Valósidejű rendszerek

Elektronikus jegyzet

2. fejezet

Készítette: Bányász Gábor tanársegéd
BME Automatizálási és Alkalmazott Informatikai Tanszék
1117. Budapest, Magyar Tudósok körútja 2.
QB.218.
Tel: 463-2597
Fax: 463-2871 (adminisztráció)
Mail: banyasz.gabor@aut.bme.hu

Hallgatják: Villamosmérnöki és Informatikai Kar
Nappali tagozat
Villamosmérnöki mesterszak (MSc)
I. évfolyam, 1. félév
Számítógép alapú rendszerek szakirány hallgatói

COPYRIGHT

Tartalomjegyzék

1 Bevezetés

1.1 A Linux

A Linux kiváló, ingyenes platformot ad a programok fejlesztéséhez. Az alapvető fejlesztőeszközök a rendszer részét képezik. Unix-szerűségéből adódóan, programjainkat könnyen átvihetjük majdnem minden Unix- és Unix-szerű rendszerre. További előnyei:

- A teljes operációs rendszer forráskódja szabadon hozzáférhető, használható, vizsgálható és szükség esetén módosítható;
- Ebbe a körbe bele tartozik a kernel is, így komolyabb, a kernel módosítását igénylő problémák megoldására is lehetőségünk nyílik;
- Mivel a stabil és a fejlesztői kernel vonala jól elkülönül, ezért célrendszerek készítésénél szabadon választhatunk, hogy stabil, vagy a legújabb fejlesztéseket tartalmazó rendszerre van szükségünk;
- A Linux fejlesztése nem profitorientált fejlesztők kezében van, így fejlődésekor csak technikai szempontok döntenek, marketinghatások nem befolyásolják;
- A Linux felhasználók és fejlesztők tábora széles és lelkes. Ennek következtében az interneten nagy mennyiségű segítség és dokumentáció lelhető fel.
- A Linux rendszer nagy mértékben skálázható az egészek kis beágyazott rendszerektől a kiterjedt szerver rendszerekig.
- Tudásban és képességekben nem marad el versenytáraitól, illetve sok szempontból túl is tesz rajtuk.

Az előnyei mellett természetesen meg kell említenünk hátrányait is. A decentralizált fejlesztés és a marketing hatások hiányából adódóan a Linux nem rendelkezik olyan egységes, felhasználóbarát kezelői felülettel, mint versenytársai, beleértve a fejlesztői eszközök felületét is. Emellett az információk begyűjtése a szoftver csomagok illesztése sokszor többlet energiát igényel.

1.2 A szabad szoftver és a Linux története

A számítástechnika hajnalán a cégek kis jelentőséget tulajdonítottak a szoftvereknek. Elsősorban a hardvert szándékozták eladni, a szoftvereket csak járulékosan adták hozzá, üzleti jelentőséget nem tulajdonítottak neki. Ez azt eredményezte, hogy a forráskódok, algoritmusok szabadon terjedhettek.

Azonban ez az időszak nem tartott sokáig, a gyártók hamar rájöttek a szoftverben rejlő üzleti lehetőségekre, és ezzel beköszöntött a zárt forráskódú programok korszaka. Az intellektuális eredményeket a cégek levédik, és szigorúan őrzik.

Ezek a változások nem nyerték el Richard Stallman (Massachusetts Institute of Technology, MIT) tetszését. Ennek hatására megalapította a Free Software Foundation (FSF) nevű szervezetet Cambridge-ben. Az FSF célja szabadon terjeszthető szoftverek fejlesztése.

A kifejezésben a free szabadságot jelent, nem ingyenességet. Hite szerint a szoftvernek és a hozzátartozó dokumentációnak, forráskódnak szabadon hozzáférhetőnek és terjeszthetőnek kell lennie. Ennek elősegítésére megalkotta (némi segítséggel) a General Public License-t (GPL, magyarul általános felhasználói licensz).

A GPL három fő irányelve:

- Mindenkinek, aki GPL-es szoftvert kap, megvan a joga, hogy ingyenesen továbbadja a forráskódját. (Leszámítva a terjesztési költségeket.)
- Minden szoftver, amely GPL-es szoftverből származik, szintén GPL-es kell, hogy legyen.
- A GPL-es szoftver birtokosának megvan a joga, hogy szoftvereit olyan feltételekkel terjessze, amelyek nem állnak konfliktusban a GPL-lel.

A GPL egyik jellemzője, hogy nem nyilatkozik az árról. Vagyis a GPL-es szoftvertermékek tetszőleges áron adhatóak el a vevőnek. Egyetlen kikötés, hogy a forráskód ingyenesen jár a szoftverhez. A vevő ezután azonban szabadon terjesztheti a programot és a forráskódját. Az internet elterjedésével ez azt eredményezte, hogy a GPL-es szoftvertermékek ára alacsony (sok esetben ingyenesek), de lehetőség nyílik ugyanakkor arra is, hogy a termékhez kapcsolódó szolgáltatásokat, támogatást térítés ellenében nyújtsák.

Az FSF által támogatott legfőbb mozgalom a GNU's Not Unix (röviden GNU) projekt, amelynek célja, hogy szabadon terjeszthető Unix-szerű operációs rendszert hozzon létre.

A Linux története 1991-re nyúlik vissza. Linus Torvalds, a helsinki egyetem diákja ekkor kezdett bele a projektbe. Eredetileg az Andrew S. Tanenbaum által tanulmányi célokra készített Minix operációs rendszerét használta a gépén. A Minix az operációs rendszerek működését, felépítését volt hivatott bemutatni, ezért egyszerűnek, könnyen értelmezhetőnek kellett maradnia. Emiatt nem tudta kielégíteni Linus igényeit, aki ezért belevágott egy saját, Unix-szerű operációs rendszer fejlesztésébe.

Eredetileg a Linux-kernelt gyenge licensszel látta el, amely csak annyi korlátozást tartalmazott, hogy a Linux-kernel nem használható fel üzleti célokra. Azonban ezt rövidesen GPL-re cserélte. A GPL feltételei lehetővé tették más fejlesztőknek is, hogy csatlakozzanak a projekthez, és segítsék munkáját.

A GNU C-library projektje lehetővé tette alkalmazások fejlesztését is a rendszerre. Gyorsan követték ezeket a gcc, bash, Emacs programok portolt változatai. Így az 1992-es év elején már aránylag könnyen lehetett installálni a Linux 0.95-et a legtöbb Intel-gépen.

A Linux-projekt már a kezdetektől szorosan összefonódott a GNU-projekttel. A GNU-projekt forráskódjai fontosak voltak a Linux-közösség számára rendszerük felépítéséhez. A rendszer további jelentős részletei a kaliforniai Berkley egyetem nyílt Unix-forráskódjaiból, illetve az X konzorciumtól származnak.

A különböző Unix-fajták egységesített programozói felületének létrehozására született meg a C nyelven definiált POSIX (Portable Operating System Interface) a szabvány (letölthető a http://www.pasc.org/ címről) – a szó végén az X a Unix világra utal –, amit a Linux implementálásakor is messzemenőkig figyelembe vettek[1].

Ahogy a Linux fejlődött, egyre többen foglalkoztak az installációt és a használatot megkönnyítő disztribúciók készítésével. A Slackware volt az első olyan csomag, amelyet már komolyabb háttértudás nélkül is lehetett installálni és használni. Megszületése nagyban elősegítette a Linux terjedését, népszerűségének növekedését.

A RedHat disztribúció (http://www.redhat.com) a társaihoz képest viszonylag későn született, de nagyon népszerű lett. Fejlesztőinek célja egy stabil, biztonságos, könnyen installálható és használható csomag készítése volt. A RedHat cég termékeihez támogatást, tanfolyamokat, könyveket is nyújt. Ennek következtében a RedHat disztribúció az üzleti Linux-felhasználás egyik vezetőjévé nőtte ki magát.

Ezt követően számos új disztribúció született, így manapság széles listából választhatjuk ki a szükségleteinknek leginkább megfelelőt. Mindegyiknek megvannak az előnyei, hátrányai, hívői. Azonban mindegyik ugyanazt a Linux-kernelt és ugyanazokat a fejlesztői alapkönyvtárakat tartalmazza.

A Linux-disztribúciók általában a fejlesztői könyvtárakat, fordítókat, értelmezőket, parancsértelmezőket, alkalmazásokat, segédprogramokat, konfigurációs eszközöket és még sok más komponenst is tartalmaznak a Linux-kernel mellett.

1.3 A Linux skálázhatósága

A Linux skálázhatóságának felső határa lényegében nincsen. Kialakíthatóak belőle sokszerveres elosztott rendszerek is.

A skálázhatóság alsó határa a BasicLinux disztribúció ad jó példát. Ez a rendszer két változatban elérhető az alábbi igényekkel:

- 2.8 MB disk – 12 MB RAM
- 20 MB disk – 3 MB RAM

A saját kezű mini Linux rendszer építésével kapcsolatban elmondható, hogy komolyabb módosítások nélkül egy 8 MB háttértár és 16 MB RAM igényű rendszert könnyen összeállíthatunk.

A Linux rendszer mérete az itt említetteknél tovább is csökkenthető, azonban ez már a komoly módosításokat is igényel.

[1] A POSIX1 szabványt tartalmazó az ANSI/IEEE 1003.1-1990 jelű szabványt (ez definiálja a C nyelvű felületet) egy időben a 1003.2-1003.13 szabványokkal egészítették ki, amelyeket POSIX.x szabványoknak neveznek. A POSIX szabványok implementációjukban eltérhetnek, ugyanis a szabvány csak felületet határoz meg, nem implementációt.

2 A Linux és a valósidő

A Linuxból léteznek valós idejű módosított változatok. Ezek a normál rendszertől lényegében csak kernel módosításokban térnek el. Bár tartalmazhatnak segédeszközöket a valósidejű szoftverek teszteléséhez.

Számos valósidejű Linux disztribúció létezik: RT Linux, Linux/RT, LynxOS, BlueCat, stb.

A normál Linux kernel mostanában a soft real-time-tól egyre inkább tart a hard real-time felé. Az eredeti 2.4-es kernelnél csak soft real-time-ról beszélhetünk, azonban a kernel módosításával (patch-ek) a rendszerek átalakíthatóak hard real-time rendszerré.

Ugyanakkor a kernel fejlődésével és a valósidejű módosítások integrálásával az aktuális (>2.6.23) verzió már közel valósidejű, így ott ez az átalakítás nem feltétlenül szükséges. A kritikus pontok a következőek:

- A folyamatok ütemezése.
- A kernel folyamatok megszakíthatósága.
- A kernelen belüli szinkronizációk.
- A megszakítások kezelése.

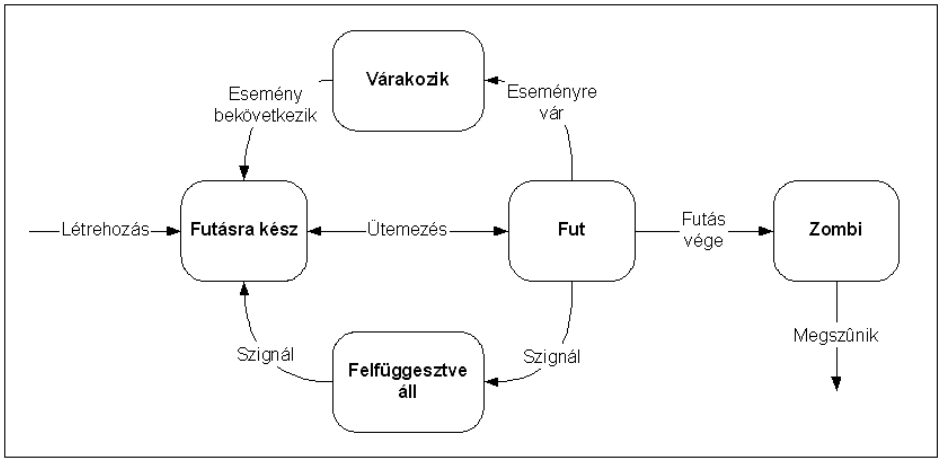
2.1 A folyamatok állapotai

A futás során a processzek különböző állapotokba juthatnak. Az állapot függhet a processz aktuális teendőjétől, és külső hatásoktól. A processz aktuális állapotát a processzhez rendelt leíró struktúra állapotleíró változója tárolja.

Linux alatt egy processz lehetséges állapotai a következők:

- A processz éppen fut az egyik processzoron. (Az állapotváltozó értéke: RUNNING)
- A processz futásra kész, de másik foglalja a processzort, ezért várakozik a listában. (Az állapotváltozó értéke ilyenkor is: RUNNING)
- A processz egy erőforrásra vagy eseményre várakozik. Ilyenkor attól függően kap értéket az állapotváltozó, hogy a várakozást megszakíthatja egy jelzés (INTERRUPTABLE), vagy sem (UNINTERUPTABLE).
- A processz felfüggesztve áll, általában a SIGSTOP jelzés következtében. Ez az állapot hibajavítás esetén jellemző. (Az állapotváltozó értéke: STOPPED)
- A processz kész az elmúlásra (már nem élő, de még nem halott: zombi), de valami oknál fogva még mindig foglalja a leíró adat struktúráját. (Az állapotváltozó értéke: ZOMBIE).

Az állapotok kapcsolatait a következő ábra szemlélteti:



1. ábra A processz állapotainak diagramja

A processzeket a fenti állapotdiagram szerint a **processzütemező** (scheduler) kezeli.

2.2 Ütemezés

A Linux-rendszerek párhuzamosan több folyamatot futtathatnak. Ritka az eset, amikor a processzorok számát nem haladja meg a folyamatok száma. Ezért, hogy az egyes processzek többé-kevésbé párhuzamosan fussanak, az operációs rendszernek folyamatosan váltogatnia kell, hogy melyik processz kapja meg a processzort. Az ütemező feladata, hogy szabályozza a processzoridő kiosztását az egyes folyamatok számára.

Ahhoz, hogy az ütemezés felhasználói szempontból jó legyen, szükségesek a következők: a felhasználó az egyes folyamatok prioritását szabályozhassa, és az ő szemszögéből nézve a processzek párhuzamosan fussanak. A rossz ütemezési algoritmus az operációs rendszert döcögőssé, lassúvá teheti, az egyes processzek túlzott hatással lehetnek egymásra a processzoridő elosztása során. Ezért a Linux fejlesztői nagy figyelmet fordítottak az ütemező kialakítására.

A processzek váltogatásával kapcsolatban két kérdés merül fel.

- Mikor cseréljünk le egy processzt?
- Ha már eldöntöttük, hogy lecserélünk egy processzt, melyik legyen az a processz, amelyik a következő lépésben megkaphatja a CPU-t?

Ezekre a kérdésekre adunk választ a továbbiakban.

2.2.1 Klasszikus ütemezés

A klasszikus Linux ütemezés időosztásos ütemezést használ. Ez azt jelenti, hogy az ütemező az időt kis szeletekre osztja (time-slice), és ezekben az időszeletekben valamilyen kiválasztó

algoritmus alapján adja meg az egyes processzeknek a futás lehetőségét. Azonban az egyes processzeknek nem kell kihasználnia az egész időszeletet, akár le is mondhatnak a processzorról, ha éppen valami rendszer eseményre kell várakozniuk. Ezek a várakozások a rendszerhívásokon belül vannak, amikor a processz éppen kernel módban fut.[2]

A Linuxban a nem futó processzeknek nincs előjoga az éppen futó processzekkel szemben, azaz a nem futó folyamatok nem szakíthatják meg a futó folyamatokat annak érdekében, hogy átvegyék a processzort, vagyis a folyamatok csak önkéntes alapon adhatják át egymásnak a futás lehetőségét.

A fenti két alapelv gondoskodik arról, hogy megfelelő időpontban bekövetkezzen a váltás. A másik feladat a következő processz kiválasztása a futásra készek közül. Ezt a választási algoritmust a kernelen belül a `schedule()` függvény implementálja.

A klasszikus ütemezés három ütemezési fajtát (policy) támogat: a szokásos Unix ütemező metódust, és két, valósidejű (real-time) processzek ütemezésére szolgáló algoritmust. A valósidejű folyamatok a szokásos értelmezés szerint azt jelentenék, hogy az operációs rendszer garantálja, hogy az adott processz azonnal megkapja a processzort, amikor szüksége van rá, ezáltal a külső eseményekre azonnal reagálhat. Ezt hívjuk „hard real-time”-nak. Azonban a Linux csak egy ún. „soft real-time” metódust támogat, amely a valósidejű folyamatokat a kiválasztás során előre veszi, ezáltal a lehetőség szerinti legkisebb késleltetéssel juttatja processzorhoz.[3]

A Linux prioritásos ütemező algoritmust használ a választáshoz. A normál folyamatok két prioritásértékkel rendelkeznek: statikus és dinamikus prioritás. A valósidejű folyamatok ehhez hozzáadnak még egy prioritásértéket, a valósidejű prioritást. A prioritásértékek egy-szerű egész számok, amelyeknek segítségével a kiválasztó algoritmus súlyozza az egyes processzeket.

A statikus prioritás

A névben szereplő statikus szó jelentése, hogy értéke nem változik az idő függvényében, csak a felhasználó módosíthatja. (A processzinformációk közt a neve: nice)

A dinamikus prioritás

A dinamikus prioritás lényegében egy számláló, értéke a processz futásának függvényében változik. Segítségével az ütemező nyilvántartja, hogy a processz mennyi futási időt használt el a neki kiutaltból. Ha például egy adott processz sokáig nem jutott hozzá a CPU-hoz, akkor a dinamikus prioritás értéke magas lesz. (A processzinformációk közt a neve: counter)

A valósidejű prioritás

Jelzi, hogy a processz valós idejű, és ezáltal minden normál folyamatot háttérbe szorít a választáskor. További funkciója, hogy a valós idejű folyamatok közötti prioritásviszonyt megmutassa. (A processzinformációk közt a neve: rt_priority).

---

[2] Minden processz részben felhasználói üzemmódban, részben kernel üzemmódban fut. A felhasználói üzemmódban jóval kevesebb lehetősége van a processznek, mint kernel üzemmódban, ezért bizonyos erőforrások, rendszer szolgáltatások igénybevételéhez át kell kapcsolnia. Ilyenkor az átkapcsoláshoz a processz egy rendszerhívást hajt végre (a Linux manuál 2. szekciója tartalmazza ezeket, pl. *read()*). Ezen a ponton a kernel futtatja a processz további részét.

[3] Léteznek azonban olyan megoldások, amelyek elérhetővé teszik a hard real-time ütemezést is Linux alatt. A következő címeken találhatunk rá példákat: `http://www.rtai.org/`, `http://www.rtlinux.org/`

Normál processzek esetén Linux ütemezési algoritmusa az időt ciklusokra (epoch) bontja. A ciklus elején minden processz kap egy meghatározott mennyiségű időegységet, vagyis a számlálóját egy, a statikus prioritásából meghatározott értékre állítjuk. Amikor egy processz fut, ezeket az időegységeket használja el, vagyis a számlálója csökken. Ha elfogyott az időegysége, akkor már csak a következő ciklusban futhat legközelebb. A ciklusnak akkor van vége, amikor minden RUNNING állapotú folyamatnak elfogyott az időegysége (tehát a várakozó processzek nem számítanak). Ilyenkor új ciklus kezdődik. Ezzel a módszerrel elérhető, hogy minden processz kapjon több-kevesebb futásidőt, vagyis egyiket se éheztessük ki.

A FIFO valósidejű ütemezés esetén csak a valósidejű prioritásnak van szerepe az ütemezésben. A legnagyobb prioritású processzek közül a sorban a legelső futhat egészen addig, amíg át nem adja másnak a futás lehetőségét (várakoznia kell, vagy véget ért), vagy nagyobb prioritású processz nem igényli a processzort.

A round-robin (körbenforgó) valósidejű ütemezési algoritmus a FIFO továbbfejlesztett változata. Működése hasonlít a FIFO ütemezéshez, azonban egy processz csak addig futhat, amíg a neki kiutalt időegysége el nem fogy, vagyis a számlálójának az értéke el nem éri a 0-t. Ilyenkor a sor végére kerül, és a rendszer megint kiutal számára időegységeket. Ezáltal lehetővé teszi a CPU igazságos elosztását az azonos valósidejű prioritással rendelkező folyamatok között.

2.2.2 Az O(1) ütemezés

Az O (1) ütemezés a 2.6-os kernellel együtt jelent meg. A Java virtuális gépek miatt volt rá szükség a sok párhuzamosan futó szál miatt. Mivel a korábbi ütemező algoritmus futási ideje egyenes arányban állt a processzek/szálak számával, ezért nagy mennyiségű szál esetén a hatékonysága jelentősen csökkent. Az O(1) ütemező erre a problémára jelentett megoldást, mivel az ütemezési algoritmus futás ideje konstans.

2.2.3 Completely Fair Scheduler (CFS)

A 2.6.23-as kerneltől kezdődően ez a normál kernel alapértelmezett ütemező algoritmusa. A klasszikus ütemezési algoritmushoz képest az egyik legjelentősebb eltérése, hogy nem használ időszeleteket, vagy ha úgy vesszük, ezek dinamikusan változnak. Csak egy ún. granulity érték van, amely nanosec felbontású és konfigurálható. Ezt nevezhetjük az ütemező felbontásának.

Ahogy az algoritmus neve is jelzi, a fő működési elve a CPU igazságos elosztása a folyamatok között. Vagyis valójában azt könyveli, hogy ha egy taszk fut, akkor mennyi idő jár ezek után a többieknek. Majd igazságosan átadja annak a folyamatnak, amelyiknek leginkább jogos az igénye.

A várakozási listát egy teljesen kiegyensúlyozott, piros-fekete bináris fában tárolja. Ennek a leggyakrabban használt bal oldali elemét gyorsító tárazza is a hatékonyság érdekében.

Az ütemezés ennél az algoritmusnál O(log n) komplexitású, azonban tényleges értéke kisebb, mint a korábbi ütemezőnél volt. A taszkok váltása konstans idő, viszont a leváltott taszk beillesztése a bináris fában O(log n) időt igényel.

Az ütemező további újítása a korábbiakkal szemben, hogy az ütemezési stratégiák implementációja moduláris felépítésű lett. Így szabadon bővíthető további algoritmusokkal. Ugyanakkor tartalmaz egy új stratégiát is, amelynek neve „batch”. Ez lehetővé teszi olyan alkalmazások futtatását, amelyet hosszabban érdemes futtatni a processzoron. Ezek többnyire szerver alkalmazások.

A CFS ütemezőben is a valósidejű folyamatok megelőzik a normál ütemezésűeket. Ugyanakkor mivel nincsenek időszeletek, ezért a rendszer szinte azonnal válthat, ha egy valósidejű folyamat futásra késszé válik. A taszkváltás is a gyorsító tárazás miatt rövid idő alatt végrehajtódik.

2.3 Megszakításkezelés

A Linuxban két megszakítás kezelő típust különböztetünk meg. A „gyors” megszakítás kezelő letiltja a megszakításokat, ezért gyorsra kell elkészítenünk. A „lassú” megszakítás kezelő nem tiltja le a megszakításokat, így lehet kicsit lassabb is az implementációja. Ugyanakkor bizonyos helyzetekben szinkronizálási problémákat vethetnek fel.

Azért, hogy a megszakítás kezelő rutinok gyorsan lefuthassanak, a Linux fejlesztők egy olyan megoldást alkalmaznak, amely a feladatot szétválasztja két részre:

- „Top half” – Ez a tényleges megszakítás kezelő rutin. A feladata, hogy az adatokat letárolja gyorsan utólagos feldolgozáshoz, majd bejegyezze a másik fél futtatására vonatkozó kérelmét.
- „Bottom half” – Ez a rész ténylegesen már nem a megszakítás kezelőben fut le, hanem utána kicsivel. A komolyabb, időigényesebb számításokat itt végezzük el. Technikailag két eszköz közül választhatunk az implementáció során:
    - Tasklet
    - Workqueue

Mostanában egy új megoldás is kezd elterjedni, amelynél a megszakítás kezelő még kisebb, mert nincs szükség a taszkok regisztrálására. Helyettük a tényleges feldolgozás kernel szálakkal történik. Ebben az esetben a feldolgozás még hosszabb lehet, mivel az ütemező képes a kernel szálakat megszakítani és ütemezni. További nyereség, hogy a kernel szálakból akár felhasználói alkalmazásokkal is kommunikálhatunk, vagy elindíthatjuk őket.

2.4 A valósidejűség felé mutató további elemek

Az eddig felsorolt lépésekkel a normál kernel elég közel került a valósidejűséghez, azonban további lépéseket is tettek a fejlesztők:

- A kernel szálak a 2.6-os kernelben megszakíthatóak, így egy hosszabb művelet sem tudja lefogni a processzort.
- A szinkronizálásokat optimalizálták a kernelben.
- A rendszer nagyfelbontású időzítőt használ.

2.5 Különbség a normál és az RT kernel között

Jelenleg már aránylag kevés eltérés van a normál és a valósidejű kernel között. A valósidejű kernel az alábbi plusz funkciókkal rendelkezik:

- Tartalmaz egy direkt hozzáférési lehetőséget a fizikai memóriához.
- Tartalmaz néhány memória menedzsmentbeli módosítást.
- A gyenge pontok felderítésére pedig tartalmaz egy késleltetés monitorozó eszközt.

2.6 Az RT-Preempt patch

Az RT-Preempt patch a standard Linux kernelt valósidejű kernellé változtatja. Ezt a következő módosításokkal éri el:

- A kernel szinkronizációs eszközök megszakíthatóvá válnak, mivel rtmutex-el újraimplementálták őket.
- A spinlock és rwlock által védett kritikus szakaszok megszakíthatóak. A nem megszakítható szakaszokhoz raw_spinlock-ot használhatunk.
- Prioritás öröklést támogatják a kernel szinkronizációs eszközök. Ezzel megakadályozzák a prioritás inverzió jelenségét.
- A megszakítás kezelők megszakítható kernel szálakká válnak.
- Az eredeti Linux timer API-t átkonvertálja. A nagy felbontású időzítő infrastruktúra az eredeti POSIX időzítő függvényeket sokkal pontosabbá teszi.

3 A Linux rendszer felépítése

A Linux rendszer az alábbi fő építőkövekből áll össze:

- Kernel (modulokkal)
- Fejlesztői könyvtárak
- Segédprogramok
- Shell(ek)
- Shell scriptek
- Alacsony szintű grafika (framebuffer)
  - directfb, SDL
- X Window (Xorg, KDrive)
  - widget library: Xlib, Qt, Gtk++
  - Desktop Environment: KDE, GNOME

3.1 A Linux betöltése

Egy operációs rendszer betöltődése, elindulása első pillantásra mindig is kicsit mágikus dolognak tűnik. Általában ha egy, a háttértárolón lévő programot szeretnénk betölteni, lefuttatni, akkor beírjuk a nevét a parancsértelmezőbe, vagy rákattintunk az ikonjára, és az operációs rendszer elindítja nekünk. Azonban hogyan történik mindez, amikor a gép indulásakor magát az operációs rendszert szeretnénk betölteni, elindítani?

A Linux-kernel betöltését és elindítását az ún. kernelbetöltő végzi el. Ilyen program a LILO (The Linux Loader), a Grub, az uBoot, a LOADLIN, és még sorolhatnánk. Ahhoz, hogy ez a program betöltődjön, szükséges egy kis hardveres segítség. Általában a gép csak-olvasható memóriájában van egy kis program (az x86 architektúra esetén ennek neve BIOS, ARM esetén Bootstrap), amely megtalálja, betölti és futtatja ezt a kernelbetöltőt. Vagyis összegezve, egy kicsi beégetett program lefut, és elindít egy valamivel nagyobb betöltő programot. Ez a betöltő program pedig elindít egy még nagyobb programot, nevezetesen az operációs rendszer kernel programját.

A kernel manapság általában tömörített formában van a lemezeken, és képes önmagát kitömöríteni. Így az első lépés a kernel kitömörítése, majd a processz a kitömörített kernel kezdő címére ugrik. Ezt követi a hardver inicializálása (memóriakezelő, megszakítástáblák stb.), majd az első C függvény (`start_kernel()`) meghívása. Ez a függvény, amely egyben a 0-s azonosítójú processz, inicializálja a kernel alrészeit. Az inicializáció végén a 0-s processz elindít egy kernelszálat (neve init), majd egy üresjárati ciklusba (idle loop) kezd, és a továbbiakban a 0-s processz szerepe már elhanyagolható.

Az init kernelszál vagy processznek a processz azonosítója az 1. Ez a rendszer első igazi processze. Elvégez még néhány beállítást (elindítja, a fájlrendszert szinkronizáló és

lapcsere-kezelő folyamatokat, felélesztí a rendszer konzolt, felcsatolja (`mount`) a root fájlrendszert), majd lefuttatja a rendszerinicializáló programot (nem keverendő a korábban említett processzel). Ez a program az adott disztribúciótól függően a következők valamelyike: `/etc/init`, `/bin/init`, `/sbin/init`. Az init program az `/etc/inittab` konfigurációs állomány segítségével új processzeket hoz létre, és ezek további új processzeket. Például a getty processz létrehozza a login processzt, amikor a felhasználó bejelentkezik. Ezek a processzek mind az init kernelszál leszármazottai.

3.2 Initrd és initramfs

Az initrd és az újabb initramfs olyan állományok, amelyek egy összetömörített állományrendszert tartalmaznak. Ezt a kernel induláskor kitömöríti egy ramdiskre, majd képes állományrendszerként használni.

Felhasználási területei:

- Kis memóriában futó Linux rendszerek esetén. Ezek a kis rendszerek alkalmasak egy normál rendszer feltelepítésére vagy megjavítására, partíciók klónozására, archiválására.
- A „Live” rendszerek olyan rendszerek, amelyek CD-ről, vagy pendriveról indulnak és futnak. Egy kis ramdiskes részből és az eszközön tárolt állományrendszerből állnak. Lehetővé teszik, hogy a rendszer telepítése nélkül is legyen Linuxunk.
- A telepített rendszerek is használják arra, hogy a bootoláshoz szükséges kernel modulokat betölthessék.

3.3 SysV init

A kernel a kezdeti feladatainak végrehajtása és a root állományrendszer csatolása után elindítja az init programot, amely gondoskodik a további rendszer inicializálási lépések végrehajtásáról. Ilyen feladatok a hálózati eszközök beállítása, a konzol konfigurálása, az USB eszközök inicializálása, az állományrendszerek ellenőrzése és felcsatolása, stb. Majd ezután következik a különböző szolgáltatások elindítása, a virtuális konzolok, és esetleg az X felület létrehozása. Ezeket a feladatokat az `/etc/inittab` állomány alapján végzi el.

Az init program több futás szintet definiál. A futási szint egy konfigurációs összeállítás, amelyben összeállíthatjuk, hogy a rendszer egyes állapotaiban milyen szolgáltatások fussanak és melyek nem. A használható futási szintek a következők:

Szint	Leírás
0	Fenntartott. A rendszer leállítása.
1	Fenntartott. Egy felhasználós mód. Minden hálózati és felhasználói szolgáltatás leáll. A felhasználók nem léphetnek be, csak a rendszergazda kap egy shellt a konzolon.

Szint	Leírás
2	Több felhasználós mód hálózat nélkül.
3	Több felhasználós mód.
4	Nem használt.
5	Több felhasználós mód, X felülettel.
6	Fenntartott. A rendszer újraindítása.
7-9	A Linux rendszerek esetén ezek is működő futás szintek, azonban ezt a dokumentumok általában nem említik. Ennek oka, hogy a tradicionális Unix rendszerekben csak a 0-6 futás szintek léteznek.

A 0, 1, 6 szintek a rendszer működéséhez vannak fenntartva. A többi futási szint lényegében az adminisztrátor által szabadon konfigurálható. Leírásuk inkább ajánlás, amely alapján célszerű beállítani a rendszerünket, illetve amit a disztribúció készítői is szem előtt tartottak az alapértelmezett konfiguráció összeállításakor.

A futásszintekhez tartozó scriptek, illetve a scriptekre mutató szimbolikus linkek a futásszintnek megfelelő `/etc/rc.d/rcN.d/` könyvtárban találhatóak, ahol az N a futásszint száma. Emellett a rendszer indulásakor lefut még az alábbi két script:

```
/etc/rc.d/rc.sysinit
/etc/rc.d/rc.local
```

3.4 Linux állományrendszerek

A Linux számos fájlrendszer típust támogat. Ezek egy része használható a rendszerállományok tárolására is, másik része pedig az egyéb operációs rendszerekkel való együttműködés érdekében került a rendszerbe (FAT, VFAT, NTFS).

A következőkben végigtekintjük - a teljesség igénye nélkül - a Linux által támogatott állományrendszereket.

3.4.1 Minix

A Linux születésének hajnalán a fejlesztés a Minix operációs rendszer alatt történt. Egyszerűbb volt megosztani a háttértárolót a két rendszer között, mint egy új állományrendszert kifejleszteni. Ezért legelőször ennek a fájlrendszernek a támogatása került bele a Linuxba. Azonban a minix állományrendszer túl sok korlátozást tartalmazott, ezért a fejlesztők idővel szükségesnek látták a lecserélését, ezért fejlesztették ki az extended filesystems nevű állományrendszert.

3.4.2 Extended filesystems (ext, ext2, ext3, ext4)

Az ext, ext2, ext3, ext4 állományrendszerek elsősorban a Linux számára lettek kifejlesztve. Az ext a minix egy továbbfejlesztése. Azonban még így is számos hiányossággal rendelkezett, és teljesítmény problémái is voltak, ezért szükségessé vált egy új rendszer kifejlesztése, amely az ext2 nevet kapta.

Az ext3 állományrendszer az ext2 továbbfejlesztése. Lényegében az ext2 kibővítése a journaling mechanizmussal. Az ext4 állományrendszer az ext3 továbbfejlesztése és egyben a legújabb verzió.

3.4.3 Journaling állományrendszerek

A komolyabb állományrendszerek adatainak frissítése, összetettségükből adódóan több különálló írás műveletből áll. Ahhoz, hogy az állományrendszer konzisztens maradjon, az összes írás műveletnek végre kell hajtódnia. Ha eközben egy áramkimaradás miatt a rendszer hirtelen összeomlik, akkor a fájlrendszer egy köztes állapotban maradhat. Ilyenkor a következő bootolásnál a rendszer végig ellenőrzi az állományrendszert és megpróbálja ezeket a hibákat kijavítani. Azonban ez egy nagyobb partíción nagyon sokáig eltarthat, és nem is jár mindig sikerrel, ha a helyreállításhoz nincsenek meg a szükséges információk.

A journaling állományrendszereken ennek a problémának a megoldására születtek. Az állomány rendszer tartalmaz egy elkerített területet, melyet journal-nak, vagy log-nak neveznek. Mielőtt az állományrendszerhez hozzányúlna a rendszer, hogy az egyik állapotból eljusson egy újabb állapotba, előtte a műveleteket feljegyzi ebbe az elkerített területbe. Majd ezek után végrehajtja a folyamatot. Ha eközben a rendszer összeomlana, akkor a journal tartalmazza a szükséges információkat, amely alapján a művelet visszajátszható. Természetesen, ha a journal terület írása közben omlana össze a rendszer, akkor az állományrendszer konzisztens állapotban van, ezért nem igényel beavatkozást.

Természetesen a journaling fájlrendszer hátránya, hogy a több írásműveletből adódóan valamennyivel lassabb, mint a könyvelés nélküli társai.

A Linux alatt a következő journaling állományrendszerek vannak:

- JFS: Az IBM által kifejlesztett, elsősorban az IBM Enterprise szerverekben használatos fájlrendszer.
- XFS: A Silicon Graphics által kifejlesztett és használt fájlrendszer.
- ext3: Az ext2 fájlrendszer kiegészítése.
- ReiserFS
- jffs
- jffs2

3.4.3.1 Journaling Flash File System (JFFS)

A NOR flash memóriák rendelkeznek néhány tulajdonsággal, amelyet az állományrendszer tervezőinek figyelembe kell vennie:

- Törölni kell őket írás előtt, amely lassú művelet.
- A törlést nagyobb egységekben lehet végezni.
- A törlések-írások száma véges, így az ilyen jellegű terhelést el kell osztani a tárterületen.

Ezen jellemzők miatt egy hagyományos állományrendszer, pláne egy journaling állományrendszer használata nem célszerű az ilyen eszközökön. A hagyományos állományrendszerek az állományokat, illetve a hozzájuk kapcsolódó információkat egy helyen tárolják és frissítéskor a módosításokat ugyanott végzik. A journaling állományrendszer esetén a journal állomány pedig különösen sokszor módosul.

A felsorolt problémák megoldására és a beágyazott rendszereknél szükséges journaling megtartására új koncepcióra van szükség. Ezt nyújtja a JFFS állományrendszer. Ez egy naplózás jellegű állományrendszer, amely az állományok új verzióit cirkulárisan írja fel az eszközre. Mintha egy naplóállományt írnánk a módosításokról.

Így egyben a journaling mechanizmust is megvalósítja, mivel az utolsó ponttól vissza is játszható az utolsó félbehagyott művelet.

Az üres helyet a napló eleje és vége közötti terület jelenti. Ha ez túlságosan lecsökken, akkor egy szemétgyűjtő algoritmus összerendezi a még aktuális állomány darabokat. Lényegében defragmentál.

