Tíz fontos parancs, amit minden linuxos fejlesztőnek ismernie kell

Ha más kódját kell karbantartanunk, néhány egyszerű segédprogrammal komolyan megkönnyíthetjük az életünket.

Ebben a cikkben néhány gyakorlatilag minden Linux-telepítésben megtalálható parancsot szeretnék ismertetni. Segítségükkel javíthatjuk kódjaink minőségét, divatos kifejezéssel élve termelékenyebbekké válhatunk. A lista összeállítását saját programozói tapasztalataim alapján végeztem, és olyan eszközökből áll, amelyeket magam is rendszeresen használok. Némelyik magának a kódnak az elkészítésében nyújt segítséget, mások hibakeresésre használhatók, megint mások pedig az ölünkbe pottyant idegen kód visszafejtését teszik lehetővé.

1. ctags

Aki szereti az integrált fejlesztői környezeteket (IDE), talán sosem hallott erről az eszközről, vagy ha hallott is, idejétmúltnak vélheti. Pedig egy a címkéket ismerő szerkesztő mindig is értékes eszköz marad.
Ha címkékkel látjuk el kódunkat, akkor például `vi` vagy ***Emacs*** alatt hiperszövegként tudjuk kezelni. (*1. ábra*) A kód minden objektuma egy a saját meghatározásához vezető hiperhivatkozást kap. Ha például `vi` alatt turkálunk a kódban, és szeretnénk tudni, hogy a „`pelda`” változó hol van megadva, beírjuk a `:ta foo` parancsot. Ha a kurzor éppen a változón pihen, elég a Ctrl+] kombinációt használnunk. A `vi`-t kevésbé kedvelők számára jó hír, hogy a `ctags` már nem csupán C kóddal és a `vi` szerkesztővel használható. A `ctags` GNU változata olyan címkéket állít elő, amelyek ***Emacs***, illetve sok egyéb, a címkefájlok felismerésére képes szerkesztő alatt is kezelhetők. A `ctags` a ***C***-n és a ***C++***-on kívül számos más nyelvet is ismer, olyat mint a ***Perl*** és a ***Python***, sőt, még vastervező nyelvekkel is boldogul, mint például a ***Verilog***. Segítségével emberi szem számára is könnyen átlátható utalásokat lehet készíteni, amelyek alapján könnyebb a kód megértése, áttekintése. Lehet, hogy a `ctags` használata szerkesztés közben még nem nagyon érdekel valakit, az utalásokat azonban mégis érdemes átnéznie. Ilyenkor a `ctags -x *.c*` parancsot kell kiadnia.
Én azért is szeretem ezt a kis programot, mert mindegy, hogy egy vagy száz fájlt vizsgáltatunk meg vele, mindenképpen hasznos adatokhoz jutunk, ellentétben sok IDE-vel, amelyek csak akkor érnek valamit, ha a teljes alkalmazást látják. Mivel a programok ellenőrzésére nem alkalmas, ha hibás bemenetet adunk neki, értelemszerűen a kimenete is hibás lesz.

2. strace

Az `strace` segítségével hibakereső és forráskód hiányában is kifürkészhetjük, pontosan mi történik egy program futása közben. Nekem például az abszolút kedvencem az olyan program, ami el sem indul, de nem mondja meg, hogy miért. Lehet, hogy valamilyen fájlt hiányol, de az is előfordulhat, hogy csak a jogosultságokkal van baja. Az `strace` elárulja nekünk, hogy mit csinál a program a háttérben, vagyis milyen rendszerhívásokat indít és ezek milyen eredménnyel járnak, egészen addig a pontig, míg a futása véget nem ér. A `fork` hívások követésére is képes.
Én úgy tapasztaltam, hogy az `strace` sokszor gyorsabban megadja az engem érdeklő válaszokat, mint bármelyik hibakereső; főleg, ha ismeretlen kóddal kell dolgoznom. Néha az is előfordul, hogy éles, hibakereső nélküli rendszeren kell valamilyen kód hibáit megkeresnem. Ilyenkor az `strace`-t előkapva elkerülhetem a rendszer foltozását vagy a kód `printf` hívásokkal való teleszórását. Íme egy egyszerű példa. Mi történik, ha egy normál felhasználó megpróbál törölni egy védett fájlt:

```
strace -o strace.out rm -f /etc/yp.conf
A kimenetből megtudjuk, mi volt a baj:
lstat64("/etc/yp.conf", {st_mode=S_IFREG|0644,
➥st_size=361, ...}) = 0
access("/etc/yp.conf", W_OK) = -1 EACCES
(Hozzáférés megtagadva)
unlink("/etc/yp.conf") = -1 EACCES
➥(Hozzáférés megtagadva)
```

