Kernelfordítás

Alan Ward Full Circle magazine 88-91. számában
megjelent cikksorozata alapján készült összefoglaló

A Linux őskorában a kernelfordítás elkerülhetetlen feladat volt, ha működésre akarták bírni a rendszert egy adott hardverű gépen. Manapság a különféle disztribúciók legalább egy univerzális kernelt biztosítanak és esetleg néhány alternatív változatot is. A kérdés, napjainkban szükség van-e – és ha igen, miért – kernelfordításra? A kérdések megválaszolásához a cikkíró egy rövid áttekintéssel kezdi a kernel funkciójáról, felépítéséről és működéséről. A második részben körüljárja, hogy mire van szükség a sikeres kernelfordításhoz, illetve betekintést ad a kernel tartalmába. Ezt követően végigvezet egy kernel kialakítása és fordítása menetén, illetve annak gépünkre telepítése és használata módján. Végül bemutat egy konkrét esetet. Az utolsó részben – remélhetően – bemutat néhány beállítási megoldást, a különféle processzor opciókkal egyetemben és a kernelmodulok készítésének módját.

Mi a Linux kernel?

Az operációs rendszert egy egyszerűsített ábrázolás hagymához hasonlítja, ahol a legfelső burok (réteg) a felhasználói programok, ami alatt a rendszerkönyvtárak és -programok találhatók. A felhasználói programok az utóbbiak segítségével kommunikálnak a rendszer legalsó szintjét jelentő és a hardver kezeléséért, eléréséért felelős kernellel.

A kernel alapvető feladatai:

1. a folyamatok (process-ek) CPU-hoz jutásának időzítése;
2. a memóriakezelés;
3. a ki- és bemeneti eszközökhöz hozzáférés kezelése.

A felügyeletre azért van szükség, hogy a programok meghatározott elvek alapján és sorrendben férjenek (adott esetben ne férjenek) hozzá a szűkös erőforrásokhoz – CPU-idő, memória, kimeneti eszközök (képernyő, nyomtató stb.).

A konkrét telepítéstől, illetve a programok feladataitól is függ, hogy milyen rendszerprogramokra és -könyvtárakra lehet szükség. Ezek határozzák meg, hogy a programtelepítés során milyen „függőségeket” kell vizsgálni és telepíteni. Pl. ha a webböngésző https kapcsolatot létesít, akkor szüksége van telepített openssl-re is.

A szerző felhívja a figyelmet arra, hogy csak a Linus Torvalds által indított projekt www.kernel.org cím alatt található kernele nevezhető igazából Linux kernelnek. A Linux operációs rendszer néhány fontos része viszont a GNU projekt keretében készült a www.gnu.org-on, amit jelenleg a Free Software Foundation (FSF) kezel. Ide tartozik a C fordító, a gcc. A GNU projektnek van saját kernele is, a Hurd nevű, ami eltér a Linux kernelétől. A kernelek kombinálásából tudjuk elkészíteni a saját, jól ismert GNU/Linux kernelünket (csakúgy mint a GNU/FreeBSD kernelt adott esetben). Vagyis az általunk használt kernelt igazából GNU/Linux névvel kellene illetni.

Miért lehet szükség saját kernel fordítására? A rendszer, adott hardver-összeállításhoz igazítása érdekében. Kisebb méret és gyorsabb működés érhető el vele.

A legjelentősebb eltérés a processzorok terén van. (32 bit vs. 64 bit, Intel vs. AMD, egy-, illetve többmagos processzorok stb.) A fejlettebb processzorokra készült kernel nem működik a korábbi típusokkal, illetve a korábbi típusokra készített kernel nem használja ki az új processzorok újabb szolgáltatásait. Ilyen pl. PAE-képesség, amit a Pentium Pro generációval vezettek be és a régi processzorok, illetve a Pentium III alapú M (mobile) processzorok nem ismernek. M processzor található pl. az eredeti eeePC-ben is, amit a szerző fordítási példában is használ. Bizonyos disztribúciók (pl. Debian, PCLinuxOS) rendelkeznek nem PAE-képes processzoroknak szánt külön kernellel is.

A Gentoo ennél messzebb megy, ami lehetővé teszi programok a kernelbe fordítását is. Ld. www.gentoo-org/wiki/FAQ.

Előrebocsátja, hogy a fordításhoz erőteljesebb processzorra lesz szükség, és adott esetben mintegy 20 GB méretű szabad helyre a /usr könyvtárban.

A kernel forráskódjának elérése

1. Telepíteni kell a forrást (source). Disztribúciótól függően linux-source (.buntu), kernel-source (PCLinuxOS) más és más lehet a neve. Az apt csomagkezelő esetén terminált nyitva és root-jogot szerezve telepíteni kell az adott csomagot (**`apt-get install linux-source`** v. **`apt-get install kernel-source`**). Esetleg csomagkezelővel (pl. synaptic) felrakni.

2. A fájlt ki kell csomagolni bunzip2-vel (**`bunzip2 kernel-source.tar.bz2; tar xf kernel-source.tar`** – a kernel-source helyére a megfelelő csomagnevet kell írni. Egy menetben is megtehető: **`tar xjvf kernel-source.tar.bz2`**). Ezzel létrejön egy **`/usr/src/linux-source.x.x.x`** (**`kernel-source-x.x.x`**) könyvtár, benne a forrásfájlokkal.

* A kernel forrása letölthető közvetlenül a kernel.org-ról is.

A fájlstruktúra:

```
# cd /usr/src
# ls linux
arch      Documentation include   kernel            net      security  virt
block     drivers       init      lib               README             sound
COPYING   dropped.txt   ipc       MAINTAINERS       REPORTING-BUGS     tools
CREDITS   firmware      Kbuild    Makefile          samples            ubuntu
crypto    fs            Kconfig   mm                scripts  usr

# ls linux-3.15.4
```

```
arch      Documentation init      lib               README            sound
block     drivers       ipc       MAINTAINERS       REPORTING-BUGS    tools
COPYING   firmware      Kbuild    Makefile          samples  usr
CREDITS   fs            Kconfig   mm                scripts  virt
crypto    include       kernel    net               security
```

Az első az Ubuntu kernel, a második pedig a kernel.org oldalról leszedett. A kettő az ubuntu alkönyvtárban különbözik. A két kernel mérete közel azonos, az előbbi az ubuntu könyvtárban található patch-ek (foltok) miatt 8 MB-vel nagyobb. l. https://wiki/ubuntu.com/Kernel.

A README fájt érdemes elolvasni, bár egyes részei már elavultak – l. LILO-ra vonatkozó rész. A **`DOCUMENTATION`** könyvtárban haladóknak szóló technikai részletek olvashatók. A **`kernel`** könyvtárban az alapvető forráskódok találhatók. További specializált könyvtárak az **`fs`** (fájlrendszer) az **`ipc`** (inter-process communication), az **`mm`** (memory management), a **`net`** (networking), a **`sound`** (meghajtók) stb. Az **`arch`** könyvtárban az egyes CPU architektúrák kezeléséhez szükséges assembly kódok vannak, a többi könyvtár C nyelvben íródott fájlokat tartalmaz (kivéve még a **`firmware`**-t is). Az arch könyvtárban látható, hogy hányféle CPU-t képes a kernel kiszolgálni.

