakoznod kell, de nem akarsz monitort és billentyűzetet vinni magaddal a robotra.

Az `scp` parancs

Emlékszünk a `cp` parancsra? Fájlok másolására szolgált. Ugyanaz az `scp` is, csak *Secure Copy*. Mint az `ssh` parancs, ez is gépek közötti másolásra szolgál. Ha `ssh`-val csatlakozol a távoli gépre, akkor sok dolgot megtehetsz *azon* a gépen, de ha a saját gépedről akarsz odamásolni valamit, akkor valami egyéb parancsra van szükséged.

Az alábbi példa egy fájlt másol a gépemről a `jerry-s` szerverre, a saját felhasználói fiókomba.

```
scp some_file.txt bolyki@jerry.iit.uni-miskolc.hu:~/foldername/
```

Ugyanúgy jelszót kell majd beírni, mint `ssh` esetén, de a jelszó beírása után a `some_file.txt` nevű fájl *erről a gépről* átmásolódik *arra a gépre*. A másik gép címe után kettőspontot rakunk, és a kettőspont után írjuk a fájl új elérési útját. A fenti esetben a `home` mappámban lévő `foldername` nevű mappába fog kerülni a fájl.

Megjegyzem, hogy az `rsync` parancs is ilyesmire szolgál, de soha nem tanultam meg a működését. Az `scp` egyszerű, egyértelmű, és mindenre elég volt, amire nekem kellett.

`sudo apt install`

Amint saját rendszert telepítesz, szemben fogod magad találni ezzel a paranccsal. Ez Ubuntu-n így néz ki, bár Arch Linuxon máshogy fog. Az `apt` az Ubuntu csomagkezelője. A hivatalos repository-ból tudsz csomagokat letölteni és installálni vele. Szinte minden installációs tutorialban találkozni fogsz vele, úgyhogy nem ragozom tovább. Jobb, mint a `next-next-finish`.

A `tar`, `zip` és `unzip` parancsok

A JupyterLab szerveren jártam többször úgy, hogy nem tudtam mappát letölteni a felhasználói felületről. A JupyterLabbal akkor fogtok találkozni, ha neurális hálókat tanítotok fel egy távoli gépen (egy erős szerveren, masszív videokártyákkal). A JupyterLabot el lehet érni böngészőben, és lehet Python kódokat futtatni benne. Ennek a dokumentumnak az írásakor viszont nem lehetett mappát letölteni róla. Sem feltölteni.

Szerencsére akad terminál is a szerveren, így lehet Bashben utasításokat adni. A megoldás az volt, hogy a mappából csinálok egy `zip` fájlt, és azt töltöm le, vagy töltöm fel. Kitömöríteni sem lehet kattintgatással, csak terminálból.

Gondolom mostanra mindenki kitalálta, mire jó a `tar` és az `unzip` parancs.

A `tar` parancs neve a *Tape ARchive* szavakból jön. Eredetileg szalagra írt archívumokhoz találták ki. Lehet vele tömöríteni és kitömöríteni fájlokat, mappákat. Még saját fájl kiterjesztése is van (*tar*). A tömörített archívumokat kétszeres kiterjesztéssel szokták használni, `archive_name.tar.gz` szintaxissal. Egy mappát az alábbi módon tudunk tömöríteni:

```
tar cfzv archive.tar.gz foldername
```

A létrejött archívum neve `archive.tar.gz` lesz. Kitömöríteni az alábbi módon tudjuk:

```
tar xfv archive.tar.gz
```

A `tar` parancs irodalma ennél jóval bővebb, de én 95%-ban csak ennyit használtam belőle. Az `xfzv` 4 darab kapcsoló összevonva, a `cfzv` szintűgy. Érdekeség, hogy melyik mit jelent: - `x`= eXtract (csomagold ki). - `c`= Compress (csomagold be). - `f`= File (a következő fájlnevével hozd létre az archívumot). - `z`= gZip (gzip tömörítésű fájlra fog dolgozni). - `v`= Verbose (“bőbeszédű”, kiírja milyen fájlokon dolgozik, miközben csomagol / kicsomagol).