Azonban hátrányai is vannak az állományrendszernek:

- Igényli, hogy a memóriában nyílván tartsa a rendszer az állományok aktuális verziójának pozícióit.
- Induláskor az előbb említett információkat össze is kell gyűjteni, így a felcsatolása lassabb, mint a hagyományos állományrendszereké.
- A cirkuláris működés miatt a nem módosított állományokat is időnként át kell helyezni, ezáltal felesleges törléseket és írásokat is végez, csökkentve az élettartamot. (Természetesen ez arányos az írás műveletek számával.)

3.4.3.2 JFFS2

A JFFS2 a korábbi JFFS állományrendszer tovább fejlesztett változata. Az általa hozott plusz funkciók:

- A NAND flash eszközök támogatása.
- A hard link-ek kezelése.
- Tömörítés támogatása.
- Jobb teljesítmény

A JFFS2 nem egy cirkuláris napló állományként kezeli az eszközt, hanem blokkokra osztja, amelyekből mindig egyet tölt fel, amíg az meg nem telik, és át nem vált egy másikra. A működés alapelve ugyanaz, mint korábban, lényegében csak a tárterület kezelési algoritmusa

módosult. A módosításokat nem a korábbi pozícióra írja vissza, hanem az éppen aktuális blokkba. Ez alapján a blokkoknak három állapota lehet:

- Tiszta blokk: Csak aktuális információkat tartalmaz.
- Piszkos blokk: Tartalmaz aktuális és már elavult állományokat is.
- Szabad blokk: Nem tartalmaz adatot.

A háttérben futó szemétgyűjtő algoritmus a piszkos blokkok információit összepakolja egy tiszta és szabad blokkokat alkotva.

A működéséből látszik, hogy a statikus állományokra előnyösebb, mint a korábbi JFFS állományrendszer, mivel ilyenkor nem végez fölösleges törléseket.

Hátrányai:

- Továbbra is lassú a felcsatolás, mivel ilyenkor végig kell vizsgálni az állományrendszert az állomány információk összegyűjtéséhez.
- És természetesen a memória igény is megmaradt, illetve kicsit tovább nőtt.

3.4.3.3 UBIFS

Az UBIFS (Unsorted Block Image File System) a JFFS2 továbbfejlesztése. Az UBIFS egyik lényegi eltérése a JFFS2-höz képest, hogy gyorsítótárat használ az írásnál. Emellett egy pesszimista algoritmust használ a szabad terület megbecsüléséhez. A szabad terület pontos felmérése idő és számítás igényes, ezért a rendszer nem pontos értéket ad vissza. Ez a JFFS2 esetében is így van, azonban az UBIFS sosem mond több szabad területet, mint amennyi ténylegesen lehet.

Az UBIFS-nél gyorsult a felcsatolás, a nagy állományok elérése és az írás művelet. Emellett jobb a visszaállíthatósága és a tápfeszültség hiba tűrése, mint a JFFS2-nek.

3.4.4 Speciális állományrendszer típusok

A Linux rendszer rendelkezik virtuális állományrendszer típusokkal is, amelyek speciális funkciókat látnak el.

Proc

A proc virtuális állományrendszer a kernel belső állapotáról szolgáltat információkat, illetve lehetőséget nyújt a beállítások menet közbeni módosítására. A `/proc` könyvtárban láthatjuk.

Sysfs

A sysfs virtuális állományrendszer a rendszerben található eszközöket teszi elérhetővé fa struktúrába szervezve. Megnézhetjük az eszköz állapotát, illetve kontrollálhatjuk az eszközvezérlő beállításait. A `/sys` könyvtárban találhatjuk.

Tmpfs

A tmpfs egy ramdisk eszközt valósít meg. Vagyis tényleges tárolás nem történik, mivel az eszköz újraindítása után az állományok nem elérhetőek. Azonban gyakran használjuk

átmeneti állományok tárolására, vagy a folyamatok közötti kommunikációra, mivel gyors és nem terheli a háttértárként használt eszközt.

3.4.5 Más operációs rendszerek támogatása

A Linux számos, más operációs rendszerek által használt állományrendszer típust is támogat:

- DOS FAT 12/16/32 (msdos néven), illetve a VFAT bővítés, amelyet az MS Windows rendszerek vezettek be a hosszú állománynevek támogatására.
- NTFS, amelyet az MS Windows NT vonal használ. (Használata csak olvasható módban ajánlatos, mivel a Microsoft nem közli a specifikációját, és bármikor önhatalmúlag megváltoztathatja. A fejlesztők csak az állományrendszer elemzésével készítették a támogatást.)
- HFS, amelyet a Machintos rendszerek használnak.
- HPFS, amelyet az OS2 operációs rendszer használ.
- AFFS, az Amiga gépek támogatására.
- Qnx4, a QNX operációs rendszer támogatására.

3.4.6 A CD és a DVD állományrendszere

A CD-ROM állományrendszerének neve Linux alatt az iso9660, mivel ez a szabvány definiálja a működését.

A DVD-k esetén az udf a használatos.

3.4.7 Hálózati állományrendszerek

A Linux három hálózati megosztásokra használatos állományrendszer támogat. A Unix rendszereknél használatos NFS-t (Network Filesystem), az MS Windows család által bevezetett SMB protokoll, valamint a Novell Netware szerverek által használt ncpfs-t.

3.5 Állomány típusok

A Linux fájlrendszerében a következő állománytípusokat különböztetjük meg:

Egyszerű állomány

Az egyszerű állományok azok, amelyekre először gondolnánk az állomány szó hallatán, vagyis bájtok sorozata, adatok, program, stb. A Linux minden állományt egyszerűen byte-ok véletlenszerűen címezhető szekvenciájának tekinti. A rendszer eltakarja az állományt tároló fizikai eszköz tulajdonságait. Az állomány mérete az általa tartalmazott byte-ok számával azonos.

Könyvtár

A könyvtár speciális állomány, amely olyan információkat tartalmaz, amire a rendszernek szüksége van az állományok elérésére. Pl. az ext(2,3,4) állományrendszerek esetén tartalmazza az állomány nevét és a hozzá tartozó i-node indexet.

Eszközök

A legtöbb fizikai eszköz a Linux rendszerben, mint állomány jelenik meg[4]. Ez azt jelenti, hogy ezekre az eszközökre állomány nevekkel hivatkozhatunk, és ugyanúgy kezelhetjük őket, mint a közönséges állományokat. Írhatjuk, olvashatjuk. Azonban ezeket a műveleteket természetesen az adott eszközre értelmezhető műveletre képezi le a kernel. (Például egy hangkártya eszköz olvasása a hang bedigitalizálását jelenti, az írása pedig a hang állomány lejátszását.)

Két eszköztípust különböztetünk meg: **blokk** és **karakter** típusú eszközök. A **blokkos eszköz** állomány egy olyan hardver eszközt reprezentál, amelyet nem olvashatunk bájtonként, csak bájtok blokkjaiként. A blokkos eszközök leggyakrabban háttértárak, és fájlrendszert tartalmaznak. A **karakteres eszköz** bájtonként olvasható. A modemek, terminálok, printerek, hangkártyák, és az egér mind-mind karakteres eszköz.

Tradicionálisan az eszköz leíró speciális állományok a `/dev` könyvtárban találhatóak.

Szimbolikus link

A szimbolikus link (röviden symlink) egy speciális állomány, amely egy másik állomány elérési információit tartalmazza. Amikor megnyitjuk, a rendszer érzékeli, hogy szimbolikus link, kiolvassa az értékét, és megnyitja a hivatkozott állományt. Ezt a műveletet a szimbolikus link követésének hívjuk.

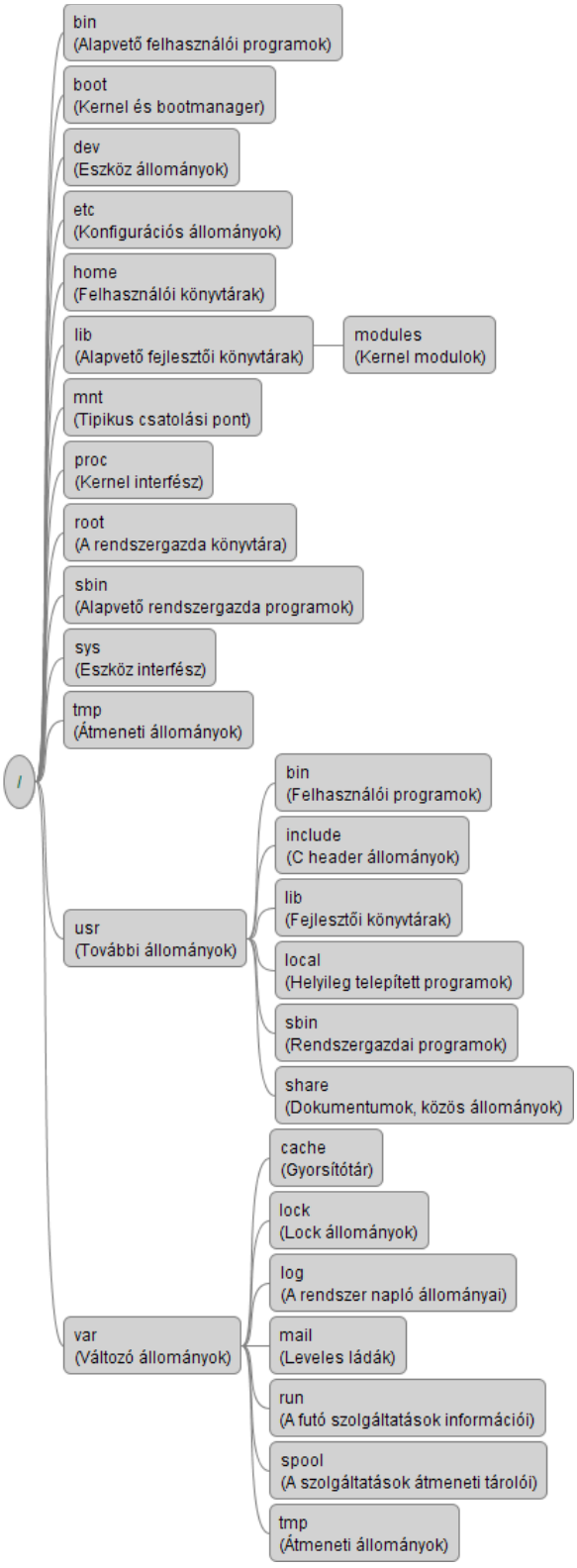
Csővezeték (pipe) és socket állományok

A csővezeték és a socket állományok a Linux két egyszerű IPC (Inter Process Communication, folyamatok közötti kommunikáció) mechanizmusát alkotják. Speciális állományok, amelyek egy kommunikációs csatornát írnak le a memóriában.

3.6 Könyvtár struktúra

A Linux hierarchikus állományrendszerében való könnyebb eligazodás érdekében az alábbi ábra egy rövid áttekintést ad a legfontosabb könyvtárakról és azok funkcióiról.

[4] Linux esetén kivételt képeznek ez alól a hálózati interfészek, amelyeket nem láthatunk az eszköz állományok között.



2. ábra A könyvtár hierarchia

3.7 Linkek

Sok operációs rendszerben egy az egyes összerendelés van az állományok és az állománynevek között. (Minden állománynak egy neve van, és minden állománynév egy állományt jelöl.) A Linux a nagyobb rugalmasság érdekében nem ezt a koncepciót követi. Egy állománynak lehet több neve is, akár különböző könyvtárakban. Megkülönböztetünk ún. hard linket, amikor különböző névvel hivatkozunk egy állományra és szimbolikus link-et, amikor egy mutató állományt hozunk létre.

4 A Linux rendszer kezelése

4.1 Belépés

A szokásos bejelentkezés nem tér el a más rendszereknél is használt egyszerű felhasználói név - jelszó alapú azonosítástól. Azonban általában a Linux rendszerek egyszerre két felületen is nyújtanak a felhasználó felé. Egyrészt a szöveges, virtuális konzolokat, másrészt egy grafikus felületet (X Window).[5] Mivel a Linux nagymértékben konfigurálható, ezért az egyes rendszereknél a beállítások eltérhetnek. Mi a disztribúciók által általánosan használtakat tekintjük most át.

4.1.1 Virtuális konzolok

A terminálok használata a korai időkre vezethető vissza. Eredetileg a Unix nagy számítógépek tipikus kezelői felületei a soros terminálok voltak, amelyek egyszerű karakter bevitelt, és a válasz karakterek megjelenítését tették lehetővé. Ez az idők folyamán fejlődött, és a terminál karakterek segítségével barátságosabb felületet nyújtottak. A személyi számítógépek, illetve a hálózatok fejlődése révén a soros terminálok használata háttérbe szorult, azonban egyszerűségük, uniformitásuk, elterjedtségük révén virtuális terminálokként tovább élnek.

A Linux rendszerek a konzolon általában egyszerre 6 ilyen virtuális terminált szimulálnak. Ezek között az ALT + F1-F6[6] gombokkal váltogathatunk. (Grafikus módról történő átváltás esetén CTRL + ALT + F1-F6.) Mindegyik egy teljes értékű szöveges terminál. Külön-külön mindegyiken bejelentkezhetünk valamely felhasználóként, és párhuzamosan dolgozhatunk.

4.1.2 Grafikus felület

A rendszer indulásakor manapság elsőként a grafikus felülettel találjuk magunkat szembe. Azonban a virtuális konzolok használata után is visszakapcsolhatunk ide az ALT + F7[7] segítségével. Belépés után egy grafikus ablakos felülettel találjuk magunkat szembe (más modern operációs rendszerekhez hasonlóan). Linuxnál ez a felület változatos lehet az aktuális ablakozó rendszertől függően. Jelenleg a legelterjedtebbek a KDE és a Gnome.

Ezeknél a felületeknél, a más rendszereknél megszokott mechanizmusok ugyancsak használhatóak. Kisebb konfigurációk után a működést teljesen az általunk megszokott rendszerhez igazíthatjuk.

A grafikus felületeken is elérhetjük a terminál emulációt. Ezt az xterm, kterm, stb. programok teszik lehetővé. Így a terminál parancsokat itt is használhatjuk.

---

[5] A Linux rendszerek támogatják a hagyományos, soros portos terminálokat is, azonban a beágyazott rendszerek kivételével ezek használata manapság már nem tipikus.

[6] Fedora 10+ esetén F2-F7

[7] Fedora 10+ esetén F1

4.2 A jelszó beállítása

Unix rendszereknél a jelszót a **passwd** parancs segítségével módosíthatjuk. A felhasználók csak a saját jelszavukat módosíthatják. Ehhez meg kell adniuk a régi jelszót (hogy egy magára hagyott terminálnál mások ne írhassák át), illetve kétszer az új jelszót.[8]

Rendszergazdaként bármelyik felhasználó jelszavát módosíthatjuk. Ilyenkor a **passwd <név>** szintaxis a használatos.

4.3 A legfontosabb parancs: man

A Linux (és Unix) elektronikus segítség rendszerét a manuál oldalak adják. Ezeket az oldalakat a **man** paranccsal érhetjük el. Minden, a rendszerben található parancsról, segédprogramról, de még a függvényekről is az alábbi módon kérhetünk segítséget.

```
man <parancs>
```

Magáról a man parancsról is kaphatunk információt az alábbi módon:

```
man man
```

A leírások 8 fejezetre (kötetre) oszlanak:

1. Általánosan használható programok
2. Rendszerhívások
3. Felhasználói szintű könyvtári függvények
4. Speciális állományok
5. Állományformátumok
6. Játékok
7. Különféle leírások
8. A rendszer karbantartásához szükséges információk

Több fejezetben is lehetnek azonos nevű leírások. Pl. lehet azonos nevű program és rendszerfüggvény is. Ilyenkor az egyértelműség érdekében a fejezet számát is meg kell adni:

```
man <fejezet> <parancs>
```

Kulcsszókra is kereshetünk:

```
man –k <kulcsszó>
```

4.4 A parancs formátum

A Linux parancsok a következő formátumot használják:

[8] A rendszer a beállításától függően tesztelheti a megadott új jelszót és visszautasíthatja, ha túl "gyengének" találja. Linux alatt a legelterjedtebb a *cracklib* könyvtár alkalmazása, amely különböző szempontokból teszteli a megadott jelszót, illetve összehasonlítja a szó adatbázisával.

```
parancs opció(k) argumentum(ok)
```

A sor elején áll a parancs, vagy programnév. Ezt helyközzel elválasztva követik az opciók. Az opciók "-" jellel kezdődnek. Több opció is állhat egy "-" jel mögött, vagy megadhatjuk őket külön "-"-el helyközökkel elválasztva. Időnként használatos a dupla kötőjel használata is. Az opciókat követik az argumentumok, szintén helyközzel elválasztva. Az argumentumok száma és sorrendje a parancstól függ.

Például:

```
ls -l /tmp
```

4.5 Gyakran használt parancsok

Linux alatt a leggyakrabban a következő, egyszerű parancsokkal találkozhatunk:

Fájlrendszer parancsok:

- cd – Könyvtárváltás.
- cmp – Bináris állományok összehasonlítása.
- cp – Állományok másolása.
- dd – Állományok konvertálása és másolása.
- diff – Szöveges állományok összehasonlítása.
- du – Lemezhasználat összegzése.
- find – Állományok keresése.
- head – Állományok elejének megtekintése.
- ls – Könyvtár tartalmának listázása.
- mkdir – Új könyvtár létrehozása.
- mv – Állományok átnevezése vagy mozgatása.
- pwd – A munkakönyvtár elérési útjának kiírása.
- rm – Állományok törlése.
- rmdir – Könyvtárak törlése.
- tail – Állományok végének megtekintése.

Szöveges állományok megtekintése:

- cat – Állományok kiírása, összefűzése, esetleg létrehozása.
- less – Állományok megjelenítése. Görgethetjük, lapozhatjuk az állományt, és kereshetünk is benne.
- more – Egyszerű program, amely a szöveges állományt oldalakra tördelve jeleníti meg.

4.6 Jogosultságok

A Unix rendszerek alatt minden felhasználó rendelkezik egy egyedi azonosító számmal (**uid** = user identifier), és egy vagy több csoport azonosító számmal (**gid** = group identifier).

A felhasználó azonosítója és az alapértelmezett csoport azonosítója az `/etc/passwd` állományban van eltárolva. A többi csoportazonosító az `/etc/group` állomány bejegyzései alapján rendelődik hozzá a felhasználóhoz.

Létezik egy kitüntetett felhasználó, a rendszergazda (super-user, a login neve "**root**"). A rendszergazdára nem vonatkoznak a felhasználókra beállított hozzáférési jogok, a rendszer még az állományok tulajdonosánál is bővebb lehetőségeket engedélyez számára. A rendszer az alapján ismeri fel, hogy az uid-je 0.

A Linux és más Unix operációs rendszerek a felhasználói és csoport azonosítókat használják a védelmi korlátozások meghatározására. A rendszer minden állománya egy tulajdonos és egy csoport azonosítóval rendelkezik:

- A tulajdonos azonosító az állományt létrehozó felhasználó azonosítója. (A rendszergazda ezt a paramétert állíthatja a chown paranccsal.)
- A csoportazonosító az állomány létrehozásakor a felhasználó alapértelmezett csoportja. Ez később a chgrp paranccsal állítható.

Ezek az azonosítók szabályozzák, hogy az egyes folyamatok hozzáférhetnek-e az adott állományhoz és elvégezhetik-e a kérdéses műveleteket, vagy sem. A folyamat (felhasználó) és az állomány között 3 féle reláció állhat fenn:

- Azon felhasználó, akinek az azonosítója megegyezik az állomány azonosítójával, az a tulajdonos (owner).
- Akinek a felhasználói azonosítója más, de valamelyik csoport azonosítója megegyezik a bejegyzés azonosítójával az a csoporttárs (group).
- Akiknek pedig a felhasználói és csoport azonosítója egyaránt eltér a bejegyzés azonosítóitól, azok a többiek (others).

Minden egyes relációra 3 féle jogosultsági engedélyt állíthatunk be: olvasás, írás, és végrehajtás (könyvtárak esetén ez utóbbi keresési jogot jelent). Nézzük, hogyan is néz ez ki! Az ls –l paranccsal részletes listát kérhetünk az adott könyvtár tartalmáról:

```
drwxr-xr-x    2 root     root         4096 Sep 23 23:24 dir1
-rw-r--r--    1 root     root            5 Sep 23 23:24 file1
```

A lista sorainak első mezője tartalmazza az adott állományra/könyvtárra érvényes jogokat a következők szerint:

```
drwxrwxrwx
```

Ahol az első karakter az állomány típusa:

Jelzés	Típus
b	a bejegyzés egy blokk-orientált speciális állomány
c	a bejegyzés egy karakter-orientált speciális állomány
d	a bejegyzés egy katalógus
l	a bejegyzés egy szimbolikus link
-	a bejegyzés egy egyszerű állomány

A következő 9 karakter hármas csoportokra osztható. Az első három az owner, a második a group, a harmadik az other jogait jelenti.

A hármas csoporton belül a három karakter sorrendben a következő lehet: **"r", "w"** és "**x**". ezek bármelyike helyén "-" is állhat. Jelentésük pedig rendre az olvasási (read), írási (write), végrehajtási (execution) jogot vagy "-' esetén annak hiányát jelenti.

- Az olvasási jogkör szabályozza, hogy van-e joga az adott felhasználónak az állomány tartalmának megtekintésére, illetve a katalógus tartalmának ki listázására.
- Ha egy felhasználó rendelkezik írási joggal egy bejegyzésre, akkor bővítheti, kicserélheti, megváltoztathatja az állomány tartalmát, illetve létrehozhat, törölhet bejegyzést katalógus esetén.
- A végrehajtási jog (amely csak állományra vonatkozhat, ha katalógusra vonatkozik, akkor keresési jognak nevezzük) azt jelzi, hogy az adott felhasználó futtathatja-e az állományt. Maga a végrehajthatósági jog önmagában még nem jelenti azt, hogy az állomány ténylegesen futtatható kódot tartalmaz.
- A katalógusokra vonatkozó keresési jog azt jelzi, hogy van-e jogunk a katalógus bejegyzéseinek eléréséhez, használatához.

Megjegyzés: Szimbolikus linkek esetén a jogok nem értelmezettek. Az eredeti állomány jogai az irányadóak.

A tulajdonos lehetőségeit ismertető csoportban "x" helyén "s" áll, ha az állomány setuid módban van. Hasonlóképpen a csoporttárs lehetőségeit ismertető csoportban "x" helyén "s" áll, ha a bejegyzés setgid módú.

Az ún. setuid mód lehetővé teszi, hogy a rendszer a programot futtató felhasználót saját uid-je helyett a programállomány tulajdonosának uid-jével azonosítsa (természetesen csak a program futásának időtartamára). Így a felhasználó által futtatott program mindazt végrehajthatja, amit a programállomány tulajdonosa végrehajthat. A setgid mód teljesen azonos a setuid móddal: a rendszer a felhasználó valódi gid je helyett a programállomány tulajdonosának gid-jét (gid-jeit) használja a hozzáférési jogok megállapításánál a program futásának időtartama alatt.

A jogokat a `chmod` paranccsal módosíthatjuk. A parancsot két féle képen használhatjuk: egyrészt a jogok egzakt, oktális számokkal való megadásával, másreszt szimbolikus jelekkel. A `chmod` parancs használatának általános formája:

```
chmod <mód> <fájlnév>
```

Azonban a rendszernek már az állományok létrehozásakor be kell állítania valamilyen engedélyeket. Lehetne szigorú és elvehetne minden jogot, de lehetne engedékeny is, és megadhatná őket. Azonban a legjobb megoldás, ha a felhasználó megadhatja. Ezt meg is tehetjük az `umask` segítségével. Ez egy egész szám, amely azt tartalmazza, hogy mely biteket töröljön a rendszer az engedélyek közül. A legegyszerűbb, ha oktális számokkal adjuk meg. Ilyenkor lényegében a `chmod` parancsnál használt szám inverzét kell megadni:

```
umask <mód inverze>
```

4.6.1.1 A jogok megváltoztatása (szimbolikus jelekkel)

A mód szimbolikusan a következő módon adható meg:

```
[kinek] op jogok [op jogok] ...
```

Ahol a “kinek” a következő betűk kombinációja kehet:

Jelzés	Kinek állítjuk a jogait?
u	a tulajdonosnak
g	a csoportnak
o	a többieknek
a	mindenkinek

Ha a “kinek” elmarad, akkor az mindenkit jelöl (mint az “a”), de ilyenkor az aktuális umask-ot figyelembe veszi, és azokra a bitekre, ahol az umask 1 az állítás nem hat.

Az op +, -, vagy = lehet:

Jelzés	Művelet
+	jogokat adunk hozzá az állomány jogaihoz
-	jogokat veszünk el az állomány jogaiból
=	pontosan ezt az értéket akarjuk beállítani

A jogok a következő betűk kombinációi lehetnek:

Jelzés	Jogok
r	olvasási jog

Jelzés	Jogok
w	írási jog
x	végrehajtási jog
X	végrehajtási jog, de csak akkor, ha az állomány katalógus, vagy van már másik x bitje
s	*setuid* vagy *setgid* bit
t	*sticky* bit[9]
u	a tulajdonos mezőt az eredeti módból veszi
g	csoport mezőt az eredeti módból veszi
o	többiek mezőt az eredeti módból veszi

4.6.1.2 A jogok állítása (oktális számokkal)

Az abszolút kód egy oktális szám, ami a következő kódok vagy kapcsolatával állítható elő:

Jelzés	Jogok
4000	*setuid* mód a végrehajtáskor
2000	*setgid* mód a végrehajtáskor
1000	*sticky* bit
0400	olvasás a tulajdonos által
0200	írás a tulajdonos által
0100	végrehajtás (keresés a katalógusban) a tulajdonos által
0040	olvasás a csoportnak
0020	írás a csoportnak
0010	futtatás a csoportnak

[9] A *sticky* bit állományok esetén nem értelmezett. Csak a régi Unix rendszerek használták, a Linux figyelmen kívül hagyja. Könyvtárak esetén a jelentése, hogy a benne található állományokat csak a tulajdonos vagy a *root* nevezheti át vagy törölheti. Jellemzően a /tmp könyvtár használja ezt a jog bitet. Nélküle bárki átnevezhetné vagy törölhetné az állományokat.

Jelzés	Jogok
0004	olvasás a többieknek
0002	írás a többieknek
0001	futtatás a többieknek

4.6.1.3 Alapértelmezett állomány jogok

Azok a programok, amelyek állományokat hoznak létre, a létrehozáskor beállítják a létrehozandó állomány védelmi kódját. Így történik ez akkor is, ha pl. editorral létrehozunk egy új állomány. Ilyenkor a jogokat az *umask* beállítások alapján állítja be a rendszer. Mégpedig úgy, hogy azokat a védelmi biteket (csak az alsó 9 bitet), amelyek értéke 1 az umask-ban, a létrehozandó állomány védelmi kódjából kitörli.

Például a 002 *umask* érték azt jelenti, hogy állományok estén a jog 664 lesz, könyvtárak esetén 775. (Látható a fentiekből ez az alapértelmezett értéke az umask-nak.)

Állományok esetén futásjogot nem tudunk az umask-al adni.

4.7 Állománynév-helyettesítés

A parancs argumentumában megadott "*", "?", és "[...]" jelek ún. állománynév-helyettesítő jelek. Ezek jelentése:

Jel	Jelentés
*****	Nulla vagy tetszőleges számú tetszőleges karakter.
?	Pontosan egy tetszőleges karakter.
[abc]	Az "a" vagy "b" vagy "c" karakter egyike.
[m-n]	m-n intervallumból egy karakter.

Az olyan argumentumok, amelyek állománynév-helyettesítő karaktereket tartalmaznak, állománynevekből álló egyszerű argumentumok sorozatára cserélődnek le. Például “*f”-ből azoknak az aktuális katalógusban található állományneveknek a sorozata lesz, amelyek az “f” karakterekre végződnek. A helyettesítő karakterek közül több is szerepelhet egyidejűleg ugyanabban az argumentumban. Pl: az "[a-z]*" a kisbetűvel kezdődő neveket jelenti.

Az, hogy a kifejtő metódust a shell tartalmazza, több előnnyel jár:

- A kifejtést megvalósító program csak egyszer szerepel a rendszerben (helytakarékosság).
- A programoknak a kifejtéssel nem kell foglalkozniuk.

- A kifejtő algoritmus alkalmazása így bizonyosan mindig egységes lesz.

4.8 A find parancs

A `find` parancs rekurzívan végignézi az adott könyvtárból nyíló alkönyvtárakat. Keresi bennük a kifejezésnek megfelelő állományokat, majd kiírja a talált állományok listáját, vagy feladatokat hajt végre az adott állományokon.

A parancs formája:

```
find <elérési út> <kifejezés>
```

A leggyakrabban használt eset, amikor megadott nevű állományokat keresünk a könyvtárak sűrűjében:

```
find . -name fontos
./doksik/fontos
./fontos
```

Nézzünk meg még néhány opciót, amelyekkel összetettebb keresési feltételeket adhatunk meg:

Paraméter	Leírás
`-name fájl`	Az állomány megfelel, ha a neve a fájl paraméterrel egyezik. A *shell* által használt állománynév helyettesítő szintaxis használható, de idézőjelek közé kell tenni, hogy ne a *shell* értelmezze.
`-type típus`	Az állomány típusát specifikálhatjuk, ahol típus az alábbiak közül az egyik karakter: b - blokk speciális állomány, c - karakter speciális állomány, d - könyvtár, f - normál állomány, l - szimbolikus link, p - *named pipe*, s - *socket*.
`-user uname`	Az állomány megfelel, ha az állomány tulajdonosa az uname nevű felhasználó.
`-nouser`	Az állomány megfelel, ha nincs tulajdonosa regisztrálva.
`-group gname`	Az állomány megfelel, ha az állomány csoportja a megadott.

Paraméter	Leírás
`-nogroup`	Az állomány megfelel, ha az állomány csoportja nincs regisztrálva.
`-perm onum`	Az állomány megfelel, ha a joga megegyeznek a megadottal.
`-perm -onum`	Az állomány megfelel, ha az onum érték minden 1-es bitje az állomány jogaiban is 1.
`-perm +onum`	Az állomány megfelel, ha az onum érték valamelyik 1-es bitje az állomány jogaiban is 1.
`-size N`	Az állomány megfelel, ha az állomány N blokk hosszúságú (512 byte van egy blokkban).
`-empty`	Az állomány vagy könyvtár megfelel, ha üres.
`-amin N`	Az állomány megfelel, ha utoljára N perce fértek hozzá.
`-atime N`	Az állomány megfelel, ha utoljára N napja fértek hozzá.
`-cmin N`	Az állomány megfelel, ha utoljára N perce módosult a státusza.
`-ctime N`	Az állomány megfelel, ha utoljára N napja módosult a státusza.
`-mmin N`	Az állomány megfelel, ha utoljára N perce módosították.
`-mtime N`	Az állomány megfelel, ha utoljára N napja módosították.
`-exec parancs`	Az állomány megfelel, ha a végrehajtandó program 0 visszatérési értékkel tér vissza. A parancs végét egy pontosvessző ";" jelzi, amit a *shell* miatt idézőjelek közé kell zárni. Az állománynevet "{}" jelekkel helyettesíthetjük. Használhatjuk úgy is, hogy a talált állományokra a megadott parancsot végrehajtsa.
`-ok parancs`	Megegyezik az -exec paranccsal, de a parancs végrehajtása előtt jóváhagyást kér a felhasználótól.
`-print`	Hatására az aktuális állomány teljes nevét kiírja a képernyőre.
`-xdev`	Hatására a find program nem megy át másik kötetre (csatolt eszközre).

Az opciókat a következő jelekkel kombinálhatjuk: **`!`** (nem)**,** **`-a`** (és)**,** **`-o`** (vagy). Továbbá zárójelekkel csoportosíthatjuk, azonban a zárójeleket idézőjelekbe kell tenni, hogy a shell ne értelmezze.

Az **N** értékek helyén a számok mellett használhatjuk a + és - jeleket:

- n - pontosan az adott szám
- +n - több (nappal ezelőtt) mint n
- -n - kevesebb (nappal ezelőtt) mint n

A **`find`** parancs **`-exec`** paraméterével programokat is lefuttathatunk az egyes állományokra. Ez lehet a keresést szolgáló állománytesztelő program, de ha egy **`-a`** paraméterrel illesztjük a keresés végére, akkor keresés eredményein végrehajtott művelet is. A **`find`** a **{}** jelek helyére a talált állományok nevét helyettesíti be és így hajtja végre a parancsot minden állományra. A parancsot "**;**" jellel kell zárnunk. Hogy a shell ne értelmezze, idézőjelbe kell tennünk, vagy "\" jelet kell tennünk elé.

Nézzünk egy példát, ahol a program futtatása az állomány tesztelését szolgálja:

```
find -exec grep -q "alma" {} ";" -a -print
```

És egy másik példát, ahol az eredmény állományokon végzünk el műveletet:

```
find -name "*fontos" -a -exec ls -l {} ";"
```

Ha nem vagyunk benne biztosak, hogy minden állományra szeretnénk végrehajtani a parancsot, akkor hasznos ötlet az **`-ok`** paraméter használata az **`-exec`** helyett.

4.9 A grep parancs

A **`grep`** (Global Regular Expression Print) parancs a megadott mintát keresi az állományokban, és az illeszkedő sorokat kiírja a standard kimenetre. A minta lehet egy egyszerű szöveg, azonban a **`grep`** képes a reguláris kifejezésnek nevezett, speciális jelentésű karakterekkel kiegészülő szövegminta használatára is.

```
grep [opciók] minta [állomány ...]
```

Ha egynél több állománynevet adtunk meg, akkor a megtalált sorok elé kiírja azt is, hogy melyik állományban vannak.

Az alábbi opciókkal befolyásolhatjuk a **`grep`** működését, a mintaillesztési szabályt (**-i, -w**), vagy az eredményként megjelenő információkat (**-l, -n, -v**).

Paraméter	Leírás
-i	A kis- és nagybetűket azonosnak tekinti.
-l	Csak az állományok nevét írja ki.
-n	Minden kiírt sor előtt a sor sorszáma is szerepel.

Paraméter	Leírás
-v	Inverz működés: a nem illeszkedő sorokat írja ki.
-w	A mintát, mint szót keresi.

Létezik két módosult változata az alap **grep** utasításnak:

- Az egrep mintaillesztője a kiterjesztett reguláris kifejezéseket is ki tudja értékelni. (Egyenértékű a grep -E utasítással.)
- Az fgrep mintaillesztője csak egyszerű szöveget használ. (Egyenértékű a grep -F utasítással.)

4.9.1 Egyszerű reguláris kifejezések

A szövegek feldolgozása során gyakori feladat a mintaillesztés. A minta megadására az egyszerűbb használhatóság érdekében közös szintaktikát használnak a programok. Ezt hívjuk reguláris kifejezésnek.

Egy szövegmintában általában minden karakter önmagát jelenti, azaz a mintában az adott helyen elő kell fordulnia, de egyes karaktereknek speciális a jelentése, így adhatunk meg bonyolultabb mintát.

Karakter	Jelentés
\	Az utána írt speciális karakter is karakterként kerül a mintába, kivéve: újsor-jel, számjegy, "(" vagy ")".
^	Sor eleje.
$	Sor vége.
.	Egy darab valamilyen karakter.
c*	A c karakter 0 vagy többször. (Használható a .* kombinációban, ha bármilyen karaktert szeretnénk megadni.)
[abc]	Valamelyik karakter a listából, hasonlóan, mint a *shell*-nél. Használható a [a-c] módon is.
[^abc]	Bármely karakter, ami nem "a" vagy "b" vagy "c".

Ezen karakterek egy részét a shell is értelmezi. Hogy ez biztosan ne okozzon problémát a minta stringet egyszeres idézőjelek (' ') közé célszerű tenni.

A reguláris kifejezések meta karakterei hasonlóak a shell-éhez, azonban akad néhány eltérés. Továbbá a reguláris kifejezések jóval több speciális karaktert tartalmaznak. Ez a lista a fenti

táblázatban megadottnál bővebb, azonban itt csak a legáltalánosabbak, a **grep** parancs által is értelmezhetőeket soroltuk fel. Az **egrep** esetében lehetőségünk van a többi kifejezést is használni, azonban ezekre most nem térünk ki.

Nézzünk néhány példát:

`grep "minta$" szoveg`	Azokat a sorokat írja ki a "szoveg" állományból, amelyeknek a végén a "minta" szó szerepel.
`cat szoveg | grep '[pt]i.os'`	Olyan sorokra illeszkedik, amelyekben szerepel egy szövegrészlet, ahol az első betű "p" vagy "t", majd "i", egy meghatározatlan karakter és "os". Például piros vagy tilos.
`ls -l | grep "^d"`	A long listából azokat a sorokat írja ki, amelyek "d" betűvel kezdődnek, vagyis könyvtárakat.

4.10 A parancsértelmező (shell)

A UNIX rendszerek és így a Linux egyik érdekessége, hogy a felhasználói felületet megvalósító parancsértelmező (shell) egy közönséges felhasználói program. Ez a parancsértelmező nem integráns része az operációs rendszernek, és nem élvez speciális megkülönböztetéseket. Így bárki lecserélheti a sajátját egy másik programra.