A **`firmware`** könyvtár az egyéb eszközvezérlők működtetéséhez EEPROM-ban, vagy flash memóriában tárolandó bináris kódokat tartalmazza. Ezeket nem tekintjük a kernel részének. Rendszerfilozófiától függően itt lehetnek nem nyílt forráskódú binárisok is!

Janu megjegyzése: *Ezzel az a bibi, hogy ha van ilyen, a kernel már nem felel meg a GNU-nak. Persze most nagy vita folyik éppen a Nvidia miatt ennek újraértelmezéséről.*

A hardver-meghajtók alapvetően a drivers könyvtárban találhatók – kivéve a hang- és néhány más eszközét. Itt ellenőrizhető, hogy a kernel kezel-e egy adott hardvert – vezérlő chip-et. Pl. a **`drivers/ethernet/realtek/`**-ben található 8139cp.c a RealTek RTL-8139C+ sorozat vezérlője, amit számos eszközbe beépítettek.

A kernelen kívüli további szükséges eszközök: c fordító és egyebek (A továbbiakban leírja mi az alapvető különbség a fordítót (c) és az értelmezőt (python) használó nyelvek között. Az előbbiben az adott nyelven megírják a forráskódot, majd egy fordítóval létrehozzák a futtatható bináris fájlt. Az értelmező típusú nyelveknél a bináris fájl létrehozása elmarad, az adott nyelven leírt forráskódot az értelmező utasításról utasításra, menet közben fordítja le.)

Janu megjegyzése: *Van egy harmadik eset is: betöltéskor előfordít a futtató bináris (pl. Perl). A Python ráadásul hermafrodita, mert képes félig binárist előállítani (.pyc fájlok)*

A hozott példa a „Hello world” üdvözlés kiíratása. A program c-ben megírva:

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

int main(int argc, char *argv[])
{
printf("Hello, world!\n");
}
```

Az első két sor meghatározza, hogy melyik fejlécfájlokra van szükség. A program fő része az egy sor, a **`printf`** függvény meghívása. A c-ben megírt forráskód legyen a **`hello.c`**, amit a „**`cc hello.c -o hello`**” paranccsal fordítunk le. A futtatás ezután a program nevével (hello) történik –

„**./hello**”. A hosszabb programok esetén, mint a Linux kernel, sok száz forráskódra és fejléc fájlra lehet szükség. Azt, hogy ezek közül adott esetben melyiket használják a fordításhoz, egy „**makefile**” nevű fájl határozza meg. *(**Janu**: U*x alatt Makefile a neve (nagy kezdőbetűvel).)*

A példánk makefile-ja lehet:

```
hello:
cc hello.c -o hello
```

A fordítás ez esetben a

```
make hello
```

utasítással történik. Tehát a kernel fordításához a c fordítón kívül még szükség van néhány eszközre:

```
gcc binutils make bzip2 coreutils
```

A make rendszer használható a kernel fordítási folyamat beállítására is. A különféle beállító szkriptek pl.

make config – szövegalapú
make xconfig és **make gconfig** – grafikus

beállító eszközök meghívásával. Az **xconfig**-hoz **qt4-default** és **qt4-qmake**, a **gconfig**-hoz **libgtk2.0-dev, libglib2.0-dev** és **libglade2-dev** kell. A szerző kedvence a **menuconfig**, curses alapú megoldás, amihez az **ncurses** csomag kell. Ez felhasználóbarátabb, de szövegalapú megoldás.

***Janu megjegyzése:** a menuconfig további előnye, hogy van Help-je, ami csak kérésre jön fel (nem úgy, mint a make config-nál).*

A kernel forráskönyvtárait és a könyvtárak fájljait átnézve a .c kiterjesztésű c forrásfájlok és a .h kiterjesztésű fejlécfájlok mellett make fájlt is láthatunk minden könyvtárban és alkönyvtárban, ami meghatározza a forráskód fordításának menetét – hogyan és mit fordítson, mi legyen a kimeneti fájl neve, illetve milyen fordítási paramétereket alkalmazzon. A könyvtárak és alkönyvtárak make fájljai csak az adott helyen található forráskódokra vonatkoznak, ami mutatja, hogy a kernel egyes elemei külön is lefordíthatóak. Ez jusson eszünkbe, amikor a kernel moduláris jellegével foglalkozunk, vagyis ha nem muszáj, nem kell az egész kernelt lefordítani, egyetlen modul is lefordítható.

De mik azok az egyes könyvtárakban szintén megtalálható kconfig fájlok? A kernel konfigurációs utasításait tartalmazzák. A Linux kernel forráskódja 2011-ben 15 millió, 2013-ban pedig már 17 millió sort tartalmazott (A cikkben található két cikk-hivatkozás, ami a Linux kernel készítőinek összetételét mutatja. Érdemes átfutni: fejlesztők 2011-ben – http://arstechnica.com/business/2012/04/linux-kernel-in-2011-15-million-total-lines-of-code-and-microsoft-is-a-top-contributor/, illetve 2013-ban – http://www.extremetech.com/computing/175919-who-actually-develops-linux-the-answer-might-surprise-you).

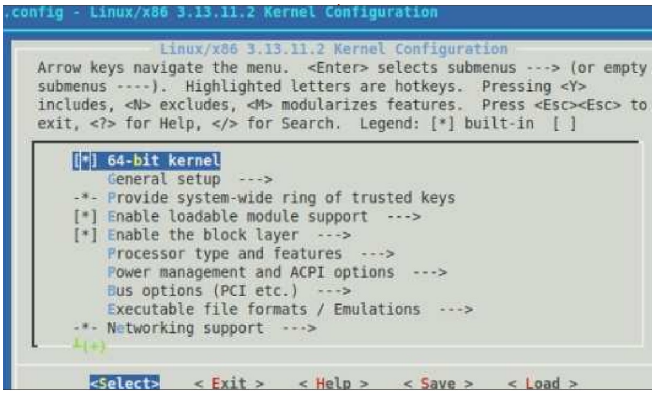
A kernel hatalmas mérete miatt szükséges az automatizált konfigurációs rendszer, és a kconfig fájlok ehhez tartoznak meghatározva, hogy az adott könyvtárban melyek az elérhető paraméterek. A cikkben szereplő példa a security/selinux kconfig fájlját mutatja be, amivel a selinux (megnövelt biztonságú Linux kernel modul) kikapcsolható indításkor – alapállapot az -n = no (azaz bekapcsolt selinux).

A második példa összetettebb. Az IPv6-csatorna alkalmazását határozza meg, három lehetséges kimenettel. Y – a modult egyenesen a kernelbe fordítja bele, azaz mindig betöltődik. N – az új kernelbe nem kerül bele. M – betölthető fájlként fordítja le a modult, boot-kor nem, de a működés során, szükség esetén betöltődik.

A kernel konfigurálása

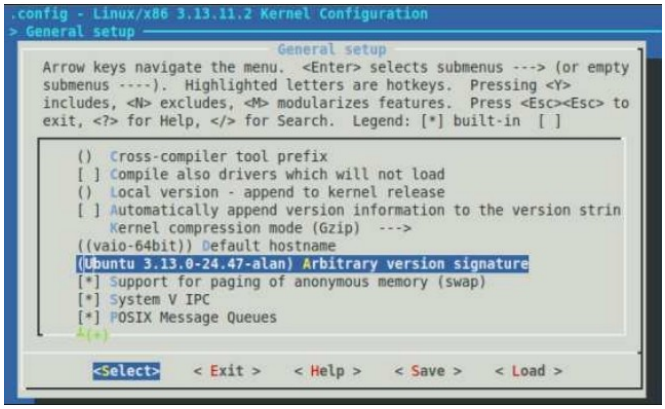
A konfigurációs fájlokon keresztül történik, amit megelőzően a korábbi beállítások maradványait el kell távolítani: „make mrproper”, majd elindítani a korábban már említett beállító eszközök egyikét. A szerző a menuconfig-ot kedveli, amit a „make menuconfig” paranccsal indít el.

A paraméterek mellett található jelek értelmezése (a fejlécben kezelési segítség olvasható):
[] - aktiválható paraméter. *, ha bekapcsolt;
-*- - nem változtatható, korábban meghatározott érték;
---> - az opcióhoz almenü tartozik;
Y, N, M – belefordított, kikapcsolt, szükség esetén betölt (modulba fordít);
le- és felfelé mutató **nyilak** – pozicionálás;
szóköz – váltás az opciók között.

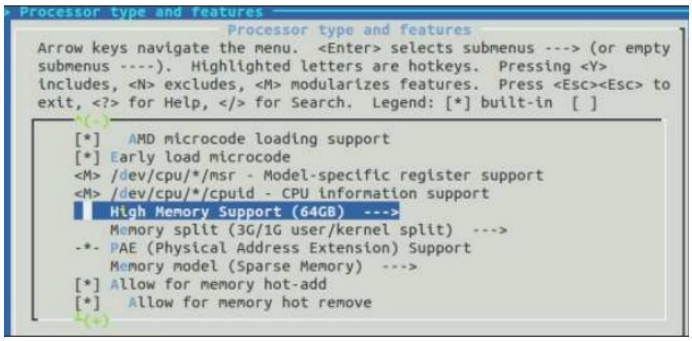


Az egyes paraméterek alapbeállításainak változtatásában kezdetben legyünk visszafogottak! Az **első opció a 64 bites kernelre** vonatkozik (l. kép). Megjegyzi, hogy a Linux ugyan lehetővé teszik a keresztfordítást (cross compile), vagyis a 64 bites kernel fordítását 32 bites processzorral és viszont, de nem ajánlja.

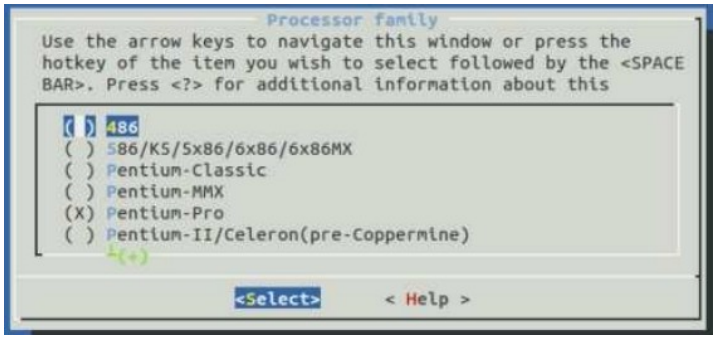
Második opció – General setup – az új kernel számos alapvető beállítására vonatkozik. A többségét hagyjuk békén, kivéve a Default hostname -et (gépnév alapbeállítása) és a Arbitrary version signature -t (verzió aláírása). Jó jelezni, hogy saját kernelváltozatról van szó. (Lekérdezés működő rendszernél „cat /proc/version_signature”.)



Processor type and features opció (almenü). Itt lehet a kernelt adott rendszer-összeállításhoz finomítani. Például a Symmetric multi-processing support-nál kikapcsolható a többmagos processzorhasználat. Emlékeztet, hogy régebben az MS rendszereknél külön fizetni kellett a többprocesszoros működés bekapcsolásáért, míg a Linuxnál a 2.0-ás kernel óta alapbeállítás. A modern processzorok többmagosak, vagy legalább a HyperThreading elérhető – fizikai magonként két logikai mag szimulálása. Ezt az SMP kezeli. Korlátozott kapacitású gépek esetén ezt a kernelből kihagyva néhány tíz kB RAM megtakarítható.



A processzor kiválasztásánál említi meg, hogy itt kapcsolható ki a PAE támogatás – ha a „High Memory Support” bekapcsolt, akkor jelenik meg a PAE opció, a vége felé. Annyit érdemes tudni róla, hogy ez csak a 32 bites kernelekre vonatkozik, a 64 bites kernel mindig tartalmazni fog valami PAE-hoz hasonló eljárást, mivel azokat a processzorokat 4 GB-nél nagyobb memória kezelésére készítették fel.



A .buntuk-hoz a Canonical további meghajtói a „Ubuntu Supplied Third-Party Device Drivers” almenüben tekinthetők meg és állítható be. (Ez csak az .buntu által biztosított forráskód esetén igaz!)

Az új kernel fordítása

A kernel fordítása két lépést takar. Első magának a kernelnek a fordítása. A második a betölthető modulok fordítása – amennyiben van ilyen aktiválható. A fordítás parancsa:

```
make
```

Sokáig és nagy processzorhasználat mellett zajló folyamat! Legjobb nagyobb teljesítményű asztali gépen végezni, a processzorok hűtése miatt. Egy kétmagos Intel i5 processzoron ez kb. két órát is igénybe vehet. Ha a változtatás csak néhány modult érintett, és a kernelt magát csak ellenőrizni, de újrafordítani nem kell, akkor a parancs

make modules

ami sokkal gyorsabban lefuthat.

A kernel telepítése

Az új kernel és a lefordított modulok a forrás könyvtáraiban találhatóak meg. Az egyes lefordított fájlokat össze kell fűzni „linkekkel” egyetlen végrehajtható fájlba a kernelnek, illetve modulonként egy-egy betölthető modul fájllá átalakítani. A kernel maga a **vmlinuz** fájl a root-ban, amit be kell csomagolni és a **/boot** könyvtárban elhelyezni. Ekkor a mérete ~150 MB-ról 5-6 MB-re csökken. A betölthető modulok .ko (kernel object) kiterjesztést kapnak és a forrásban a .c és .o fájljaik mellett találhatók. Pl. az IPv6 a ne/ipv6/ip_6tunnel.ko lesz. Az ilyen kernel modulokat a **/lib/modules/<kernel neve>/kernel** könyvtárba kell elhelyezni.

Tehát az új kernel végrehajtásához 4 műveletet kell elvégeznünk:

1. a modulokat el kell különíteni a forrásfájloktól és a /lib/modules/<kernel neve>/kernel könyvtárba másolni;
2. a kernelt tömöríteni és a tömörített fájlt a /boot könyvtárba kell elhelyezni;
3. a modulokat egyesíteni kell egy initrd (initial fájl system) tömörített fájlba és a /boot-ban elhelyezni;

   Janu megjegyzése: *A dolog ennél bonyolultabb! Az initrd-be csak a betöltés során szükséges modulok kellenek, azt majd felváltja a valódi kernel betöltése után beépülő modulok a /lib/… alól!*

4. a GRUB-ot frissíteni szükséges, hogy az új kernelt indítási opcióként el lehessen érni.

Az 1. ponthoz, a modulok telepítéséhez adminisztrátorként ki kell adni a

make modules_install

parancsot. Ezzel a modulok a megfelelő /lib könyvtárba kerülnek. Ezután a

make install

parancs a maradék 2.-4. lépéseket végzi el egyben.

Az új kernel kipróbálása

A gépet újraindítva a boot menüben az első elem az új kernel bejegyzése lesz. A régebbi kernel is megmarad a menüben. Ha hiba nélkül lefutott az indítási folyamat, akkor terminálban a

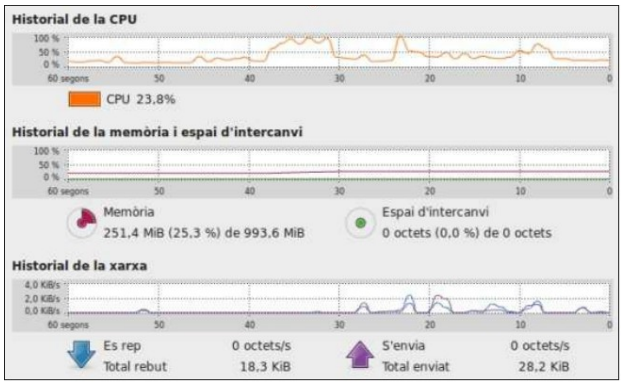
uname -a, vagy a
cat /proc/version_signature # **Megjegyzés:** *a leírás a .buntukra készült, így nem minden disztribúcióban van ilyen!*

parancsot kiadva megnézhetjük, hogy valóban az új kernel fut-e. Lehet próbálgatni, hogy az új kernel mennyivel lett másabb (jobb) a réginél.

Egy példa

A sorozat negyedik részében egy 2007-es Asus eeePC 701-esen keresztül bemutatja hogyan lehet egy meglévő kernelt frissítve új funkciókat adni a számítógépnek. Magát a fordítást egy másik, asztali gépen hajtja végre a szükséges hely és a sebesség miatt.

Első lépés a számítógép architektúrájának, építő elemeinek megismerése → Internet. Második a legfrissebb kernel (ő Ubuntu-ban utazik – a Full Circle Magazine .buntu orientált) letöltése. Harmadik lépés a jelenlegi állapot ellenőrzése – Mate system monitor adatainak ellenőrzése. A felmérés alapján a probléma a processzor környékén lesz – egymagos, 32 bites – a leterheltsége gyakran eléri a 100%-ot.



Az adott hardverre méretezéssel radikális változásokat lehet elérni. Ne felejtsük el, hogy a 32 bites kernelt 32 bites rendszeren fordítsuk, ha el akarjuk kerülni az esetleges problémákat.

1. lépés: **`make mrproper`** – az előzmények eltávolítása.
2. lépés: **`make menuconfig`** (vagy bármelyik másik beállító program) betöltése.
3. lépés: végig megyünk a **beállítások**on az elejétől kezdve. **`64 bit`** kikapcsolása. Csak a 32 bit Celeron-t bejelölni. **`Cross compiler`** nem kell. Kiválasztani a szükségtelen, vagy helyileg töltendő modulokat. **`Kernel compression mode`** – a kernel tömörítése. Alaphelyzetben gzip. Maradhat, mert többnyire preferált bzip nagyobb tömörítést ad ugyan, de betöltéskor jobban terheli a processzort, mert röptében csomagol ki. Írjuk át a kernel nevét – **`Arbitrary version signature`**. A többi almenü elemmel nem kell foglalkozni.

Kapcsoljuk ki a **`Support for paging anonymous memory`** – SWAP – javaolja. Ő 1 GB memoria mellett nem szereti használni. 512 GB esetén maradhat. Az **`initrd`** fájl beállításainál a tömörítés szintén maradhat gzip. Vissza a főmenübe. **`Enable loaded module support`** – modulbetöltés engedélyezése – bekapcsolandó. Elméletileg minden modul befordítható a kernelbe és akkor ez nem kellene, de ezáltal a boot-kor a betöltődő modulok a memóriát terhelnék. **`Enable block layer on`** – maradjon bekapcsolva. A blokkeszközök, mint pl. merevlemezek eléréséhez kell.

Most jön a processzor rész – **`Processor type and features`**. **`Symmetric multi processing support`** – kikapcsol. **`Support for extended (non-PC) x86 platforms`** – kikapcsolni. **`Intel Low Power Subsystem Support`** – kikapcsolni. **`Linux guest support`** – kikapcsolni. Ez egy fizikai eszköz és nem virtuális. **`Memtest`** – kikapcsolható.

`Processor family` – **`PentiumIII/Celeron(Coppermine)`**. A teljes Pentium-III utasítás készletet megkapjuk, amit alapból nem. (A csomagolók hajlamosak a Pentium-MMX-et választani, mivel az működik a Pentium-I 166-tól, vagy 200-tól.) A **`Generic x86 support`** legyen kikapcsolt, hiszen egy konkrét, ismert gépünk van. Lejjebb a **`Toshiba`**..., **`Dell`**... **`support`** és a **`CPU microcode`**

loading support kikapcsolható, ha csak nem tervezed a CPU mikrókódját frissíteni.

High Memory Support almenü fontos. **64 GB** opció nem kell. 1 GB-nél a **High Memory** mehet off-ba. Azt azonban tudni kell, hogy ezt kikapcsolva a kernel fenntartja magának a RAM felső 128 MB-jét. További részleteket erről a http://unix.stackexchange.com/questions/4929/what-are-high-memory-and-low-memory-on-linux oldalon lehet olvasni. A **PAE** marad kikapcsolva. Az almenü többi része maradhat, ahogy van. Csak arról győződj meg, hogy a **MTRR support** be van kapcsolva és az **EFI runtime support** kikapcsolva.

Tovább le a **Power management and ACPI options**-ra, a Suspend to RAM kikapcsolható. Csakúgy a **Power Management Debug**. A **CPU frequency scaling** kikapcsolható.

Vissza a főmenübe. A **Networking support**-ban – hacsak nem akarod USB-ről, vagy szoftverben használni – kikapcsolható az Amateur Radio, CAN bus, IrDA, Bluetooth, Wimax, Plan 9, CAIF és NFC.

A **Device drivers** szakaszban a többsége **M** módban meghagyható. A **File systems** résznél ízlés szerint eltávolítható a Reiserfs, JFS, XFS, GFS2, OCFS2, btrfs és az NILFS2 csakúgy mint CD-ROM/DVD Filesystems, mehet. A DOS/FAT/NT-nél nem árt a VFAT-ot bekapcsolva hagyni az USB meghajtók miatt.

A **Kernel hacking** – inkább hagyjuk békén. **Security options** – ő nem használja a SELinux-ot, így kikapcsolja. Hasonlóképpen a Tomoyo..., AppArmor, Yama, Integrity...-t.

Ismét vissza a főmenübe. A **Cryptographic API**. Jobb bekapcsolva hagyni a meghajtókat. A **Virtualization**-ban minden kikapcsolható. A **Library routines**-ban maradhatnak az alapértékek.

Fordítás, tesztelés és telepítés

A fordítást végző gépen