Ha menet közben akarunk hibákat felderíteni a `strace` segítségével, futó folyamatokhoz is csatlakozhatunk vele. Tegyük fel például, egy folyamat egy csomó időt tölt valamivel. Az `strace -c -p folyamatazonosító` paranccsal pillanatok alatt megtudhatjuk, mi folyik odabent. Néhány másodperc után nyomjunk Ctrl-C-t, és az alábbihoz hasonló kiírást fogunk látni:

```
% time  seconds  usecs/call calls errors syscall
------ -------- ---------- ----- ------ --------------
 91.31  0.480456       3457   139         poll
  6.66  0.035025        361    97         write
  0.91  0.004794         16   304         futex
  0.52  0.002741         14   203         read
  0.31  0.001652          3   533         gettimeofday
  0.26  0.001361          4   374         ioctl
  0.01  0.000075          8    10         brk
  0.01  0.000064         64     1         clone
  0.00  0.000026         26     1         stat64
  0.00  0.000007          7     1         uname
  0.00  0.000005          5     1         sched_get_priority_max
  0.00  0.000002          2     1         sched_get_priority_min
------ -------- ---------- ----- ------ --------------
100.00  0.526208            1665         total
```

Ebben az esetben a várakozást a `poll` rendszerhívás idézte elő, valószínűleg itt egy foglalatra várt a folyamat.

3. fuser

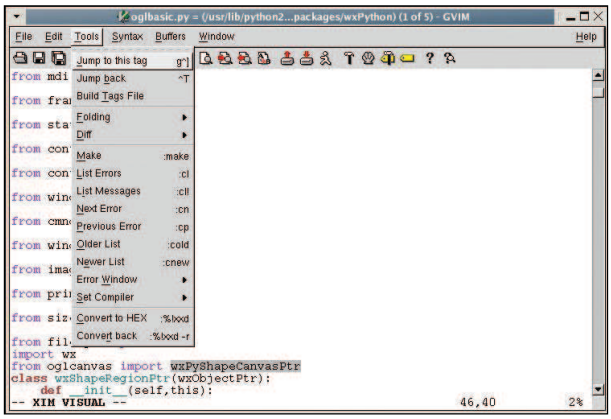
A név a „file user” (fájl használója) kifejezésből állt elő, vagyis a programmal azt tudhatjuk meg, hogy adott fájlt mely folyamatok nyitottak meg. Arra is alkalmas, hogy az összes ilyen folyamatnak jelzést küldjünk. Tegyük fel például, hogy törölni szeretnénk egy fájlt, de nem tudjuk megtenni, mert valamelyik program megnyitotta, és esze ágában sincs lezárni. Ilyenkor megúszhatjuk a gép újraindítását, ha kiadjuk a `fuser -k` fájl parancsot, ez `SIGTERM` jelzést küld az összes olyan folyamatnak, amely megnyitotta a fájlt.
Előfordulhat, hogy a `kill` paranccsal kell megölnünk egy `fork` hívásokkal – bármilyen okból – jónéhány példányra osztódott folyamatot. Egy kezdő programozó ilyenkor valószínűleg egy a célnak megfelelő `ps | grep` parancsot adna ki, majd szorgalmas másol-beilleszt játékba kezdene az egérrel. Sokkal egyszerűbb azonban, ha kiadjuk a `fuser -k ./program` parancsot, ahol a program a futtatható fájl elérési útja. A `fuser` általában a felügyeleti eszközök számára fenntartott `/sbin` könyvtárban található. Ha gondoljuk, a `/usr/sbin` és a `/sbin` könyvtárakat adjuk hozzá a `$PATH` környezeti változóhoz.

5. ps

A `ps`-t általában a folyamatok állapotának megvizsgálására használják, és sokan nem is tudják, hogy hibakereső eszközként is kiválóan megállja a helyét. Mindezeket a lehetőségeket a `-o` kapcsolóval érhetjük el, amely számos adatot bocsát rendelkezésünkre a folyamatokról, ideértve a processzorhasználatot, a képzetes memória iránti igényt, az aktuális állapotot stb. Mindezen mutatók jelentős része a ***POSIX*** szabványban is szerepel, vagyis többféle rendszeren is elérhető.
Ha a futó programokat, folyamatazonosítót és folyamatállapotot szeretnénk kilistázni, adjuk ki a
`ps -e -o pid,state,cmd`

parancsot. A kimenet így fog kinézni:

```
4576 S opt/OpenOffice.org1.1.0/program/
➥soffice.bin -writer
```