A rendszerbe való bejelentkezéskor a rendszer a login-név (user azonosító) és a jelszó alapján ellenőrzi a felhasználót, majd a **password** állomány hozzá tartozó sorának megfelelő mezőjében megadott programot elindítja. Ez rendszerint valamelyik shell (sh, csh, ksh, tcsh stb.), de lehet valami más program is. Mindegyik felhasználónak joga van ezt a programot lecserélni (**chsh** parancs).

Az egyes shell változatok között a különböző kényelmi szolgáltatásokban, valamint a programozói felületükben van eltérés. Ugyanis a UNIX shell-ek nem csupán parancsértelmezők, hanem programozási nyelvek is egyben.

A következőkben áttekintjük a **bash** (Bourne again shell) különböző funkcióit.

4.10.1 Parancssor értelmezés

A UNIX shell lényegében egy parancsértelmező program, amely beolvassa a felhasználó által begépelt sorokat, és azt más programok végrehajtását előíró kérésekként értelmezi. Egy parancssor a legegyszerűbb esetben egy parancsból és a hozzá tartozó, szóközökkel elválasztott, paraméterekből áll:

```
parancs arg1 arg2 ... argn
```

A parancsértelmező különálló karaktersorozatokká bontja fel a parancs nevét és argumentumait. Ezután egy beállítható keresési út (PATH) szerint a különböző

katalógusokban megkeresi a parancs nevű állományt. Ha sikerült megtalálni, akkor azt végrehajtja. A parancs végrehajtásának befejezését a prompt kiírásával jelzi.

A parancssorban szereplő állománynév után következő argumentumok a következő négy kategória valamelyikébe esnek:

- Egyszerű karakterlánc.
- Állománynév, amit "<",">" vágy ">>" előz meg.
- Egy karakterlánc, amely állománynév helyettesítő karaktert is tartalmaz.
- "` `" jelek közé zárt újabb parancs, ami ún. parancshelyettesítést eredményez.

Az egyszerű argumentumokat a program string tömbjeként kapja meg. A *shell* állítja elő ezt a tömböt, ezáltal könnyebb és egységes a paraméter kezelés implementálása a programokban.

4.10.2 A shell beépített parancsai

A *shell*-ek tartalmaznak néhány beépített parancsot. Ezek többnyire a *shell* működésével szorosan összefüggő parancsok, amelyeket nem érdemes más módon implementálni.

Azonban léteznek olyan *shell*-ek, amelyek számos parancs megvalósítását integrálják magukba. Teszik ezt azért, hogy a rendszerhibák esetén is működhessenek, és ezáltal lehetőséget nyújtsanak az adminisztrátornak a nehéz helyzetből való kilábalásra. Ilyen shell program a `sash` (stand-alone shell). Ez egy statikusan linkelt program, amely tartalmazza a legszükségesebb programok egyszerűsített változatait. Így akkor is működőképes, amikor az alap programkönyvtárak meghibásodása miatt a rendszer lényegében használhatatlan.

Az ilyen *shell*-ek telepítése feltétlenül javasolt, hogy a problémás helyzeteket egyszerűbben megoldhassuk.

4.10.3 Állománynév-helyettesítés

A parancs argumentumában megadott "*", "?", és "[...]" jelek ún. állománynév-helyettesítő jelek. Ezek jelentése:

Jel	Jelentés
*	Nulla vagy tetszőleges számú tetszőleges karakter.
?	Pontosan egy tetszőleges karakter.
[abc]	Az "a" vagy "b" vagy "c" karakter egyike.
[m-n]	m-n intervallumból egy karakter.

Az olyan argumentumok, amelyek állománynév-helyettesítő karaktereket tartalmaznak, állománynevekből álló egyszerű argumentumok sorozatára cserélődnek le. Például “*f”-ből azoknak az aktuális katalógusban található állományneveknek a sorozata lesz, amelyek az “f”

karakterekre végződnek. A helyettesítő karakterek közül több is szerepelhet egyidejűleg ugyanabban az argumentumban. Pl: az "[a-z]*" a kisbetűvel kezdődő neveket jelenti.

Az, hogy a kifejtő metódust a shell tartalmazza, több előnnyel jár:

- A kifejtést megvalósító program csak egyszer szerepel a rendszerben (helytakarékosság).
- A programoknak a kifejtéssel nem kell foglalkozniuk.
- A kifejtő algoritmus alkalmazása így bizonyosan mindig egységes lesz.

4.10.4 Standard input/output átirányítás

A shell által végrehajtott programok eleve három megnyitott állománnyal indulnak. Ezek a 0, 1 és 2 állományleírókhoz vannak hozzárendelve:

- 0 – standard input
- 1 – standard output
- 2 – error output

Ezek alapértelmezésben a felhasználó terminájához vannak rendelve, de átirányíthatjuk őket. A “>” vagy “>>” jellel a standard kimenetet irányíthatjuk át. A parancs végrehajtása alatt az 1-es állományleíró a “>” jel után megadott nevű állományra mutat. Például az

```
ls > kimenet
```

parancs létrehozza a “kimenet” nevű állományt és a listát abba írja. Ha a “>>” jelet használjuk

```
ls >> kimenet
```

akkor az ls parancs kimenetét a “kimenet” állomány végéhez fűzi.

A hiba kimenetet a “2>” jelöléssel irányíthatjuk át.

Hasonlóan a standard input is átirányítható. Ezt a “<” jellel tehetjük meg. A “< bemenet” jelölés azt jelenti, hogy a standard input a “bemenet” nevű állományból jöjjön.

4.10.5 Csővezeték

A Linuxban gyakran előfordul, hogy az egyik program kimenetét szeretnénk használni egy másik program bemeneteként. Például ha egy program kimenetét szeretnénk megszűrni, rendezni, tördelni. Ezt megtehetjük az előző fejezetben ismertetett módon átmeneti állományok létrehozásával. Azonban a Linux tartalmaz egy sokkal hatékonyabb eszközt is, amelyet csővezetéknek (pipe) nevezünk. A csővezeték egy olyan speciális állomány, ami egy FIFO-t (First In First Out) valósít meg. Az érdekessége abban áll, hogy ez a FIFO állományként kezelhető: Két állományleíróval hivatkozhatunk rá, egyiken írhatunk bele, a másikon pedig a beírási sorrendben kiolvashatjuk a beírt adatokat.

A csővezeték-szervezés lehetőségeinek előnyei:

- Nincs szükség ideiglenes állományokra, amit később úgy is letörölnénk.
- Mivel a folyamatok "párhuzamosan" futnak, a FIFO-nak nem kell nagynak lenni.

- A pipe használata gyorsabb, mivel az a legtöbb memóriában keletkező állományként jelenik meg (egy FIFO csupán 8 KB).
- Az eredmény a cső végén már azelőtt megjelenhet, mielőtt az első program az összes bemenetét feldolgozná.

A csővezetéket a shell számára a "|" jellel adhatjuk meg:

```
ls | grep "minta" | sort | more
```

Ebben a példában az **ls** parancs kimenetéből a **grep** program kiszűri azokat, amelyekben szerepel a "minta" string, majd az eredményt rendezi, és oldalakra tördelve megjeleníti.

4.10.6 Parancshelyettesítés

Igen hasznos és érdekes szolgáltatása a shell-eknek az ún. parancshelyettesítés. Ez lehetővé teszi, hogy az egyik program kimenetét a másik program paraméter listájaként használjuk. Ilyenkor a megfelelő parancsargumentumot "`" (visszafelé dőlő aposztróf) párba kell zárni, vagy zárójelekbe írni és elé egy „$" jelet tenni. Például a

```
du `cat parameters.txt`
```

vagy

```
du $(cat parameters.txt)
```

parancs esetén a du parancs megmondja azon fájlok méretét, amelyek a "parameters.txt" állományban fel vannak sorolva.

4.10.7 Parancssorozatok

Ha a parancsokat egymás után pontosvesszővel elválasztva írjuk, akkor úgynevezett parancssorozathoz jutunk. Így például az

```
date; ls
```

parancs először kiírja az aktuális dátumot, majd kilistázza a pillanatnyi katalógust.

Parancssorozatot nem csak pontosvessző segítségével alkothatunk, hanem a "||" és a "&&" jelekkel is. Ekkor egy feltételesen végrehajtódó sorozathoz jutunk. A "||" és a "&&" jelek jelentése azonos a C nyelvben megszokottal. Vagyis a

```
parancs1 || parancs2

parancs3 && parancs4
```

sorozatból a parancs2 csak akkor fog végrehajtódni, ha a parancs1 hamis megállási státusszal állt meg. A parancs4 pedig csak akkor fog végrehajtódni, ha a parancs3 igaz megállási státusszal állt meg.

4.10.8 Szinkron és aszinkron folyamatok

Az eddigi példákban az egymás utáni parancsok egymást követve szinkron módon hajtódtak végre. Lehetőség van azonban aszinkron végrehajtásra is. Ha egy parancsot az "&" jel követ, akkor a shell nem várja meg a parancs befejeződését, hanem a prompt jel kiadásával újabb

parancsra várva azonnal visszajelentkezik, miközben a kiadott parancs aszinkron módon a shell-től leválasztva fut tovább. Pl.:

```
find / -name 'core' > eredmeny &
```

Ebben az esetben a *shell* azonnal visszatér és új parancsra vár. A fájlkeresés a háttérben fut és az eredményt az "eredmeny" állományba írja. (A háttérben futó folyamatok kimenetét érdemes fájlba irányítani, hogy ne keveredjen össze más parancsok kimenetével.)

A parancs futatásakor a rendszer kiír egy ún. process ID-t amellyel hivatkozhatunk később a háttérben futó folyamatra.

4.10.9 Csoportosítás

A parancsokat csoportosíthatjuk is zárójelekkel.

```
( parancs1 ; parancs2 )
```

Például:

```
(date; ls) > lista &
```

Ez a parancs kiírja a dátumot és az állományok listáját a "lista" állományba, és mindezt a háttérben teszi.

4.11 A Bash shell további funkciói

A következőkben röviden áttekintjük a Bourne *shell* további lehetőségeit, programozását.

4.11.1 Változók kezelése

A *shell*-ek az eddig megismert funkciókon túl változók kezelésére is alkalmasak. A változókat egyszerű string típusú tárolóként kezeli.

Érték adás *shell* változónak:

Az érték beállítása a "=" jellel történik. (Előtte és mögötte nem lehet üres hely.)

```
változó=érték
```

Például:

```
kutya=ugat
```

Hivatkozás a változóra:

Ha már beállítottuk a változót, akkor a "**$**" jellel a változó neve előtt hivatkozhatunk az értékre.

Például:

```
kutya=ugat
echo $kutya
ugat
```

A példánkban az **echo** parancsot használtuk, amely kiírja a paraméterként kapott szöveget. Mint látható a változó helyére a *shell* behelyettesíti a változó értékét. Az **echo** parancs már az értéket kapja meg, és azt írja ki. Előfordul, hogy közvetlenül a változó után kell írnunk valamit. Ebben az esetben meg kell határoznunk a *shell* számára, hogy meddig tart a változó neve. Ezt a "{" és "}" jelekkel tehetjük meg:

```
kati=zsuzsi
echo ${kati}ka
zsuzsika
```

A változót az **unset** paranccsal törölhetjük. Ebben az esetben, ha továbbra is használjuk, akkor a *shell* egy üres szöveget helyettesít be.

A **set** parancs használatával kiírattathatjuk a beállított változók listáját.

Szükség lehet arra is, hogy a *shell* által indított programok is megkapják a környezeti változók értékét, mert a rendszerről, vagy a felhasználó preferenciájáról tudhatnak meg információkat. Ilyenkor az **export** kulcsszóval tesszük számukra elérhetővé:

```
TERM=”VT100”
export TERM
```

vagy összevonva:

```
export TERM=”VT100”
```

A programok számára kiexportált környezeti változókat az **env** paranccsal listázhatjuk ki.

4.11.2 Parancsállományok

A legtöbb felhasználó azt hiszi, hogy a shell csak egy interaktív parancsértelmező, pedig a parancsértelmezés valójában egy programnyelv része. Mivel a parancsvégrehajtásnak az interaktív és a programozott módját egyaránt a shell-nek kell végrehajtania, ezért furcsa nyelvvé vált.

A Linux rendszerben gyakran találkozhatunk parancsállományokkal, más néven shell scriptekkel. Egyszerűbb feladatokat gyakran gyorsabban elvégezhetünk velük, mintha programot írnánk. A rendszer adminisztrátorok különösen sokszor használják őket egyes funkciók elvégzésére.

A legegyszerűbb változata, amikor parancsokat egymás után egy szöveg állományba írjuk. Ekkor a shell egymás után végrehajtja őket.

Az állományt kétféle képen futathatjuk:

- Meghívunk egy shell-t és paraméterként átadjuk az állományt és a futtatási paramétereit:

```
bash állomány [argumentum…]
```

- Vagy adunk futás jogot az állományra (x bit) és csak futtatjuk az állományt:

```
állomány [argumentum…]
```

4.11.3 Feltételek kiértékelése

A programok visszatérési értéke alapján elágazásokat, ciklusokat szervezhetünk. A 0 az igaz értéket, a nem 0 a hamis értéket jelenti. Azonban a test parancs segítségével lehetőségünk nyílik arra, hogy feltételeket adhassunk meg, és ez szolgáljon alapként az elágazáshoz.

A feltételeket három csoportra oszthatjuk. Az első csoport a string-eket vizsgálja:

Parancs	Jelentés
test s	igaz, ha s nem null szöveg
test -z s	igaz, ha s nulla hosszúságú szöveg
test -n s	igaz, ha s nem nulla hosszúságú szöveg
test s1 = s2	igaz, ha s1 szöveg egyezik s2 szöveggel
test s1 != s2	igaz, ha s1 szöveg nem egyezik s2 szöveggel

A feltételek második csoportja az numerikus értékeket vizsgálja:

Parancs	Jelentés
test n1 -eq n2	igaz, ha n1 szám egyenlő n2-vel (numerikus =)
test n1 -ne n2	igaz, ha n1 szám nem egyenlő n2-vel (numerikus !=)
test n1 -lt n2	igaz, ha n1 kisebb mint n2 (numerikus <)
test n1 -le n2	igaz, ha n1 kisebb vagy egyenlő mint n2 (numerikus <=)
test n1 -gt n2	igaz, ha n1 nagyobb mint n2 (numerikus >)
test n1 -ge n2	igaz, ha n1 nagyobb vagy egyenlő mint n2 (numerikus >=)

A feltételek harmadik csoportja az állományok tulajdonságainak ellenőrzésére szolgál:

Parancs	Jelentés
test -f f	igaz, ha f egy létező állomány vagy könyvtár
test -r f	igaz, ha f olvasható
test -w f	igaz, ha f írható

Parancs	Jelentés
test -d f	igaz, ha f könyvtár
test -s f	igaz, ha f létezik, és nem nulla hosszúságú
test -t fd	igaz, ha fd egy megnyitott állomány leírója, és az egyben egy terminál

Ezeket a műveleteket kombinálhatjuk is a **!** (tagadás), **-o** (vagy), **-a** (és) logikai műveletekkel, továbbá zárójelekkel csoportosíthatjuk is.

A **`test`** parancsot "**[**", "**]**" jelekkel is helyettesíthetjük az alábbi módon:

```
[ n1 -eq n2 ]
```

A szögletes zárójelek esetén figyeljünk arra, hogy a szögletes zárójelek és a feltétel között egy-egy szóköz van.

4.11.4 Vezérlési szerkezetek

A shell tartalmaz olyan eszközöket, amellyel elágazásokat és ciklusokat hozhatunk létre a parancsállományon belül (de akár egy parancssorban is a “;” jel használatával). Most ezeket vesszük sorra.

4.11.4.1 if feltétel

Az **`if`** feltétel használata esetén, ha a feltétel teljesül, vagyis a parancs visszatérési értéke 0, akkor a **`then`** ág hajtódik végre. Ha a feltétel nem teljesül, vagyis a parancs visszatérési értéke nem 0, akkor az **`else`** ág hajtódik végre.

```
if parancs
   then parancsok
   else parancsok
fi
```

```
if test $1 -eq 1
   then echo 'egy'
fi
```

```
if [ $1 -eq 1 ]
   then echo 'egy'
fi
```

Általában a **test** szögletes zárójeles változatával találkozhatunk az **if** feltételeként, de használhatunk más programot is, ha követi a fenti visszatérési érték szabályokat.

4.11.4.2 for ciklus

A **`for`** ciklus eltér a más programozási nyelvekben általában használt szintaktikától. Itt egy érték listát adunk meg, amelyen egyesével végigmegy a ciklus. A megadott változónak átadja az aktuális értéket és meghívja a **`do`** és **`done`** közötti parancsokat.

```
for i in w1 w2 ...
   do parancsok
done
```

```
for i in egy ketto harom
   do echo $i
```

```
done
```

4.11.4.3 while ciklus

A `while` ciklus működése olyan, mint más programozási nyelvek esetén: amíg a feltétel igaz végrehajtja a `do` és `done` közti parancsokat.

```
while parancs
   do parancsok
done
```

```
while test $i -le 5
   do i=`expr $i + 1`
done
```

```
while [ $i -le 5 ]
   do i=`expr $i + 1`
done
```

4.11.4.4 until ciklus

Az `until` ciklus is úgy működik, ahogy már más programozási nyelvek esetén megszokhattuk, vagyis a while-hoz képest negált a feltétel vizsgálat.

```
until parancs
   do parancsok
done
```

```
until test $i -le 5
   do i=`expr $i + 1`
done
```

```
until [ $i -le 5 ]
   do i=`expr $i + 1`
done
```

4.11.4.5 case szerkezet

A `case` szerkezet a változó értéke szerint kiválasztja az azzal megegyező minta ágát, és végrehajtja a parancsokat. A "*" minta az alapértelmezett.

```
case word in
   minta1) parancsok;;
   minta2) parancsok;;
   ...
esac
```

```
case $1 in
   1) echo egy;;
   2) echo ketto;;
   *) echo sok
esac
```

4.11.5 Shell script példa

Egy script, amely megszámolja a könyvtárban található állományokat az alábbiak szerint néz ki:

```
#!/bin/bash
#
# file szamlalo
#
```

```
n=0
for i in *
do
        if [ -f $i ]
        then
              n=`expr $n + 1`
        fi
done
```

A script-ek első sorában mindig megadjuk a futtató *shell* nevét általában elérési úttal a következő formában:

```
#!<shell>
```

A kommenteket a "#" jellel kezdődően írhatunk az állományba.

A példában az **n** változó értékét 0-ra állítottuk.

Ez után a `for` ciklus végig lépked az adott könyvtár bejegyzésein (mivel a * helyére a *shell* behelyettesíti a könyvtár állományait és könyvtárait).

Az `if` feltételében megvizsgáljuk, hogy a bejegyzés egy állomány-e. Ha igen, akkor a számláló értékét eggyel megnöveljük.

4.12 Folyamatok

A programok elindításakor létrejön egy folyamat, amely tartalmazza a program kódját, adatait és a környezeti beállításokat is (munkakönyvtár, környezeti változók, stb.). Minden folyamathoz az operációs rendszer egy egyedi **folyamatazonosítót** (**P**rocess **ID**) és egy **folyamat csoportazonosítót** (**P**rocess **Gr**ou**p ID**) rendel.

Egy folyamatot (kivéve a legelső folyamatot, az úgynevezett init-et) mindig egy másik folyamat hoz létre. A létrehozó folyamatot **szülő folyamat**nak (Parent Process), a létrehozott folyamatot **gyermek folyamat**nak (Child Process) nevezzük. Az *init* folyamatot kivéve minden folyamatnak van szülője. A gyermek a szülő másolata, csak az azonosítója különbözik. Így többek közt örökli a megnyitott állományok leíróit, az umask, ulimit értékeit, szignálok kezelésére vonatkozó beállításokat.

A szülő-gyermek reláció alapján egy fa struktúrát is felrajzolhatunk. Ezt a `pstree` paranccsal nézhetjük meg.

Amennyiben egy szülő folyamat előbb szűnik meg, mint annak gyermek folyamatai, a gyermek folyamatok **árvák**ká (orphans) válnak, és szülőjük automatikusan az *init* folyamat lesz. A folyamatok tudnak még a szülő folyamat azonosítójáról (Parent Process ID) is.

4.12.1 A folyamatok monitorozása

A folyamatokat a `ps` paranccsal listázhatjuk ki.

```
ps [opciók]
```

Alapértelmezésben a `ps` parancs csak azokat a folyamatokat listázza ki, amelyeket az adott terminálról indítottunk. A listában csak a folyamat azonosítója, a terminál, a státusz, az eltelt idő, és a parancs neve szerepel. Azonban opciókkal nagyban befolyásolhatjuk a kilistázott folyamatokat és a megjelenített információkat.

Folyamatok kiválasztása:

Paraméter	Leírás
-A	Hatására az összes folyamatról információt kaphatunk.
-N	Negálja a folyamatválasztást.
-a	A terminálokhoz kapcsolódó összes folyamat listáját kapjuk, csak a *shell*-ek nem szerepelnek benne.
a	Az adott terminál folyamatait listázza ki.
r	Csak a futó folyamatokat írja ki.
-u user	A megadott felhasználó folyamatait listázza ki.
x	Azokat a folyamatokat is kiírja, amelyek nem tartoznak terminálhoz.

A kimenet formázása:

Paraméter	Leírás
-f	un. teljes listát ad.
-j	Kiírja az ún. job control információkat is.
-l	Hosszú listaformátummal jeleníti meg.
-o formátum	Felhasználó által definiált formátum.
j	lásd -j
l	lásd -l
o formátum	lásd -o

A `top` parancs segítségével folyamatosan is monitorozhatjuk a folyamatokat. Alapértelmezésben a CPU használatának sorrendjében listázza ki az egyes processzeket, azonban ezt könnyen átrendezhetjük. Segítséget a “h” vagy “?” gombokkal kaphatunk.

4.12.2 Háttérfolyamatok

A háttér folyamat indítását már korábban is tárgyaltuk.

```
parancs &
```

Azonban ezt a módszert csak **rövid folyamatok** esetén használjuk, mert ha kilépünk a shell-ből, akkor a rendszer szignáljára leáll! A másik jellemzője, hogy a program kimenete ilyenkor a terminál.

Hosszú folyamatok esetén a `nohup` parancsot használhatjuk:

```
nohup parancs &
```

A `nohup` parancs megmondja a folyamatnak, hogy ne vegye figyelembe a 01 és 03 szignált (HANGUP és QUIT). Ezáltal a folyamat tovább fut akkor is, ha a felhasználó kilép. (A szignálokat rövidesen bővebben is áttekintjük.)

A `nohup` által indított folyamat kimenete nem kerülhet a terminálra, mert lehet, hogy a felhasználó rövidesen kijelentkezik. Ha a parancssorban nem irányítjuk át a kimenetet, akkor a `nohup` parancs automatikusan átirányítja a `nohup.out` állományba. Ha már létezik a `nohup.out` fájl, akkor hozzáfűzi a végéhez.

Mivel minden folyamatnak kell, hogy legyen szülő folyamata, ezért amikor kilépünk a rendszerből, a `nohup` által indított folyamatnak az `init` folyamat lesz a szülője.

4.12.3 Kommunikáció a folyamatokkal, megszüntetés

A `kill` parancs segítségével szignálokat küldhetünk a folyamatnak, így kommunikálhatunk velük. A `kill` parancs szintakszisa:

```
kill [-szignál] folyamatszám ...
```

Ha nem adunk meg szignált, akkor az alapértelmezés a TERM (15) szignál. Ezzel a szignállal leállíthatjuk a folyamatot, ha nincsen maszkolva. Ha ez nem sikerül, akkor keményebb eszközhöz kell nyúlnunk, ez a KILL (9) szignál. Ezt már nem maszkolhatják a folyamatok. Azonban óvatosan bánjunk vele, mert nyitott állományokat, vagy *lock*-okat hagyhat maga mögött.

A szignálokat a `kill -l` paranccsal listázhatjuk ki. Néhány fontosabb:

Szignál	Leírás
SIGHUP (1)	Akkor generálódik, ha kilépünk, miközben a folyamat még fut.
SIGINT (2)	interrupt karakter (ctrl-c)
SIGQUIT (3)	quit karakter (ctrl-\)
SIGKILL (9)	A legerősebb szignál, ezt nem hagyhatja figyelmen kívül a program. (Csak végső esetben javasolt.)

Szignál	Leírás
SIGTERM (15)	Folyamatok leállítása. Az alapértelmezett érték.

A hangup (SIGHUP (1)) szignált akkor kapja a folyamat, amikor a szülője leállt. Például ha a shell-ből indítottunk egy háttérfolyamatot, majd kilépünk.

Az interrupt szignált (SIGINT (2)) akkor generálódik, amikor az interrupt gombot (ctrl-c) használjuk.

A quit szignál (SIGQUIT (3)) a quit kombináció esetén (ctrl-\) jön létre.

4.12.4 Folyamat vezérlése a Bash shell-ben

Néhány további műveletre is lehetőségünk nyílik a *bash shell* használata esetén:

Az előtér folyamatot felfüggeszthetjük a <ctrl-*z*> kombinációval. Ez nem szünteti meg a folyamatot, csak megállítja a pillanatnyi állapotában. Amikor megállítjuk a folyamatot, a *shell* megjelenít egy *job* számot, amivel később hivatkozhatunk rá.

Ha háttér folyamatként szeretnénk folytatni, akkor az a **`bg`** paranccsal tehetjük meg.

```
bg <job>
```

Ha megint előtérbe akarjuk hozni, akkor azt az **`fg`** paranccsal tehetjük meg.

```
fg <job>
```

A **`jobs`** paranccsal kilistázhatjuk a felfüggesztett- és háttérfolyamatokat.

A job számot használhatjuk a **`kill`** parancs paraméterezésére is, így nem kell a processz azonosítót megjegyezni.

```
kill [-szignál] %job
```

4.12.5 Prioritás állítás

Lehetőségünk van arra, hogy az egyes folyamatok prioritását módosíthassuk. Ezáltal megmondhatjuk a kernel processz ütemezőjének, hogy mely taszkokat részesítsen előnyben, és mely taszkok nem fontosak. A prioritást a **`renice`** paranccsal állíthatjuk be. A **`renice`** szintaxisa:

```
renice <prioritás> [[-p] pid ...] [[-g] pgrp ...] [[-u] user ...]
```

Egy vagy több folyamatot is megadhatunk egyszerre. Felsorolhatjuk a folyamatok azonosítóját (**`-p pid`**), hivatkozhatunk folyamat csoportokra is (**`-g pgrp`**), vagy egy felhasználó összes folyamatának állíthatjuk a prioritását (**`-u user`**), vagy kombinálhatjuk is ezeket. A leggyakrabban használt alak, amikor csak egy folyamat prioritását módosítjuk:

```
renice 20 1033
```

Ahol az 1033-as azonosítóval rendelkező folyamat *nice* szintjét 20-ra állítjuk.

A 0 prioritás a folyamatok alapértelmezett prioritása. Ehhez képest a pozitív értékek egyre nagyobb előzékenységet jelentenek. A maximum érték a 20. Ha egy folyamat *nice* értéke 20,

az azt jelenti, hogy csak akkor fut, amikor más folyamatnak nincs szüksége a processzorra. A felhasználók csak 0 és 20 között állíthatják az értéket.

A rendszergazda ellenben mínusz értékeket is beállíthat egészen -20-ig. Ezzel megnövelheti a folyamat által kapott processzoridő arányát. Természetesen ezzel óvatosan kell bánnunk, hogy ne szívjuk el az erőforrásokat más processzek elől.

5 Beágyazott rendszer készítése x86 platformon

Az következő fejezet példát mutat egy x86 alapú beágyazott rendszer elkészítésére.

5.1 Hozzávalók:

- Linux Kernel
- Libc (glibc, vagy uClibc)
- busybox
- e2fsprogs
- dropbear (ssh-hoz)
- bootmanager (grub, lilo, uBoot)

Az egyes csomagok web oldalát Google segítségével könnyen megtalálhatjuk és letölthetjük a forrás állományokat.

5.2 Előkészületek

A rendszer összeállításához hozzunk létre egy üres könyvtárat. Az állományrendszer felépítése során egyes állományokra speciális jogosultságokat is be kell állítanunk. Ezért a példában rendszergazdaként állítjuk össze a Linux rendszerünket. A másik megoldás a fakeroot csomag használata, amely felhasználók számára szimulál egy olyan környezetet, ahol a szükséges jogosultság beállításokat elvégezhetik.

5.2.1 Kernel fordítás

Tömörítsük ki a kernel forrást, majd lépjünk be a kernel könyvtárába. Első lépésként konfigurálnunk kell a kernelt. Erre több felület közül választhatunk. Az X-es megoldás a következő:

```
make xconfig
```

A konfiguráció összeállításához szükséges, hogy tudjuk, milyen hardver eszközöket tartalmaz a gépünk, illetve nem árt némi tapasztalat is ilyen téren. Ha elvégeztük a konfigurálást és lementettük a beállításokat, akkor következhet a fordítás és a kész állományok elhelyezése a rendszerünkben.

```
make
install -D -m 0644 arch/i386/boot/bzImage /root/MyLinux/system/boot
make INSTALL_MOD_PATH=/root/MyLinux/system/ modules_install
```

5.2.2 Libc

A fordításokhoz és a rendszer összeállításhoz le kell fordítanunk a glibc library csomagot. Azonban a műveletek megkönnyítése érdekében a példában a desktop rendszer könyvtárait használjuk a fordításokhoz, ezért az elkészített rendszerbe is be kell másolni a desktop gépről a glibc csomag állományait. Ha eltérne a munkagép és a célgép architektúrája, akkor természetesen ezt a trükköt nem követhetjük el.

5.2.3 Grub

A grub fordítása és telepítése a következő:

```
./configure --prefix=/usr
make
make DESTDIR="/root/MyLinux/system/" install
```

A telepített állományok azt a célt szolgálják csak, hogy segítségükkel felinstalláljuk a grub-ot. Így ha helyszűkében vagyunk, akkor a telepítés lépését ki is hagyhatjuk, illetve más helyre telepíthetjük az állományokat. Ezt követően a szükséges állományokat másoljuk be a Linux rendszerünkbe.

Erre a célra hozzuk létre egy könyvtárat:

```
/root/MyLinux/system/boot/grub
```

A /boot/grub tartalmának összeállítása:

```
cp /root/MyLinux/system/usr/lib/grub/i386-pc/{stage1,stage2,e2fs_stage1_5} /root/MyLinux/system/boot/grub/
```

Ezt követően ugyanabban a könyvtárban hozzunk létre egy konfigurációs állományt `grub.conf` néven:

```
serial --unit=0 --speed=115200
terminal --timeout=0 serial console

timeout 3
default 0

title Linux
root (hd0,0)
kernel /bzImage ro root=/dev/hda1 ramdisk_size=0 noinitrd console=ttyS0,115200n8
```

A kernel paraméterezés során feltételeztük, hogy a Compact Flash a számítógép primary IDE buszán található master-ként. Ha más a hardver összeállítás, akkor a `/dev/hda` helyett mást kell megadnunk.

Történelmi okokból célszerű egy szimbolikus linket létrehoznunk a konfigurációs állományra az alábbi módon:

```
ln -s grub.conf menu.lst
```

5.2.4 Busybox

A busybox csomag szolgáltatja a rendszerünkhöz szükséges alapvető programokat. A kernelhez hasonlóan konfigurálhatjuk. A konfigurálás során kiválaszthatjuk, hogy milyen programokat szeretnénk a rendszerünkbe. A fordítás eredményeként egy bináris állományt kapunk, és az egyes programok valójában szimbolikus linkek lesznek erre az állományra.

```
make xconfig
make
make install
```

Az `_install` alkönyvtárból másoljuk ki az állományokat. (Archív copy! vagyis `cp -a`)

5.2.5 e2fsprogs

A forrás kitömörítése után a következő módon fordítjuk és telepítjük:

```
./configure
make
make DESTDIR="/root/MyLinux/system" install
```

5.2.6 Dropbear

A forrást az alábbi módon fordíthatjuk és telepíthetjük:

```
./configure --prefix=/usr
make PROGRAMS="dropbear dbclient dropbearkey scp"
make PROGRAMS="dropbear dbclient dropbearkey scp" DESTDIR="/root/MyLinux/system" install
```

5.2.7 Debug információk

A rendszerünkben található binárisok (futtatható programok és libraryk) debug információkat tartalmazhatnak. Mivel a kis méretre törekszünk, ezeket eltávolíthatjuk a **strip** program használatával. Az alábbi könyvtárakban kell elkövetnünk (a `/root/MyLinux/system` könyvtártól számítva): `/bin`, `/lib`, `/sbin`, `/usr/bin`, `/usr/lib`, `/usr/sbin`

5.3 A könyvtárfa kiegészítése

Az alábbi hiányzó könyvtárakat hozzuk még létre a `/root/MyLinux/system` alatt:

```
/dev
/mnt
/mnt/pendrive
/proc
/root
```

```
/sys
/tmp
/var
/var/lock
/var/log
/var/run
/var/spool
/var/spool/cron
/var/tmp
```

5.4 Konfiguráció

5.4.1 Alap konfigurációs állományok

Az `etc` könyvtárban létre kell hoznunk a következő alapvető konfigurációs állományokat.

5.4.1.1 fstab

```
/dev/hda1       /               ext3    defaults        1 1
none            /dev/pts        devpts  defaults        0 0
none            /dev/shm        tmpfs   defaults        0 0
none            /proc           proc    defaults        0 0
none            /sys            sysfs   defaults        0 0
```

5.4.1.2 group

```
root:x:0:
```

5.4.1.3 gshadow

```
root:::root
```

5.4.1.4 hosts

```
127.0.0.1       localhost.localdomain localhost Ctrl
```

5.4.1.5 grub.conf

```
ln -s /boot/grub/grub.conf
```

5.4.1.6 issue

```
Kernel \r on an \m
```

5.4.1.7 mtab

```
ln -s /proc/mounts mtab
```

5.4.1.8 nsswitch.conf

```
passwd:     files
shadow:     files
group:      files

hosts:      files dns

bootparams: files

ethers:     files
netmasks:   files
networks:   files
protocols:  files
rpc:        files
services:   files

netgroup:   files

publickey:  files

automount:  files
aliases:    files
```

Figyelem! A beállítások függvényében a /lib könyvtárba be kell másolnunk a megfelelő `libnss_xxx.so` állományokat, illetve azok függőségeit.

5.4.1.9 passwd

```
root:x:0:0:root:/root:/bin/sh
```

5.4.1.10 profile

```
# /etc/profile

PATH="/sbin:/bin:/usr/sbin:/usr/bin"

if [ -x /usr/bin/id ]; then
        USER="`/usr/bin/id -un`"
        LOGNAME=$USER
```

```
fi


export PATH USER LOGNAME
```

5.4.1.11 protocols

```
ip      0       IP
icmp    1       ICMP
tcp     6       TCP
udp     17      UDP
raw     255     RAW
```

5.4.1.12 services

```
tcpmux  1/tcp
udpmux  1/udp
echo    7/tcp
echo    7/udp
ftp     21/tcp
ssh     22/tcp
ssh     22/udp
telnet  23/tcp
domain  53/tcp  nameserver
domain  53/udp  nameserver
```

5.4.1.13 shadow

```
root::0:0:99999:7:::
```

5.4.1.14 shells

```
/bin/sh
/bin/ash
/bin/false
```

5.4.1.15 termcap

A linux és vt100 terminálok beállításait másoljuk bele a desktop gép hasonló állományából.

5.4.2 Egyéb

További konfigurációs állományok.

5.4.2.1 dropbear

```
mkdir dropbear
```

5.4.2.2 mdev.conf

```
urandom  0:0 644
random   0:0 644

#sda[1-4] 0:0 664 @ mount -t vfat /dev/$MDEV /mnt/pendrive
```

5.4.3 Indító scriptek

A rendszer elindulásához létre kell hoznunk az indító scripteket. Ezeket célszerű könyvtárakba rendeznünk.