```
make
```

Adminisztrátorként

```
make modules_install
make install
```

Ha fordítást Pentium III-nál jobb processzoros gépen végezted, akkor mindennek futnia kell, így kipróbálható. **Kernel Panic** esetén olvasd át figyelmesen a hibajelzést és kutakodj a neten, mi okozhatja. (Indítsd a régi kernelt és javíts.)

Ha elindul az új kernellel a gép, de később áll le

```
Starting init: /bin/sh exists but couldn't execute it (error -8)
```

üzenettel, akkor azt jelenti, hogy a kernel betöltésekor elfogyott a memória, így az initrd nem tölthető be. A lehetséges hibaok származhat GPT-t tartalmazó merevlemez miatt stb. Lehet túl méretes initrd miatt is, ami **ls -lh /boot** paranccsal ellenőrizhető. Ha az initrd mérete (>20 MB) 100 MB-re felmegy, akkor az előbbi a hiba. Erről továbbiakat itt:

http://unix.stackexchange.com/questions/30345/why-is-my-initial-ramdisk-so-big olvasható.

Lehetséges megoldás:

A kezdő fordítás

```
make INSTALL_MOD_STRIP=1
```

utána root-ként

```
make INSTALL_MOD_STRIP=1 modules_install
make INSTALL_MOD_STRIP=1 install
```

Ezáltal az initrd kb. 10-15%-kal kisebb lesz.

Ha minden rendben van, akkor az új kernelt telepíteni kell az eeePC-re. A fordítást egy 8 GB SD kártyára másolta át, amiről gyakorlatilag a rendszerét futtatja. Aki a belső meghajtóról használja a gépet, vagy más disztribúciót használ, annak értelemszerűen kell a másolásokat végrehajtani!

Root-ként

```
cp /boot/*3.13.11.2 /media/<felhasználónév>/<meghajtó neve>/boot/
cp -r /lib/modules/3.13.11.2 /media/<felhasználónév>/<meghajtó
neve>/lib/modules/
```

Ez helyükre másolja a előbb a kernelt és az initrd-t, majd a modulokat az új rendszer érdekében. Ezután – ő kihúzta a fordításra használt gépből az SD kártyát és arról elindította – akkor még a régi kernellel – az eeePC-t. (Aki másképpen másolta át a fájlokat a helyükre, annak is az újra indítás természetesen szükségtelen.) A még régi kernellel futó rendszeren root-ként

`grub-update` – ha GRB2-t használsz, Megjegyzés: a grub-legacy-n ez szükségtelen, nincs ilyen parancs.

A kerneles sorozat ötödik részében folytatódik a példaként használt Asus eeePC-re szabott kernel piszkálása a kód módosításával. A kernel kódjának apró módosításaival és azok hatásának vizsgálatán keresztül nyújt betekintést a forráskód belső megszakításaiba. A példa kedvéért apró módosításokat hajt végre a /proc fájlrendszeren. Emellett láthatjuk, hogyan kap tájékoztatást a rendszerfelhasználó a kernelben zajló folyamatokról.

A proc fájlrendszer

A korábbiakban már használtuk a /proc virtuális fájlrendszert, amikor a /proc/version_signature fájlt beolvastuk, hogy lássuk pontosan milyen kernel verziót futtatunk.

```
$ cat /proc/version_signature # ugyanazt adja vissza, mint a ‚uname -a'

Ubuntu 3.13.0-24.47-generic 3.13.9
```

Azt azért nem árt tudni, hogy a /proc fájlrendszer fájljai és könyvtárai valójában nem léteznek a lemezen. Ellentétben a „valódi” fájlokkal és könyvtárakkal, amiket ha el akarunk érni, a kernel felépít a memóriában a lemezen lévő szerkezet visszatükrözésére egy belső adatstruktúrát, ellenben a /proc adatstruktúrája csak a kernel memóriájában létezik, a lemezen nem. Ezért lehet virtuális

fájlrendszernek minősíteni. Amikor ezekhez a fájlokhoz fordulunk, azokat – a bennük foglalt adatokkal együtt – a kerneltől magából kapjuk.

Ez a rugalmas fájlrendszer teszi lehetővé a kétoldalú kommunikációt a felhasználó és kernel között – adatokat közöl és utasításokat (paramétereket) fogad ezen keresztül a kernel. Pl. a hálózati IP-cím továbbítása alaphelyzetben kikapcsolt, ám szerverként működés esetén az interfészek közötti továbbítás, az alábbi (rendszergazda) paranccsal bekapcsolható

```
# echo 1 > /proc/sys/net/ipv4/ip_forward # IPv4

# echo 1 > /proc/sys/net/ipv6/conf/all/forwarding # IPv6
```

A „0” kikapcsolja, bármilyen pozitív érték bekapcsolja a szolgáltatást.

A változtatás a rendszer újraindítását igényli!

Egyszerű /proc bejegyzés anatómiája

A proc fájlrendszert az fs/proc/ alkönyvtárban található forráskód vezérli. Nézzünk egy egyszerű példát, a /proc/version fájlt menet közben előállító – az fs/proc/version.c-ban található – kódot (a 3.13-as kernel szerint):

```
$ cat /proc/version

Linux version 3.13.0-24- generic (buildd@batsu) (gcc version 4.8.2 (Ubuntu
4.8.2-19ubuntu1) ) #47-Ubuntu SMP Fri May 2 23:30:00 UTC 2014

#include <linux/fs.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/utsname.h>
static int version_proc_show(struct seq_file *m, void *v)
{
seq_printf(m, linux_proc_banner,
utsname()->sysname,
utsname()->release,
utsname()->version);
return 0;
}
static int version_proc_open(struct inode *inode, struct file *file)
{
return single_open(file, version_proc_show, NULL);
}
static const struct file_operations version_proc_fops = {
.open = version_proc_open,
.read = seq_read,
.llseek = seq_lseek,
.release = single_release,
};
static int __init proc_version_init(void)
{
proc_create("version", 0, NULL, &version_proc_fops);
return 0;
}
```