1. kép Címkékkel ellátott fájl szerkesztése gvimmel

```
4618 D dd if /dev/cdrom of /dev/null
4619 S bash
4645 R ps -e -o pid,state,cmd
```

A kimenetben láthatjuk, hogy a `dd` parancs nálam megszakíthatatlan alvó (`D`) állapotban volt. Lényegében arról van szó, hogy amíg a program vár a */dev/cdrom*-ra, addig blokkolt állapotba kerül. Az ***OpenOffice.org writer*** éppen alvó (`S`) állapotban van, a `ps` pedig futóban (`R`).
Ha tudni akarjuk, hogy egy futó program hogyan teljesít, a következő parancsot kell kiadnunk:

```
ps -o start,time,etime -p folyamatazonosító
```

Ekkor a `time` parancs – később még lesz róla szó – alapszintű kimenete jelenik meg, azzal a különbséggel, hogy nem kell megvárnunk a program futásának befejeződését.
A `ps` által átadott adatok túlnyomó része a */proc* fájlrendszeren keresztül is elérhető, de ha parancsfájlt írunk, a `ps` használatával jobban hordozható kódot kapunk. Soha nem tudhatjuk, hogy a rendszermag egy komolyabb módosítása miatt mikor válik működésképtelenné a */proc* fájlrendszerben kutakodó parancsfájlunk. Maradjunk inkább a `ps`-nél.

5. time

A `time` parancs segítségével leginkább kódunk teljesítményét tudjuk vizsgálni. Alapszintű kimenetében a tényleges, a felhasználói és a rendszeridő szerepel. A tényleges idő az az időmennyiség, amely a kód elindulásának és kilépésének időpontja között telt el. A felhasználói és a rendszeridő rendre a felhasználó által a kód és a rendszermag futtatására fordított idő.
A `time` parancs kétféle változatban létezik. A héj tartalmaz egy beépített változatot, amely csak ütemezési adatokat szolgáltat. A */usr/bin* könyvtárban található változat több adatot ad, és a kimenet formázását is lehetővé teszi. Ha a beépített helyett ez utóbbit akarjuk használni, akkor a parancs kiadásakor egy visszafelé hajló perjelet kell annak elejére írnunk, ahogy az a példákban is látható.
A Linux ütemezőjének alapszintű ismerete jelentősen megkönnyíti a kimenet értelmezését, ám ha éppen az ütemező működését akarjuk tanulmányozni, akkor ezzel az eszközzel ezt is megtehetjük. A valós idő például jellemzően na-

gyobb, mint a felhasználói és a rendszeridő összege. Ennek oka az, hogy a rendszerhívásokban blokkolva eltelt idő nem számít bele a folyamat futási idejébe, ilyenkor ugyanis az ütemezőnek megvan a lehetősége arra, hogy más folyamatoknak adja az erőforrásokat. Az alábbi példában a `sleep` parancs futása egy másodpercig tart, de mérhető rendszer- vagy felhasználói időt nem vesz igénybe:

```
\time -p sleep 1
real 1.03
user 0.00
sys 0.00
```

A következő példával azt szemléltetem, hogy egy folyamat hogyan töltheti teljes futási idejét a felhasználói térben. Ebben az esetben a *Perl* a `log()` függvényt hívja meg hurkolva, ennek futásához a rendszermagra gyakorlatilag nincs szükség:

```
\time perl -e ´log(2.0) foreach(0..0x100000)´
real 0.40
user 0.20
sys 0.00
```

További példaként lássunk egy nagy mennyiségű memóriát lekötő folyamatot:

```
\time perl -e ´$x = ´a´ x 0x1000000´

0.06 user 0.12 system 0:00.22 elapsed 81% CPU
(0 avgtext + 0 avgdata 0 maxresident) k
0 inputs + 0 outputs (309 major+8235 minor)
pagefaults
0 swaps
```

Ebben az esetben a `pagefaults` érték, vagyis a laphibák száma érdekel bennünket. Noha a GNU `time` parancs rengeteg adatot tesz elérhetővé, a 2.4-es sorozatú *Linux* rendszermagok csak a fő- és az allaphibákat követik figyelemmel. Főlaphiba az, amelynél szükség van be/kivitelre, allaphiba pedig az, amelyiknél nincs.