Hozzuk létre az alábbi könyvtárakat:

```
rc.d
rc.d/init.d
rc.d/rc.startup.d
rc.d/rc.shutdown.d
```

5.4.3.1 rc.d/rc.sysinit

```
#!/bin/sh

echo "Mount /proc"
mount -t proc /proc /proc
mount -t usbfs /proc/bus/usb /proc/bus/usb

echo "Mount /sys"
mount -t sysfs /sys /sys

e2fsck -y /dev/hda1

echo "Remount /"
mount -o remount,rw /

echo "Mount /dev"
mount -t tmpfs mdev /dev
mkdir /dev/pts
```

```
mount /dev/pts
mkdir /dev/shm

echo "Start mdev"
echo /sbin/mdev > /proc/sys/kernel/hotplug
/sbin/mdev -s

# dropbear miatt
rm -f /dev/random
ln -s /dev/urandom /dev/random

echo "Mount all"
mount -a

for ii in /etc/rc.d/rc.startup.d/S* ; do
  $ii start
done
```

5.4.3.2 rc.d/rc.shutdown

```
#!/bin/sh

for ii in /etc/rc.d/rc.shutdown.d/K* ; do
  $ii
done

/bin/umount -a -r
```

5.4.3.3 inittab

```
::sysinit:/etc/rc.d/rc.sysinit
::respawn:/sbin/getty -L ttyS0 115200 vt100
#tty1::respawn:/sbin/getty 38400 tty1
::restart:/sbin/init
::ctrlaltdel:/sbin/reboot
::shutdown:/etc/rc.d/rc.shutdown
```

5.4.4 További indító scriptek

5.4.4.1 rc.d/init.d/crond

```
#!/bin/sh

[ -f /usr/sbin/crond ] || exit 0

case "$1" in
  start)
      if ! pidof crond ; then
          crond
      fi
      ;;
  stop)
      killall crond 2>/dev/null
      ;;
  restart)
      $0 stop
      $0 start
      ;;
  status)
      if pidof crond >/dev/null ; then
          echo "running"
      else
          echo "stopped"
      fi
      ;;
esac
```

5.4.4.2 rc.d/init.d/syslog

```
#!/bin/sh

[ -f /sbin/syslogd ] || exit 0
[ -f /sbin/klogd ] || exit 0

case "$1" in
```

```
  start)
      if ! pidof syslogd | sed "s/$$\$//" | grep -q [0-9] ; then
          syslogd -m 0 -C64
          klogd -c 1
      fi
      ;;
  stop)
      killall klogd 2>/dev/null
      killall syslogd 2>/dev/null
      ;;
  restart)
      $0 stop
      $0 start
      ;;
  status)
      if pidof syslogd | sed "s/$$\$//" | grep -q [0-9] ; then
          if pidof klogd >/dev/null ; then
              echo "running"
          else
              echo "stopped"
          fi
      else
          echo "stopped"
      fi
      ;;
esac
```

5.4.4.3 rc.d/init.d/network

```
#!/bin/sh

case "$1" in
  start)
      . /etc/sysconfig/network.conf
      /bin/hostname ${HOSTNAME}
```

```
      # configure interfaces
      for ii in /etc/sysconfig/network/*.conf ; do
          DEV=`basename $ii | sed 's/.conf//g;'`
          /etc/sysconfig/network/ifup $DEV
      done
      ;;
  stop)
      for ii in /etc/sysconfig/network/*.conf ; do
          DEV=`basename $ii | sed 's/.conf//g;'`
          /etc/sysconfig/network/ifdown $DEV
      done
      ;;
  restart)
      $0 stop
      $0 start
      ;;
  status)
      ip address
      ip route
      ;;
esac
```

5.4.4.4 rc.d/init.d/dropbear

```
#!/bin/sh

[ -f /usr/sbin/dropbear ] || exit 0
[ -f /usr/bin/dropbearkey ] || exit 0

case "$1" in
  start)
      if [ ! -f /etc/dropbear/dropbear_rsa_host_key ]
      then
        echo $"Generate rsa host key"
        dropbearkey -t rsa -f /etc/dropbear/dropbear_rsa_host_key
      fi
```

```
      if [ ! -f /etc/dropbear/dropbear_dss_host_key ]
      then
        echo $"Generate dss host key"
        dropbearkey -t dss -f /etc/dropbear/dropbear_dss_host_key
      fi

      dropbear
      ;;
  stop)
      killall dropbear 2>/dev/null
      ;;
  restart)
      $0 stop
      $0 start
      ;;
  status)
      if pidof dropbear | sed "s/$$\$//" | grep -q [0-9] ; then
          echo "running"
      else
          echo "stopped"
      fi
      ;;
esac
```

5.4.5 Az indító scriptek belinkelése

5.4.5.1 rc.d/rc.startup.d/

```
S11network -> ../init.d/network
S12syslog -> ../init.d/syslog
S40dropbear -> ../init.d/dropbear
S90crond -> ../init.d/crond
```

5.4.5.2 rc.d/rc.shutdown.d/

```
K10crond -> ../init.d/crond
K21dropbear -> ../init.d/dropbear
K42syslog -> ../init.d/syslog
```

```
K44network -> ../init.d/network
```

5.4.6 A hálózati kártyák konfigurálása

5.4.6.1 sysconfig/network.conf

```
HOSTNAME=Ctrl
```

5.4.6.2 sysconfig/network/ifup

```
#!/bin/sh

error_exit()
{
  error_exit "missing dev config" >&2
}

DEV=$1
if [ ! -f /etc/sysconfig/network/$DEV.conf ] ; then
    exit 1
fi
. /etc/sysconfig/network/$DEV.conf

if ! ip link set up dev $DEV ; then
  error_exit "cannot enable interface $dev."
fi

if [ ! $DEV = lo ] ; then
  if ! arping -q -c 2 -w 3 -D -i ${DEV} ${IPADDR} ; then
     error_exit "another host already uses address ${IPADDR} on $
{DEV}."
  fi
fi

if ! ip address add $IPADDR/$NETMASKBITS brd + dev $DEV; then
  error_exit "failed to configure $IPADDR/$NETMASKBITS on $DEV."
fi

# update ARP cache of neighboring computers
```

```
if [ ! $DEV = lo ] ; then
    arping -q -A -c 1 -i $DEV $IPADDR 2>/dev/null >/dev/null
</dev/null
    arping -q -U -c 1 -i $DEV $IPADDR 2>/dev/null >/dev/null
</dev/null
fi

if [ ! -z "$DEFAULTGW" ] ; then
  ip route add default via $DEFAULTGW
fi
```

5.4.6.3 sysconfig/network/ifdown

```
#!/bin/sh

error_exit() {
    echo "ifdown: error: $@" >&2; exit 1
}

DEV=$1
if [ ! -f /etc/sysconfig/network/$DEV.conf ] ; then
    error_exit "missing dev config" >&2
fi
. /etc/sysconfig/network/$DEV.conf

if [ $DEV = eth1 ] ; then
    ip route delete
fi
if ! ip address delete $IPADDR/$NETMASKBITS brd + dev $DEV ; then
    error_exit "failed to remove $IPADDR/$NETMASKBITS on $DEV."
fi
```

5.4.6.4 sysconfig/network/lo.conf

```
IPADDR=127.0.0.1
NETMASKBITS=24
```

5.4.6.5 sysconfig/network/eth0.conf

```
IPADDR=192.168.0.10
```

```
NETMASKBITS=24
DEFAULTGW=192.168.0.254
```

5.5 Az induló eszközállományok összeállítása

Minden, amit az mdev indulása előtt használunk. A következők eszközállományokat célszerű létrehoznunk, vagy bemásolnunk a `dev` könyvtárban:

```
console
hda
hda[1-4]
null
tty[0-6]
ttyS[0-3]
zero
```

5.6 Az állományrendszer előállítása

Következő lépésként csatlakoztassuk a Compact Flash kártyát a számítógéphez és végezzük el a következő műveleteket.

5.6.1 Partícionálás

Az `fdisk` program segítségével hozzunk létre egy Linux (83) partíciót.

5.6.2 Állományrendszer létrehozása

```
mke2fs -j -L / /dev/sdb1
```

5.6.3 Az állományok másolása

Csatoljuk fel a partíciót és archív másolással (`cp -a`) másoljuk rá az összeállított állományrendszerűnket, majd csatoljuk le!

5.6.4 A grub installálása

Ahhoz, hogy a Compact Flash-ről el tudjon indulni a PC, még be kell installálnunk a grub-ot a MBR-be. Ezt a következő parancsokkal tehetjük meg (ha a desktop gépünkre nincs grub felinstallálva, akkor a lefordított programot kell meghívnunk):

```
grub

grub> find /boot/grub/stage1
grub> root (hd1,0)
grub> setup (hd1)
grub> quit
```

5.7 Az első teszt

Az elkészült CF kártyát tegyük be az eszközbe, kapcsoljuk be és szorítsunk.

5.7.1 Ellenőrzés

Ha bootol, akkor `logread`-el mindenképpen ellenőrizzük a logot, és vizsgáljuk körbe, hogy minden működik-e.

5.7.2 Jelszó

Célszerű első lépésként beállítani a jelszót, majd visszamásolni azt az összeállított állományrendszerünkbe.

6 Beágyazott rendszer készítése más platformra

Ha a rendszer elkészítéséhez használt és a cél platform eltér egymástól, akkor a fordítási folyamat jelentősen bonyolódik az előző példához képest.

6.1 Keresztfordítás

Ha a fejlesztői gép és a célgép architektúrája eltér egymástól, akkor szükséges egy keresztfordító környezet kialakítása. A fordításokat ebben a környezetben kell elvégeznünk, hogy a binárisok majd fussanak az eszközön.

Azonban a keresztfordító környezetnek akkor is van létjogosultsága, ha a két gép architektúrája megegyezik. Ilyenkor segít abban, hogy az alkalmazásokat a beágyazott eszközben található library verziókkal készíthessük el, és ehhez ne kelljen a desktop rendszert módosítanunk.

Keresztfordító környezet kialakítása az alábbi lépésekkel történhet. A fordításoknál a --target=<cél platform> opcióval meg kell adnunk a célgép architektúráját, hogy a megfelelő binárisok készülhessenek. A lépések:

1. Szükségünk van a Linux kernel forrásban található header állományokra. Ehhez a kernelt be kell konfigurálnunk a célgépnek megfelelően, mivel ez hatással lehet a header állományokra is.
2. A következő lépés a **`binutils`** csomag lefordítása.
3. Szükségünk van egy C fordítóra. Ehhez le kell fordítanunk a **`gcc`** csomagot. Azonban a program lefordítására szükség van a lefordított **`glibc`** csomagra, amelyet csak a következő lépésben tudunk előállítani a fordító programmal. A megoldást az jelenti, hogy a **`gcc`** elkészítése bár sikertelen lesz, de elkészít egy C fordítót, amivel megtörténhet a **`glibc`** fordítása.
4. A **`glibc`** lefordítása.
5. Mivel a **`glibc`** elkészült, ezt követően a **`gcc`** csomag lefordítása már teljes lehet.
6. Ezt követően le kell fordítanunk a további fejlesztői könyvtárakat is, amelyeket a rendszerben használni szeretnénk.

6.2 Automatizált eszközök

6.2.1 Buildroot

A Buildroot Makefile-ok és patch-ek gyűjteménye, amely egyszerűvé teszi a keresztfordító környezet és a beágyazott rendszer létrehozását a cél platformra. C fejlesztői könyvtárnak az uClibc-t használja.

A fejlesztése, módosítása nehézkes, ezért manapság az OpenEmbedded rendszer veszi át a helyét.

6.2.2 OpenEmbedded

Az OpenEmbedded rendszer a Buildroot-al szemben a fordítási műveleteket, a csomagok előállításának módját BitBake receptekben írja le. Számos cél architektúrára készíthetünk a segítségével keresztfordító környezetet, csomagokat, vagy operációs rendszert.

Könnyen bővíthető új receptekkel, vagy módosíthatóak a már meglévő részei.

6.2.3 Yocto Project

Az OpenEmbedded rendszeren alapuló, annak hibáit kiküszöbölő rendszer. Strukturált, réteges felépítésének köszönhetően lehetővé teszi az elosztott fejlesztést. Elsődleges célja, hogy egy hobby fejlesztők által használt rendszerből egy a cégek számára is jól használható eszközzé váljon. Elősegíti, hogy például a hardver fejlesztő cégek elkészítsék a paneljeikhez a támogatást, illetve a rendszer fejlesztők a saját kiegészítéseiket, módosításaikat könnyen hozzáadhassák az alaprendszerhez.

Jól dokumentált, könnyen integrálható az Eclipse fejlesztői környezetbe, készíthetünk emulációs környezetet és SDK eszközkészletet is a rendszerünkhöz.

6.2.4 Scratchbox

A Scratchbox szemben az előző eszközökkel nem a keresztfordító környezet előállítására, vagy az operációs rendszer összeállítására való. Feladata elsősorban, hogy a célrendszert emulálja és ebben az emulált környezetben történik a fordítás. A parancssor ebben az esetben a legegyszerűbb, mert a szimulált rendszer alapértékei megoldják ez a feladatot.

Megoldható akár az is, hogy az OpenEmbedded rendszerrel megalkotjuk a keresztfordító környezetet, majd beillesztjük azt a Scratchbox-ba.

6.3 Yocto project

A Yocto project egy fejlesztői keretrendszert nyújt számunkra, amelynek segítségével könnyen lefordíthatjuk az egyes programokat különböző hardver architektúrákra. Továbbá támogatja teljes disztribúciók összeállítását és későbbi újrafordítását is. Segítségével összeállíthatjuk a keresztfordító környezetet, előállíthatunk szoftver telepítő csomagokat, kész rendszer image állományokat[10], és fejlesztői környezetet.

Az előállítható fejlesztői környezet tartalmazhat egy emulált rendszert, publikálható fejlesztői SDK-t, Eclipse integrációt.

Használatához Linux rendszerre van szükségünk. A műveletek többsége adminisztrátori jogok nélkül is elvégezhető. Ugyanakkor jelentős lehet a háttértár igénye.[11]

6.3.1 Telepítés

A Yocto Project állományait és dokumentációját a projekt weboldalán keresztül érhetjük el legkönnyebben: https://www.yoctoproject.org/

[10] Olyan állomány, amely kompletten tartalmaz egy Linux rendszert, amelyet már csak fel kell telepítenünk. Az állomány konkrét típusa és a telepítés módja hardverenként eltérhet.

[11] Több GB tárhelyre kell számítanunk a legkisebb rendszereknél is.

A Yocto Project rendszer használatához néhány szoftvert először telepítenünk kell. A konkrét csomagnevek disztribúciónként eltérnek, de a weboldalon található dokumentációkban megtalálható néhány tipikusa disztribúció esetére. Ubuntu esetén az alábbi parancsot kell kiadnunk:

```
sudo apt-get install gawk wget git-core diffstat unzip texinfo build-essential chrpath libsdl1.2-dev xterm
```

A Yocto Project rendszert a Git repository-ból kell letöltenünk. A Git egy verzió kontroll eszköz. Segítségével a Git repository-n keresztül szinkronizálják a fejlesztők a munkájukat. A jelenleg aktuális stabil verzió letöltése az alábbi paranccsal történik:

```
git clone -b dora git://git.yoctoproject.org/poky.git
```

Később a frissítést a következő paranccsal végezhetjük el (ezt a letöltéskor létrejött „poky” könyvtárban kell kiadni):

```
git pull
```

6.3.2 Új rendszer generálása

A Yocto Project feltelepítése után a teendőnk egy „build” könyvtár létrehozása, a konfigurációs állományok elkészítése és az image állomány előállítása.

Első lépésként belépünk a Yocto Project könyvtárába:

```
cd poky
```

Egy szkript segítségével legeneráljuk a „build” könyvtárunkat benne az alapértelmezett konfigurációs állományokkal:

```
source oe-init-build-env
```

A parancs hatására be is lépünk a „build” könyvtárba. Az itt található „conf” alkönyvtár tartalmazza a konfigurációs állományokat. A „conf/bblayers.conf” állományban megadhatjuk a használt rétegeket, azonban elsőre használhatjuk az alapértelmezett értéket.

A „conf/local.conf” állományban tudjuk megadni azokat a paramétereket, amelyek alapján a rendszert szeretnénk összeállítani. (El is tekinthetünk ezektől a beállításoktól, azonban ekkor a fordítás parancssorának elejére kell írnunk őket.) Elsősorban az alábbi paramétereket kell megadnunk:

- A cél architektúra megadása (MACHINE)
- A disztribúció megadása (DISTRO)
- Csomagformátum (PACKAGE_CLASSES)
- A párhuzamosan futó build szálak száma

Tipp – A következő sort írjuk be a konfigurációs állományba. Ennek hatására a fordítás végén a rendszer eltávolítja azokat az átmeneti állományokat, amelyekre már nincs szükség.

```
INHERIT += "rm_work"
```

Ezt követően előállítjuk az image állományt. Például:

```
bitbake -k core-image-sato
```

Az image típus megválasztásával döntjük el, hogy tulajdonképpen mi kerüljön az image állományba. Az image helyett választhatunk még egyszerű csomag recepteket, vagy metacsomagokat, amelyek több másik csomagra hivatkoznak.

Példa egyszerű csomagra:

```
bitbake nano
```

A „-k” paraméter azt jelzi a bitbake számára, hogy ha valahol elakad a fordítási folyamat, akkor is a lehető legtöbb lépést hajtsa végre. Mivel esetenként az összes lépés végrehajtása több órát vehet igénybe, ezért gyakran magára hagyjuk a gépet ezen idő alatt. Kellemetlen lenne pár óra múlva arra visszatérni, hogy az egész folyamat valamelyik korai lépésnél elakadt és azóta nem csinált a gép semmit.

6.3.2.1 Recept

Egy szoftver csomag előállítási recept tartalma:

- Adminisztratív adatok a telepítő csomaghoz
- Függőségek
- Forrás állományok (Ha URL-t adunk meg, akkor letölti.)
- A források előkészítése
- A forrás konfigurálása
- Fordítás
- Telepítése
- Telepítő csomagok összeállítása
- Esetlegesen a csomag fordító környezetbe való telepítése
- Architektúra függő kiegészítések, egyes lépések felüldefiniálása

6.3.2.2 Rétegek

A recepteket a Yocto Projectben, szemben a klasszikus OpenEmbeddeddel rétegekbe (layer) szervezzük. Ezáltal lehetővé válik az elosztott munka, mert mindenki a saját rétegében fejleszthet egymástól függetlenül. A konfiguráció során adjuk meg, hogy ténylegesen mely rétegek felhasználásával álljon össze az a recept és konfiguráció halmaz, amellyel a fordítást végezzük.

A rendszerben meglévő alapvető rétegek:

- meta
- meta-yocto
- meta-yocto-bsp

A hardver gyártók saját réteget nyújthatnak az eszközeikhez, amelyeket Board Support Packages (BSP) rétegeknek nevezünk. Ez általában az adott eszköz konfigurációs állományait, kernel konfigurációját és az esetleges recept módosításokat tartalmazza.

Mi is alkothatunk külön rétegeket a szoftvereinknek. Ebben a rétegben az egyes szoftvercsomagok előállításához készítünk recepteket, illetve az azokat összefogó metacsomag recepteket. Természetesen saját image állomány receptet is készíthetünk.

A kész rétegeket le is tölthetjük, összeválogathatjuk a fordításhoz. Sok népszerű hardverhez és szoftvercsomaghoz találhatunk ilyen rétegeket.

A rétegek amellett, hogy új recepteket adhatnak a fordításnál használt halmazhoz egymás receptjeit felül is definiálhatják, módosíthatják. Köztük prioritás sorrend állítható fel.

6.3.2.3 Disztribúció

A disztribúció az OpenEmbedded esetében egy lista szoftver változatokról, verziókról, amelyek a tesztek alapján működőképesek együtt. Mivel a különböző szoftver csomagok különböző verziói sokszor nem működnek együtt, ezért szükséges egy válogatás, amely meghatározza, hogy az adott rendszerbe milyen változatokat akarunk használni.

Ugyanakkor a válogatás nehéz feladat, és nem biztos, hogy az összeállító ember tesztjei mindenre kiterjedtek. Így nem garantált, hogy nem találkozunk problémával.

A Yocto Projectben a két elterjedt disztribúció a „poky” és az „angstrom”.

6.3.2.4 Architektúra

A cél hardver meghatározza a következőket:

- A kernelt:
    - A verziót
    - A konfigurációt
    - A telepítendő modulokat, és járulékos szoftvereket
    - A formátumot
    - Az alapértelmezett paramétereket
- A preferált grafikus felületet
- Az image állomány állományrendszerét

Az OpenEmbedded rendszerben az architektúra konfigurációs állományok az itt felsorolt paramétereket tartalmazzák. Emellett a rendszerben az egyes csomagok recept állományaiban definiálhatunk architektúra függő módosításokat, konfigurációkat is.

6.3.2.5 A csomag fajták

A fordítás során célnak az alábbi csomag fajtákat választhatjuk:

- Egyszerű szoftver csomag

- Task metacsomag – Több szoftver csomagot fog össze egy feladathoz, szoftver rendszerhez, teljes operációs rendszer felépítéséhez
- Image – Teljes operációs rendszer image

Ezeket a csomagokat az OpenEmbedded „recipes” alkönyvtárában találhatjuk.

6.3.3 A Yocto Project felépítése

A Yocto Project rendszer könyvtára az alábbi alkönyvtárakat tartalmazza:

- bitbake – A program, ami a recepteket végrehajtja.
- build* – A könyvtár, amit a munkához létrehozunk.
- documentation – Dokumentációk
- Rétegek: meta, meta-yocto, meta-yocto-bsp
  - classes – BBClass állományok, amelyek receptekben felhasználható műveleteket implementálnak
  - conf – Konfigurációs állományok
    - bitbake.conf – A BitBake fő konfigurációs állománya
    - distro – Disztribúciók
    - machine – Architektúrák
  - files – Egyéb állományok
  - recipes-* – Receptek
  - site – Operációs rendszer függő beállítások
- scripts – Szkriptek a művelek egyszerűsítéséhez.

A `recipes` könyvtár szoftver csomagonként egy-egy könyvtárat tartalmaz. Ezen könyvtárak tartalma:

- Receptek a különböző szoftver verziókhoz.
- Verzió független állományok egy könyvtárban.
- Verzió függő állományok (patch, config) külön könyvtárakba szervezve.

Emellett az egyes könyvtárak tartalmazhatnak további architektúra, vagy disztribúciófüggő alkönyvtárakat.

6.3.4 Saját kiegészítések

A saját receptjeinket, kiegészítéseinket egy új réteg létrehozásával adhatjuk a rendszerhez.

Új réteget az alábbi paranccsal hozhatunk létre:

```
yocto-bsp create <bsp-name> <karch>
```

A „bsp-name” az új réteg neve. A „karch” a kernel architektúrát adja meg, amelyhez a konfiguráció és a receptek készülnek. A paraméterezésből látható, hogy a szkript elsősorban BSP jellegű rétegek létrehozására szolgál. Azonban később könnyen átstruktúrálhatjuk a rétegünket, akár kitörölhetjük a hardver konfigurációs részeket, vagy akár többet is létrehozhatunk.

A létrehozott réteg könyvtárában a „conf/layer.conf” állomány tartalmazza a réteg beállításait:

- réteg neve
- receptek elérhetősége
- réteg prioritása

Ezt követően létrehozhatunk

- új architektúra konfigurációt (conf/machine/)
- saját recepteket (recipes-*/)

6.3.5 Valósidejű rendszer készítése

A Yocto Project segítségével valósidejű Linux rendszert is készíthetünk. Ez a normál rendszertől csak a kernelben tér el, illetve hozzáadhatunk néhány szoftver eszközt a valósidejű használathoz.

A linux-yocto kernel repository tartalmaz preempt-rt ágakat. A rendszerhez a kernelt ezekből az architektúrának megfelelő felhasználásával kell generálnunk. A művelethez egy kernel receptet kell készítenünk, azonban a népszerűbb architektúrákhoz ez már készen megtalálható a Yocto rendszerben. Egy új BSP réteg létrehozásakor kiválaszthatjuk a listából.

A létrehozott BSP réteget hozzá kell vennünk a fordítási konfigurációhoz. Ezt követően az elkészített image már a valósidejű kernelt tartalmazza, ami által valósidejű Linux rendszernek számít. Azonban célszerű a rendszert néhány valósidejű teszt programmal és eszközzel is kiegészítenünk. Ezt úgy érhetjük el, hogy vagy a „core-image-rt” recept segítségével generáljuk a rendszer imaget, vagy egy ebből származó recepttel.

6.3.6 A létrehozott könyvtárak

A fordítás során a „build” könyvtárunkban a rendszer létrehoz egy `tmp` könyvtárat, amely az alábbi alkönyvtárakat tartalmazza:

Könyvtár	Leírás
`cache`	A receptek feldolgozása során a rendszer előállít olyan átmeneti állományokat, amely később gyorsítja a az elindítását. Ezek a gyorsító állományok tárolódnak ebben a könyvtárban.
`deploy`	A telepíthető csomagokat (a konfigurációtól függ a formátum), és az image állományokat találhatjuk itt külön könyvtárakban architektúrának megfelelően.

Könyvtár	Leírás
`pkgdata`	A csomagok adatai.
`rootfs`	Ha teljes beágyazott rendszert készítünk, akkor ebben a könyvtárban állítja össze az OpenEmbedded a rendszerünk root állományrendszerét. Majd ezt követően a beállításoknak megfelelően összeállítja az image állományokat belőle.
`stamps`	Az elvégzett műveleteket üres állományok létrehozásával könyveli a rendszer. Ha valamelyik műveletet szeretnénk megismételtetni, akkor az állomány törlésével elérhetjük. Azonban ilyen esetben ügyeljünk arra, hogy a műveletek egymásra épülnek, így ha meggondolatlanul cselekszünk, akkor nem tudható az eredménye.
`sysroots`	A keresztfordító környezet állományai találhatóak itt alkönyvtárakba szervezve. Ha szeretnénk a saját forrás állományunkat lefordítani az eszközre, akkor itt keressük meg hozzá a gcc programot. Emellett a library és header állományok is itt találhatóak a fordításhoz.
`work`	Az egyes csomagok lefordításához a rendszer ezen a könyvtáron belül létrehoz egy külön könyvtárat.

A csomagok fordításához, illetve a telepítő csomagok elkészítéséhez a rendszer a `work` könyvtáron belüli architektúrának megfelelő alkönyvtárban létrehoz egy-egy külön könyvtárat. Ebben további alkönyvtárak találhatóak az egyes műveletek elvégzéséhez.

Könyvtár	Leírás
`temp`	Az egyes lépések lefuttatásához a rendszer létrehoz egy-egy script állományt. Ezt futtatva hajtja végre az adott lépést. A scriptek mellett a műveletek log állományait tárolja itt a rendszer. Ezekből visszanézhetjük az esetleges hibák esetén az üzeneteket.
forrás könyvtár	A könyvtár neve változó attól függően, hogy a betömörített forrás csomagban milyen néven szerepel. A patch-elés, konfigurálás, fordítás ebben a könyvtárban történik.
`image`	A fordítás végén a telepítést ebben a könyvtárban végzi el a rendszer. Ebből történik a telepítő csomag állományok összeválogatása.
`packages-split`	Minden telepítő csomaghoz létrehoz a rendszer egy alkönyvtárat, amelybe belemásolja az `image` könyvtárból a csomaghoz tartozó állományokat. Ezt követően elkészíti az adott telepítő csomagokat.

Könyvtár	Leírás
`staging-pkg`	Ha valamilyen fejlesztői könyvtárat fordítunk le, akkor a keresztfordítói környezetben is szükség van mind a library mind a header állományokra. Ezeket az állományokat, amelyeket a `staging` könyvtárba szán ebben a könyvtárban rendezi össze a rendszer egy telepítő csomaggá.

6.3.7 Telepítés

A `deploy` könyvtárban található egy alkönyvtár az image állományoknak `images` néven. Ebben találhatjuk meg az előállított kernelt illetve a beágyazott rendszert tartalmazó állományt. Emellett található még itt egy vagy több könyvtár, amely telepítő csomagokat tartalmaz. A csomagok formátuma lehet `ipk`, `rpm`, `deb`, így egy-egy hasonló nevű könyvtárt találhatunk.

Ezek mellett ha előállítottunk fejlesztői eszközkészletet is, akkor azt az `sdk` könyvtárban találjuk.

A beágyazott rendszer telepítéséhez általában az alábbi lépéseket kell végigcsinálnunk:

- A cél tároló eszközt (pl. egy CompactFlash kártya, amely majd a beágyazott eszköz háttértára lesz) meg kell partícionálnunk. Tipikusan egy partíciót kell létrehoznunk rajta és be kell állítanunk a Linux típusúra. (Az azonosítója 0x83.)
- El kell készítenünk az állományrendszert a partíción.
- Fel kell csatolnunk a partíciót.
- A `deploy/images` könyvtárban található tömörített állományrendszer állományt ki kell tömörítenünk a felcsatolt partícióra.
- El kell végeznünk a konfiguráció esetleges módosítását, és ezt követően a partíció lecsatolását.

Azonban egyes architektúrák esetében az image állomány komplett tároló eszköz (SD, CF) képet tartalmazhat. Ekkor a telepítés arra egyszerűsödik, hogy dd paranccsal rámásoljuk az image állományt a céleszközre.

A kernel telepítése függ az architektúrától:

- X86: A kernelt el kell helyeznünk az állományrendszerben és be kell konfigurálnunk a bootmanager-t. Ugyanakkor szükséges a bootmanager telepítése is, amely révén a tároló eszköz bootolható lesz.
- ARM, MIPS, stb.:
    - Belső flash tároló:
        - Az U-Boot vagy más hasonló program segítségével át kell töltenünk a céleszköz RAM-jába.
        - Törölnünk kell a szükséges helyet a flash-ben.

        - Át kell másolunk a RAM-ból a flash-be.
        - Be kell állítanunk a boot opciókat, illetve le kell mentenünk.
    - SD/MMC/CF kártyán való elhelyezése egy speciális partíción

Az előző példák csak néhány tipikus esetet mutatnak be. Az adott kártya esetében követendő metódust a hardver gyártó dokumentációjából tudjuk meg pontosan. Kivételt képez ez alól az x86 architektúra, amelynek a leírását a Yocto Project dokumentációjában találhatjuk.

Miután feltelepítettünk és beüzemeltünk a rendszert azt követően további telepítő csomagokkal bővíthetjük. A Yocto Project által előállított telepítő csomagot csak át kell vinnünk az eszközre valamelyik módszerrel (adathordozón, hálózaton, soros kábelen, stb.). Ezt követően a rendszerünkben használt csomag menedzsment eszközzel telepíthetjük is. A telepítés során az eszköz vizsgálja a függőségeket, és az esetleges ütközéseket is.

6.3.8 Keresztfordítás

Ha szeretnénk a saját programunkat lefordítani és nem akarunk receptet készíteni hozzá, akkor kézzel is elvégezhetjük a keresztfordítást. A `sysroots` könyvtár tartalmazza a szükséges programokat és a keresztfordító környezetet:

- Keresztfordító programok
- Fejlesztői könyvtárak (library)
- Header állományok
- Segédprogramok
- Konfigurációs állományok a fordításhoz

A fordító programok könyvtárneve függ a hoszt és a cél architektúrától. Például:

```
sysroots/x86_64-linux/usr/bin/i586-poky-linux/i586-poky-linux-gcc
```

Ha nem használunk semmi további library-t a libc-n kívül, akkor elég csak a fordító programot lefuttatnunk. Nem szükséges extra paraméterekkel megadni az elérési utakat, mivel az alapértelmezetten használja.

Ellenben ha további fejlesztői könyvtárakat is használunk, akkor paraméterekkel meg kell adnunk a fordításhoz a header könyvtárat, a linkeléshez pedig a library könyvtárat. Annak érdekében, hogy egyszerűbb legyen a paraméterezés, a rendszer tartalmazza a pkg-config segédprogramot, amely konfigurációs állományok alapján legenerálja a fordító és linkelő szükséges paraméterezését. A pkg-config az alábbi módon használható:

- Fordításhoz:

```
sysroots/x86_64-linux/usr/bin/pkg-config --cflags glib-2.0
```

- Linkeléshez:

```
sysroots/x86_64-linux/usr/bin/pkg-config --libs glib-2.0
```

Ha egyszerre akarjuk végezni a fordítást és a linkelést is, akkor a pkg-config által generált paraméterezést is összevonhatjuk.

7 Fejlesztő eszközök

Linux alatt a fejlesztőeszközök széles választéka áll rendelkezésünkre. Ezekből kiválaszthatjuk a nekünk megfelelőt, azonban néhány fontos eszközt mindenkinek ismernie kell.

A Linux-disztribúciók számtalan megbízható fejlesztőeszközt tartalmaznak, amelyek főként a Unix-rendszerekben korábban elterjedt eszközöknek felelnek meg. Ezek az eszközök nem nyújtanak barátságos felületet, a legtöbbjük parancssoros, felhasználói felület nélkül. Azonban sok éven keresztül bizonyították megbízhatóságukat, használhatóságukat, valamint számos új felhasználóbarát program a színfalak mögött továbbra is ezeket a parancssori eszközöket használja.

7.1 Fordító

A GNU Compiler Collection (`http://gcc.gnu.org/`) a C (*gcc*), C++ (*g++*), Objective-C, Fortran, Java (*GCJ*), Ada (*GNAT*) és a Go nyelvek fordítóit foglalja egy csomagba. Ezáltal az előbb felsorolt nyelveken írt programokat mind lefordíthatjuk a GCC-vel.

További nyelvekhez (Mercury, Pascal) is léteznek előtétfelületek (front-end), azonban ezeknek egy részét még nem integrálták a GCC hivatalos csomagjába.

Mi a továbbiakban a vizsgálódásainkat a C/C++ nyelvekre korlátozzuk.

A *gcc*-nek nincs felhasználói felülete. Parancssorból kell a megfelelő paraméterekkel meghívnunk. Használata számos paramétere ellenére szerencsére egyszerű, mivel általános esetben ezeknek a paramétereknek csak egy kis részére van szükségünk.

Használat előtt érdemes ellenőriznünk, hogy az adott gépen a *gcc* mely verzióját telepítették. Ezt az alábbi paranccsal tehetjük meg:

```
gcc -v
```

Ebből megtudhatjuk a *gcc* verzióját, továbbá a platformot, amelyre lefordították.

A program gyakran használt paraméterei:

Paraméter	Jelentés
-o fájlnév	A kimeneti állománynév megadása. Ha nem adjuk meg, akkor az alapértelmezett fájlnév *a.out* lesz.
-c	Fordítás, linkelés nélkül. A paraméterként megadott forrásállományból tárgykódú (object) fájlt készít.
-Ddefiníció=x	Definiálja a *definició* makrót x értékkel.

Paraméter	Jelentés
-Ikönyvtárnév	Hozzáadja a könyvtárnév paraméterben meghatározott könyvtárat ahhoz a listához, amelyben a .h kiterjesztésű (header) állományokat keresi.
-Lkönyvtárnév	Hozzáadja a *könyvtárnév* paraméterben meghatározott könyvtárat ahhoz a listához, amelyben a fejlesztői könyvtár (library) állományokat keresi.
-llibrary	A programhoz hozzákapcsolja a *library* nevű programkönyvtár metódusait.
-static	Az alapértelmezett dinamikus linkelés helyett a fordító a statikus programkönyvtárakat linkeli a programba, ha azok rendelkezésre állnak.
-g, -gN, *-ggdb, -ggdbN*	A lefordított állományt ellátja a hibakereséshez (debug) szükséges információkkal. A *-g* opció megadásával a fordító a szabványos hibainformációkat helyezi el. A *-ggdb* opció arra utasítja a fordítót, hogy olyan további információkat is elhelyezzen a programban, amelyeket csak a *gdb* hibakereső értelmez. A paraméter végén megadhatunk egy szintet (N) 0 és 3 között. 0: nincs hibakeresési információ, 3: extra információk is belekerülnek. Az alapértelmezett a 2-es szint.
-O, -ON	Optimalizálja a programot az N optimalizációs szintnek megfelelően. Az optimalizáció során kisebb/gyorsabb kód előállítására törekszik a fordító. A szint 0-tól 3-ig választható meg. 0 esetén nincs optimalizáció. Az alapértelmezett 1-es szint esetén csak néhány optimalizációt végez a *gcc*. A leggyakrabban használt optimalizációs szint a 2-es.
-Wall	Az összes gyakran használt figyelmeztetést (warning) bekapcsolja. A csak speciális esetben hasznos figyelmeztetéseket külön kell bekapcsolni.

Példaként nézzünk végig néhány esetet a *gcc* program használatára.

Egy tetszőleges szövegszerkesztőben elkészítettük a már jól megszokott kódot:

```
/* hello.c */

#include <stdio.h>

int main()
{
```

```
  printf("Hello vilag!\n");
  return 0;
}
```

Ezután kipróbáljuk jól sikerült programunkat. Ehhez az alábbi paranccsal fordíthatjuk le:

```
gcc -o hello hello.c
```

Ha nem rontottuk el a kódot, akkor a fordító hibaüzenet nélkül lefordítja a forrást, és a programra a futtatási jogot is beállítja. Hibátlan kód esetén a fordító nem ír ki semmilyen üzenetet. Innen tudhatjuk, hogy jól dolgoztunk.

Ezután már csak futtatnunk kell a programot:

```
./hello
```

A programunk fut, és a konzolon megjelenik a következő felirat:

```
Hello vilag!
```

Vagyis meghívtuk a *gcc* programot, amely lefordította (compile) a kódot, majd meghívta az *ld* nevű linker programot, amely a lefordított kódunkhoz hozzáfűzte a fejlesztői könyvtárakból a felhasznált függvényeket, és létrehozta a futtatható bináris állományt.

Ha csak tárgykódú állományt (object file) akarunk létrehozni (melynek kiterjesztése *.o*), vagyis ki szeretnénk hagyni a linkelés folyamatát, akkor a *-c* kapcsolót használjuk a *gcc* program paraméterezésekor:

```
gcc -c hello.c
```

Ennek eredményeképpen egy *hello.o* nevű tárgykódú fájl jött létre. Ez az állomány gépi kódban tartalmazza az általunk írt programot. Ezt természetesen linkelnünk kell még, hogy egy kész futtatható binárist kapjunk:

```
gcc -o hello hello.o
```

A *-o* kapcsoló segítségével tudjuk megadni a linkelés eredményeként létrejövő futtatható fájl nevét. Ha ezt elhagyjuk, alapértelmezésben egy *a.out* nevű állomány jön létre.