A `zip` és `unzip` parancs ennek az egyszerűbb verziója, ezért szeretem jobban. Ha csak egy *zip* fájlt akarsz becsomagolni és kicsomagolni, akkor könnyebbet nem is kereshetnél.

```
# Becsomagolja a my_folder mappát egy my_archive.zip fájlba
zip -r my_archive.zip my_folder # -r is for recursive
# Kicsomagolja a my_archive.zip mappát
unzip my_archive.zip
```

A `tar` parancs egy svájci bicska az archívumok készítésére, tömörítésre, kitömörítésre. A `zip` és `unzip` csak annyi, amennyit a nevük mond. Nekik is vannak kapcsolóik, amikről érdemes lehet olvasni, ha használod őket. A fenti példa azonban lefedi a használati esetek 95%-át (nálam).

Az `sl` parancs

Az előző parancsok hosszúra sikerültek, ezért ez egy fellélegzős parancs lesz. Programozó vagy és szereted megviccelni magad? Az `sl` parancsot neked találták ki!

Mi az egyik leggyakrabban beütött parancssori utasítás? Igen, az `ls`.

Mi az egyik leggyakoribb elgépelési forma? Igen, a karakterek felcserélése. Tehát `sl`.

Nem is lennél igazi mérnök, ha időről időre nem csinálnál hülyét magadból. Az `sl` parancs karaktergrafikusan meganimál egy elhaladó vonatot neked, valahányszor elgépeled az `ls` parancsot. Lehet installálnod kell, de megéri (a tanszéki gépekre az illetékes már volt kedves felrakni).

```
sl
```

A cowsay parancs

Az `sl` parancs egyik ismert párja. Karaktergrafikusan kirajzol egy tehenet, ami mond neked valamit. Miután 3 órája szívsz egy beadandóval, bizonyára valami kedveset fog mondani.

```
cowsay "You suck."
```

A history parancs

Most térjünk vissza pár olyan parancsra, amik tényleg hasznosak. A `history` parancs már mentette meg párszor az életemet. Remélem a félév végére már rájöttél, hogy a terminálban, ha fölfelé nyilat nyomsz, akkor az előző parancsokon tudsz végiggörgetni. Mi van, ha hirtelen szükséged lesz arra a parancsra, amit 200 parancssal ezelőtt ütöttél be? Nagyon hosszú parancs volt, nem emlékszel a paraméterlistájára, de tudod, hogy például `gcc`-vel kezdődött.

A `history` parancs siet a segítségedre. Kíírja neked az adott terminálhoz tartozó korábban kiadott parancsok listáját. Kombináld a `grep` parancssal, hogy még kevesebbet kelljen szívnod vele!

```
history | grep "gcc"
```

A kill és a ps parancs

Nem vagy programozó, ha még nem rontottál el semmit. Időnként annyira el lehet rontani, hogy le se tudod lőni a `sz*r` programot, amit indítottál. A terminálos programokat általában meg lehet ölni `Ctrl+C`-vel. Létezik viszont olyan eset, amikor egyenesen meg kell ölnöd egy programot a PID-je (Process Id) alapján.

Először meg kell találnod a parancs PID-jét. Többek között erre jó a `ps`. Add ki a következő kapcsolókkal, hogy mindent is láss!

```
ps awux
```

Üdv a Linuxos parancskezelőben! A gépen futó összes programot láthatod. Több ezer van. Használd a `grep`-et, hogy rátalálj a saját programodra.

```
ps awux | grep "my_crazy_program"
```

Bal szélén az a felhasználó van, akinek a jogaival fut a program. A saját felhasználónevedet és a `root` felhasználót biztosan látni fogod. A következő oszlopban a PID van. A PID alapján le tudsz állítani egy programot erőszakkal.

```
kill -9 10234
```

A fönti parancs leállítja a 10234-es PID-del rendelkező futó programot (processzt). OS-ből erről többet fogtok tanulni, de megjegyeznék itt valamit:

Megjegyzés: A `kill` parancs neve becsapós. Nem programok leállítására szolgál, hanem arra, hogy jelet küldjünk nekik. A 9-es jel a `SIGKILL`, tehát az

azonnali leállításra vonatkozó jel. Ha másféle jelet küldünk, akkor a program másként reagál. Például ha lenyomjuk a **Ctrl+C** billentyűket, akkor is jelet küldünk, csak egy másikat, a **SIGINT** parancsot (Interrupt). Általában ez is megöli a programot, de **SIGKILL** egészen biztosan.