7. nm

Segítségével a megadott futtatható vagy objektumfájlon belüli szimbólumnevekről gyűjthetünk adatokat. Alapesetben a kimenetben egy szimbólumnév és annak képzetes címe jelenik meg. Hogy mindez mire jó? Tegyük fel, hogy éppen kódot fordítunk, és a fordító hibát jelez, miszerint a `_pelda` szimbólum feloldatlan. Átfésüljük a kódunkat, de nem jövünk rá, hogy hol használjuk ezt a szimbólumot. Lehet, hogy valamelyik sablonból származik, esetleg a kóddal együtt lefordított beemelt fájlok sokaságának egyikében szereplő makróról van szó. A kiadandó parancs:

```
nm -guA *.o | grep pelda
```

Ezzel minden a `pelda` szimbólumra hivatkozó modult megkapunk. Ha azt akarjuk tudni, melyik könyvtárban található a `pelda` megadása, a következő parancsot adjuk ki:

```
nm -gA /usr/lib/* | grep pelda
```

Az `nm` parancs a *C++* nevek kibontására is alkalmas, ami főleg *C* és *C++* kód keverésekor jöhet jól. Például, ha egy *C* függvényt `extern` „C” nélkül adunk meg, akkor a következőhöz hasonló fordítási hibát kapunk:

```
undefined reference to `cfunc(char*)´
```

Egy nagyobb méretű, szegényesen megadott fejlécekkel ellátott tervezetnél nem kis feladat a hibás modult megkeresni. Ha az összes feloldatlan szimbólumra rá szeretnénk keresni az objektumfájlokban (a nevek kibontásával), a következő parancsot állítsuk össze:

```
nm -guC *.o
extern-c.o:cfunc
no-extern-c.o:cfunc(char*)
```

Az első modul rendben van, a második viszont nem.

7. strings

A `strings` segítségével bináris fájlokba ágyazott ASCII karakterláncokat tudunk keresni. Tulajdonképpen jóra és rosszra egyaránt alkalmas. A jó oldal: kinyomozhatjuk, hogy az szabványos kimenetre kerülő rejtélyes szöveg melyik könyvtárból származik:

```
strings -f /usr/lib/lib* | grep “rejtélyes üzenet”
```

A rossz oldal: a karakterláncok alapján ki lehet deríteni, hogy milyen formátumot meghatározó karakterláncokat használunk, amivel akár sebezhető pontokat is lehet találni programunkban. Ez az oka annak, hogy a programokba soha nem szabad felhasználói neveket és jelszavakat beépíteni. Ugyanezért nem rossz ötlet saját programunkat is megvizsgálni vele, így legalább tudjuk, hogy egy rafináltabb programozó mennyire lát bele a kártyáinkba. A GNU ***binutils*** részeként elérhető `strings`-változat számtalan hasznos lehetőséget biztosít.

8. od, xxd

Ez a két parancs lényegében ugyanazt csinálja, ám némileg eltérő szolgáltatásokat nyújtanak. Az `od` bináris fájlokat tud tetszőleges formátumra alakítani. Ha olyan programokkal dolgozunk, amelyek nyers bináris fájlokat állítanak elő, az `od` rendkívül nagy segítséget jelenthet. Bár neve az ***octal dump***, az oktális kiíratás kifejezésből származik, az adatokat decimális és hexadecimális formátumba is képes kiírni. Az `od` egészeket, IEEE lebegőpontos értékeket és egyszerű bájtokat tud kezelni. Több bájtos egész vagy lebegőpontos értékeknél a futtató gép bájtsorrendjétől függ a kimenet.

Az `xxd` szintén bináris fájlokat írat ki, de nem próbálja egészekként vagy lebegőpontos értékekként értelmezni azokat. Így a futtató gép bájtsorrendje nem befolyásolja a kimenetet, ami a fájl jellegétől függően zavaró és előnyös is lehet. Példaként hozzunk létre egy négybájtos fájlt egy Intel alapú gépen:

```
$ echo -n abcd > pelda.bin
$ od -tx4 pelda.bin
0000000 64636261
0000004
```

```
$ xxd -g4 pelda.bin
0000000: 61626364                  abcd
```

Az od kimenete egy bájtfelcserélt 32 bites egész, az xxd-é viszont egy négy bájtból álló csoport, amelynek bájtsorrendje a fájléval megegyező. Ha az abcd karakterláncot akarjuk megkeresni, az xxd-t kell használnunk, ha viszont a 0x64636261 32 bites számot szeretnénk fellelni, akkor az od-ot vegyük elő.
Az xxd a kimenetet binárisan is képes formázni, valamint bináris fájlt képes C tömbbé alakítani. Tegyük fel, van egy bináris fájlunk, amit egy C program egy tömbjébe szeretnénk elhelyezni. Ezt egyetlen módon tehetjük meg: az alábbi módon létrehozunk egy szöveges fájlt.

```
$ xxd -i pelda.bin

unsigned char pelda_bin[] = {
  0x61, 0x62, 0x63, 0x64
};

unsigned int pelda_bin_hossz = 4;
```

Ha olyan programokkal dolgozunk, amelyek nyers bináris fájlokat állítanak elő, az od rendkívül nagy segítséget jelenthet.