A következőkben megvizsgáljuk azt az esetet, amikor a programunk több (esetünkben kettő) forrásállományból áll. Az egyik tartalmazza a főprogramot:

```
/* sincprg.c */

#include <stdio.h>

double sinc(double);

int main()
{
  double x;
```

```
  printf("Kerem az x-et: ");
  scanf("%lf", &x);
  printf("sinc(x) = %6.4f\n", sinc(x));
  return 0;
}
```

A másik implementálja a függvényt:

```
/* sinc.c */

#include <math.h>

double sinc(double x)
{
  return sin(x)/x;
}
```

A teljes program lefordítása fordítása:

```
gcc -o sincprg sincprg.c sinc.c -lm
```

Így a *sincprg* futtatható fájlhoz jutunk. Ugyanezt megoldhattuk volna több lépésben is:

```
gcc -c sincprg.c
gcc -c sinc.c
gcc -o sincprg sincprg.o sinc.o -lm
```

Az *-lm* kapcsolóra azért van szükségünk, hogy a *sin()* függvényt tartalmazó matematikai programkönyvtárat hozzáfűzzük a programunkhoz. Ha ezt elmulasztjuk, akkor egy hibaüzenetet kapunk a fordítótól, hogy nem találja a *sin()* függvény referenciáját. A *-l* kapcsoló általában arra szolgál, hogy egy programkönyvtárat hozzáfordítsunk a programunkhoz. A Linux-rendszerhez számos szabványosított programkönyvtár tartozik, amelyek a */lib*, illetve a */usr/lib* könyvtárakban találhatóak. Amennyiben a felhasznált programkönyvtár máshol található, az elérési útvonalat meg kell adnunk a *-L* kapcsolóval:

```
gcc prog.c -L/home/myname/mylibs mylib.a
```

Hasonló problémánk lehet a *.h*-kiterjesztésű állományokkal (header file). A rendszerhez tartozó header állományok alapértelmezetten a */usr/include* könyvtárban (illetve az innen kiinduló könyvtárstruktúrában) találhatóak. Így amennyiben ettől eltérünk, a *-I* kapcsolót kell használnunk a saját header útvonalak megadásához:

```
gcc prog.c -I/home/myname/myheaders
```

Ez azonban ritkán fordul elő, ugyanis a C programokból általában az aktuális könyvtárhoz viszonyítva adjuk meg a saját *.h*-állományainkat, vagyis relatív útvonal megadást használunk.

```
gcc prog.c -I../myheaders
```

A C előfeldolgozó (preprocessor) másik gyakran használt funkciója a *#define* direktíva. Ezt is megadhatjuk közvetlenül parancssorból. A

```
gcc –DMAX_ARRAY_SIZE=80 prog.c -o prog
```

hatása teljes mértékben megegyezik a *prog.c* programban elhelyezett

```
#define MAX_ARRAY_SIZE 80
```

preprocesszor direktívával. Ennek a funkciónak gyakori felhasználása a

```
gcc -DDEBUG prog.c -o prog
```

paraméterezés. Ilyenkor a programban feltételes fordítási direktívákat helyezhetünk el a hibakeresés megkönnyítésére:

```
#ifdef DEBUG
  printf("A szal elindult.");
#endif
```

A fenti esetben látjuk az új szálak indulását debug üzemmódban, amikor a *DEBUG* makrót definiáljuk. Viszont ez az információ szükségtelen a felhasználó számára egy már letesztelt program esetén, ezért ebben az esetben újrafordítjuk a programot *DEFINE* nélkül.

7.2 A programkönyvtárak alapfogalmai

A programkönyvtár (program library) olyan lefordított (compiled) forráskódú programok gyűjteménye, amelyeket nem közvetlen futtatásra, hanem más programok összeállítása során lehetséges későbbi felhasználás céljából készítettek. Ilyen például a libc, amely a tárgyalt függvények többségét tartalmazza, például a C nyelv szabványos függvényeinek jó részét, a rendszerhívásokat stb. Linux alatt háromfajta programkönyvtár létezik:

- statikus könyvtárak (static libraries)
- megosztott könyvtárak (shared libraries)
- dinamikusan betöltött könyvtárak (Dynamically Loaded Libraries) – a továbbiakban DL.

Linux alatt is használatos a dinamikus kapcsolású programkönyvtár (Dynamically Linked Libraries – DLL), amelynek jelentése nem egyértelmű: van, aki az utolsó két típus valamelyikére, illetve mindkettőre alkalmazza a DLL kifejezést, de mivel a dinamikusan betöltött könyvtárak a megosztott könyvtáraktól lényegében csak használatuk módjában különböznek, ez nem okoz különösebb félreértést. A programkönyvtár fajtájának megkülönböztetésére a Linux a következő táblázatban látható konvenciót alakította ki.

Utótag	Típus
`.a`	statikus könyvtár
`.so.x.y.z`	megosztott könyvtár x fő, y mellék verziószámmal és z kibocsátásszámmal

`.sa`	a.out formátumú megosztott könyvtár

Gyakran szükségünk van arra, hogy egy adott program milyen könyvtárakat használ. Nézzük meg például a lynx programot:

```
ldd /usr/bin/lynx
```

Ennek kimenetét a használt könyvtárak alkotják:

```
libz.so.1 => /usr/lib/libz.so.1 (0x40027000)
libncurses.so.5 => /usr/lib/libncurses.so.5 (0x40036000)
libssl.so.2 => /lib/libssl.so.2 (0x40074000)
libcrypto.so.2 => /lib/libcrypto.so.2 (0x400a1000)
libc.so.6 => /lib/i686/libc.so.6 (0x42000000)
libdl.so.2 => /lib/libdl.so.2 (0x40167000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Ezzel a segédprogrammal meg tudjuk állapítani, hogy mely könyvtárakat fog a lynx program betölteni futási időben.

A könyvtárak elhelyezkedését illetően a következő konvenciók alakultak ki:

- A rendszer által használt programkönyvtárak a /usr/lib könyvtárban találhatók, azok, amelyek a rendszer indulásához szükségesek, a /lib könyvtárban vannak, végül azok, amelyek nem részei a rendszernek, a /usr/local/lib könyvtárban helyezkednek el.
- Ha programkönyvtárunk olyan programokat hív meg, amelyeket csak program-könyvtárakon keresztül lehet hívni, azt a /usr/local/libexec könyvtárba helyezzük el. Az XWindow rendszer által használt programkönyvtárak a /usr/X11R6/lib könyvtárban vannak.

7.2.1 Statikus programkönyvtárak

Történetileg ez a könyvtártípus jelent meg legelőször. Gyakorlatilag lefordított (compiled) tárgykódú (object) fájlok gyűjteménye. Eredetileg az volt a céljuk, hogy gyorsabbá tegyék a fordítást, de ez ma már nem jelent különösebb előnyt a gyors fordítóprogramok miatt. Amennyiben valaki nem szeretné kiadni a forráskódot, a lefordított kódot statikus könyvtárak formájában is rendelkezésre bocsáthatja.

Felhasználásuk a gcc fordító -l paraméterével lehetséges. Célszerű minden lib fájlhoz egy vagy több fejállományt készíteni a fordító számára. Így használatkor az #include direktíva segítségével megadjuk a prototípust, majd a linker számára elérhetővé tesszük a .a utótagú könyvtárfájlt, majd a paraméterezésről sem feledkezünk meg.

A gcc fordító alapértelmezése az, hogy ha talál .so kiterjesztésű megosztott könyvtárat, akkor azt kapcsolja a programba, ellenkező esetben a statikusat. A szabványos könyvtárak általában mindkét formátumban rendelkezésre állnak.

7.2.2 Megosztott programkönyvtárak

Ezek a könyvtárak a program indulásakor töltődnek be, és a futó programok osztoznak rajtuk, memóriát és lemezterületet takarítva meg ezzel. Számos lehetőséget nyújtanak:

- újabb verziók installálása, (ugyanakkor a programok a régi verziót is használhatják);
- már felinstallált programkönyvtárak egyes részeinek, függvényeinek átdefiniálása. Ez megtehető az alatt, amikor a futó programok már létező könyvtárakat használnak.

A megosztott könyvtárak könnyű használatához nagyon fontos a konvenciók betartása, erre a tárgyaláskor mi is nagy hangsúlyt helyezünk.

7.2.3 Elnevezési szintek

Sokszor előfordul, hogy csak a programkönyvtár belső működésén változtattunk, és a változások nem érinthetik a programkönyvtárat használó programokat sem fordítási, sem pedig felhasználói szinten. Ha az elosztott könyvtárakra az őket használó programokból fizikai fájlnévvel hivatkozunk, lehetetlenné válna a verziószámok változtatása. Ugyanez a probléma fennáll a fejlesztők oldalán is: folyton újra kellene paraméterezni a gcc-t, ha egy újabb verziót állítunk elő.

Ezért egy adott megosztott programkönyvtárnak három – a konvenció szerint különböző – neve lehet.

- Fizikai név – a könyvtárat tartalmazó fájl neve a fájlrendszerben.
- Az úgynevezett sonév (soname) – erre a programkönyvtár betöltésekor hivatko-zunk
- Linkernév – a fordításkor használjuk a gcc paramétereként

A sonév a konvenció szerint lib előtaggal kezdődik, amit a név után .so követ, végül pont után a fő verziószám, ami csak az interfésszel (a benne lévő függvények prototípusaival vagy az osztályok deklarációival) változik.

A fizikai nevet a sonévből képezzük, úgy, hogy hozzáadunk egy pontot, a mellék verziószámot, egy újabb pontot és végül a kibocsátás (release) számát.

A linkernév a sonévből keletkezik, ha elhagyjuk a fő verziószámot.

Ez a hierarchia teszi lehetővé, hogy a fizikai fájlnévtől függetlenül hivatkozzunk egy könyvtárra, valamint a fizikai névtől függetlenül használjuk a fordításhoz. Ha a fizikai névre nincs közvetlen programbeli hivatkozás, a programkönyvtár bármikor frissíthető. A programok belül a sonevekkel hivatkoznak a könyvtárakra. Ezért alakult ki az a konvenció, hogy az azonos interfészt használó programkönyvtárak azonos sonevekkel rendelkeznek.

7.3 Make

A Unix-os fejlesztések egyik oszlopa a *make* program. A Linux rendszereken a *GNU make* terjedt el (`http://www.gnu.org/software/make/`). Ez az eszköz lehetővé teszi, hogy a programunk fordításának menetét könnyen leírjuk és automatizáljuk. Nemcsak nagy programok esetén, hanem akár egy forrásállományból álló programnál is egyszerűbb a

fordítás, ha csak egy *make* parancsot kell kiadnunk az aktuális könyvtárban a fordító paraméterezése helyett.

További szolgáltatása a *make* programnak, hogy egy sok forrásállományból álló program fordítása esetén, csak a módosított állományokat fordítja újra, illetve azokat, amelyekre ezek hatással lehetnek.

Ahhoz, hogy mindezeket a funkciókat megvalósíthassa a *make* egy ún. *Makefile*[12]-ban le kell írnunk a programunk fordításának menetét, a forrásállományok megadásával. Lássunk erre egy példát:

```
 1: # Makefile
 2:
 3: objs = sincprg.o sinc.o
 4: libs = -lm
 5:
 6: sincprg: $(objs)
 7:         gcc –o sincprg $(objs) $(libs)
 8:
 9: install: sincprg
10:         install –m 644 sincprg /usr/local/bin
11:
12: .PHONY: install
```

Az **1.** sorban egy **megjegyzést** (comment) láthatunk. A Unix-os tradíciók alapján a megjegyzést # karakterrel jelöljük.

A **3.** sorban definiáljuk az *objs* **változót**, melynek értéke: *sincprg.o sinc.o*

A **4.** sorban ugyanezt tesszük a *libs* **változóval**. A későbbiekben ezt fogjuk majd felhasználni a linkelési paraméterek beállítására.

A **6.** sorban egy **szabály** (rule) megadását láthatjuk. Ebben a *sincprg* állomány az *objs* változó értékeitől függ. A *sincprg* állományt hívjuk itt a szabály **cél**jának (target) és a *$(objs)* adja a **függőségi listát** (dependency list). Megfigyelhetjük a változó használatának módját is.

A **7.** sorban egy parancssor található, de ez több soros is lehet. Azt mondja el, hogy hogyan készíthetjük el a **célobjektumot** a függőségi lista elemeiből. Itt állhat több parancssor is szükség szerint, azonban minden sort TAB karakterrel kell kezdeni.

A **9.** sornak speciális a célja. Ebben a szabályban valójában nem állomány létrehozása a célunk, hanem az installációs művelet megadása.

A **10.** sorban végezzük el a programunk telepítését az *install* program meghívásával bemásoljuk az */usr/ local/bin* könyvtárba.

[12] Ha alkalmazzuk a *make –f fájlnév* szintaxist, akkor a *make* a megadott fájlt dolgozza fel az alapértelmezett *Makefile* állománynév helyett.

A **12.** sor egy problémának a megoldását rejti. A 9. sorban a cél nem állomány volt, hanem egy parancs. Azonban, ha mégis létezik egy *install* nevű állomány, és az frissebb, mint a függőségi listában szereplő *sincprg* állomány, akkor nem fog lefutni a szabályunk. Ezt küszöböljük ki a *.PHONY* kulcsszóval, amely módosítja a *make* működését. Ebben az esetben megadhatjuk vele, hogy az *install* cél esetén ne figyelje az állomány létét, hanem minden esetben hajtsa végre a szabályt, ha kérjük.

Általánosan megfogalmazva a *Makefile*-ok ötféle dolgot tartalmazhatnak. Ezek:

- megjegyzések
- explicit szabályok
- implicit szabályok
- változódefiníciók
- direktívák

7.3.1 Megjegyzések

A megjegyzések magyarázatul szolgálnak, a *make* program gyakorlatilag figyelmen kívül hagyja őket. A # karakterrel kell kezdődniük.

7.3.2 Explicit szabályok

A **szabály** (rule) azt határozza meg, hogy mikor és hogyan kell újrafordítani egy vagy több állományt. Az így keletkező állományokat a szabály **cél**jának vagy **tárgy**ának (target) nevezzük.[13] Hogy egy állomány létrejöjjön, általában más állományokra is szükség van. Ezek listáját nevezzük **függőségi listá**nak, **feltételek**nek, vagy **előkövetelmény**nek. Például:

```
foo.o: foo.c defs.h
      gcc -c -g foo.c
```

A fenti példában a szabály tárgya a *foo.o* fájl, az előkövetelmény a *foo.c,* illetve a *foo.h* állomány. Mindez azt jelenti, hogy szükség van a *foo.c* és a *defs.h* állományokra, továbbá a *foo.o* fájlt akkor kell újrafordítani, ha

- a *foo.o* fájl nem létezik,
- a *foo.c* időbélyege későbbi, mint a *foo.o* időbélyege,
- a *defs.h* időbélyege későbbi, mint a *foo.o* időbélyege.

Azt, hogy a *foo.o* fájlt, hogyan kell létrehozni, a második sor adja meg. A *defs.h* nincs a *gcc* paraméterei között: a függőséget egy – a *foo.c* fájlban található – *#include „defs.h”* C nyelvű preprocesszor direktíva jelenti.

A szabály általános formája:

```
TARGY: ELOKOVETELMENYEK; RECEPT
      RECEPT
```

[13] Azonban mint láthattuk a cél nem minden esetben állomány létrehozása.

```
...
```

A *TARGY* egy állománynév. Az *ELOKOVETELMENYEK* fájlnevek szóközzel elválasztva. A fájlnevek tartalmazhatnak speciális jelentésű, ún. helyettesítő karaktereket (wildcard characters), mint például a „.” (aktuális könyvtár), „*” vagy „%” (tetszőleges karaktersorozat), „~” (*home* könyvtár). A *RECEPT* szerepelhet vagy az *ELOKOVETELMENYEK*-kel egy sorban pontosvesszővel elválasztva, vagy a következő sorokban, amelyek mindegyikét egy **TAB karakterrel kell kezdeni**.[14] A parancs sorozat végét egy olyan sor jelzi a *make*-nek, amelynek az elején nincs TAB karakter.

Mivel a $ jel már foglalt a változókhoz, ezért ha a parancsban $ jel szerepelne, akkor helyette $$t kell írnunk.

Ha egy sor végére „\” jelet teszünk, a következő sor az előző folytatása lesz, teljesen úgy, mintha a második sort folytatólagosan írtuk volna. Erre azért van szükség, mert a *make* minden parancssort külön subshell-ben futtat le. Ezáltal például a *cd* parancs hatása is csak abban a sorban érvényesül, ahova írjuk. Például:

```
cd konyvtar; \
     gcc –c –g foo.c
```

Ha a *make* programnak paraméterként megadjuk egy tárgy nevét, akkor az a szabály hajtódik végre, amely kell a tárgy előállításához. A *make* felépít egy függőségi fát, amelyet bejárva ellenőrzi, hogy mely részeket kell újrafordítani. Majd elvégzi a szükséges lépéseket, hogy előállítsa az aktuális célt.

Ha a *make* programot paraméterek nélkül futtatjuk, akkor automatikusan az első szabály hajtódik végre, valamint azok, amelyektől valamilyen módon függ.

A *make* program használatánál ügyeljünk arra, hogy mivel a lépések szükségességét a tárgy és a függőség állományok időbélyege alapján vizsgálja, ezért ha az idő beállítások megkavarodnak, akkor nem működik jól. Ilyen helyzet leggyakrabban akkor fordul elő, ha a fejlesztés során gépet váltunk és a két számítógép órája nem jár szinkronban. Természetesen az idő visszaállítása is okozhat ilyen problémát. Ilyenkor célszerű az időbélyeget aktualizálni a *touch* paranccsal és a köztes állományok letörlésével újra fordítani az egész projektet.

7.3.3 Hamis tárgy

Gyakran előfordul, hogy valamilyen műveletre egy szabályt készítünk, amelynek a recept része tartalmazza a végrehajtandó parancsokat. A szabály célja ilyenkor nem egy előállítandó állomány, hanem csak egy névként funkcionál, amellyel a szabályra hivatkozunk. Az ilyen tárgyakat hívjuk hamis tárgynak.

Azonban ha tényleg valamelyik művelet létrehozná a tárgyként megadott állományt és az frissebb lenne a feltételeknél, akkor a szabályhoz tartozó recept parancsai egyáltalán nem hajtódnak végre. Gyakori, hogy nincs megadva feltétel a hamis szabályhoz. Ilyenkor egyáltalán nem hajtódik végre a szabály ha a tárgy állomány létezik.

[14] Ügyeljünk arra, hogy sok szövegszerkesztő hajlamos a TAB gomb lenyomására space karaktereket elhelyezni a szöveg állományba, ami később hibát eredményez. Újabban azonban sok szövegszerkesztő az állomány nevéből detektálja, hogy Makefile-t készítünk. Ebben az esetben azonban az állomány név megadásával célszerű kezdenünk.

Azért hogy elkerüljük a korábban vázolt problémákat a *.PHONY* kulcsszóval egyértelműen deklarálnunk kell a hamis célokat. Például:

```
.PHONY: clean

clean:
	rm *.o prg
```

Ha egy könyvtárban több program forrása is szerepel, akkor gyakran készítünk egy hamis szabályt „all” néven, ami mindent lefordít és előállít. És hogy ez az alapértelmezett szabály is legyen egyben, ezért első szabályként írjuk meg. Például:

```
all: prg1 prg2

.PHONY: all

prg1: prg1.o
	gcc –o prg1 prg1.o

prg2: prg2.o
	gcc –o prg2 prg2.o
```

7.3.4 Változódefiníciók

Mint az első példában is láthattuk, gyakran előfordul, hogy egyes állományneveknek több helyen is szerepelniük kell. Ilyenkor, hogy könnyítsük a dolgunkat, **változó**kat használunk. Ez esetben elegendő egyszer megadnunk a listát, a többi helyre már csak a változót helyezzük. A változók használatának másik célja, hogy a Makefile-unkat rendezettebbé tegyük, megkönnyítve ezzel a dolgunkat a későbbi módosításoknál.

A Makefile-ban a változók típusa mindig szöveg. A változók megadásának szintaxisa:

```
VALTOZO = ERTEK
```

Erre a változóra később a

```
$(VALTOZO)
```

vagy a

```
${VALTOZO}
```

szintaxissal hivatkozhatunk.

A változó neve nem tartalmazhat „:”, „#”, „=”, és semmilyen üres mező karaktert. Azonban a gyakorlatban még szűkebb halmazt használunk: betűk, számok, és a „_” karaktert. A változók nevénél a rendszer figyeli a kis és nagybetűket. A tradíció az, hogy a változó nevét csupa

nagybetűkkel írjuk. Azonban jelenleg gyakrabban kis betűs neveket használunk a Makefile-ban és a nagybetűs nevek inkább rendszer funkciókra vannak fenntartva. Egyes nagybetűs változók felüldefiniálásával módosíthatjuk a rendszer működését. Emellett néhány karakter kombinációnak speciális jelentése van. Ezekkel az automatikus változókról szóló fejezetben találkozunk.

Lehetőségünk van arra, hogy a változók értékének megadásakor más változók értékeit is felhasználjuk. Például:

```
prgs = $(prg1) $(prg2)
```

Azonban van egy olyan tulajdonsága a rendszernek, amelyre figyelnünk kell. **A változók kiértékelés a felhasználás helyén történik.** Azokat az értékeket veszi figyelembe a rendszer, amelyeket addig definiáltunk és szükség szerint **rekurzívan kifejti**. Például:

```
objs = $(obj1) $(obj2)
obj1 = elso.o
obj2 = masodik.o


prg: $(objs)

```

Amikor a „prg” szabályt meghívjuk, akkor a feltételeknél kiértékeli a program az „objs” változót. Ennek értéke „$(obj1) $(obj2)”, kiértékeli mindkét változót, így megkapjuk az „elso.o masodik.o” szöveget, amelyet behelyettesít.

Azonban ennek a megoldásnak van egy másik oldala is. A változók használata során kerülnünk kell a következő megoldást:

```
objs = elso.o
objs = $(objs) masodik.o
```

Reflexből arra számítanánk, hogy szekvenciálisan hajtódnak végre a műveletek és a végeredményként az „elso.o masodik.o” szöveg helyettesítődik majd be a „$(objs)” helyére. Ezzel szemben egy végtelen ciklust idézünk elő és a behelyettesítés helyén végtelen „masodik.o” szöveg lesz, mivel ez az utoljára beállított érték.

7.3.5 A változó értékadásának speciális esetei

Láthatóan a felhasználás helyén elvégzett rekurzív kiértékelés csapdákat rejt magában. Erre megoldás az **egyszerű kiértékelés**, amit a definiálás helyén végez el a rendszer. Ennek jele a „:=” karakterkombináció. A működést a következő példa szemlélteti:

```
a := egy
b := $(a) ketto
a := harom
```

Ez egyenlő azzal, mintha a következőket írtuk volna.

```
b := egy ketto
```

```
a := harom
```

Az előzőekben említett hibás példára is van helyes megoldás. Vagyis megoldható, hogy a **változó értékéhez hozzáadjuk** további szövegeket. Ezt a „+=” jellel tehetjük meg. A korábban látott példa helyesen:

```
objs = elso.o
objs += masodik.o
```

Így az „objs” változó használatakor az eredmény nem végtelen ciklus lesz, hanem az „elso.o masodik.o” szöveg.

Az eddig látottak mellett van lehetőségünk **feltételes értékadás**ra is. Ebben az esetben a változónak csak akkor adunk értéket, ha még nincs definiálva. Ennek jele a „?=”. Az alábbi példán nézzük meg a működését:

```
a = egy
a ?= ketto
```

Ez egyenlő a következővel:

```
a = egy
```

Bár ebben a példában nem látszik közvetlenül az eszköz haszna, azonban egy összetett, elágazásokat vagy további állományokat tartalmazó *Makefile* esetén nagyon hasznos tud lenni.

7.3.6 Több soros változók definiálása

A *define* direktíva használatával is adhatunk a változóknak értéket. Szintaxisa lehetővé teszi, hogy az érték új sor karaktereket is tartalmazzon:

```
define ket-sor =
echo $(a)
echo ketto
endef
```

A *define* szót a változó neve követi. Ezt követi a művelet. Az alapértelmezett a „=” jel, amit el is hagyhatunk. Azonban használható a „+=” jel is, ezáltal bővíthetjük a változó korábbi értékét. A műveleti jel után mást már nem írhatunk az első sorba. Az értékek a következő sorokban következnek és a blokkot az *endef* szó zárja. Az *endef* előtt szereplő újsor karakter már nem számít bele az értékadásba.

Láthatóan az értékadásnál hivatkozhatunk más változókra is, ahogy a korábbi esetekben is.

Szükség esetén meg is szüntethetjük a definiált változót az *undefine* kulcsszóval.

```
undefine ket-sor
```

7.3.7 A változó hivatkozásának speciális esetei

A **helyettesítő hivatkozás** révén lehetőségünk van arra is, hogy a változóra való hivatkozás során, az értékét módosítva helyettesítsük be. Ilyenkor egy konverziós szabályt

definiálhatunk. A konverzió során az értéket szavanként kezeljük, és a szavak végén található szöveg részleteket cseréljük le más karaktersorozatokra. Nézzünk egy példát:

```
srcs = elso.c masodik.c
objs := $(srcs:.c=.o)
```

Ebben a példában az „objs” változó értéke „elso.o masodik.o” lesz. Vagyis egy szó listából könnyen előállíthatunk módosított listákat is. A helyettesítő hivatkozás hivatalos formátuma:

```
$(valtozo:a=b)
```

vagy

```
${valtozo:a=b}
```

Ahol a „valtozo” egy változó, amely szavakból álló listát tartalmaz. A művelet minden szó végén lecseréli az „a” szövegrészletet „b” szövegre.

Egy másik speciális változó hivatkozásfajta a **számított változónevek** használata. Ebben az esetben a hivatkozott változó nevét egy másik változó tartalmazza. Egy példán demonstrálva:

```
a = b
b = c
eredm := $($(a))
```

A „$(a)” értéke „b”. Ezt behelyettesítve a „$($(a))” kifejezésbe „$(b)”-t kapunk. Ennek értéke „c”. Vagyis az „eredm” változó értéke „c” lesz.

A számított változóneveket a helyettesítő hivatkozásokkal is kombinálhatjuk.

```
src_1 := elso.c
src_2 := masodik.c

objs := $(src_$(a):.c=.o)
```

Az „a” változó értékétől függően vagy az „src_1” vagy az „src_2” értékéből állítjuk elő az „objs” értékét, és ennek során a „.c” kiterjesztést „.o”-ra cseréljük.

7.3.8 Automatikus változók

A receptek készítésénél gyakran sokat segít, ha a forrás és cél állományok nevét nem kell minden esetben beírnunk, hanem a szabály céljából és feltételeiből generálhatjuk. Ez az eddig tárgyalt esetekben könnyebbséget hoz, azonban több később tárgyalt esetben nélkülözhetetlen eszköz. Általános szabályokat nem tudunk úgy definiálni, hogy az állományneveket ne generálnánk, hiszen ezekben az esetekben nem tudunk konkrét állományokat megadni.

Az állománynevek előállítását az **automatikus változók**kal végezzük. Az alábbi táblázat foglalja össze az automatikus változókat a jelentésükkel.

Aut. változó	Jelentés
$@	A cél állomány neve.
$<	A függőségi lista első elemének neve.
$?	Az összes olyan feltételfájl neve (sorban, szóközzel elválasztva), amely frissebb a célállománynál.
$^	Az összes feltétel fájl neve szóközzel elválasztva.
$+	Jelentése nagyrészt egyezik az előzővel, azonban ha a feltételek közt többször szerepel a fájl, akkor azt ugyanúgy több példányban tartalmazza.
$*	A célfájl nevének utótag nélküli része.

A következő fejezetekben találkozunk az alkalmazási területekkel is.

7.3.9 Többszörös cél

Egy szabály megadásakor nem csak egy tárgyat, hanem tárgyak listáját is megadhatjuk szóközökkel elválasztva. Ebben az esetben minden tárgyra külön-külön végrehajtódik a szabály. Vagyis olyan, mintha ugyanazt a szabályt lemásoltuk volna annyiszor, ahány célállományunk van és mindegyik szabály tárgyához beírnánk egy-egy célállományt. Természetesen ennek csak úgy van értelme, hogy ha nem pontosan ugyanazt a parancsot hajtjuk végre minden esetben. Az ellentétet, hogy a recept egyforma, de a végrehajtott parancsok mégis különbözőek legyenek, az automatikus változókkal tudjuk feloldani.

Nézzünk egy példát, ahol egy forrás állományból két különböző programot fordítunk eltérő paraméterezéssel:

```
debug_flags := -g3 -O0
release_flags := -g0 -O3
prgs := debug release

all: $(prgs)

.PHONY: all

$(prgs): hello.c
	gcc $($(@)_flags) -o $@ $<
```

A *make* parancs kiadása után két fordítás történik. Létrejön egy „debug” és egy „release” nevű állomány.

```
gcc -g3 -O0 -o debug hello.c
gcc -g0 -O3 -o release hello.c
```

A két fordítás eltérő paraméterezéssel történik. A „$($(@)_flags)” kifejezésben láthatunk egy automatikus változót „$(@)”, amely az aktuális tárgy értékét veszi fel. Ezt a „_flags” szöveggel kiegészítve egy másik változó nevét adja ki, amit felhasználva beillesztjük annak értékét. A létrehozott állomány neve a tárgy („$@”), míg a forrás a feltétel lista első eleme („$<”).

7.3.10 Minta szabályok

Az előző fejezet példájában láthattuk, hogyan tudunk több állományt is előállítani ugyanazon recept alapján. Azonban ha megfigyeltük a módszernek van egy kellemetlen megkötése. Az előkövetelmény lista közös. Vagyis csak olyan esetekben működik, amikor minden célállomány előállításához ugyanazokra a forrás állományokra van szükség.

A problémánkat a **statikus minta szabály**ok oldják meg, amelyek lehetővé teszik, hogy a cél állományok nevéből generáljuk a feltételeket. Ennek általános alakja:

```
TÁRGYAK: TÁRGY-MINTA: ELŐKÖVETELMÉNYEK-MINTÁJA; RECEPT
	RECEPT
	…
```

Hasonlóan az egyszerű szabály megadáshoz a tárgyak listájánál használhatunk helyettesítő karaktereket. A TÁRGY-MINTA és az ELŐKÖVETELMÉNYEK-MINTÁJA megadja, hogyan számítható ki egy tárgy nevéből a hozzá tartozó előkövetelmények neve. A minták egy helyen tartalmazzák a „%” karaktert. A tárgy nevére illesztve a tárgy mintáját a „%” helyére eső szöveget a rendszer az előkövetelmény mintájának „%” karakterrel jelölt helyére illeszti és így előállít egy új feltételt.

A könnyebb érthetőség kedvéért egy példán reprezentáljuk a statikus minta szabályok használatát:

```
objs := elso.o masodik.o

all: $(objs)

$(objs): %.o: %.c
	gcc –c –o $@ $<
```

Ez a szabály az alábbi parancsok végrehajtását fogja eredményezni:

```
gcc –c –o elso.o elso.c
gcc –c –o masodik.o masodik.c
```

Vagyis a szabály meghívódik először az „elso.o” céllal. A minták alapján ebben az esetben az előkövetelmény az „elso.c” állomány lesz. A fordítás során a parancssorba behelyettesítődik a cél állomány neve és az előkövetelmény lista első eleme mint forrás. Ez eredményezi az első parancssort. Majd ezt követően a „masodik.o” célra is végrehajtódnak ezek a műveletek. Ez eredményezi a második parancssort.

A statikus minta szabálynak létezik egy általánosított formája, amit egyszerűen **minta szabály**oknak nevezünk. Míg a statikus minta szabályoknál megadtunk egy tárgy listát és a szabály csak azokra az esetekre érvényes, addig az általános minta szabályoknál ezt elhagyjuk és amolyan alapértelmezett szabályt alkotunk. Ez a szabály azonban csak akkor lesz érvényes, ha a rendszer nem talál egy specifikus szabályt az adott cél állomány előállítására. Az általános minta szabály szintaxisa:

```
TÁRGY-MINTA: ELŐKÖVETELMÉNYEK-MINTÁJA; RECEPT
     RECEPT
     …
```

Az előző példánál maradva:

```
objs := elso.o masodik.o

all: $(objs)

%.o: %.c
     gcc -c -o $@ $<
```

A hatás látszólag ugyanaz, mint a statikus minta szabály esetében. Azonban ebben az esetben egy általános szabályt fogalmaztunk meg, amely leírja, hogy a „.o” végű állományok előállításához szükség van egy „.c” végű állományra, illetve leírja az előállítás metódusát. Vagyis minden további „.o” végű állomány előállítására is működni fog ez a szabály.

7.3.11 Klasszikus ragozási szabályok

A minta szabályok elődje a ma már háttérbe szorult **ragozási szabály**ok. A *make* programnak megadható az utótagok listája, illetve hogy az egyik utótaggal rendelkező állományból hogyan állítható elő egy másik utótaggal rendelkező állomány. Hasonlóan a minta szabályokhoz ezek a szabályok is csak akkor jutnak árvényre, ha az adott célállomány előállítására nincs specifikus szabály.

Példaként nézzük egy ragozási szabály megadást:

```
.c.o:
     $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<

.SUFFIXES: .c .o
```

Ez a példa a „.c” kiterjesztésű forrásállományokból a hozzájuk tartozó „.o” kiterjesztésű állományok előállítására fogalmaz meg általános szabályt. A példából látható, hogy az

utótagokat a *.SUFFIXES:* speciális tárgy után fel kell sorolnunk, majd páronként adhatjuk meg, hogy hogyan áll elő az egyik utótagú állományból a másik, jelen esetben a .c kiterjesztésű C forrásfájlból a „.o” kiterjesztésű tárgykódú állomány. A ragozási szabályok általános alakja:

```
FsCs:
RECEPT
      ...
```

Az **Fs** a forrás utótag, a **Cs** a cél utótag. A forrásból a cél előállításához a *make* a RECEPT sorokban megadott utasításokat hajtja végre. A ragozási szabályoknak nem lehetnek további előkövetelményei, csak az az egy, ami a generálódik.

Láthatóan a ragozási szabályok kötöttebbek, mint a minta szabályok, ezért mostanában az utóbbiakat használjuk.

7.3.12 Implicit szabályok

A C, C++, Pascal és más nyelvek forrásállományainak lefordítására megvannak a kialakult fordító programok, illetve tipikus parancsok. Ezért nagyban egyszerűsítené a fejlesztők munkáját, ha ezeket a szabályokat nem kellene mindig gépiesen beleírni a *Makefile*-ba. Ezt a *make* készítői is felismerték, ezért a tipikus műveletek szabályait beleépítették a programba.

Ha a *make* programnak elő kell állítania egy állományt, mert arra utasítjuk, vagy szerepel egy feldolgozandó szabály feltételei között, akkor megpróbálja megkeresni a rá vonatkozó szabályt. Azonban ha nem szerepel az állomány egyetlen szabály tárgyaként sem, akkor megpróbál egy alapértelmezett szabályt találni rá, és ha ez sikerül, akkor végre is hajtja azt. Ezeket az alapértelmezett szabályokat **implicit szabályok**nak nevezzük.

Az implicit szabályok nagy része előre adott, de mi is írhatunk ilyeneket. Az előre adott implicit szabályokat a

```
make –p
```

paranccsal nézhetjük meg. Azonban ez a parancs valójában a beépített szabályok és az általunk írt *Makefile* unióját adja vissza. Vagyis ha csak a beépített szabályokat akarjuk látni, akkor nem lehet a könyvtárban *Makefile*.

Az implicit szabályok nagy része ragozási, illetve minta szabály. Hiszen ezek olyan általános szabályok, amelyek mindenki számára használhatóak. A recepteket azonban úgy fogalmazták meg, hogy egyes változók értékével befolyásolhatjuk a működésüket. Például a C állományok fordítása esetén ez így néz ki:

```
%.o: %.c
$(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c $(OUTPUT_OPTION) $<
```

Láthatóan például a *CFLAGS* változó értékével befolyásolhatjuk, milyen opciókkal történjen a C fordító meghívása. Vagyis elég a *Makefile*-unkban ennek a változónak más értéket adni, ha az alapértelmezett értékek nem megfelelőek és nem szükséges a szabályt újra megalkotni. Az ilyen jellegű váltózók nevénél a konvenció a nagybetűk használata, ahogy a példában is szerepel.

7.3.13 Speciális tárgyak

Az eddigi fejezetek során láthattunk már két speciális célt is. Ilyen volt a *.PHONY* és a *.SUFFIXES*. Összegezzük a legfontosabbakat.

Név	Jelentés
.PHONY	Ennek a célnak az előfeltételei hamis céloknak minősülnek. Vagyis a célhoz tartozó szabályt időbélyeg ellenőrzés nélkül mindenképpen lefuttatja a *make*.
.SUFFIXES	A cél előfeltételei állománynév kiterjesztések, amelyeket a *make* a ragozási szabályok keresésénél használ.
.DEFAULT	Ehhez a célhoz megadott recept fut le azokban az esetekben, ha egy adott célhoz sem explicit, sem implicit szabályt nem talál a rendszer.
.SILENT	Ha megadunk előfeltételt ehhez a szabályhoz, akkor az azok előállításánál végrehajtott parancsokat nem írja ki a *make*. Ha nem adunk meg előfeltételt, akkor minden recept végrehajtása „csendes” lesz.
.ONESHELL	Korábban láthattuk, hogy a recept végrehajtása során minden sor külön shell példányban hívódik meg. Ennek a speciális célnak a használatával ezt módosíthatjuk. Ebben az esetben a teljes recept egy shell példány hajtódik végre.

7.3.14 Direktívák

A **direktívák** nagyon hasonlóak a C nyelvben használt preprocesszor direktívákhoz. Arra utasítják a *make* programot, hogy a *Makefile* olvasása közben valamilyen speciális műveletet végezzen el. Ezek a műveletek a következők lehetnek:

Más *Makefile*-ok beolvasása.

Feltételek alapján a *Makefile* egyes részeinek használata, vagy figyelmen kívül hagyása.

Változók definiálása.

Elsőként nézzük más állományok hozzáadását a *Makefile*-unkhoz:

```
include FAJL_NEVEK...
```

A példában a *FAJL_NEVEK* egy kifejezés, amely szóközzel elválasztott fájlnevekből, speciális karakterekből és változókból állhat. A *make* betölti a megadott állományokat és beilleszti a *Makefile*-ba.

Feltételek alapján a *Makefile* egyes részeit kihagyhatjuk, vagy benne hagyhatjuk a végrehajtott műveletek között.