Még egy megjegyzés: Gondolom mondanom sem kell, hogy ha elkezdünk ész nélkül programokat öldödni a rendszerünkben, akkor az valószínűleg hibákhoz fog vezetni.

A gcc parancs

Van olyan eset, főleg Linux alatt, főleg az OS tantárgy gyakorlatán, hogy terminálban kell C programokat fordítani és futtatni. A C fordító a legtöbb Linux rendszeren a **gcc** lesz.

Ha van egy egyfájlos C kódod, amit le akarsz fordítani és futtatni, akkor megteheted a következőt:

```
gcc my_prog.c -o my_prog # compile
./my_prog # run
```

Nyilván nagy programoknál nem javaslom ezt a módszert, de kicsiknél elég ennyi is. Ha külső könyvtárat használsz a programodban, akkor azt esetleg linkelned kell:

```
gcc my_prog.c -o my_prog -lSDL # compile
./my_prog # run
```

Az **SDL** (vagyis inkább **SDL2**) egy videojáték készítéshez jól használható könyvtár (alacsony szintű, ezért programozni vágyóknak ajánlom).

A cloc parancs

Benne vagy egy nagy saját projektben vagy beadandóban? Írod már egy hete, és kíváncsi lettél, hogy mégis mennyit írtál eddig? Kíváncsi lettél, hány soros a kódod, mennyivel dicsekedhetsz el?

A **cloc** parancs (Count Lines Of Code) Linux alatt egy könnyen installálható kis utility.

```
cloc src
```

A fenti parancs megszámolja az **src** mappában lévő kódsorok számát, nyelv szerint.

Egyebek

A legtöbb programnak, programnyelvnek, keretrendszernek, amit használni fogtok az egyetem és a munka során, van valamilyen parancssori interfésze (CLI - Command Line Interfész). Tipikus példa a **git**. Én szeretem a parancssori **git**-et, más kevésbé. De itt nem áll meg a sor. Csak néhány parancssori felület, amit

használnom kellett: - `gcc` a C nyelvhez. - `g++` a C++ nyelvhez. - `java` és `javac` a Java nyelvhez. - `node`, `npm`, `ng` a webfejlesztéshez (Node szerver, Node csomagmenedzser, Angular CLI). - `docker` a virtuális gépek menedzseléséhez. Itt is előjött az, hogy meg kellett találni a virtuális gépek azonosítóit, majd le kellett lőni őket, esetleg törölni és újragenerálni őket. Sokat segített a `grep` parancs. - `cmake` a komplexebb C, C++ projektek buildeléséhez és installálásához.

Ezekkel vagy fogtok találkozni, vagy nem, de jó tudnotok, hogy a command line nem ér véget Számítógépi Architektúrák tantárggyal.

Parancsok, amik segíthetnek a beadandóban

A `w` és a `who` parancs

Arra valók, hogy adatokat tudjunk meg az épp bejelentkezett felhasználókról, legutóbbi belépésükről, stb. Vannak olyan beadandó feladatok, amik ezekre építenek.

A `read` parancs kapcsolói

Akadnak játékok is a beadandók között. Bár alapesetben nem javasolnám, hogy bárki Bashben készítsen játékot, de a beadandó talán izgalmasabb és élvezetesebb, ha nem egy APEH szimulátort írtok, hanem egy Pongot. Egy valós idejű játéknál (ami nem körökre osztott, mint az amőba), jól jöhet a `read` parancs `-t` kapcsolója. Ez egy *timeout*-ot ad a parancshoz, vagyis ha nem ütsz le semmit, akkor program folytatódik. Mint egy játékban.

Ezenfelül jól jöhet az is, hogy a `read` parancsnak megadható, hogy csak egy karaktert várjon. Egy billentyűlenyomást pont elég egyszerre kezelni. Ha olyan játékot akarsz írni, ahol egyszerre több billentyű kell, akkor tényleg keres másik programnyelvet.

A `read` parancsot el is lehet hallgattatni az `-s` kapcsolóval (így ha leütsz valamit, akkor az nem fog megjelenni a terminálban).