9. file

UNIX és ***Linux*** alatt a fájlnevek kiterjesztésére vonatkozóan soha nem volt kötelező előírás. A névadási szokások ugyan fejlődtek, de csak irányelvekről és nem kötelezően betartandó előírásokról van szó. Ha egy digitális képet ***kep00.exe*** névvel akarunk ellátni, megtehetjük. Linuxos fényképkezelő alkalmazásunk gond nélkül fogadni fogja a fájlt, függetlenül annak nevétől, amit nekünk viszont nehezebb lesz megjegyeznünk.
A file parancs például akkor lehet segítségünkre, ha egy összeomlott webböngészőből kell kibányásznunk egy összezavart nevű fájlt. Tegyük fel például, hogy a fájl nevének ***pelda.proba.hello.vilag.tar.gz*** kellene lennie, a valóságban viszont ***pelda.proba***. A kiadandó file parancs a következőképpen alakul:

```
$ file pelda.proba

pelda.proba: gzip compressed data, was
“pelda.proba.hello.vilag.tar”, from Unix
```

Előfordulhat, hogy kapunk egy terjesztést, aminek a ***bin*** könyvtárában tucatnyi fájl található, futtatható fájlok és parancsfájlok vegyesen. Tegyük fel, hogy ki szeretnénk válogatni a héjprogramokat. Próbálkozzunk ezzel:

```
$ file /usr/sbin/*  | grep script

/usr/sbin/ezmegmi:  a /bin/bash script text
                    executable

/usr/sbin/xconv.pl:     a /usr/bin/perl script
                        text executable
```

A file parancs a ***bin*** könyvtár összes fájlját megvizsgálja, a grep pedig kiszűri azokat, amelyek nem parancsfájlok. Néhány további példa:

```
file core.4867

core.4867: ELF 32-bit LSB core file Intel 80386,
➥version 1 (SYSV), SVR4-style, from ´abort´

file /boot/initrd-2.4.20-6.img

/boot/initrd-2.4.20-6.img: gzip compressed data,
➥from Unix, max compression

file -z /boot/initrd-2.4.20-6.img

/boot/initrd-2.4.20-6.img: Linux rev 1.0 ext2
➥filesystem data (gzip compressed data, from
➥Unix, max compression)
```

Ne feledjük, ahogy a könyvek tartalmát sem ítélhetjük meg borítójuk szerint, úgy a fájlok tartalmát sem mérhetjük fel csupán nevük alapján.

10. objdump

Ez egy különlegesebb eszköz, nem kifejezetten kezdők számára. Egyfajta objektumfájlokhoz készült adatbányász programnak tekinthető.
Az objektumkód rengeteg értékes adatot tartalmaz, az objdump futtatásával ezeket tudjuk elérni. Segítségével például kiírathatjuk a forrássorokba kevert assembly kódokat; ez az, amit a gcc -S, ki tudja miért, de nem tesz meg. Működéséhez az objektumkódot a debug (-g) kapcsolóval kell lefordítani:
objdump –demangle –source objektum.o

Az objdump-pal elhalálozás utáni hibaelemzés céljából mag (core) fájlból is kinyerhetünk bináris adatokat, ha hibakereső program éppen nem áll rendelkezésünkre. Teljes példát most helyszűke miatt nem ismertetnék, de annyit elmondanék, hogy az nm vagy az obdump segítségével először meg kell határoznunk a képzetes címet, ezután az objdump -x paranccsal az összes képzetes címhez tartozó fájleltolást ki tudjuk gyűjteni. Az utolsó lépés az objdump ismételt futtatása, ez ugyanis nem ***ELF*** fájlformátumokból is képes olvasni, ellentétben a ***gdb***-vel és az egyéb eszközökkel.
Írásom nem annyira útmutatóként, inkább kiindulópontként szolgálhat termelékenységünk növeléséhez. Az említett parancsok mindegyikéhez kimerítő leírás tartozik a linuxos ***man*** és ***info*** oldalak között. További tájékoztatást és ötleteket ezekben érdemes keresni.

Linux Journal 2004. szeptember, 125. szám

John Fusco programfejlesztő a General Electric Healthcare-nél (korábban GE Medical Systems), ahol linuxos programokat és illesztőprogramokat tervez a GE Lightspeed sorozatú számítógépes tomográfjaihoz (www.gemedicalsystems.com/rad/ct/products/light_series/index.html).