```
module_init(proc_version_init);
```

Most nézzük át az egyes részek tartalmát. Az első néhány #include sor befoglalja a fejléc fájlokat az include/linux/ forráskód könyvtárból:

• **<kernel.h>** a forráskódban használt, legalapvetőbb makró definíciókat tartalmazza.
• **<init.h>** az inicializáló kódokat tartalmazza, ezen belül a modul inicializálásához használtakat.
• **<fs.h>** a fájlrendszer részeinek alapvető meghatározását tartalmazza, mint például az olyan kódokét, amik jelzik, hogy a fájl írható, olvasható-e stb.
• **<proc_fs.h>** hasonló kódokat tartalmaz, de ezúttal kimondottan a /proc fájlrendszerre.
• **<seq_file.h>** a soros fájlműveletek végrehajtásához a közös kódokat definiálja.
• **<utsname.h>** a felhasználói felületről meghatározott adatok eléréséhez szükséges kódokat tartalmazza.

A C-ben programozók számára ez túlbonyolítottnak tűnhet, hiszen a fájlokra vonatkozó rutinok megtalálhatóak az olyan szokásos C fejlécfájlokban, mint a <stdio.h>. Meg kell azonban jegyezni, hogy az olyan szabvány I/O (ki- és beviteli) rutinok mint a „printf” történetesen a glibc könyvtárfájlba kerülnek lefordításra, amit a kernel nem ér el addig, amíg a fájlrendszer fel nem állt. Ha a rendszerbetöltő meghajtó nem működik, a kernel nem éri el a szabványos C könyvtárakat, miközben hibaüzenetet kellene megjelenítenie, tudatván a felhasználóval, hogy mi történt. Tehát, ezért kell a szövegkiíratót rutinokat a kernelbe fordítani. A kernel-parancsok hasonlóak, de nem feltétlenül egyeznek a glib-es változattal. Pl. a képernyőre író parancs a „printk”, aminek a szintaxisa megegyezik a „printf”-évél.

A kódban az utolsó sor a

```
module_init(proc_version_init)
```

Ez az, ahová a kódot vezérlő /proc/version fájl modulként betöltődik a memóriába. Akkor nézzük a „proc_version_init”-et:

```
static int __init proc_version_init(void)
{
proc_create("version", 0, NULL, &version_proc_fops);
return 0;
}
```

Ez a modul inicializálása felett deklarált eljárás, ami meghatározza, hogy a modul telepítésekor mit kell csinálni, azaz létre kell hoznia a „version” virtuális fájlt a /proc keretében és hozzárendelni egy callback függvénytáblázatot, amihez akkor fordul, amikor a szokásos műveleteket hajtják végre a virtuális fájlon.

A “file_operations version_proc_fops”-nak hívott callback függvények:

```
static const struct file_operations version_proc_fops = {
.open = version_proc_open,
.read = seq_read,
.llseek = seq_lseek,
.release = single_release,
};
```

Ezek közül csak az „open” műveletet definiálja “version_proc_open” függvényben, a többi marad alapértéken.

Közvetlenül előtte a „version_proc_open”-t definiálja ekképpen:

```
static int version_proc_show(struct seq_file *m, void *v)
{
seq_printf(m, linux_proc_banner,
utsname()->sysname,
utsname()->release,
utsname()->version);
return 0;
}
```

Ez egyszerűen egy „seq_printf”-fel kiírja fájlba a linux verziót, a futtatott rendszer nevét, valamit a kiadás és a verzió számát tartalmazó címkét.

Egy /proc tétel módosítása

A /proc/version fájl talán nem annyira érdekes a módosítás szempontjából, ellenben a /proc/cpuinfo sokkal több állítási lehetőséggel bír:

```
$ cat /proc/cpuinfo
[...]
processor : 3
vendor_id : GenuineIntel
cpu family : 6
model : 37
model name : Intel(R) Core(TM) i5 CPU M 460 @ 2.53GHz
stepping : 5
microcode : 0x2
cpu MHz : 2527.207
cache size : 3072 KB
 […]
```

(Az átláthatóság kedvéért kicsit szerkesztve!).

Ezek az információk a rendszerben található valamennyi processzorra adottak. Ez hasznos, így nem kell a chip dokumentációját átnézni ezekért az adatokért. Ugyanakkor nem árt fenntartásokkal kezelni az adatokat, illetve kevesebb, de az adott igényeknek jobban megfelelő információ jobb lenne.

Kezdjük az ismerkedést az fs/proc forráskód könyvtárban található cpuinfo.c-vel, aminek a szerkezete – a callback függvények meghívását leszámítva – hasonlít a version.c-éhez. A cpuinfo.c esetében az open műveletet „cpuinfo_open”-nek hívják és a következőképpen definiált:

```
extern const struct seq_operations cpuinfo_op;
static int cpuinfo_open(struct inode *inode, struct file
*file)
{
return seq_open(file, &cpuinfo_op);
}
```

Eszerint, most meg kell tudnunk, hogy a kernel forráskódjában a „cpuinfo_op” hol van előre kódolva. E célra használhatjuk a grep parancsot megfelelően paraméterezve, vagy fordulhatunk a Linux Cross Reference tool-hoz a Free Electrons-on: http://lxr.free-electrons.com/ Az eszköz nagyon hasznos, amikor szavakat, vagy szimbólumokat kell a Linux forráskódjában keresnünk. (Próbáld ki az „Identifier search”-öt”)

Mindenesetre a nekünk megfelelő kód az arch/x86/kernel/cpu/proc.c fájlban található ilyen formában:

```
static int show_cpuinfo(struct seq_file *m, void *v)
```

Látható, hogy a /proc/cpuinfo-nak adott információ folyamatában sorosan adódik hozzá a kimeneti fájlhoz. A 60. sortól a következőket látjuk:

```
seq_printf(m, "processor\t: %u\n"
"vendor_id\t: %s\n"
"cpu family\t: %d\n"
"model\t\t: %u\n"
"model name\t: %s\n",
cpu,
c->x86_vendor_id[0] ? c->x86_vendor_id : "unknown",
c->x86,
c->x86_model,
c->x86_model_id[0] ? c->x86_model_id : "unknown");
```

Az egészet megjegyzésbe tehetjük és lecserélhetjük a saját kódunkkal. A módosításokat nem árt megfelelő megjegyzésekkel ellátni. A szerző az arch/x86/kernel/cpu/proc.c-ban található show_cpuinfo-t módosította a következőképpen:

```
/* -- by Alan -- */
seq_printf(m,
"CPU[%d]:\n\t%s %s\n",
cpu,
c->x86_vendor_id[0] ? c->x86_vendor_id : "unknown",
c->x86_model_id[0] ? c->x86_model_id : "unknown");
if (cpu_has(c, X86_FEATURE_TSC)) {
unsigned int freq = cpufreq_quick_get(cpu);
if (!freq)
freq = cpu_khz;
seq_printf(m, "\t%ld.%06lu GHz\n",
freq / 1000000, (freq % 1000000));
}
seq_printf(m, "\tcpu cores\t: %d\n", c->booted_cores);
/* -- end by Alan -- */
```

A módosítás után a kernelt újra kell fordítani és telepíteni a /boot könyvtárba, továbbá a GRUB-ot frissíteni. Minthogy a modulokat nem módosítottuk, azokat nem kell újrafordítani és -telepíteni. Ha már csináltuk kernelfordítást legalább egyszer, akkor elég gyorsan fog menni:

```
$ make
$ sudo bash
# make install
```

Az arch/x86/kernel/cpu/proc.c babrálása közben számos, létező függvényt tettünk megjegyzésbe (comment) és a fordító panaszkodni fog a valahogy így:

```
warning: ‘show_cpuinfo_core’ defined but not used [- Wunused-function]

static void
show_cpuinfo_core(struct seq_file *m, struct cpuinfo_x86 *c,
```

(A próbánk szempontjából ez nem gond. Ám, ha egy valós kernel projektbe dolgozunk be, akkor

jobb egy kicsit kitisztázni a dolgokat, ha nem akarunk problémát magunknak.)

Amikor a fordítás és a telepítés kész, boot-olj be róla. Az asztalon semmi változás nem látható. Akkor most ellenőrizzük le a /proc/cpuinfo tartalmát újonnan. A proc.c-ben eszközölt változásokat kell visszatükröznie.