```
ifeq ($(CC),gcc)
  libs := $(gcc_libs)
else
  libs := $(normal_libs)
endif
```

A fenti példában, ha a *CC* változó értéke *gcc*, akkor a második sor hajtódik végre, egyébként a negyedik.

Változókat a *define* direktívával definiálhatunk. Erre láthattunk példát a több soros változók létrehozásánál. (Lásd: Több soros változók definiálása fejezet)

7.4 *Make* alternatívák

Bár a *make* hatékony eszköz, amely más platformokon is jól használható. Azonban nagyobb projektek esetén, amikor a célok száma megnő, akkor a megírása összetett, nehéz feladat lehet. Emellett nehezen tud adaptálódni a különböző rendszerek eltérő konfigurációihoz. Fordító platformok váltása esetén szükséges lehet a *Makefile* módosítása.

7.4.1 *Autotools*

A *GNU Autotools* kiegészíti a *GNU make* funkcionalitását kibővített szabálylistával és részletes függőség ellenőrzéssel. Fő célja a forráskódok hordozhatóságának megteremtése a Unix rendszerek között. Számos nyílt forráskódú szoftver csomag használja.

A *GNU Autotools* több különálló programot tartalmaz, mint az *Autoconf*, *Automake*, és a *Libtool*. Azonban a programok a működésük során további eszközökre is építenek, mint például a *pkg-config* és a *gettext*.

A felhasználóknak a fordításhoz nincs szükségük a teljes *Autotools* csomagra. A forráscsomagban lévő *configure* script képes az *Autotools* eszközök nélkül is ellátni a feladatát.

7.4.2 *CMake*

A *CMake* (`http://www.cmake.org/`) hasonlóan a *GNU Autotools*-hoz *Makefile*-okat hoz létre, amelyekkel elvégezhetjük a tényleges fordítást. Azonban beállítástól függően akár Visual Studio projekt állomány előállítására is alkalmas. Vagyis fordító és platform független módon gondoskodik a forráskód lefordításához szükséges állományok legenerálásáról.

7.4.3 *qmake*

A *qmake* a Qt csomag része. Fő feladata Qt programok fordításának támogatása. Automatikusan kezeli a *moc* és *uic* segédprogramok hívását. Egyszerű konfigurációs állomány és nagy mennyiségű beépített tudás jellemzi. Működése hasonlít az előző két eszközhöz. Eredményként egy *Makefile*-t állít elő, amivel elvégezhetjük a fordítást.

7.4.4 *SCons*

Az *SCons* (`http://www.scons.org/`) egy szoftver „konstrukciós” eszköz. Automatikusan vizsgálja a forráskódokban a függőségeket, elvégzi a platform adaptációs lépéseket. Eredményként kész bináris programokat állít elő.

Az *SCons* Python alapú eszköz így a konfigurációs állományok és a fordítási műveletek leírásai Python scriptek.

7.5 Hibakeresés

A C az egyik legelterjedtebb programozási nyelv, és egyértelműen a Linux-rendszerek általános nyelvének tekinthető. Van azonban jó pár olyan jellemzője, amelyek használata könnyen vezethet nehezen felderíthető programhibákhoz. Ilyen gyakori és nehezen felderíthető hibafajta a memóriaszivárgás (memory leak), amikor a lefoglalt memória felszabadítása elmarad, vagy a túlírás (buffer overrun), amikor a program a lefoglalt területen kívülre ír. Ebben a fejezetben ezeknek a problémáknak a megoldására is látunk példákat.

A legfőbb hibakereső eszközünk a **`gdb`**. Azonban a gdb, illetve az operációs rendszer önmagában a memória kezelési hibáknak csak egy kis részét deríti fel, ezért ilyen jellegű hibakereséshez be szokás vetni még egy malloc debuggert (például **`ElectricFence`**). A memória kezelési hibák felderítésére egy másik eszköz a **`valgrind`**, amely egy virtuális processzoron futtatja a gépet.

Az **`strace`** további hasznos diagnosztikai, hibakereső eszköz. Általános esetben az **`strace`** lefuttatja a paraméterként megadott programot, és monitorozza a processz rendszerhívásait és a jelzé-seket, amelyeket kap. Az összes rendszerhívást a paramétereivel együtt a szabványos hiba-kimenetre, vagy a megadott kimeneti állományba írja.

A **`lint`** segédprogram a forráskód vizsgálatában lehet segítségünkre. Megkeresi a veszélyes, nem ajánlott megoldásokat a program forráskódjában. A program írása közben gyakran követünk el olyan szinte észrevehetetlen kis hibákat, amelyek később a hibakeresés során súlyos fejtörést okozhatnak. Ilyen a nem inicializált változók használata, a tömbök túlindexelése, értékadás az egyenlőség vizsgálata helyett stb. A **`lint`** az ilyen hibák elkerülésében segít. A C/C++ forrásállományokat dolgozza fel és értelmezi, mint egy fordító, azonban több szempontból vizsgálja a kód helyességét, és keresi a hiba gyanús részeket.

7.5.1 gdb

A hibakeresőknek, mint például a gdb-nek, az a rendeltetése, hogy belelássunk egy program működésébe, követhessük a futás során lezajló folyamatokat, továbbá hogy megtudjuk, mi történt a program összeomlásakor, mi vezetett a hibához.

A gdb-nek ezt lehetővé tevő funkcióit alapvetően négy csoportba oszthatjuk:

- A program elindítása.
- A program megállítása meghatározott feltételek esetén.
- A program megálláskori állapotának vizsgálata.
- A program egyes részeinek megváltoztatása és hatásának vizsgálata a hibára.

A gdb a C-ben és C++-ban írt programok vizsgálatára használható. Más nyelveket csak részlegesen támogat.

7.5.1.1 Példa a gdb használatára

A hibakereséshez az adott programnak tartalmaznia kell a hibainformációkat, vagyis az adott programot a -g kapcsolóval kell fordítani.

Először indítsuk el a programot a

```
gdb program
```

utasítással, majd állítsuk be a program kimenetének a szélességét a

```
set width=70
```

segítségével.

Helyezzünk el egy töréspontot (breakpoint) a fuggveny nevű függvénynél:

```
break fuggveny
```

Ezután a program futtatásához adjuk ki a

```
run
```

parancsot!

Amikor elértük a töréspontot a program megáll. Ilyenkor a leggyakrabban használt parancsok:

Parancs	Rövid	Magyarázat
run	r	A folyamat elindítása.
next	n	A következő sorra (next line) ugrás
step	s	Belépés egy függvénybe (step into)
print	p	Kiírja (print) a megadott változó aktuális értékét
backtrace	bt	A stack keretek megjelenítése

Parancs	Rövid	Magyarázat
list	l	A forráskód kilistázása az aktuális pozíció környékén
continue	c	A program futásának folytatása
Ctrl+D		A program leállítása (az EOF jel)
quit	q	Kilépés a gdb programból

Sikeres hibakeresés után a quit paranccsal léphetünk ki.

7.5.1.2 A gdb indítása

A gdb-t többféle argumentummal és opcióval indíthatjuk, amivel befolyásolhatjuk a hibakeresési környezetet.

A leggyakrabban egy argumentummal használjuk, ez a vizsgálandó program neve:

```
gdb PROGRAM
```

Azonban a program mellett megadhatjuk második paraméterként a core állományt :

```
gdb PROGRAM CORE
```

A core fájl helyett azonban megadhatunk egy process ID-t is, ha egy éppen futó processzt szeretnénk megvizsgálni:

```
gdb PROGRAM 1234
```

Ilyenkor a gdb az „1234” számú processzhez kapcsolódik.

Ezeket a funkciókat a gdb elindítása után parancsokkal is elérhetjük. A gdb-t használhatjuk más gépeken futó alkalmazások távoli debugolására is.

7.5.1.3 Töréspontok: breakpoint, watchpoint, catchpoint

A töréspont, amikor a program elér egy meghatározott pontra, megállítja annak futását. Minden törésponthoz megadhatunk plusz feltételeket is. Beállításuk a break paranccsal és paramétereivel történik.

Töréspontként megadhatjuk a program egy sorát, egy függvénynevet, vagy egy egzakt címet a programon belül:

```
break FUGGVENY
```

A forrásállomány FUGGVENY függvény meghívásánál helyezi el a töréspontot. (Lekezeli a C++ függvény felüldefiniálását is.)

```
break +OFFSET
break -OFFSET
```

Az aktuális pozíciótól megadott számú sorral vissza, vagy előrébb helyezi el a töréspontot.

```
break SORSZAM
```

Az aktuális forrásállomány SORSZAM sorában helyezi el a töréspontot.

```
break FÁJLNEV:SORSZAM
```

A FÁJLNEV forrásállomány SORSZAM sorában helyezi el a töréspontot.

```
break *CIM
```

A CIM címen helyezi el a töréspontot.

```
break
```

Argumentum nélkül az aktuális stack frame következő utasítására helyezi el a töréspontot.

```
break ... if COND
```

A törésponthoz feltételeket is megadhatunk. A COND kifejezés minden alkalommal kiértékelődik, amikor a töréspontot eléri a processz, és csak akkor áll meg, ha az értéke nem nulla, vagyis igaz.

A watchpoint olyan speciális töréspont, amely akkor állítja meg a programot, ha a megadott kifejezés értéke változik. Nem kell megadni azt a helyet, ahol ez történhet. A watchpoint-okat különböző parancsokkal állíthatjuk be:

```
watch KIF
```

A gdb megállítja a programot, amikor a program módosítja, írja a KIF kifejezés értékét.

```
rwatch KIF
```

A watchpoint akkor állítja meg a futást, amikor a program olvassa a KIF kifejezést.

```
awatch KIF
```

A watchpoint mind az olvasáskor, mind az íráskor megállítja a program futását.

A beállítások után a watchpoint-okat ugyanúgy kezelhetjük, mint a normál töréspontokat. Ugyanazokkal a parancsokkal engedélyezhetjük, tilthatjuk, törölhetjük.

A catchpoint egy másik speciális töréspont, amely akkor állítja meg a program futását, ha egy meghatározott esemény következik be. Például ilyen esemény lehet egy C++ program-ban bekövetkező kivétel (exception), vagy egy könyvtár betöltése. Ugyanúgy, mint a watchpoint-oknál, itt is több lehetőségünk van a töréspont beállítására. A catchpoint általános beállítási formája:

```
catch ESEMENY
```

Ahol az ESEMENY kifejezés az alábbiak közül valamelyik:

Esemény	Jelentés
throw	Egy C++ kivétel keletkezésekor.
catch	Egy C++ kivétel lekezelésekor.

Esemény	Jelentés
exec	Az exec függvény meghívásakor.
fork	Az fork függvény meghívásakor.
vfork	Az vfork függvény meghívásakor.
load [KONYVTARNEV]	A KONYVTARNEV könyvtár betöltésekor.
unload [KONYVTARNEV]	A KONYVTARNEV könyvtár eltávolításakor.

A catchpoint-okat szintén ugyanúgy kezelhetjük a beállítás után, mint a korábbi töréspontokat.

7.5.2 Valgrind

A valgrind szintén egy parancssoros program. Lényegében egy eszközkészlet a következő feladatokra:

- Memóriakezelési hibák felderítése
- Szálkezelési hibák felderítése
- Teljesítmény analízis

A paraméterként megkapott programot egy virtuális processzoron futtatja. A virtuális processzor számos hibaellenőrzést végez a kód futtatása közben, így deríti ki az esetleges hibákat.

7.6 IDE

A Linux fejlesztések során nem csak parancssoros eszközöket használhatunk. A korábban tárgyalt fejlesztői eszközöket fogja össze egy egységgé az ún. integrált fejlesztői környezet (IDE: Integrated Development Environment). Számos grafikus felülettel rendelkező IDE eszköz közül választhatunk a Linux alatt:

- Eclipse
- Anjuta
- KDevelop
- NetBeans
- QtCreator
- ...

8 Linux alkalmazások fejlesztése

8.1 Állomány- és I/O-kezelés

Az állomány (file) a Unix-világ egyik legáltalánosabb erőforrás-absztrakciója. Ez azt jelenti, hogy a különböző erőforrásokat az operációs rendszer fölött állományként látjuk, vagyis lényegében ugyanazokkal a függvényekkel kezeljük őket, mint a fájlokat. Az olyan erőforrások például, mint a memória, lemez terület, eszközök, IPC-csatornák, mind állományként jelennek meg a Unix-rendszereken. Az interfészek általánosításával egyszerűsödik a programozók feladata, illetve egyes erőforrások kezelése kompatibilissé válik más erőforrásokkal.

A következő állománytípusokat különböztetjük meg:

Egyszerű állomány

Az egyszerű állományok azok, amelyekre először gondolnánk az állomány szó hallatán. Vagyis bájtok sorozata, adatok, kód stb.

Könyvtár

A könyvtár speciális állomány, amely a benne lévő állományok listáját tartalmazza. A régi Unix-rendszerekben az implementációk megengedték, hogy a programok az egyszerű állományok kezelésére szolgáló függvényekkel hozzá is férjenek. A könnyebb kezelhetőségért azonban egy speciális rendszerhívás-készlet került az újabb rendszerekbe. Ezeket később tárgyaljuk.

Eszközök

A legtöbb fizikai eszköz a Unix-rendszereken, mint állomány jelenik meg. Két eszköztípust különböztetünk meg: blokk és karakter. A blokkos eszköz olyan hardvereszközt reprezentál, amelyet nem olvashatunk bájtonként, csak bájtok blokkjaiként. A Linux a blokkos eszközöket speciálisan kezeli, és általában fájlrendszert tartalmaznak. A karakteres eszköz bájtonként olvasható. A modemek, terminálok, printerek, hangkártyák, és az egér mind-mind karakteres eszköz.

Az eszközleíró speciális állományok tradicionálisan a /dev könyvtárban találhatók.

Szimbolikus hivatkozás

A szimbolikus hivatkozás (symbolic link, simlink) speciális állomány, amely egy másik állomány elérési információit tartalmazza. Amikor megnyitjuk, a rendszer érzékeli, hogy szimbolikus hivatkozás, kiolvassa az értékét, és megnyitja a hivatkozott állományt. Ezt a műveletet a szimbolikus hivatkozás követésének hívjuk. A rendszerhívások alapértelmezett esetben követik a szimbolikus hivatkozásokat.

Csővezeték (pipe)

A csővezeték a Unix legegyszerűbb IPC-mechanizmusa. Mint neve is elárulja, egy virtuális csővezetéket képez memóriában, amelynek végeire egy-egy állományleíróval hivatkozhatunk. Általában az egyik processz információkat ír bele az egyik oldalán, míg egy másik processz a

másik végén a beírási sorrendben kiolvassa az adatokat. Mivel a két processz párhuzamosan kezeli, ezért kis memóriaterületre van szükség köztes tárolóként.

A parancsértelmező a csővezetékeket I/O-átirányításokra használja, míg sok más program az alprocesszeikkel való kommunikációra.

Két típusát különböztetjük meg: unnamed (névtelen) és named (megnevezett). A névtelen csővezetékek akkor kreálódnak, amikor szükség van rájuk, és amikor mindkét oldal lezárja, akkor eltűnnek. Azért névtelenek, mert nem látszódnak a fájlrendszerben, nincs nevük. A megnevezett csővezetékek ezzel szemben fájlnévvel jelennek meg a fájlrendszerben, és a processzek ezzel a névvel férhetnek hozzá. A csővezetékeket FIFO-nak is nevezik, mivel az adatok FIFO- (first in, first out – aki elsőként be, az elsőként ki) rendszerben közlekednek rajta.

Socket

A socketek, mint a csővezetékek, IPC-csatornaként használhatóak. Flexibilisebbek, mint a csővezetékek, mert lehetővé teszik a kommunikációt két, különböző gépen futó processz között is.

Sok operációs rendszerben egy–az–egyes összerendelés van az állományok és az állománynevek között. (Minden állománynak egy neve van, és minden állománynév egy állományt jelöl.) A Unix, a nagyobb flexibilitás érdekében, szakított ezzel a koncepcióval.

Az állomány egyetlen egyedi azonosítója az inode-ja (information node, információs csomópont). Az állomány inode-ja tartalmaz minden információt az állományról, beleértve a jogokat, a méretét, a hivatkozások számát. Két inode-típust különböztetünk meg. Amelyikkel nekünk dolgunk lesz, az in-core inode. Minden megnyitott állományhoz tartozik egy ilyen, a kernel a memóriában tartja, és minden állománytípushoz egyforma. A másik típus az on-disk inode, amely a lemezen tárolódik. Minden állomány rendelkezik eggyel. A tárolása, struktúrája a fájlrendszer függvénye. Amikor egy processz megnyitja az állományt, az on-disk inode betöltődik a memóriába, és in-core inode-ra konvertálódik. Amikor az in-core inode módosul, visszakonvertálódik on-disk inode-ra, és tárolódik a fájlrendszeren. A két inode-típus szinkronizálását a kernel végzi, a legtöbb rendszerhívás azzal végződik, hogy mindkettőt frissíti.

Az on-disk és az in-core inode nem teljesen ugyanazokat az információkat tartalmazza. Például csak az in-core inode könyveli az adott állományhoz kapcsolódó processzek számát. Néhány állománytípus, mint például a névtelen csővezeték (unnamed pipe), nem rendelkezik on-disk inode-dal.

Az állománynév egy adott könyvtárban csak hivatkozás az on-disk inode-ra. A fájlnév lényegében egy mutató. Az on-disk inode tartalmazza a rá hivatkozó fájlnevek számát. Ezt hívjuk link count-nak (a kapcsolatok száma). Amikor egy állományt törlünk, akkor ez a szám csökken eggyel. Ha eléri a 0 értéket, és egyetlen processz sem tartja éppen nyitva, akkor ténylegesen törlődik, és a hely felszabadul. Ha viszont egy processz nyitva tartja, csak akkor szabadul fel a tároló hely, amikor ez a processz befejezi a használatát, és bezárja.

8.1.1 Egyszerű állománykezelés

A Linux számos fájlkezelő rendszerhívással rendelkezik. Kezdetként a legegyszerűbbeket, legáltalánosabbakat nézzük.

8.1.1.1 Az állományleíró

Amikor egy processz hozzáférést szerez egy állományhoz, vagyis megnyitja, a kernel egy állományleírót ad vissza, amelynek a segítségével a processz a későbbi műveletek során hivatkozhat az állományra. Az állományleírók kicsi, pozitív egész számok, amelyek a processz által megnyitott állományok tömbjének indexeként szolgálnak.

Az első három leíró (0, 1 és 2) speciális célokat szolgál, minden processz lefuttatásakor létrejön. Az első (0) a szabványos bemenet leírója, ahonnan a program az interaktív bemenetet kapja. A második (1) a processz szabványos kimenete, a program futása során keletkezett kimenet jó része ide irányítódik. A hibajelentések kimeneteként szolgál a harmadik (2) szabványos hibakimenet leírója. Mindkét kimenet általában a képernyőt jelenti. A szabványos C könyvtár követi ezt a szerkezetet, így kimeneti, bemeneti függvényei automatikusan használják ezeket a leírókat.

Az unistd.h headerfájl tartalmazza az STDIN_FILENO, STDOUT_FILENO, és STDERR_FILENO makrókat, amelyekkel ezeket a leírókat elérhetjük a programunkból.

8.1.1.2 Hozzáférés állományleíró nélkül

Az állománykezelő rendszerhívások egyik csoportját alkotják azok a függvények, amelyek állományleírókat használnak. A rendszerhívások másik alakja, amikor paraméterként az állomány nevét adjuk meg, és így kérünk az állomány inode-jára vonatkozó műveleteket.

8.1.1.3 Állományok megnyitása

Bár a Linux több állományfajtát is támogat, a legáltalánosabbak az egyszerű állományok. A programok, a konfigurációs fájlok, és az adatfájlok mind az állományok ezen csoportjába tartoznak. Ezeknek az állományoknak a megnyitására két függvény áll rendelkezésünkre:

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

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
```

Az `open()` függvény a megadott állomány leírójával tér vissza, ez a legkisebb, nem használt leíró lesz. Ha értéke kisebb, mint 0, akkor az állomány megnyitása meghiúsul, ilyenkor a szokásos módon az errno változó tartalmazza a hibakódot.

A flags paraméter tartalmazza a hozzáférési igényünket, illetve korlátozza a később használható függvények körét. Értéke az alábbi választási lehetőségek közül az egyik:

`O_RDONLY` (csak olvasható), `O_RDWR` (írható és olvasható), `O_WRONLY` (csak írható). Továbbá ezt kiegészíti egy, vagy több az alábbi opciók közül (bites VAGY kapcsolattal):

Opció	Jelentés
`O_CREAT`	Ha az állomány nem létezik, akkor egyszerű fájlként létrehozza.
`O_EXCL`	Az O_CREAT opcióval együtt használatos. Az open() hibával tér vissza, ha az állomány már létezik.
`O_NOCTTY`	A megnyitott fájl nem lesz a processz kontrollterminálja. Csak akkor érvényesül, ha a processz nem rendelkezik kontrollterminállal, és egy terminálleszközt nyit meg.
`O_TRUNC`	Ha az állomány létezik, akkor tartalma törlődik, és a mérete 0 lesz.
`O_APPEND`	Az állomány végéhez fűzi íráskor az adatokat. (A véletlen hozzáférés ilyenkor is engedélyezve van.)
`O_NONBLOCK`	Az állományműveletek nem blokkolják a processzt. Az egyszerű állományok esetén a műveletek mindig várakoznak, mert a lemezműveletek gyorsak. Azonban egyes állománytípusok esetén a válaszidő nem meghatározott. Például egy csővezeték esetén, ha nincs bejövő adat, az normál esetben blokkolja a processzt az olvasás műveletnél. Azonban ezzel az opcióval ilyenkor azonnal visszatér, hibát jelez.
`O_SYNC`	Normál esetben a kernel write cache-t alkalmaz, ami nagyban javítja a rendszer teljesítményét. Azonban adatvesztést eredményezhet. Ezzel az opcióval elérhető, hogy az írási művelet végére az adat valóban tárolva legyen a lemezen. Ez például adatbázisok esetén nagyon fontos.
`O_NOFOLLOW`	Amennyiben a megadott állomány valójában szimbolikus hivatkozás, akkor hibával tér vissza.
`O_DIRECTORY`	Ha a megadott név nem könyvtár, akkor hibával tér vissza.

A mode paraméter tartalmazza a hozzáférési jogosultságokat új állományok létrehozása esetén. Az `open()` függvénynél csak az `O_CREAT` opció esetén van értelme.

A `creat()` függvény egyenértékű az alábbi függvénnyel:

```
open(pathname, O_CREAT|O_WRONLY|O_TRUNC, mode)
```

8.1.1.4 Állományok bezárása

Az állományok bezárása azon kevés függvények közé tartozik, amely az összes állománytípusra megegyezik.

```
#include <unistd.h>


int close(int fd);
```

Ez elég egyszerű metódus. Lezárja az állományleírót, így az nem hivatkozik többé az állományra, nem használhatjuk tovább. Megszünteti az állomány minden zárolását.

Egy fontos dolgot érdemes vele kapcsolatban megjegyezni: lehet sikertelen. Néhány fájlrendszer nem tárolja le az állományok utolsó darabját, amíg az állományt be nem zárjuk (pl. az NFS). Ha ez a végső írási művelet sikertelen, akkor a `close()` függvény hibával tér vissza. Vagyis amennyiben nem használjuk a szinkron írási módot (`O_SYNC`), akkor mindig ellenőriznünk kell a visszatérési értékét, még ha nagyon ritkán is következik be hiba.

8.1.1.5 Írás, olvasás, mozgás az állományban

Az állományból olvasásnak, illetve állományba írásnak több módja van. Itt a legegyszerűbbet vesszük. Az olvasási és az írási művelet paramétereiben igen hasonlít:

```
#include <unistd.h>


ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

Mindkettő tartalmaz egy állományleírót (fd), egy mutatót az adatpufferre (buf), és a puffer hosszát (count). A `read()` függvény beolvassa az adatokat az állományból, és a pufferbe írja őket. A `write()` a count mennyiségű bájtot a pufferből az állományba írja. Mindkét függvény az átvitt bájtok számát adja vissza, vagy hiba esetén -1-et.

```
/* hellovilag2.c - A "Hello Vilag!" szoveget az aktualis konyvtar
hello allomanyaba irja. */


#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>


int main(void)
{
  int fd;
```

```
  /* Megnyitja az allomanyt, letrehozza, ha nem letezik, es torli
a tartalmat. */
  if((fd = open("hello", O_CREAT|O_TRUNC|O_WRONLY, 0644)) < 0)
  {
    perror("open");
    return 1;
  }

  /* 13 karaktert, a szoveget, kiirjuk az allomanyba. */
  if(write(fd, "Hello Vilag!\n", 13) != 13)
  {
    perror("write");
    return 1;
  }
  close(fd);
  return 0;
}
```

A Unix-állományokat két részre oszthatjuk: pozícionálható (seekable) és szekvenciális (non-seekable) állományokra. A szekvenciális állományok FIFO-csatornák, amelyek nem támogatják a véletlen hozzáférést, és az adatok nem újraolvashatóak, illetve nem felülírhatóak. A pozícionálható állományok lehetővé teszik az írási vagy olvasási műveleteket az állomány teljes területén. A csővezetékek, a karakteres eszközök nem pozícionálható állományok, a blokkos eszközök, és az egyszerű állományok pozícionálhatóak.

Amennyiben egy program ki akarja használni a véletlen hozzáférés lehetőségét, az írási és olvasási műveletek előtt be kell állítania az állományon belüli aktuális pozíciót. Erre szolgál az `lseek()` függvény:

```
#include <unistd.h>

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

Az fd leírójú állomány aktuális pozícióját elmozgatja offset bájtnyit a whence pozícióhoz képest, ahol a whence értéke az alábbi lehet:

Opció	Jelentés
SEEK_SET	Az állomány eleje.

Opció	Jelentés
SEEK_CUR	Az aktuális pozíció.
SEEK_END	Az állomány vége.

Az utóbbi két érték (`SEEK_CUR` és `SEEK_END`) esetén az offset értéke negatív is lehet. Ilyenkor természetesen a pozíció az ellenkező irányba módosul. Például ha az állomány végétől vissza 5 bájtnyira szeretnénk állítani a mutatót:

```
lseek(fd, -5, SEEK_END);
```

Az lseek() visszatérési értéke az aktuális pozíció az állomány elejétől, illetve –1 hiba esetén. Ezáltal például az

```
size=lseek(fd, 0, SEEK_END);
```

egyszerű megoldás az állomány méretének kitalálására.

Habár a különböző processzek, amelyek egyszerre használják az állományt, nem módosít-hatják egymás aktuális pozícióértékét, ez nem jelenti azt, hogy biztonságosan tudják párhuzamosan írni a fájlt, – könnyen felülírhatják egymás adatait.

Amennyiben szükséges, hogy több processz is írjon az állományba, ezt párhuzamosan csak hozzáfűzéssel tehetik meg. Ilyenkor az `O_APPEND` opció gondoskodik arról, hogy az írásműveletek atomiak legyenek.

A POSIX szabvány lehetővé teszi, hogy az aktuális pozíciót nagyobb értékre állítsuk, mint az állomány vége. Ilyenkor az állomány a megadott értékűre növekszik, és oda kerül az állomány vége. Ilyenkor a rendszer nem allokál a kimaradt területnek lemezhelyet, és nem is írja ki. Csak az állomány logikai mérete módosul. Az állomány ezen területeit **lyukak**nak (gap) nevezzük. A lyuk területéről való olvasás során 0 értékű bájtokat kapunk. Az írás során a használt területekre megtörténik a tényleges allokáció. Az ilyen lyukas állományok leginkább akkor használatosak, amikor az adat pozíciója is információt hordoz és takarékoskodunk a lemezterülettel. Ilyenek lehetnek például a hash-táblák. Másik gyakori példa a p2p rendszerekben használatos nem szekvenciális állományletöltés. Ilyenkor a letöltés kezdetén a teljes állományterületet allokálni kellene, hogy bármelyik részletet a helyére írhassuk később. Azonban a lyukakkal elkerülhető a tényleges allokálás, és ez a hely megtakarítása mellett nagyban gyorsítja is az eljárást.

8.1.2 Inode információk

A fejezet elején már tárgyaltuk az inode-ok fogalmát. Ez lényegében egy leíró adatstruktúra, amely az adott állomány paramétereit tartalmazza. A következő oldalakon megismerünk néhány függvényt, amellyel az inode-okat kezelhetjük.

8.1.2.1 Inode információk kiolvasása

A Linux az alábbi 3 függvényt támogatja az inode információk eléréséhez:

```
#include <unistd.h>
```

```
int stat(const char *file_name, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);
int fstat(int fd, struct stat *buf);
```

Ezek a függvények a megadott állomány inode információit adják vissza. Használatukhoz nem kell semmilyen hozzáférési jog az adott állományra, azonban az adott könyvtárra, illetve rendelkeznünk kell keresési joggal az összes oda vezető könyvtárra.

Az első függvény, a `stat()`, a `file_name` paraméter által megadott állomány inode információit adja vissza. Amennyiben szükséges, követi a szimbolikus hivatkozás. Ha ez utóbbi funkciót el szeretnénk kerülni, akkor az `lstat()` függvényt kell használnunk. A legutolsó változat, a `fstat()` függvény, megnyitott állományok inode információinak elérését teszi lehetővé.

A `struct stat` az alábbi elemeket tartalmazza:

Típus	Mező	Leírás
`dev_t`	`st_dev`	Az állományt tartalmazó eszköz azonosítója.
`ino_t`	`st_ino`	Az állomány on-disk inode-száma.
`mode_t`	`st_mode`	Az állomány jogai és típusa.
`nlink_t`	`st_nlink`	A referenciák száma erre az inode-ra.
`uid_t`	`st_uid`	Az állomány tulajdonosának felhasználóazonosítója (user ID).
`gid_t`	`st_gid`	Az állomány tulajdonosának csoportazonosítója (group ID).
`dev_t`	`st_rdev`	Ha az állomány speciális eszközleíró, akkor ez a mező tartalmazza a major és minor azonosítót.
`off_t`	`st_size`	Az állomány mérete bájtokban.
`unsigned long`	`st_blksize`	A fájlrendszer blokkmérete.
`unsigned long`	`st_blocks`	Az állomány által allokált blokkok száma.
`time_t`	`st_atime`	Az állományhoz való legutolsó hozzáférés időpontja.
`time_t`	`st_mtime`	Az állomány legutolsó módosítási időpontja.

Típus	Mező	Leírás
`time_t`	`st_ctime`	A legutolsó változtatás időpontja az állományon, vagy az inode-információn.

8.1.2.2 Jogok lekérdezése

Habár az inode leíró struktúra `st_mode` eleme tartalmazza az állomány jogait, amellyel meghatározhatjuk, mihez van jogunk, és mihez nincs, azonban ezen információk kinyerése nem olyan egyszerű, mint szeretnénk. Ugyanakkor a kernel már rendelkezik a kódrészlettel, amely meghatározza a hozzáférési jogainkat.

Egy egyszerű rendszerhívással le is kérdezhetjük ezeket:

```
#include <unistd.h>


int access(const char *pathname, int mode);
```

Ez a rendszerhívás leellenőrzi, hogy az adott processz olvashatja, vagy írhatja-e a pathname paraméterként megadott állományt, illetve, hogy egyáltalán létezik-e. Amennyiben a megadott név valójában egy szimbolikus hivatkozás, akkor az általa mutatott állomány jogait vizsgálja a függvény.

A `mode` paraméter a következő értékekből egyet vagy többet is tartalmazhat:

Érték	Jelentés
`F_OK`	Az állomány létezik-e, elérhető-e?
`R_OK`	A processz olvashatja-e az állományt?
`W_OK`	A processz írhatja-e az állományt?
`X_OK`	A processz futtathatja-e az állományt? (Könyvtár esetén keresési jog.)

A visszatérési érték 0 siker esetén, amikor a vizsgált jogosultságokkal rendelkezik a folyamat.

Természetesen a vizsgálatot befolyásolja, hogy az elérési útban megadott könyvtárakra rendelkezünk-e keresési joggal.

8.1.2.3 Jogok állítása

Az állományok hozzáférési jogai a `chmod()` rendszerhívással módosíthatóak.

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


int chmod(const char *path, mode_t mode);
int fchmod(int fd, mode_t mode);
```

Habár a `chmod()` paramétere az állomány elérési útvonala és neve, ne felejtsük el, hogy valójában inode információkat állítunk. Vagyis ha több hivatkozás van az adott állományra, akkor a többi joga is változik.

A `mode` paraméter a hozzáférés vezérlő bitek kombinációja. Tipikus, hogy a programozók oktálisan adják meg, azonban lehet a következő definiált értékek kombinációjával is:

Érték névvel	Érték számmal	Jelentés
`S_ISUID`	`04000`	setuid
`S_ISGID`	`02000`	setgid
`S_ISVTX`	`01000`	sticky bit
`S_IRUSR (S_IREAD)`	`00400`	tulajdonos olvashatja
`S_IWUSR (S_IWRITE)`	`00200`	tulajdonos írhatja
`S_IXUSR (S_IEXEC)`	`00100`	tulajdonos futtathatja, kereshet benne
`S_IRGRP`	`00040`	csoport olvashatja
`S_IWGRP`	`00020`	csoport írhatja
`S_IXGRP`	`00010`	csoport futtathatja, kereshet benne
`S_IROTH`	`00004`	mindenki olvashatja
`S_IWOTH`	`00002`	mindenki írhatja
`S_IXOTH`	`00001`	mindenki futtathatja, kereshet benne

Csak a root felhasználó és az állomány tulajdonosa jogosult a jogok állítására. Ha ezzel a függvényhívással mások próbálkoznak, akkor `EPERM` hibaüzenetet kapnak.

8.1.2.4 Tulajdonos és csoport beállítása

Mint a jogok, az állomány tulajdonosa és csoportja is az inode struktúrában tárolódik. Egy rendszerhívás szolgál mindkettő állítására.

```
#include <unistd.h>

int chown(const char *path, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
```

Az owner és group paraméterek adják meg az új tulajdonost és csoportot. Ha bármelyikük is -1, akkor nem változik az érték.

Csak a root jogosult a tulajdonos állítására. Mind a tulajdonos, mind a root felhasználó módosíthatja az állomány csoportját. Azonban míg az első esetben a tulajdonosnak tagnak kell lennie az új csoportban, a második esetben, amikor a root felhasználó állítja, akkor ilyen megkötések nincsenek.

Amikor a tulajdonost, vagy a csoportot nem root állítja, akkor biztonsági okokból mindig a setuid és a setgid bit törlődik. A POSIX nem definiálja, hogy root felhasználó esetén mi a követendő, ezért kernelverziótól függ, hogy törlődik-e vagy sem.

8.1.3 Könyvtárbejegyzések módosítása

Ne felejtsük el, hogy a könyvtárbejegyzések csak egyszerű mutatók az on-disk inode-okra. Minden lényeges információ az inode-okban tárolódik. Az **`open()`** függvény lehetővé teszi a processznek, hogy új, egyszerű állományokat hozzon létre. Azonban további függvényekre van szükség egyéb típusú állományok létrehozásához, vagy a könyvtárbejegyzések módosításához. Ebben a fejezetben a szimbolikus hivatkozások, az eszközkezelő állományok, és a FIFO-bejegyzések kezeléséhez szükséges rendszerhívásokkal foglalkozunk.

8.1.3.1 Eszközállományok és Pipe- bejegyzések

A processzek a fájlrendszeren az **`mknod()`** rendszerhívás segítségével hozhatnak létre megnevezett csővezetékeket (named pipe) és eszköz állományokat.