Egy példa a használatára:

```
playing=1
while [[ $playing -eq 1 ]]; do
    read -n1 -t 0.1 -s user_keypress
    if [[ $user_keypress == "w" ]]; then
        echo UP
    elif [[ $user_keypress == "s" ]]; then
        echo DOWN
    elif [[ $user_keypress == "a" ]]; then
        echo LEFT
    elif [[ $user_keypress == "d" ]]; then
        echo RIGHT
    elif [[ $user_keypress == "x" ]]; then
```

```

        playing=0 # exit
    fi
    echo "Computer takes action."
done
echo "Game over"

```

A fenti példában a `read` parancs egy karaktert vár, és 0.1 másodperc után továbblép, ha addig nem érkezik felhasználói input. Az `-s` kapcsoló miatt a felhasználói input nem kerül ki a terminálra. Erre az alapra ráírhatod a Pongot, vagy bármi egyéb egyszerű, karaktergrafikus játékot.

Válaszok

1. Kérdéssor

1. Abszolút.
2. Relatív.
3. Relatív.
4. Abszolút.
5. Kettő.
6. Magára a mappára.
7. A szülő mappára.
8. Igen. A root mappa.
9. A jelenleg bejelentkezett felhasználó home mappáját.
10. Hamis. A `-p` kapcsolóval egyszerre több mappa is létrejön, és egyébként is felsorolásszerűen meg lehet neki adni, hogy milyen mappákat szeretnének létrehozni.
11. Hamis. Az `rm` parancs mappát is töröl, ha beadjuk neki az `-r`, tehát rekurzív kapcsolót.
12. Hamis. Nem feltétlen ugyanazon a néven másolunk; meg lehet adni más nevet is neki a célpozícióban.
13. d. Arra kell gondolni, mikor csak átnevezünk, de nem helyezünk át. Ekkor valójában elérési utat módosítunk, ami magában foglalja az áthelyezés és az átnevezés műveletét (más path fog a fájlra utalni).
14. c.
15. Hamis.
16. `rmdir` vagy `rm`, de utóbbi esetén kell a rekurzív kapcsoló (`rm -r mappanev`)

17. a.

2. Kérdéssor

1. robert
2. ruby
3. hamis
4. hamis (a ruby csoportba tartozók szerkeszthetik)
5. igaz (csak módosítani nem tudja akárki)
6. csak a fájl tulajdonosa számára
7. 666
8. 640
9. 655
10. 777
11. 700
12. 744
13. Mindenki.
14. Igen.
15. Igen.
16. Nem.
17. Mindenki.
18. Senki.
19. A tulajdonos felhasználó és a tulajdonos csoport tagjai.
20. A tulajdonos felhasználó.
21. Senki.

3. kérdéssor

1. semmit
2. OK
3. Semmit (csak a hibát)
4. Making directory
OK
5. OK
6. Making directory
Problem
7. Igen
8. Igen
9. Nem
10. Nem
11. Igen
12. 1
13. 1
14. 6

4. kérdéssor

1. c
2. a
3. semmit, hiba (-E kapcsoló lemaradt)
4. semmit, hiba ([a-z] után szükséges egy számosság megjelölő).
5. Aragorn,dog,5
Aragog,cat,2
6. Raptor,cat,10
Ducatti,parrot,2
Garfield,cat,4
Aragog,cat,2
7. Raptor,cat,10
Garfield,cat,4
Aragog,cat,2
8. Aragog
Garfield
Raptor
9. Species
10. Thor,dog,12
11. Nincs válasz, ez egy bónusz.

5. kérdéssor

1. c
2. a
3. x: \$x
4. Szintaxis hiba, ugyanis x számnak lett deklarálna.
5. x: 0 (furcsa viselkedés, mert nem sikerült számként értelmezni a Ten-t)
6. x: (egy space az x értéke)
7. Hiba az egyszeres zárójel miatt.
8. -5
9. b
10. 31.9

6. kérdéssor

1. a
2. d
3. b
4. d
5. c

7. kérdéssor

1. a

2. c
3. b

8. kérdéssor

1. c
2. d
3. b
4. a
5. d
6. a