```
$ cat /proc/cpuinfo

CPU[0]: GenuineIntel Intel(R) Core(TM) i5 CPU M 460 @ 2.53GHz 2.527462 GHz
cpu cores : 2
[…]
```

Na ez már sokkal egységesebb, szebben néz ki és számomra jobban olvasható.

Összességében megállapíthatjuk, hogy a /proc virtuális fájlrendszer alkalmas a saját kódunk kipróbálására. Kezdetben használhatjuk információk kinyerésére a kernelből, ám a vállalkozóbb szelleműek később kipróbálhatják a fordítottját is – a kernel belső folyamatainak buherálását futás közben.

Az előbbiekben módosítottunk a kernel kódján, amivel információkat szolgáltat a processzorunkról. Most teljesen új kódot készítünk, amit azután beillesztünk a kernelbe, lefordítjuk és futtatjuk egy párszor.

Az új tulajdonságot nem a kódba magába írjuk bele, hanem modult készítünk, lefordítjuk. Amellett, hogy nem kockáztatjuk a teljes kernel összeomlasztását, még gyorsabban lefordítható és bármikor beilleszthető, vagy eltávolítható, amikor csak úgy kívánjuk.

Továbbra is a /proc fájlrendszerben maradunk és megnézzük, hogyan lehet a kernellel való kétirányú kommunikációra használni, nem csak fogadni, hanem információkat, parancsokat adni.

A kód, amit példaképpen készítünk, átírja a /proc/hostname-et – a kernel fogadja az általunk megadott szöveget és átírja a hostname-et. Noha a proc fájlrendszerben már van egy ilyen eszköz (/proc/sys/kernel/hostname), de mi most a felhasználók számára is lehetővé fogjuk tenni a hostname átírását.

Az új kódot a kernel forráskód fa struktúrájában bárhol, vagy akár külön könyvtárban is elhelyezhetjük, én az fs/proc könyvtárba raktam, hostname.c néven.

Új modul készítése

Kernel modul készítésének alapfeltételei:

* Kell egy olyan kód, ami inicializálja az adatstruktúrá(ka)t és az általunk készített /proc elemeket a modul betöltésekor.
* Kell továbbá olyan kód, ami korrekt módon takarít, amikor a modult a memóriából töröljük. Eltérően más operációs rendszertől a Linuxban a kernel modulok rendben eltávolíthatóak, ha már nincs szükség rájuk.
* Végül készítenünk kell valamilyen callback függvényt, ami meghívható, ha a művelet a /proc fájlon végrehajtásra került.

A callback függvények alkalmazása operációs rendszerek készítésében gyakori eljárás. Például, ha rendszeren belül várunk valamilyen esemény bekövetkezésére (pl. billentyű lenyomására), döntenünk kell, milyen stratégiát követünk. Az első lehet, hogy rendszeres időnként lekérdezzük a

billentyűzetet, hogy van-e feldolgozható új billentyűutasítás. Ez nem túl hatékony, hiszen számos folyamat futhat a rendszerben akkor is, amikor a billentyűzet nincs használatban.

A másik lehetőség, amikor a megszakítási eljárást használjuk, ami lényegében a várakozást a hardverre bízza. A megfelelő meghajtó az indításkor készít egy, billentyűlenyomást feldolgozó függvényt, ami nem fut le, hanem a memóriában várakozik. A címe (egy jelző) átkerül a megszakításkezelő rendszerbe. Amikor billentyű lenyomás hatására megszakítást érzékel, meghívja ezt a függvényt.

A callback függvény használható akkor is, amikor a kernelen kívülről érkező eseményt várunk. Ezek lehetnek fizikai események (egér-billentyű lenyomása), vagy logikai (szoftver) események (esetünkben a felhasználó olvas egy fájlt).

A modul inicializálása és eltávolítása

Az első dolog, amit teszünk, hogy beillesztjük a modul utolsó két sorát:

```
module_init(hostname_proc_init);
module_exit(hostname_proc_exit);
```

A „module_init” jelzi a modul memóriába-töltésekor meghívandó függvényt, esetünkben a „hostname_proc_init”-et. A „module_exit” jelzi azt a függvényt, amit a modul eltávolításakor kell meghívni a takarítás érdekében.

Vedd észre, hogy az elnevezésben követem a kernel forráskódnál is alkalmazott konvenciót, miszerint az összes függvény neve a modul nevével indul, amit a proc követ – jelezvén, hogy a kód a /proc fájlrendszeren belül működik. Végül következik a függvény egyedi azonosítója, ami a használatára utal.

Ugyan ezt a rendszert alkalmazom a modul kezelésére készített hivatkozásnál is, ami a korrekt telepítés ellenőrzésekor jöhet jól.

```
static struct proc_dir_entry *hostname_entry = NULL;
```

Most akkor meg kell írni a modul inicializálására szolgáló kódot:

```
static int __init hostname_proc_init(void)
{
           printk(KERN_INFO "hostname loading\n");
           hostname_entry = proc_create("hostname", 0666, NULL,
&hostname_proc_fops);

           if (hostname_entry == NULL)
                      printk(KERN_INFO "hostname could not create /proc
entry\n");
           else {
                      hostname_entry->proc_iops = &hostname_proc_iops;
                      printk(KERN_INFO "hostname /proc entry created\n");
           }
           return 0;
}
```

A függvény kialakítás szabványos, ehhez ragaszkodnunk kell. Az új /proc bejegyzést 0666-os jogosultsággal készítem, ami minden felhasználónak olvasási és írási jogot ad (tulajdonos, csoport, többiek). A kernel naplójába (log) is elég sok információt teszek, ami a hibakeresést segíti.

Vedd észre a két tábla címét:

„hostname_proc_fops” és
„hostname_proc_iops”

Ezek, további callback függvényekre mutató hivatkozások, amiket a felhasználó hozzáférési jogosultságának ellenőrzésekor használ.

A „hostname_proc_permission” célja, hogy kiadja a szöveget, amikor a /proc bejegyzésünket a „hostname_proc_open” kiolvasta, illetve hogy, kiolvassa és feldolgozza a felhasználói adatokat, amikor /proc bejegyzésünk a „hostname_proc_write”-ba bekerül:

```
static const struct inode_operations hostname_proc_iops = {
           .permission = hostname_proc_permission,
};
static const struct file_operations hostname_proc_fops = {
      .open   = hostname_proc_open,
      .read   = seq_read,
      .write  = hostname_proc_write,
      .llseek         = seq_lseek,
      .release        = single_release,
};
```

A többi művelet megmarad az eredeti kezelő (handler) beállításaival (seq_read stb.)

A modul eltávolítása memóriából:

```
static void __exit hostname_proc_exit(void)
{
           printk(KERN_INFO "hostname unloading\n");}

           if (hostname_entry == NULL)
                     printk(KERN_INFO "hostname /proc entry does not exist, not
removing\n");
           else {
                     proc_remove(hostname_entry);
                     printk(KERN_INFO "hostname /proc entry removed\n");
           }
}
```

Esetünkben elég kevés takarítani való van, csak a /proc bejegyzésünket kell eltávolítani a „proc_remove” függvénnyel. Mint korábban is, törekedtem több információt adni a log-nak. (Ezek zömét, ha tényleg használatba vesszük a modult, el kell távolítani.)

A /proc bejegyzés olvasásának és írásának feldolgozása.

Miként a korábbi részekben a /proc bejegyzés felhasználók általi olvasását két függvény kezeli. A „function_hostname_open”, ami a „hostname_proc_fops”-ban meghatározott callback függvény. Ám ez csak ez csak a /proc bejegyzés fájlhoz és az inode mutatókhoz fér hozzá. A hozzáférés megkönnyítésére jól használható a „single_open” függvény, ami soros fájlelérési mutatót „m” biztosít, ezt lehet használni a „printf”-ban a fájlba történő formázott íráshoz.