```
#include <fcntl.h>
#include <unistd.h>


int mknod(const char *pathname, mode_t mode, dev_t dev);
```

A **`pathname`** a létrehozandó bejegyzés neve. A **`mode`** a hozzáférési jogokat (amelyet az umask értéke módosít), és az állomány típusát határozza meg. Állománytípusnak az alábbiak adhatók meg:

Érték	Jelentés
`S_IFREG`	Normál állomány
`S_IFCHR`	Karakter típusú eszköz állomány
`S_IFBLK`	Blokk típusú eszköz állomány
`S_IFIFO`	named pipe

Az utolsó **`dev`** paraméter a **`S_IFCHR`**, vagy **`S_IFBLK`** mód esetén megadja a major és minor azonosítókat a létrehozandó eszközállományhoz. (Az eszköz típusa és major száma megadja a kernelnek, hogy melyik eszközkezelőt hívja meg. A minor szám az eszközmeghajtón belüli

szelektálásra szolgál, ha az adott eszközkezelő több eszközt szolgál ki.) Csak a root hozhat létre eszköz állományokat.

A `sys/sysmacros.h` állomány tartalmaz három makrót a `dev` paraméter beállítására. A `makedev()` függvény első paramétere a major szám, a második a minor szám, és ezekből létrehozza a `dev_t` értéket. A `major()` és `minor()` makrók a `dev_t` értékéből kiszámolják az eszköz major, illetve minor számát.

8.1.3.2 Merev hivatkozás létrehozása

Amikor a fájlrendszerben több név hivatkozik ugyanarra az inode-ra, az állományokat merev hivatkozásnak (hard link) nevezzük. Minden hivatkozásnak ugyanazon az állományrendszeren kell lennie. Ezek a hivatkozások teljesen egyenértékűek, illetve egyik törlése nem vezet az állomány törléséhez.

A `link()` rendszerhívás szolgál a merev hivatkozások létrehozására:

```
#include <unistd.h>


int link(const char *oldpath, const char *newpath);
```

Az `oldpath` a már létező állománynevet tartalmazza, a `newpath` az új merev hivatkozás neve. Az új név ugyanúgy használható ezek után, mint a régi, ugyanarra az állományra hivatkoznak. Lehetetlen megmondani, hogy melyik volt az eredeti.

Minden felhasználó létrehozhat új hivatkozást egy állományra, ha van hozzá olvasási joga, továbbá írás joga a könyvtárra, ahova a hivatkozást létrehozza, és futtatási joga a könyvtárra, ahol az eredeti állomány van. Könyvtárakra hivatkozást csak a root hozhat létre, azonban ez erősen ellenjavallt.

8.1.3.3 Szimbolikus hivatkozás létrehozása

A szimbolikus hivatkozás (symbolic link) a hivatkozások egy flexibilisebb típusa, mint a merev hivatkozások, azonban nem egyenrangúak az állomány többi hivatkozásával. Vagyis, ha az adott állományra az összes merev hivatkozást megszüntetjük, akkor a szimbolikus hivatkozás a semmibe fog mutatni. A szimbolikus hivatkozások használata könyvtárak között általános, továbbá létrehozható partíciók között is.

A rendszerhívások többsége automatikusan követi a szimbolikus hivatkozásokat, hogy megtalálja az inode-ot, kivéve, ha olyan változatukat használjuk, amelyek ezt eleve kizárják. A következő függvények nem követik Linux alatt a szimbolikus hivatkozásokat:

```
chown()
lstat()
readlink()
rename()
unlink()
```

Szimbolikus hivatkozásokat a `symlink()` rendszerhívással hozhatunk létre:

```
#include <unistd.h>


int symlink(const char *oldpath, const char *newpath);
```

Paraméterezése megegyezik a merev hivatkozásnál tárgyalttal. Amennyiben a **newpath** állomány létezik, akkor nem fogja felülírni.

A szimbolikus hivatkozás értékének megtalálása:

```
#include <unistd.h>


int readlink(const char *path, char *buf, size_t bufsize);
```

A szimbolikus hivatkozás neve a **buf** paraméterbe kerül, ha belefér. A **bufsize** tartalmazza a **buf** hosszát bájtokban. Általában a **PATH_MAX** méretű puffert használjuk, hogy elférjen a tartalom. A függvény egyik furcsasága, hogy nem tesz a sztring végére '\0' karaktert, vagyis a **buf** tartalma nem korrekt C sztring. Visszatérési értéke a visszaadott bájtok száma, vagy -1 hiba esetén.

8.1.3.4 Állományok törlése

Az állományok törlése tulajdonképpen az inode-ra mutató hivatkozások törlése, és csak akkor jelenti az állomány tényleges törlését, ha az adott merev hivatkozás az utolsó, amely az állományra hivatkozik. Mivel nincs arra metódus, hogy az állományt egzaktul töröljük, ezért ezt a metódust **unlink()**-nek nevezik.

```
#include <unistd.h>


int unlink(const char *pathname);
```

Amennyiben a hivatkozás az utolsó volt az állományra, de meg egyes processzek nyitva tartják, akkor egészen addig megmarad, amíg a legutolsó állományleírót is lezárják.

Ha az állománynév egy szimbolikus hivatkozást takar, akkor az letörlődik.

Amennyiben a név egy socketre, csővezetékre, vagy egyéb más eszközre hivatkozik, akkor letörlődik. Azonban azok a processzek, amelyek ekkor még nyitva tartják, továbbra is használhatják.

8.1.3.5 Állomány átnevezése

Az állomány nevét egészen addig megváltoztathatjuk, amíg az új név is ugyanarra a fizikai partícióra hivatkozik. Ha már egy létező nevet adtunk meg új névként, akkor először azt a hivatkozást megszünteti a rendszer, és utána hajtja végre az átnevezést. A **rename()** rendszerhívás garantáltan atomi, vagyis minden processz egyszerre csak az egyik nevén láthatja az állományt. (Nincs olyan eset, hogy egyszerre mindkét néven, vagy egyik néven sem látható.) Mivel az állomány megnyitások lényegében az inode-hoz kötődnek, ezért ha más processzek éppen nyitva tartják az állományt, akkor nincs rájuk hatással az átnevezése. Továbbra is működne, mintha nem is lenne más az állomány neve.

A rendszerhívás alakja a következő:

```
#include <stdio.h>


int rename(const char *oldpath, const char *newpath);
```

Meghívás után az `oldpath` nevű hivatkozás neve `newpath` lesz.

8.1.4 I/O-multiplexelés

Sok kliens/szerver-programnak van szüksége arra, hogy párhuzamosan több speciális állományt olvasson, vagy írjon. Például egy web böngészőnek szüksége van arra, hogy szimultán hálózati kapcsolatokon keresztül egyszerre az oldal több komponensét töltse le, hogy gyorsítsa a hozzáférést.

A legegyszerűbb megoldás egy ilyen esetre, ha a böngésző minden kapcsolaton keresztül beolvassa az érkezett adatokat, majd tovább lép a következőre. Vagyis a `read()` műveleteket egy ciklusba ágyazzuk, és felváltva olvassuk a csatornákat. Ez a megoldás jól működik egészen addig, amíg minden csatornán folyamatosan érkezik adat. Ha valamelyik lemaradna, akkor elkezdődnek a problémák. Ilyenkor annak a kapcsolatnak a következő olvasásakor a böngésző blokkolódik a `read()` rendszerhívásnál és egészen addig áll és várakozik, amíg érkezik adat. Természetesen általában ez nem a kívánt működési mód.

Emlékezzünk vissza a korábban tárgyalt nem blokkolt állománykezelésre. Ha az állományokat a `O_NONBLOCK` opcióval nyitjuk meg, akkor az írási-olvasási műveletek nem blokkolják a processz futását. Ilyenkor a `read()` rendszerhívás azonnal visszatér. Ha nem tudott beolvasni adatot, akkor csak szimplán 0-t ad vissza.

A nem blokkolt I/O-kezelés megadja ugyan a lehetőségét, hogy az egyes állományleírók között gyorsan váltogassunk, de ennek nagy az ára. A program folyamatosan olvasgatja mindkét leírót, és sosem függeszti fel a futását, amivel fölöslegesen terheli a rendszert.

8.1.4.1 Select

A hatékony multiplexelést a Unix a `select()` rendszerhívással támogatja, amely lehetővé teszi, hogy a processz blokkolódjon, és több állományra várakozzon párhuzamosan. A több állomány folytonos vizsgálatával szemben itt a processz egy rendszerhívással specifikálja, hogy mely állományok olvasására vagy írására várakozik. Amennyiben a felsorolt állományok valamelyike tartalmaz rendelkezésre álló adatot, vagy képes fogadni az új adatokat, a `select()` visszatér és az alkalmazás olvashatja, vagy írhatja azokat a fájlokat a blokkolódás veszélye nélkül. Ezek után egy újabb `select()` hívással várakozhatunk az újabb adatokra.

A `select()` és a `pselect()` formátuma:

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

```
int select(int n, fd_set *readfds, fd_set *writefds, fd_set
*exceptfds, struct timeval *timeout);


int pselect(int n, fd_set *readfds, fd_set *writefds, fd_set
*exceptfds, const struct timespec *timeout, const sigset_t
*sigmask);
```

A középső három paraméter (**readfds**, **writefds**, és **exceptfds**) adja meg, hogy mely állományleírókra vagyunk kíváncsiak, melyeket kell vizsgálnia.

Mindegyik paraméter egy-egy mutató egy **fd_set** adatstruktúrára. Ezeket a következő makrókkal kezelhetjük:

```
FD_ZERO(fd_set *set);
```

Kitörli az állományleíró listát. Ez a makró használatos a lista inicializálására.

```
FD_SET(int fd, fd_set *set);
```

Az fd leírót hozzáadja a listához.

```
FD_CLR(int fd, fd_set *set);
```

Az fd leírót kitörli a listából.

```
FD_ISSET(int fd, fd_set *set);
```

Igaz visszatérési értéket ad, ha az fd benne van a listában.

Az első állományleíró lista, **readfds**, azokat a fájlleírókat tartalmazza, amelyek akkor oldják fel a **select()** várakozását, amikor olvasható állapotba kerülnek. A **writefds** hasonlóan az írásra kész állományleírókra várakozik. Az **exceptfds** azokat a leírókat tartalmazza, amelyeknek valamely különleges állapotát várjuk. Ez Linux alatt csak akkor következik be, amikor out-of-band data jelzést kapunk egy hálózati kapcsolaton keresztül. Ezen listákból bármelyik lehet NULL.

Az n változó tartalma a legnagyobb állományleíró száma a listákból + 1.

A timeout paraméter tartalmazza azt a maximális időt, ameddig a **select()** várakozhat. Ha ez letelik, akkor a **select()** mindenképpen visszatér. A **select()** és a **pselect()** két különböző timeout értékmegadást tesz lehetővé. A **select()** visszatéréskor módosítva adja vissza az értéket, jelezve, hogy mennyi idő telt el, azonban ez nem mindegyik rendszeren van így, ezért tekintsük ezt az értéket definiálatlannak. A **pselect()** semmit nem változtat a timeout paraméterén.

További különbsége a **pselect()**-nek, hogy tartalmaz egy sigmask paramétert. A **pselect()** ezzel a maszkkal helyettesíti az aktuális jelzés maszkot a **select()** rendszerhívás idejére . (A NULL érték kikapcsolja a funkciót.)

8.1.4.2 Poll

A **select()**-hez hasonlóan működő másik eszköz a **poll()**. Funkciójában megegyezik a **select()**-el, csak paraméterezésében tér el tőle. A Linux rendszerekben valójában a **select()** a **poll()** rendszerhívással van leimplementálva.

A `poll()` alakja a következő:

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);


struct pollfd {
int fd;
short events;
short revents;
};
```

A `poll()` egy tömböt vár, amelynek elemei `pollfd` struktúrák. Egy elem egy fájlleírót ad meg az `fd` mezőben illetve, hogy milyen eseményekre várunk az `events` mezőben. Az `nfds` paraméter a tömb mérete, a `timeout` pedig a maximum várakozás időtartama.

Ha meghívjuk a `poll()`-t akkor addig várakozik, amíg legalább egy megadott esemény bekövetkezik, vagy lejár a megadott idő. A poll a kapott eseményeket bejegyzi az állományokhoz kapcsolódó `revents` elembe, így a feldolgozás során ezt az értéket kell vizsgálnunk.

8.1.5 Az ioctl rendszerhívás

Az `ioctl()` függvény a speciális állományok alacsony szintű paramétereinek kezelésére szolgál. Szintaxisa:

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


int ioctl(int d, int request, ...);
```

Az első, `d` paraméter az állományleíró. A `request` paraméter az eszköztől függő parancs kód. A harmadik paraméter egy típus nélküli pointer. A `request` paraméter kódolva tartalmazza, hogy a harmadik paraméter bemenő értéket tartalmaz, vagy kimeneti bufferként használatos, továbbá a buffer méretét.

A visszatérési értékként 0 jelenti a sikeres, -1 a sikertelen végrehajtást.

8.2 Párhuzamos programozás

8.2.1 Processzek

Egy operációs rendszerrel szemben támasztott alapvető követelmények egyike a többfeladatos működés (multitasking). Ez azt jelenti, hogy a felhasználók számára – virtuálisan vagy a valóságban is – több program fut párhuzamosan. A végrehajtás alatt álló programot, beleértve a programszámlálót, regisztereket és a változók aktuális értékét is, processznek nevezzük. A fent mondottak értelmét nem csorbítjuk jelentősen, ha a processzre egyszerűen „futó program”-ként gondolunk.

A

```
#include <unistd.h>


pid_t fork(void);
```

egy processzt két egyforma processzé változtat. Ekkor a forkot hívó szülő processz mellett egy új PID-del rendelkező gyermek processz keletkezik. A fork a szülő processzhez a gyermek processz azonosítójával (PID), míg a gyermekhez 0 értékkel tér vissza. Hiba esetén a fork() visszatérési értéke -1 (természetesen a szülő felé, hiszen a gyermek processz létre sem jött), és az errno változó tartalmazza a hiba okát.

Processz megszüntetése a

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


int kill(pid_t pid, SIGINT);
```

függvénnyel lehetséges, amelyet a jelzéskezelésnél részletezünk. Elöljáróban annyit, hogy ezt az üzenetet csak az egyazon felhasználóhoz tartozó processzek küldhetik egymásnak.

Az alábbi program szemlélteti a fork() használatát:

```
/* fork1.c - Pelda a fork() hasznalatara. */


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


int main()
{
  int pid;


  printf("A program inditasa...\n\n");
  pid=fork();


  if(pid<0)
  {
    printf("fork() hiba.\n\n");
    exit(-1);
  }
  if(pid==0)
```

```
  {
    /* A gyermek processz - pid 0 */
    printf("Gyermek processz. PID: %d "
           "fork() visszateresi ertek: %d\n", getpid(), pid);
  }
  else
  {
    /* A szulo processz - a pid valtozo a gyermek processz
       azonositoja.*/
    printf("Szulo processz. PID: %d "
           "fork() visszateresi ertek: %d\n", getpid(), pid);
  }

  return 0;
}
```

Ennek a programnak a kimenete a következő:

```
A program inditasa...

Szulo processz. PID: 3843 fork() visszateresi ertek: 3844
Gyerek processz. PID: 3844 fork() visszateresi ertek: 0
```

Látjuk, hogy a szülő processzazonosítója 3843, a gyermeké 3844. A szülő processz a fork() visszatérési értékéből értesül gyermeke azonosítójáról. Egy processz saját processz azonosítóját az <unistd.h> fájlban deklarált

```
pid_t getpid(void);
```

függvénnyel, a szülő azonosítóját a

```
pid_t getppid(void);
```

függvénnyel kérdezhetjük le. Ezeket a függvényeket a fenti programrészletben is felhasználtuk.

Gyermek processzek végére várakozni, állapotukat ellenőrizni az alábbi két függvénnyel tudunk:

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

pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int options);
```

A wait() függvény akkor tér vissza, ha a hívó processz gyermek processzei közül bármelyik befejezte működését. Ha a hívás időpontjában egy gyermek processz már zombi állapotban van, akkor a függvény azonnal visszatér. A status paraméter -1, ha hiba lépett föl, egyébként a gyermek processz kilépési kódja (exit code).

A waitpid függvény sokkal flexibilisebb: Itt a pid paraméterrel egy meghatározott processzazonosítójú gyermek kilépésére várakozhatunk, az utolsó paraméterrel megadható opcióktól függően, amelyek közül a lényegesebbeket a következő táblázat foglalja össze.

Érték	Leírás
< -1	Vár bármely gyermekfolyamat végére, melynek csoportazonosítója megegyezik a pid paraméterben adottal.
-1	Ekkor a függvény hatása megegyezik a wait függvényével.
0	Vár bármely gyermekfolyamat végére, melynek csoportazonosítója megegyezik a hívó folyamatéval.
> 0	Vár bármely gyermekfolyamat végére, melynek folyamatazonosítója megegyezik a pid paraméterben adottal.
WNOHANG	Felfüggesztés nélkül (no hang) visszatér, ha még egy gyermekfolyamat sem ért véget. Ezt a konstanst OR kapcsolatba kell hozni a fenti három lehetőség alapján választott értékkel.

8.2.2 Processzek közötti kommunikáció (IPC)

A UNIX operációs rendszer család processzek közötti kommunikációjának (Interprocess Communication) egyik fő alappillére a System V IPC. A System V IPC-t az AT&T fejlesztette ki saját UNIX verziójához, ezt Linux alá is implementáltak. A System V IPC-ről részletes útmutatást ad a ipc(5) man oldal. A másik alternatíva a POSIX szabvány által definiált lehetőségek használata. Mivel Linux alatt mindkettőt implementálták, szabadon használhatjuk bármelyiket.

A System V IPC programozása közben segítségünkre lehet néhány hasznos segédprogram. Az ipcs program kiírja a memóriában lévő olyan IPC-objektumokat, amelyekhez a hívó processznek olvasási joga van. Az ipcs paramétereit a következő táblázat foglalja össze.

Parancs	Magyarázat
ipcs –q	Csak az üzenetsorokat mutatja.
ipcs –s	Csak a szemaforokat mutatja.
ipcs –m	Csak az osztott memóriát mutatja.
ipcs –h	További segítség.

Nézzünk erre egy példát:

```
$ipcs –s


------ Semaphore Arrays --------
key              semid      owner      perms      nsems       status
0x5309dbd2 81985536   schspy      644            1
```

Fejlesztés közben az is hasznos lehet, ha el tudjuk távolítani az egyes IPC-objektumokat a kernelből. Erre szolgál az

```
ipcrm <msg | sem | shm>  <IPC ID>
```

segédprogram. Például a fent megjelenített szemafort a

```
ipcrm sem 81985536
```

paranccsal szüntethetjük meg.

A System V IPC-t Linux alatt alapvetően egy kernelhívás hajtja végre:

```
int ipc(unsigned int call, int first, int second, int third, void
*ptr, long fifth);
```

Az első argumentum határozza meg a hívás típusát (milyen IPC-funkciót kell végrehajtani), a többi paraméter a funkciótól függ. Ez a függvény értelemszerűen Linux-specifikus, lehetőleg csak a kernel programozása közben használjuk.

8.2.2.1 Szemaforok

Gyakori szinkronizációs lehetőség a szemafor (semaphore) használata. A szemafor unsigned int számlálónak képzelhető el, amelynek megváltoztatása oszthatatlan művelet kell, hogy legyen, vagyis más szálak és processzek nem tudják megszakítani a szemafort állító processzt. Vizsgáljuk most meg, miként használható a szemafor szinkronizációs célokra!

A szemafort egy meghatározott kezdeti értékre állítjuk. Valahányszor egy processz lefoglalja a szemafort (lock), annak értéke eggyel csökken. Ha eléri a nullát, több processz már nem foglalhatja le. Amennyiben egy processznek már nincs több szemafor által védendő dolga, növeli eggyel a szemafor értékét, vagyis elengedi (release) a szemafort.

A fentiekből adódik, hogy kezdeti értéke határozza meg azt, hogy egyszerre hány processz foglalhatja le a szemafort. Amennyiben ez 1, egyszerre csak egy processz foglalhatja le a szemafort, ami kölcsönös kizárást (mutual exclusion – mutex) jelenti.

Többféle szemaforkezelés lehetséges, az egyik a System V IPC által definiált, a másik a POSIX által támogatott.

A System V IPC szemaforjai lényegében szemafortömbök. Létrehozásuk a

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

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

függvényhívással történhet, ahol az első paraméter egy egyedi azonosító, az nsems a létrehozandó szemaforok száma, a semflg a hozzáférési jogosultság beállítására szolgál. Amennyiben a key paraméter egy már létező szemaforhoz lett hozzárendelve, akkor a semget függvény a létező szemafor azonosítóját adja vissza, egyébként az új szemaforét. Tehát már létező szemaforhoz egy másik processzből a key paraméter segítségével kapcsolódhatunk hozzá. Ha új szemafort szeretnénk létrehozni, a kulcsgenerálásban segítségünkre lehet az

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

key_t ftok(const char *pathname, int proj_id);
```

függvény, amely a kötelezően létező fájlra mutató pathname és egy nem nulla proj_id paraméterekből létrehoz egy egyedi kulcsot.

A szemafor törlését a következő sorral hajthatjuk végre:

```
semctl(semid, 0, IPC_RMID, 0);
```

A semctl függvény több szemaforvezérlő funkciót lát el.

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

int semctl(int semid, int semnum, int cmd, ...);
```

Az első argumentum a szemafor azonosítója, amivel a semopen függvény tért vissza. A semnum argumentum a szemafortömbből választ ki egy elemet, ez egy nulla alapú index. A semnum és az összes utána következő argumentum a cmd értékétől függ. A lehetséges műveleteket a következő táblázat foglalja össze.

Művelet	Leírás
`IPC_STAT`	Szemaforinformáció lekérdezése. Olvasási jogosultság szükséges.
`IPC_SET`	Jogosultság, felhasználó- és csoportazonosítók megváltoztatása.
`IPC_RMID`	Szemafortömb megszüntetése, felébresztve a várakozó processzeket.
`GETALL`	A szemafortömb elemeinek értékét adja vissza.
`GETNCNT`	Egy szemaforra várakozó processzek száma.
`GETPID`	A szemafortömb utolsó módosítójának processzazonosítóját kérdezi le.
`GETVAL`	Egy szemafor értékét adja vissza.

Művelet	Leírás
GETZCNT	Egy szemafor nulla (foglalt) értékére várakozó processzek száma.
SETALL	A szemafortömb összes elemének értékét állítja be.
SETVAL	Egy szemafor értékét állítja be.

A szemafor létrehozása és tulajdonságainak beállítása után vizsgáljuk meg, hogyan várakozhatunk egy szemaforra. Ezt a

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

int semop(int semid, struct sembuf *sops, unsigned nsops);
```

függvénnyel tehetjük meg. Az első paraméter a megszokott módon a szemafortömb azonosítására szolgál. A második paraméter sembuf típusú struktúrák egy tömbje, amelyek a végrehajtandó műveleteket írják elő. A semop függvény garantálja, hogy ezek közül a műveletek közül vagy mind, vagy egyik sem lesz végrehajtva. A sembuf struktúra felépítése a következő:

```
struct sembuf
{
  ushort  sem_num;     /* a szemafor indexe a tömbben */
  short   sem_op;      /* szemafor értékének változása */
  short   sem_flg;     /* a művelet jelzőbitjei */
};
```

A sem_op tagváltozó értéke előjelesen hozzáadódik a szemafor értékéhez. Amennyiben ez az érték negatív, úgy az erőforrás foglalásnak felel meg, ha pozitív, akkor az erőforrás elengedését jelenti. Nulla sem_op esetén a processz azt ellenőrzi, hogy a szemafor értéke nulla-e.

A jelzőbitek értéke IPC_NOWAIT, SEM_UNDO, illetve a kettő bitenkénti vagy kap-csolata lehet. Az IPC_NOWAIT esetén a műveletet megkísérli végrehajtani, ha azonban ez nem sikerül, a semop azonnal hibával tér vissza. Ha ez a jelzőbit nincs beállítva, akkor a semop várakozik a szemafor kedvező állapotára. Ha a sem_op értéke nulla volt és az IPC_NOWAIT nem volt beállítva, akkor a processz várakozik addig, amíg a szemafor értéke nulla nem lesz (összes erőforrás lefoglalva). A SEM_UNDO jelzőbit bekapcsolása esetén a művelet visszavonódik amikor a hívó processznek vége lesz.

A semop függvény utolsó argumentumaként a sops tömbben lévő sembuf típusú struktúrák számát adjuk meg.

8.2.2.2 Üzenetsorok

Az üzenetsor (message queues) olyan FIFO-jellegű kommunikációs csatorna, amelybe a programozó által meghatározott formátumú adatcsomagokat lehet belerakni. Az üzenethez hozzárendelhetünk egy pozitív számmal jelölt típust, amely alapján virtuálisan egy üzenetsoron több üzenetcsatornát használhatunk. Fizikailag ez láncolt listaként jelenik meg a kernel címterében, amelyet a következő adatstruktúra ír le:

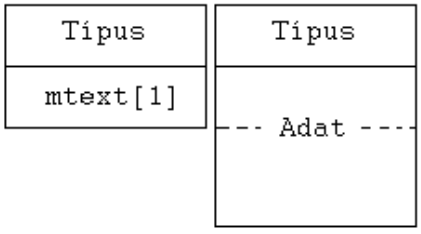
```
struct msg
{
  struct msg *msg_next;  /* a közvetkező üzenet a sorban */
  long  msg_type;        /* az üzenet típusa */
  char *msg_spot;        /* maga az üzenet (a kernel nem tud
                            semmit a formátumról) */
  short msg_ts;          /* az üzenet mérete */
};
```

A fentiekből jól látható, hogy a kernel csak az üzenet típusát kezeli (msg_type), a többi adatot egy memóriaterületre mutató pointer képviseli (msg_spot), amelynek tudja a méretét (msg_ts) és amelyet nem kell értelmezni.

Nézzük most meg, hogyan adhatjuk meg saját üzenetformátumunkat! Elsőként vizsgáljuk meg azt az alapvető struktúrát, amelyet ki kell bővítenünk, és a sys/msg.h állományban van definiálva:

```
/* üzenetformátum az üzenetküldő és üzenetfogadó függvényeknek
   (ld. később) */
struct msgbuf
{
  long mtype;         /* üzenettípus */
  char mtext[1];      /* adat */
};
```

Vagyis azok a függvények, amelyek üzenetet küldenek, illetve fogadnak, a fenti struktúrára mutató pointert várnak, és a tényleges, a programozó által definiált struktúra méretét. Az alábbi ábra szemlélteti mindezt.



3. ábra Az msgbuf és a programozó által definiált üzenetstruktúra összehasonlítása

Tegyük fel, hogy a programozó a következő struktúrát definiálta:

```
struct studentinfo
{
  char nev[40];       /* Nev */
  char cim[80];       /* Cim */
  char igszam[20];    /* Igazolvanyszam */
  char megj[80];      /* Megjegyzes */
};

struct studentinfo_msg
{
  long mtype;
  struct studentinfo data;
};
```

Ha a studentinfo_msg struktúrát explicite konvertáljuk msgbuf típusú struktúrára, abból az IPC-függvények csak a szaggatott vonalig tartó részt látják, és szükségünk van még arra az információra, amely megadja az ábrán jobb oldalon szereplő, programozó által meghatározott adatstruktúra méretét, méghozzá az mtype nélkül. A fenti példa egyben be is mutatta, hogyan kell saját üzenetformátumot leíró struktúrát deklarálni .

A System V IPC üzenetsorainak a kezelése koncepcióiban nagyon hasonlít a szemaforokéra. Üzenetsort a

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

int msgget(key_t key, int msgflg);
```

függvénnyel hozhatunk létre, melynek az első paramétere egy egyedi kulcs. Ezt ezúttal is létrehozhatjuk az ftok függvénnyel. A jelzőbitek a jogosultságokat állítják be, illetve megad-hatjuk az IPC_CREAT és az IPC_EXCL jelzőbiteket, amelyeket a System V szemaforok leírásánál mutattunk be. Amennyiben a megadott kulcs létezik, akkor a függvény a már létező üzenetsor azonosítójával tér vissza, egyébként az azonosító az újonnan létrehozotté. Hiba esetén a visszatérési érték -1.

Üzenetet küldeni, illetve fogadni az

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

```
int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);


ssize_t msgrcv(int msqid, struct msgbuf *msgp,  size_t msgsz, long msgtype, int msgflg);
```

függvények segítségével tudunk. A küldés során megadjuk az üzenetsor azonosítóját, a saját üzenetünk címét msgbuf struktúrára konvertálva, az adatméretet (a struktúra méretéből le kell vonnunk az mtype méretét). Ha az üzenetsor tele van, akkor az msgsnd függvény hibával tér vissza, vagy várakozik a küldésre. Első esetben jelzőbitként beállíthatjuk IPC_NOWAIT bitet, egyébként nullát adunk meg. Üzenet kiolvasásánál megadjuk az üzenetsor azonosítóját, annak a memóriaterületnek a pointerét msgbuf típusúra konvertálva, ahova szeretnénk, hogy az msgrcv függvény lemásolja az üzenetet. Kiolvasásnál a méret a megadott memóriaterület méretét jelenti. Ha ennél kisebb az üzenet nem történik hiba, ha nagyobb akkor igen. Ha azonban beállítjuk az MSG_NOERROR jelzőbitet, akkor az üzenet vége le lesz vágva, csak az első msgsz számú byte lesz az msgp által mutatott memóriaterületre másolva. Az msgtype argumentumot szűrésre használhatjuk az üzenet típusa alapján.

- Ha az msgtype nulla, akkor az msgrcv a soron következő üzenetet olvassa ki az üzenetsorból
- Ha pozitív, és
- ha az MSG_EXCEPT jelzőbit nincs bekapcsolva, akkor azt a legelső üzenetet, melynek típusa msgtype;
- ha az MSG_EXCEPT jelzőbit be van kapcsolva, akkor azt az első üzenetet, melynek típusa nem msgtype.
- Ha az msgtype negatív, akkor az a legalacsonyabb típusú üzenet kerül kiolvasásra, melynek típusa kisebb vagy egyenlő, mint az msgtype abszolút értéke.

Az üzenetsor vezérlését a

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


int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

függvénnyel végezhetjük. Az első paraméter az üzenetsor azonosítója, a második paramétert a következő táblázat foglalja össze:

Parancs	Leírás
`IPC_STAT`	Információt másol a buf argumentum által mutatott struktúrába.

Parancs	Leírás
`IPC_SET`	A buf által mutatott struktúra némely tagja alapján átállítja az üzenetsor tulajdonságait. A figyelembe vett tagok a következők: • msg_perm.uid • msg_perm.gid • msg_perm.mode (az alsó 9 bit) • msg_qbytes
`IPC_RMID`	Az üzenetsor megszüntetése.

Az utolsó paraméter típusa az üzenetsor tulajdonságait rögzítő struktúra:

```
struct msqid_ds
{
  struct ipc_perm msg_perm; /* Hozzáférési jogosultságok */
  struct msg *msg_first;    /* Az első üzenet a üzenetsor láncolt
                               listájában */
  struct msg *msg_last;     /* Az utolsó üzenet az üzenetsor
                               láncolt listájában */
  time_t msg_stime;         /* A legutolsó küldés ideje */
  time_t msg_rtime;         /* A legutolsó olvasás ideje */
  time_t msg_ctime;         /* A legutolsó változtatás ideje */
  struct wait_queue wwait;
  struct wait_queue rwait;
  ushort msg_cbytes;        /* Az üzenetsorban lévő bájtok száma
                               (összes üzenet) */
  ushort msg_qnum;          /* Az éppen az üzenetsorban lévő
                               üzenetek száma */
  ushort msg_qbytes;        /* Az üzenetsorban levő bájtok
                               maximális száma  */
  ushort msg_lspid;         /* A legutolsó küldő processz
                               azonosítója */
  ushort msg_lrpid;         /* A legutolsó olvasó processz
                               azonosítója */
};
```

Például üzenetsor eltávolítását a

```
msgctl(mqid, IPC_RMID, 0);
```

programsorral végezhetjük.

8.2.2.3 Megosztott memória

A megosztott memória (shared memory) közös memóriatartomány, amelyhez több processz is hozzáférhet. Ez a leghatékonyabb módja a processzek közti kommunikációnak, mert a közös memóriatartomány a hívó processz címterébe lapolódik, és a közvetlen memóriahozzáférés gyorsabb, mint bármely más eddig tárgyalt System V IPC mechanizmus.

Az exit és exec rendszerhívások esetén a rendszer az összes megosztott memória-erőforrást lecsatolja, fork esetén a gyermek processz örökli a szülőhöz csatolt összes megosztott memóriatartományt.

Az eddigiekkel összhangban módon megosztott memóriát az

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

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

függvénnyel hozhatunk létre. A key létrehozásnál egyedi azonosító, amelyet legtöbbször egy ftok hívással generálunk. Ha a key argumentum nem egyedi, a jelzőbitek (shmflag) értékétől függően hiba történik, vagy egy már létező megosztott memóriatartomány azonosítóját kapjuk vissza. A size argumentum a kívánt memória mérete bájtokban, a lefoglalt memória mérete a size érték felkerekítve a PAGE_SIZE legkisebb többszörösére. Az shmflg értéke a már ismert hozzáférési jogosultság beállításaiból és az opcionális IPC_CREAT és IPC_EXCL jelzőbitekből.

Amennyiben szeretnénk használni egy már létrehozott memóriatartományt az azonosító segítségével, akkor előbb hozzá kell csatolnunk (attach) a processz címtartományához. Ezt a

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

void *shmat(int shmid, const void *shmaddr, int shmflg);
```

hívással tehetjük meg, amely visszaad egy pointert a lefoglalt és a processz címtartományába leképezett megosztott memóriatartományra. Ha az shmaddr paraméterben megadunk egy memóriacímet, és beállítjuk az SHM_RND jelzőbitet, akkor a visszaadott cím az shmaddr értéke lesz lekerekítve a legközelebbi laphatárra. Ha az SHM_RND nincs beállítva, akkor shmaddr argumentumnak címhatárra igazított értéket kell tartalmaznia. A gyakorlati esetek többségében azonban ez a paraméter nulla, ami a rendszerre bízza az megfelelő címtartomány kiválasztását. Az shmflg argumentum a már említett SHM_RND értéken kívül SHM_RDONLY lehet, ami csak olvasásra csatolja a megosztott memóriát a processz címtartományához.