```
static int hostname_proc_open(struct inode *inode, struct file *file)
{
           return single_open(file, hostname_proc_show, NULL);
}
```

```
static int hostname_proc_show(struct seq_file *m, void *v)
{
            seq_printf(m, "system hostname is currently: %s\n", utsname()-
>nodename);
            seq_printf(m, "write new name to this file to change hostname\n\n");
            return 0;
}
```

Itt tud a felhasználó olvasni a /proc bejegyzésünkből:

```
cat /proc/hostname
system hostnaem is currently:
alan-vaio

write new name to this file to change hostname
```

A válasz a bejegyzésünkbe történő beírásra kicsit összetettebb. A felhasználói eljárás (process) által írt adat elérhetővé válik a „user_data” mutató számára és a rendelkezésre álló karakterek száma pedig a „len”-ben lesz. Ugyanakkor ez az adat a felhasználói adatstruktúrában lesz, amit át kell másolni a megfelelő kernel-területre feldolgozás előtt.

Nagyon gondosan le kell ellenőriznünk a felhasználói bevitel tartalmát. Bármilyen művelet, amit hibás felhasználói adatok alapján végzünk a kernelen, a teljes rendszer működésére hatással lehet. Fokozott óvatosság kell!

Esetünkben csak egyszerűen átmásoljuk a felhasználó által megadott első karaktereket a nem nyomtatható karakterig (azaz bármilyen szóköz előtti kódig). Ha ezeknek a karaktereknek a száma kevesebb a megengedhetőnél (ellenőrizd!), akkor másold az új hostname-et a megfelelő utsname táblába.

A függvény bezárása előtt a sikeresen beolvasott karakterek számát vissza kell adnunk. Ezt a rendszer használja fel – ha a rendelkezésre álló karaktereknél kevesebbet kezelt, akkor a rendszer ismét meghívja ezt a függvényt – vagy annyiszor, amennyiszer csak kell –, hogy feldogozza a teljes beírást. Vagyis egyszerűen annyi karaktert adunk vissza, amennyi a „len”, így nem kell többször meghívni a függvényt.

A kód:

static ssize_t hostname_proc_write(struct file *file, const char __user *user_data, size_t len, loff_t *offset)

```
{
            int buffer_size = 0;
            int i;
            char buffer[MAX_BUFFER_SIZE];
            printk(KERN_INFO "hostname_proc_write, len=%ld\n", len);

            buffer_size = len > MAX_BUFFER_SIZE ? MAX_BUFFER_SIZE : len;
            i = 0;
            buffer[0] = '\0';
            while ((i < buffer_size) && (user_data[i] > ' ')) {
                        buffer[i] = user_data[i];
                        i++;
            }
            buffer[i] = '\0';
            buffer_size = i+1;

            printk(KERN_INFO "wrote %d bytes\n", buffer_size);
```

```
            printk(KERN_INFO "hostname=%s\n", buffer);

            if (buffer_size <= __NEW_UTS_LEN)
                        for (i = 0; i < buffer_size; i++)
                                    utsname()->nodename[i] = buffer[i];
            return len;
}
```

A modul végleges forráskódja

A forráskódunk egyelőre nem fordulna le, mert kihagytam belőle az összes „include” utasítást, amik a függvény prototípusok definiálásához kellenek. Ezeket be kell illeszteni a fájlunk elejére. Nem árt még a fordítónak némi információt átadni, hogy mire való a modulunk, ki a szerzője és a licenc dolgokat:

```
MODULE_AUTHOR("Alan Ward");
MODULE_LICENSE("GPL v2");
MODULE_DESCRIPTION("hostname module for Full Circle Magazine");
```

Ezek az információk a modulba magába kerülnek bele és ellenőrizhetők a „module_info” paranccsal.

```
$ modinfo hostname.ko
filename: /home/alan/Escriptori/linux/fs/proc/hostname.ko
description: hostname module for Full Circle Magazine
license: GPL v2
author: Alan Ward
srcversion: 431F7E34A05708273893D24
depends:
vermagic: 3.13.0-24-generic
SMP mod_unload modversions
```

Ha már minden elemed és kódod megvan, akkor a végleges modul-kód e szerint állítható össze:

http://pastebin.com/5d6KxCRZ

Az új modul fordítása és telepítése

Amikor a modult már lefordítottuk, be kell kötni hivatkozásokkal a kernel adatstruktúrájába és függvényei közé. A legegyszerűbben úgy oldható meg, hogy integráljuk a meglévő forráskód könyvtár make rendszerébe.

```
obj-m := hostname.o
```

Most már nekiláthatunk a modulok fordításának, közötte a mi új modulunkat úgy, hogy a kernel forráskönyvtárában futtatjuk a make parancsokat:

```
cd ../..
make modules
```

Alternatív lehetőség és talán célszerű is csak az ebben a könyvtárban lévő modulokat fordítani a következő paranccsal:

```
make -C /lib/modules/`uname-r`/build M=$PWD modules
```

Ha a fordítás lezajlott, akkor telepíthetjük a modult a futó kernelbe:

```
su -
insmod hostname.ko
```

és lehet használni. Ha szükséges a modul eltávolítható:

```
su -
rmmod hostname
```

Érdemes elolvasni a kernel log-ját, hogy lássuk, minden rendben működik-e:

```
dmesg | tail

[ 7501.047170] hostnameloading
[ 7501.047178] hostname /procentry created
[ 8095.253713]hostname_proc_write, len=17
[ 8095.253722] wrote 17 bytes
[ 8095.253726] hostname=our-new-hostname
[ 8381.501772] hostnameunloading
[ 8381.501784] hostname /procentry removed
```

Ezzel vége a kernelfordítási sorozatnak. Láthatjuk, hogy a kernel megváltoztatása, fordítása és telepítése talán az egyik legnehezebb feladat, de türelemmel és szisztematikus munkával kevés ismerettel is megoldható. Vagyis nem kell hacker-nek, vagy számítógépes zseninek lenni hozzá, ugyanakkor nagy mértékben szélesíti az ismereteket.

Emellett még azt is megmutatta, hogy a nyílt forráskódú kernelt futtató operációs rendszert sokan ellenőrizhetik és fejleszthetik.