Mivel most már van egy mutatónk, annak segítségével tudjuk írni és olvasni a megosztott memóriatartományt. Ha már nincs szükségünk többet a megosztott memória erőforrásra, azt le kell csatolni (detach), amit a

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

int shmdt(const void *shmaddr);
```

függvény meghívásával kell végrehajtanunk, melynek egyetlen paramétere az shmat függvény által visszaadott mutató.

A vezérlést ezúttal a

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

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

függvénnyel végezhetjük. Az első paraméter a megosztott memória azonosítója, a második paramétert a következő táblázat foglalja össze.

Parancs	Leírás
`IPC_STAT`	Információ a megosztott memóriáról.
`IPC_SET`	Hozzáférési jogosultságok és azonosítók megváltoztatása.
`IPC_RMID`	A megosztott memória megszüntetése.
`SHM_LOCK`	A megosztott memória mindvégig a fizikai memóriában marad. Linux specifikus.
`SHM_UNLOCK`	Engedélyezi a swappinget.

Az utolsó paraméter felépítése:

```
struct shmid_ds
{
  struct ipc_perm shm_perm; /* Hozzáférés és azonosítók beállítása
                               */
  int shm_segsz;    /* A memóriatartomány mérete (bájtokban) */
  time_t shm_atime; /* A legutolsó felcsatolás ideje */
  time_t shm_dtime; /* A legutolsó lecsatolás ideje */
  time_t shm_ctime; /* Az utolsó változás ideje */
```

```
  unsigned short shm_cpid; /* A létrehozó processz azonosítója */
  unsigned short shm_lpid; /* Az utolsó művelet végrehajtójának
                              azonosítója */
  short shm_nattch; /* Az aktuális felcsatolások száma */

  /*---------- A többi tagot a rendszer használja ---------*/
  unsigned short shm_npages; /* A tartomány mérete (lapok száma)
                              */
  unsigned long *shm_pages;
  struct vm_area_struct *attaches; /* A felcsatolások leírója  */
};
```

A megosztott memória megszüntetése, például, a

```
shmctl(shmid, IPC_RMID, 0);
```

hívással történhet. Ha nincs olyan processz, amelyik csatolva tartaná az erőforrást, azonnal megszűnik, ha van, akkor csak mintegy megjelöli megszüntetésre a megosztott memória-erőforrást, a tényleges eltávolítás csak az utolsó lecsatolás után történik meg.

8.2.3 Szálak és szinkronizációjuk

A szálak könnyűsúlyú processzek (Lightweight Processes).

A processzeknek öt alapvető része van:

- kód
- adat
- verem
- fájlleírók
- jelzéstáblák

Amikor a processzek közt váltani kell (context switching), ezeket az adatstruktúrákat fel kell tölteni az új processznek megfelelően, ami időt vesz igénybe. A szálak esetén ezek az adatstruktúrák közösek, tehát gyorsabban lehet váltani közöttük. A processzek között csak a kódrész közös, míg a szálak ugyanabban a címtartományban futnak. Ez sokkal könnyebben használható kommunikációs lehetőséget biztosít a szálak között.

Linux alatt alapvetően kétféle száltípus van:

- felhasználói módban és
- kernel módban

futó szálak.

A felhasználói módban futó szálak nem használják a kernelt az ütemezéshez, maguk bonyolítják le az ütemezést. Ezt a megoldást kooperatív multitaszkingnek nevezik, ahol egy processz definiál olyan függvényeket, amelyekre átkapcsolhat a rendszer. A szál valamilyen módon expliciten jelzi, hogy átadja a futás lehetőségét egy másik szálnak, vagy a váltás valamilyen időzítés alapján történik. A felhasználói módban futó szálak esetén a váltás általában gyorsabb, mint a kernel módban futónál. Ennek a megoldásnak további hátrányai, hogy a szálak nem adják át a CPU-időszeletet más szálaknak, így éhezés (starving) alakulhat ki a várakozó szálak között. Továbbá nem tudja kihasználni az operációs rendszer szimmetrikus multiprocesszor (SMP) támogatását. Végül amikor egy szinkron I/O-hívás blokkol egy szálat, akkor a többi szál nem tud futni. Mindezekre a problémára léteznek megoldások (külső monitorozás az éhezés ellen, a szálak különböző processzoron futtatása, szinkron I/O műveletre csomagoló függvények (wrapper) írása, amely nem blokkol), de ezek a megoldások egy sor, az operációs rendszertől elvárt funkciót implementálnak az operációs rendszer fölé, ami nehézkessé teszi őket. A felhasználói módban futó szálakat ma már szinte egyáltalán nem alkalmazzák.

A kernel módban futó szálkezelés esetén a szálak automatikusan ki tudják használni az SMP előnyeit, az I/O-blokkolás nem probléma, és Linux alatt a váltás nem sokkal lassúbb a felhasználói módban implementált szálvezérlésnél tapasztaltnál. Ilyenkor a kernel tartja számon a szálakhoz tartozó adatstruktúrákat és ütemezi a szálakat.

A Linuxban ugyan találhatunk programkönyvtárakat a felhasználói módban futó szálkezeléshez, de a kernel a 1.3.56 verzió óta a kernel módban futó szálkezelést támogatja. A továbbiakban a kernel módú szálkezelést vizsgáljuk, bemutatva a POSIX-kompatibilis pthread programkönyvtárat.

8.2.3.1 Szálak létrehozása

A Linux kernel módú szálkezelésének a kulcsa a

```
#include <sched.h>


int clone(int(*fn)(void*),void *child_stack,int flags,void *arg)
```

függvény. Ez a már részletezett fork függvény kiterjesztésének tekinthető. Az első paraméter az indítandó processz vagy szál belépési pontja, a második a veremmutató, a harmadik argumentum pedig a megadott függvénynek átadandó paraméter. A flags argumentum finomítja a processz vagy szál létrehozását.

A clone függvényhívásra épül a Linux pthread könyvtárának inicializáló függvénye:

```
#include <pthread.h>


int pthread_create(pthread_t *   thread, pthread_*attr_t attr, void
* (*start_routine)(void *), void * arg);
```

Ezzel a függvénnyel egy szálat indíthatunk el, melynek belépési pontja a harmadik argumentumban megadott start_routine függvény, melynek prototípusa

```
void* start_routine(void* param);
```

formában deklarálható. Ez a függvény egy paramétert vár, amelyet a pthread_create függvény utolsó paramétereként adhatunk meg. A thread argumentumban a szál leíróját kapjuk vissza. Az attr paraméter NULL esetén az alapértelmezett beállításokat jellemzi.

Nézzünk most egy egyszerű példát szál indítására!

```
/* thread1.c - Szal inditasa. */

#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

void *thread_function(void *arg)
{
  int i;

  printf("A szal indul... \n");
  for ( i=1; i<=20; i++ )
  {
    printf("%d. Hello szal vilag!\n",i);
    sleep(1);
  }
  printf("A szal kilep... \n");
  return NULL;
}

int main(void)
{
  pthread_t mythread;

  if ( pthread_create( &mythread, NULL, thread_function, NULL) )
  {
    fprintf(stderr,"Hiba a szal letrehozasaban.\n");
    exit(1);
  }
  sleep(5);
```

```
  printf("A foszal kilep... \n");
  return 0;
}
```

Fordítsuk le a fenti programot:

```
gcc thread1.c -o thread1 –lpthread
```

Amikor a pthread könyvtárat használjuk, azt hozzá is kell linkelnünk a programunkhoz, amit a –lpthread kapcsolóval tehetünk meg.

Fordítás után indítsuk is el:

```
$ ./thread1
A foszal kilep...
```

ami feltehetőleg nem az az eredmény, amelyre számítottunk, ugyanis nem jelent meg a terminálon a szálban található printf hívások eredménye. Viszont hibát sem kaptunk, a főprogram sikeresen befejezte működését.

A magyarázat abban rejlik, hogy a főprogram külön szálként tovább fut a következő, a

```
  printf("A foszal kilep... \n");
```

sorra, majd a main függvényből kilép a vezérlés, ami megszünteti a processzt, kilépéskor leállítja a szálakat, és felszabadítja a processzhez tartozó összes erőforrást. Próbaképpen helyezzünk el egy késleltetést a programban a mielőtt a main függvényből kilépünk:

```
    ...
  sleep(5);
  printf("A foszal kilep... \n");
  return 0;
   ...
```

Ezután a futási eredmény már hasonlít az eredeti elképzeléshez:

```
$ ./thread1
A szal indul...
1. Hello szal vilag!
2. Hello szal vilag!
3. Hello szal vilag!
4. Hello szal vilag!
5. Hello szal vilag!
A foszal kilep...
```

A késleltetés nem volt elég ahhoz, hogy a szál befejezze működését, de már látjuk, hogy elindult, és futott egy darabig. Ezután persze növelhetnénk az időzítést, de mivel minden hardveren más az ütemezés időbeli lefolyása, nem építhetünk rá. Szükségünk van egy olyan

függvényre, amellyel a szálat indító függvényből (példánk esetében a main függvényből) meg tudnánk várni a szál lefutását. Erre kínál megoldást a

```
#include <pthread.h>


int pthread_join(pthread_t thread, void **thread_return);
```

függvény, amely felfüggeszti a hívó szál működését mindaddig, amíg a thread argumentumban megadott szál be nem fejezi futását. Ha a thread_return nem NULL, akkor az a szál visszatérési értékére mutat a sikeres visszatérés után. Módosítsuk a main függvényt mindezeknek megfelelően:

```
int main(void)
{
  pthread_t mythread;

  if ( pthread_create( &mythread, NULL, thread_function, NULL) )
  {
    fprintf(stderr,"Hiba a szal letrehozasaban.\n");
    exit(1);
  }

  if ( pthread_join ( mythread, NULL ) )
  {
    fprintf(stderr,"Hiba a szal megvarasaban.\n");
    exit(1);
  }

  printf("A foszal kilep... \n");
  return 0;
}
```

8.2.3.2 Kölcsönös kizárás (mutex)

A kölcsönös kizárás (Mutual Exclusion – mutex) a szálak szinkronizációjának hasznos eszköze. A mutexnek két lehetséges állapota van: foglalt (locked), amikor egy szál birtokolja a mutexet, illetve szabad (unlocked) állapot, amikor egy szál sem birtokolja a mutexet. A mutex egyszerre csak egy szálé lehet.

Linux alatt a háromfajta mutex van:

- gyors,

- rekurzív
- hibaellenőrző.

A mutex fajtája azt mondja meg, hogy mi történik akkor, ha egy olyan szál próbál lefoglalni egy mutexet, amelynek az már a birtokában van.

Gyors mutex esetén az adott szál arra várakozik, hogy a saját maga által már lefoglalt mutex felszabaduljon, amit csak ez a szál tudna felszabadítani, vagyis ekkor végtelen ciklusba kerülünk.

A hibaellenőrző mutex esetén a lefoglalást végző függvény rögtön hibával tér vissza.

Rekurzív mutex esetén a szál újra lefoglalja a mutexet, és ahhoz, hogy a mutex más szálak által újra lefoglalhatóvá váljon, mindannyiszor el kell engednie, ahányszor lefoglalta.

A mutex fajtáját a változó inicializálásakor is megadhatjuk:

```
#include <pthread.h>


pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t recmutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
pthread_mutex_t                    errchkmutex                    =
PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
```

Az NP utótag a nem hordozható (non-portable) makrókra utal.

Mutexet a fenti statikus inicializáció helyett függvényhívással is létrehozhatunk:

```
#include <pthread.h>


int      pthread_mutex_init(pthread_mutex_t      *mutex,      const
pthread_mutexattr_t *mutexattr);
```

Az utolsó paraméterrel jelenleg Linux alatt csak az attribútum fajtáját állíthatjuk. A mutex argumentumban kapjuk vissza a mutex objektumra mutató pointert.

Mutex felszabadítását a

```
#include <pthread.h>


int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

függvénnyel kell elvégeznünk, melynek feltétele, hogy a hívás pillanatában a mutex ne legyen foglalt. A jelenlegi Linux-implementáció mindössze ellenőrzi a szabad állapotot.

Mutex lefoglalását a

```
#include <pthread.h>


int pthread_mutex_lock(pthread_mutex_t *mutex);
```

függvénnyel végezhetjük. Ha a mutex éppen szabad, akkor a pthread_mutex_lock lefoglalja, egyébként a hívó szál addig fel lesz függesztve, amíg a mutex szabaddá nem válik, és akkor kerül a hívó szál birtokába.

Ennek a függvénynek az a hátránya, hogy ha nem szabad a mutex, akkor a szál várakozik. Ha nem szeretnénk, hogy a függvény várakozzon, hanem csak annyit, hogy ha szabad a mutex, akkor lefoglalja, egyébként rögtön visszatér jelezve, hogy a mutex foglalt, akkor a

```
#include <pthread.h>


int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

függvényt használjuk, amely EBUSY értékkel tér vissza, ha a mutex foglalt, és nem blokkolja a szálat.

Miután megszereztük a mutexet, a

```
#include <pthread.h>


int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

függvénnyel engedhetjük el.

8.2.3.3 Feltételes változók (condition variable)

Tekintsük a következő szituációt!

Példa. Egy egész típusú számláló állapota arányos egy víztartály vízszintjével. Amennyiben a számláló eléri a 10-es értéket, írjunk ki egy figyelmeztetést a terminálra, ha pedig eléri a 15-ös szintet, a program szólítsa fel az úszni nem tudókat, hogy hagyják el a gyár területét.

A fenti példa implementációjában legalább egy szál foglalkozik a tartály állapotával, és frissíti az unsigned int típusú számlálót. A feltételes változók illusztrációjának érdekében tegyük fel, hogy a tartállyal foglalkozó szálak túl vannak terhelve, és a program több processzoron fut, és szeretnénk kihasználni párhuzamosítás lehetőségeit. Ezért a számláló állapotát külön szálból ellenőrizzük, és ugyaninnen küldjük a figyelmeztetéseket. Mivel a számlálóhoz több szál is hozzáfér, ezért azt egy mutex védi. Amikor az ellenőrző szál lefoglalja a mutexet, hogy olvasni tudja a számlálót, és szeretne várakozni a 10 vagy a 15 értékek valamelyikére. Azonban, ha fogva tartja a mutexet, akkor a tartály állapotát frissítő szálak nem tudják lefoglalni a mutexet, így nem tudják megváltoztatni a számláló értékét, vagyis minden szál a másikra vár, miközben a gyárat lassan, de annál határozottabban elborítja a víz. Jó lenne, ha az ellenőrző szál a mutexet lefoglalván rendelkezne egy olyan atomi művelettel, amely elengedné a mutexet, és várakozna egy eseményre (megváltozott a számláló értéke), és ahogy az esemény bekövetkezik, egy atomi művelettel felébredne és vissza tudná szerezni a mutexet. Erre használhatjuk a feltételes változókat.

A feltételes változók olyan szinkronizációs objektumok, amelyek lehetővé teszik, hogy a kritikus szekcióban levő szálak felfüggesszék futásukat mindaddig, amíg egy erőforrásra igaz nem lesz valamilyen állítás, és várakozás közben ne fogják a kritikus szekciót biztosító

mutexet, ugyanakkor, ha az állítás igaz lesz, rögtön felébrednek, és egyben vissza tudják szerezni a mutexet.

A feltételes változókon két művelet végezhető, az egyik a várakozás a jelzésre, a másik maga a jelzés. Ilyenkor a feltételes változóra várakozó szálak közül egy elindul. A jelzés egy pillanatnyi esemény, a jelzés időpontjában az éppen jelzésre várakozó szálakat érinti, azok a szálak, amelyek a jelzés után kezdik meg várakozásukat, csak a következő jelzést veszik figyelembe.

Mivel a jelzés csak pillanatnyi esemény, előfordulhat, hogy az egyik szál már készül a várakozásra, már meghívta a megfelelő várakozó függvényt, de a jelzést már lekési, ha a várakozó szál nem kap időszeletet a futásra, illetve nem kési le, ha a várakozó szál előbb kap időszeletet a várakozásra. Ez tehát ütemezéstől függő kritikus versenyhelyzetet eredményez.

Erre a problémára megoldást jelent, ha olyan mutexet használunk, amelyet lefoglaltunk, mi¬előtt meghívjuk a várakozó függvényt, és amikor az ténylegesen megkezdi a várakozást, automatikusan elengedi a mutexet, majd amikor visszatér, újra megszerzi. Mindez természetesen nem elég, a versenyhelyzet elkerüléséhez az is szükséges, hogy mielőtt kiadunk egy jelzést, foglaljuk le a mutexet, majd a jelzés kiadása után engedjük is el. Így, ha a jelzés kiadása előtt valamelyik szál már elkezdett készülődni a várakozásra, az már megszerezte a mutexet, így a jelzés kiadása előtt megvárjuk (a mutex szabad állapotára várakozva), amíg az megkezdi várakozását, és a várakozó függvény automatikusan elereszti a mutexet, és csak akkor adjuk ki a jelzést. Amikor a jelzés kiadása előtt lefoglaljuk a mutexet, akkor „aki bújt, aki nem” jelleggel kiadjuk a jelzést a feltételes változóra már várakozó szálaknak, amelyek ez után szeretnének várakozni a feltételes változóra. Nekik meg kell szerezniük a mutexet, ami már csak a jelzés után lehetséges. Vagyis ismét – ezúttal más szemszögből – eljutunk oda, hogy egy feltételes változóhoz mindig hozzá kell rendelnünk egy mutexet.

A feltételes változókhoz kapcsolódó függvények nem jelzésbiztosak (signal safe), ami azt jelenti, hogy jelzéskezelő függvényekből (signal handler) nem hívhatók.

Feltételes változókat statikusan az alábbi kódrészlettel hozhatjuk létre:

```
#include <pthread.h>


pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Egyébként pedig a

```
#include <pthread.h>


int   pthread_cond_init(pthread_cond_t   *cond,   pthread_condattr_t
*cond_attr);
```

függvénnyel inicializáljuk a feltételes változót, ahol, ha a cond_attr NULL, akkor a feltételes változó az alapértelmezett paraméterekkel jön létre, egyébként, mivel a jelenlegi Linux-implementáció nem definiál attribútumokat a feltételes változók számára, a cond_attr paramétert a pthread_cond_init függvény figyelmen kívül hagyja.

Feltételes változót a

```
#include <pthread.h>


int pthread_cond_destroy(pthread_cond_t *cond);
```

függvénnyel szüntethetünk meg. Ekkor egy szál sem várakozhat a feltételes változóra, ez utóbbi feltétel ellenőrzésében ki is merül a függvény jelenlegi Linux-implementációjának tevékenysége.

Jelzést a

```
#include <pthread.h>


int pthread_cond_signal(pthread_cond_t *cond);
```

függvénnyel küldhetünk.

Ha azt szeretnénk, hogy az összes szál, amely egy feltételes változóra várakozik, megkezdje futását, akkor a

```
#include <pthread.h>


int pthread_cond_broadcast(pthread_cond_t *cond);
```

függvényt használjuk.

Ne felejtsük el a fenti két függvény használata előtt lefoglalni a mutexet, majd utána felszabadítani! (Ha rendeltetésszerűen használjuk a feltételes változót, nem valószínű, hogy ezt a hibát elkövetjük).

Ha várakozni szeretnénk egy feltételes változóra, akkor a

```
#include <pthread.h>


int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

függvényt használjuk. A függvény meghívása előtt le kell foglalnunk a mutexet, majd a meghívás után el kell engednünk, ugyanis visszatérés előtt ez a függvény „visszaszerzi”, azaz újra lefoglalja a mutexet.

Amennyiben csak meghatározott ideig szeretnénk várakozni, akkor a

```
#include <pthread.h>


int  pthread_cond_timedwait(pthread_cond_t  *cond,  pthread_mutex_t
*mutex, const struct timespec *abstime);
```

függvényt használjuk, a mutex szempontjából ugyanúgy, mint a pthread_cond_wait hívást. Az egyetlen különbség, hogy a függvény az abstime argumentumban megadott időnél többet nem vár, hanem újra lefoglalja a mutexet, és ETIMEDOUT értékkel tér vissza.

8.2.3.4 Szemaforok

A POSIX-szemaforok egyetlen szinkronizációs objektumot jelentenek (szemben a System V IPC szemafortömbjével).

Névtelen szemafort a következő függvénnyel hozhatunk létre:

```
#include <semaphore.h>


int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Ha a pshared nem 0, akkor a szemaforhoz más processzek is hozzáférhetnek, egyébként csak az adott processz szálai. A szemafor értéke a value paraméterben megadott szám lesz. A Linux nem támogatja a processzek közötti szemaforokat, ezért nem nulla pshared argumentum esetén a függvény mindig hibával tér vissza.

Egy sem szemafort a

```
#include <semaphore.h>


int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

függvényekkel foglalhatunk le. Ha a sem szemafor értéke pozitív, mindkét függvény lefoglalja, és visszatér 0-val. Ha a szemafort nem lehet lefoglalni (vagyis a szemafor értéke 0), a sem_trywait azonnal visszatér EAGAIN értékkel, míg a sem_wait várakozik a szemaforra. Ez utóbbi várakozást vagy a szemafor állapota (az értéke nagyobb lesz, mint nulla), vagy egy jelzés szakíthatja meg.

A szemafort használat után a

```
#include <semaphore.h>


int sem_post(sem_t *sem);
```

függvénnyel engedhetjük el. A szemafor aktuális értékét a

```
#include <semaphore.h>


int sem_getvalue(sem_t *sem, int *sval);
```

függvénnyel kérdezhetjük le. Ha a szemafort lefoglalták, akkor a visszatérési érték nulla vagy egy negatív szám, melynek abszolút értéke megadja a szemaforra várakozó processzek számát. Ha a sval pozitív, a szemafor aktuális értékét jelenti.

Szemafort a

```
#include <semaphore.h>


int sem_destroy(sem_t *sem);
```

függvénnyel szüntethetünk meg.

8.3 Valósidejű Linux alkalmazások fejlesztése

Az RT-Preempt Kernel patch készítői számára alapvető cél volt, hogy a fejlesztői API a valósidejű kernelek esetén ne térjen el a normál kerneltől. A valósidejű rendszer különbségeit a kernel lekezeli, így az alkalmazás fejlesztők számára a valósidejű rendszer nem jelent különbséget. Minden rendszerhívást ugyanúgy használhatunk.

Azonban annak érdekében, hogy az alkalmazás futásideje determinisztikus legyen néhány óvintézkedést meg kell tennünk az alkalmazásunkban:

- Az alkalmazásnak valósidejű ütemezést és prioritást kell használnia.
- Az alkalmazás memória területét végig a fizikai memóriában kell tartanunk.
- A stack terület növeléséből fakadó laphibákat meg kell előznünk.

8.3.1 Valósidejű ütemezés és prioritás

A valósidejű ütemezés két fajtáját is megismertük korábban. A FIFO és a Round Robin metódusok abban különböznek, hogy ha a legmagasabb valósidejű prioritással rendelkező folyamatokból több is futásra kész, akkor milyen sorrendben fussanak.

A valósidejű prioritás értéke 1-99 között állítható be. A magasabb érték magasabb prioritást jelent. A Kernel megszakítás rutinok és a taskletek 50-es prioritás értéken futnak.

Az ütemezési stratégiát és a prioritást közös függvényekkel állíthatjuk be. Folyamatok esetén a következő függvényt használjuk:

```
#include <sched.h>


int   sched_setscheduler(pid_t   pid,   int   policy,   const   struct
sched_param *param);
struct sched_param {
            ...
            int sched_priority;
            ...
       };
```

Ahol a pid a folyamat azonosítója, a policy az ütemezési stratégia, és a sched_priority a prioritás érték.

Szálakra is beállíthatjuk ezt az értéket. A függvény paraméterezése hasonló, mint az előzőé:

```
#include <pthread.h>


pthread_setschedparam(pthread_t  thread,  int  policy,  const  struct
sched_param *param);
```

8.3.2 Memória terület fizikai memóriában tartása

A folyamatunk memóriaterületének lapjai nem kerülhetnek ki a fizikai memóriából a háttértárra. Ha megengednénk a lapcsere partíció, vagy állomány használatát a folyamatra, akkor az egyes algoritmusok futásideje kiszámíthatatlanná válik, mert előfordulhat, hogy be kell egy vagy több lapot töltenie a rendszernek a háttértárról. Még rosszabb a helyzet, ha fizikai lapokat is fel kell szabadítani, hogy a betöltendő lapok elférjenek.

Valósidejű folyamat esetén nem engedhetünk meg ilyen kiszámíthatatlan késedelmeket. Ezt a folyamat lapjainak zárolásával érhetjük el:

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


int mlockall(int flags);
```

A flag paraméterek az alábbiak lehetnek:

- MCL_CURRENT: A jelenleg használt lapokat benntartja
- MCL_FUTURE: A később lefoglalt lapokat is benntartja

Vagyis a kódban az alábbi formában használjuk:

```
mlockall(MCL_CURRENT|MCL_FUTURE)
```

Ez az aktuálisan használt és a jövőben lefoglalt memórialapokat is a fizikai memóriában tartja. Természetesen az új lapok allokálása kiszámíthatatlan késleltetéseket okozhat, ezért azokat a nem kritikus részekben kell megtennünk.

8.3.3 A stack okozta laphibák megelőzése

Az előző fejezetben említett metódusnak még egy hiánya van. Futás közben az algoritmusunk stack használata nőhet. A stack növekedése esetén ha már betelt egy memórialap, akkor keletkezik egy laphiba, aminek hatására a Kernel újabb lapot allokál. Ez késleltetést okoz az algoritmus futásában. Megelőzni úgy tudjuk, hogy előre elvégezzük az allokációt. Természetesen ehhez ki kell számolni a program stack igényét. Ha ezt előre lefoglaljuk, akkor a későbbiek során nem allokál a rendszer új memórialapokat.

Az előzetes allokáció elvégzéséhez létre kell hoznunk egy megfelelően nagy tömböt és a tömb írásakor megtörténnek a laphibák és a foglalások.

Egy példa implementáció:

```
#define MAX_STACK (4*1024)
void stack_prefault()
{
unsigned char tmp[MAX_STACK];
memset(tmp, 0, MAX_STACK);
}
```

A program elején meg kell hívnunk ezt a függvényt.

8.4 Hálózat kezelés

A Linux a hálózat kezeléshez a Berkley socket API-t implementálja, illetve használja.

Mint a Linux más erőforrásai, a socketek is a fájlabsztrakciós interfészen keresztül vannak implementálva a rendszerbe. Létrehozásuk a socket() rendszerhívással történik, amely egy állományleíróval tér vissza. Miután a socketet inicializáltuk, a read() és write() függvényekkel kezelhetők, mint minden más állományleíró. (Ez azonban nem célszerű.) Használat után pedig a close() függvénnyel le kell zárnunk.

8.4.1.1 Socketek létrehozása

Új socketeket a socket() rendszerhívással hozhatunk létre, amely a nem inicializált socket állományleírójával tér vissza. Létrehozásakor a sockethez egy meghatározott protokollt rendelünk, azonban ezek után még nem kapcsolódik sehova. Ebben az állapotában meg nem olvasható vagy írható.

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


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

Mint az open(), a socket() is 0-nál kisebb értékkel tér vissza hiba esetén, és ha sikeres, az állományleíróval, amely 0 vagy nagyobb.

Három paraméter definiálja a használandó protokollt. Az első a protokollcsaládot adja meg és értéke a következő lista valamelyike:

Protokoll	Jelentés
`PF_UNIX, PF_LOCAL`	Unix domain (gépen belüli kommunikáció)
`PF_INET`	IPv4 protokoll
`PF_INET6`	IPv6 protokoll
`PF_IPX`	Novell IPX
`PF_NETLINK`	A kernel felhasználói interfésze
`PF_X25`	X.25 protokoll
`PF_AX25`	AX.25 protokoll, az amatőr rádiósok használják
`PF_ATMPVC`	Nyers ATM-csomagok
`PF_APPLETALK`	AppleTalk

Protokoll	Jelentés
`PF_PACKET`	Alacsony szintű csomaginterfész

A következő paraméter, a type, a protokollcsaládon belül a kommunikáció módját definiálja az alábbi értékek egyikével:

Típus	Jelentés
`SOCK_STREAM`	Sorrendtartó, megbízható, kétirányú, kapcsolatalapú bájtfolyam-kommunikációt valósít meg.
`SOCK_DGRAM`	Datagramalapú (kapcsolatmentes, nem megbízható) kommunikáció.
`SOCK_SEQPACKET`	Sorrendtartó, megbízható, kétirányú, kapcsolatalapú kommunikációs vonal, fix méretű datagramok számára.
`SOCK_RAW`	Nyers hálózati protokoll hozzáférést tesz lehetővé.
`SOCK_RDM`	Megbízható datagramalapú kommunikációs réteg. (Nem sorrendtartó.)
`SOCK_PACKET`	Elavult opció.

Az utolsó paraméter, a protocol paraméter, a protokollcsaládon belül konkretizálja a protokollt. A családokon belül általában csak egy protokoll létezik, amely egyben az alapértelmezett. Így ennek a paraméternek az értéke leggyakrabban 0. A PF_INET családon belül a TCP az alapértelmezett folyamprotokoll, és az UDP a datagramalapú.

8.4.1.2 A hardverfüggő különbségek feloldása

A hálózati kommunikáció bájtok sorozatán alapszik. Azonban egyes processzorok különböző módon tárolják a különböző adattípusokat. Ennek a nehézségnek az áthidalására definiáltak egy hálózati bájtsorrendet (network byte order), ami a magasabb helyértékű bájtot küldi előbb („a nagyobb van hátul” – big endian). Azoknál az architektúráknál, ahol az ún. hosztbájt-sorrendje ellenkező („a kisebb van hátul” – little endian) – ilyenek például az Intel 8086 alapú processzorok –, konverziós függvények állnak rendelkezésünkre, amelyeket a következő táblázat foglal össze.

Függvény	Leírás
`ntohs`	Egy 16-bites számot a hálózatibájt-sorrendből a hosztbájtsorrend sorrendjébe (big-endian–little-endian) vált át.
`ntohl`	Egy 32-bites számot a hálózatibájt-sorrendből a hosztbájtsor-rendjébe (big-endian–little-endian) vált át.

Függvény	Leírás
`htons`	Egy 16-bites számot a hosztbájtsorrendjéből hálózatibájt-sorrendbe (little-endian–big-endian) vált át.
`htonl`	Egy 32-bites számot a gép bájtsorrendjéből hálózati hosztbájtsorrendbe (little-endian–big-endian) vált át.

Azokon az architektúrákon, ahol nem szükséges ez a konverzió, ezek a függvények a megadott argumentum értékekkel térnek vissza, vagyis hordozható kód esetén mindenképpen alkalmazzuk ezeket a függvényeket.

8.4.1.3 A socketcím megadása

A legalacsonyabb szintű címreprezentáció a sockaddr struktúra:

```
struct sockaddr
{
  unsigned short sa_family;    /* címcsalád, AF_xxx */
  char           sa_data[14];  /* 14 bájtos protokoll cím */
};
```

Hátránya a fenti adatszerkezetnek, hogy a sa_data adattagot kézzel kell kitöltenünk a port számával és az IP-címmel, amely legalábbis elég nehézkes feladat. Ennek megkönnyítésére rendelkezésre áll a

```
struct sockaddr_in
{
  short int          sin_family;  /* Címcsalád = AF_INET */
  unsigned short int sin_port;    /* A port száma */
  struct in_addr     sin_addr;    /* IP cím */
  unsigned char      sin_zero[8]; /* struct sockaddr vége */
};
```

struktúra, amelynek nevében az in az internet rövidítése. Fontos, hogy a sin_port paramétert hálózati byte sorrendben adjuk meg! Az in_addr adattag típusa:

```
struct in_addr
{
  unsigned long int s_addr;
}
```

Ez utóbbi struktúra és az IP-cím megszokott reprezentációja közötti konverziót több függvény is segíti. Az

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

```
#include <netinet/in.h>
#include <arpa/inet.h>

in_addr_t inet_addr(const char cp);
```

pontozott formátumú IP-cím string reprezentációjából in_addr típusú struktúrát készít hálózati bájtsorrendben. Például:

```
struct sockaddr_in internet_address;
internet_address.sin_addr.s_addr = inet_addr("152.66.70.136");
```

Hasonló funkciót lát el a

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_aton(const char *cp, struct in_addr *inp);
```

függvény, amelynek visszatérési értéke jelzi a hibát.

Az

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char *inet_ntoa(struct in_addr in);
```

hívás a hálózati bájtsorrendben megadott 32-bites IP-címet alakítja pontozott formátummá. A visszatérési érték egy statikusan foglalt pufferben tárolódik, amit a soron következő hívások felülírnak.

8.4.1.4 Név- és címfeloldás

Ha tudjuk egy hosztnak a nevét, szükségünk lehet IP-címére is. Ha a hoszt nevéből az IP-címét szeretnénk megállapítani, akkor névfeloldásról beszélünk. Ezt a funkciót Linux alól a

```
#include <netdb.h>
extern int h_errno;

struct hostent *gethostbyname(const char *name);
```

függvénnyel érhetjük el. Ennek visszatérési értéke:

```
#include <netdb.h>
```

```
struct hostent
{
  char* h_name;     /* A hoszt hivatalos neve */
  char** h_aliases;  /* NULL terminált tömb a host más neveire */
  int   h_addrtype; /* A cím típusa, jelenleg mindig AF_INET */
  int   h_length;   /* A cím hossza bájtokban */
  char**h_addr_list;/* NULL terminált lista – az IP címek */
};
```

```
#define h_addr  h_addr_list[0] / *Kompatibilitás miatt */
```

Az inverz névfeloldást AF_INET típusú socketek esetén a

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


struct hostent *gethostbyaddr(const char *addr, int len,  int type);
```

függvényen keresztül érhetjük el. Az első paraméter egy in_addr típusú struktúrára mutató pointer, a második argumentum az első argumentumban megadott struktúra mérete, a harmadik paraméter általában AF_INET.

8.4.1.5 A kapcsolat felépítése

Ha egy összeköttetés-alapú kapcsolatot (stream socketet) hoztunk létre, akkor hozzá kell kapcsolódnunk egy másik géphez. A socket kapcsolódása aszimmetrikus művelet, vagyis a létrejövő kapcsolat két oldalán a feladatok eltérnek egymástól.

A kliensoldalon a socket létrehozása után kapcsolódunk (connect) a szerverhez. Ha a kapcsolódás sikeres (a szerver elfogadja a kapcsolódási kérelmet), akkor a socketen keresztül tudunk adatot küldeni és fogadni. Ezt a típusú socketet klienssocketnek hívjuk.

A szerver és a kliens közti lényeges különbség a kapcsolat felépítésében és a kezelendő socketek számában rejlik. Ennek megfelelően a szerveroldalon kétfajta socket található. Az egyiket a továbbiakban szerversocketnek hívjuk. Ennek az a funkciója, hogy várakozik (listen) egy adott porton, amelyet előre megadtunk (bind), és arra vár, hogy a kliensoldali socketek kapcsolódjanak (connect) hozzá. Amennyiben egy kapcsolódási kérelem érkezik, a szerver elfogadja (accept), minek eredményeképpen létre jön egy klienssocket. Ezen keresztül történik az adatcsere. A kapcsolódási kérelem elfogadása után a kommunikáció a szerversockettől függetlenül zajlik, a szerversocket csak vár további kapcsolódási kérelmekre. A szerveroldalon ezért gyakran több klienssockettel kell törődnünk.

8.4.1.6 A socket címhez kötése

Mind a szervernek, mind a kliensnek meg kell adnia, hogy a rendszer melyik címet használja a socketjéhez. Ez egy helyi cím, ahol a szerver várja a bejövő kapcsolatokat (a kliensnél ezt a címet adjuk meg, amikor kapcsolódni szeretnénk). A kliensprogramnál is lehetőség van

lokális cím megadására, azonban ez nem kötelező, mert a rendszer automatikusan is tud generálni egyet kapcsolódáskor.

A címhozzárendelés műveletét kötésnek (binding) nevezzük, és a bind() rendszerhívással hajthatjuk végre.

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


int bind(int sock, struct sockaddr *my_addr, socklen_t addrlen);
```

Az első paraméter a socket leírója, a második a címet leíró struktúra, az utolsó a címet leíró struktúra hossza.

8.4.1.7 Várakozás a kapcsolódásra

A socket-et létrehozása után a szerver processz a bind() rendszerhívással hozzárendeli egy címhez, ahol várhatja a kapcsolódókat. A processz a listen() rendszerhívással állítja be a socketre, hogy fogadja ezeket a kapcsolódásokat. Ezek után a kernel kezeli a kapcsolódási igényeket. Ilyenkor azonban még nem épül fel azonnal a kapcsolat. A várakozó processznek az accept() rendszerhívással kell elfogadni a kapcsolódást. A még az accept()-tel el nem fogadott kapcsolódási igényeket pending connection-nek nevezzük.

Normál esetben az accept() függvény blokkolódik, amíg egy kliens kapcsolódni próbál hozzá. Természetesen állíthatjuk az fcntl() rendszerhívás segítségével nem blokkolódó mód-ban is. Ilyenkor az accept() azonnal visszatér, amikor egyetlen kliens sem próbál kapcsolódni. A select() rendszerhívást is alkalmazhatjuk a kapcsolódási igények észlelésére.

A listen() és az accept() függvények formája:

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


int listen(int sock, int backlog);
int accept(int sock, struct sockaddr *addr, socklen_t *addrlen);
```

Első paraméterként mindkét függvény a socket leíróját várja. A listen() második paramétere, a backlog, amely megadja, hogy hány kapcsolódni kívánó socket kérelme után utasítsa vissza az újakat, ami a pending connection várakozási lista mérete.

Az accept() fogadja a kapcsolódásokat. Visszatérési értéke az új kapcsolat leírója, egy kliens socket. Az addr és addrlen paraméterekben a másik oldal címét kapjuk meg. Az addrlen egy integer szám, amely megadja az addr változó méretét.

8.4.1.8 Kapcsolódás a szerverhez

A kilens a létrehozott socketet a szerverhez hasonlóan a bind() rendszerhívással hozzárendelheti egy helyi címhez. Azonban ezzel a kliens program általában nem törődik, a kernelre bízza, hogy ezt automatikusan megtegye.

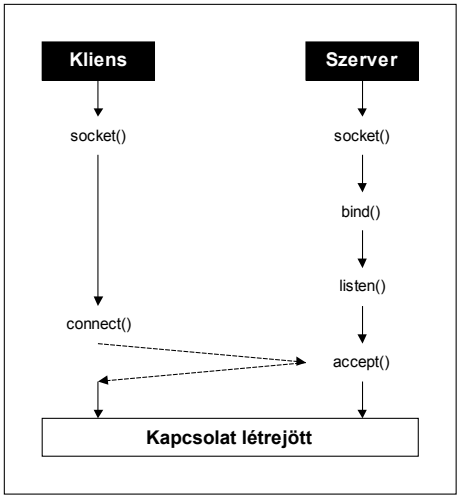
Ezek után a kliens a connect() rendszerhívással kapcsolódhat a szerverhez.

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

```
int connect(int sock, struct sockaddr *addr, socklen_t addrlen);
```

Az első paraméter a socket leírója, a további paraméterek a szerversocket címét adják meg.

A kapcsolat felépülését az alábbi ábrán láthatjuk.



4. ábra A socket kapcsolat felépülése

8.4.1.9 Összeköttetés alapú kommunikáció

A kapcsolat létrejötte után megkezdődhet az adatok átvitele, a tényleges kommunikáció. Összeköttetés alapú kapcsolat esetén a

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

int send(int s, const void *msg, size_t len, int flags);
```

függvényt használjuk adatok küldésére. Az első argumentum a socket leírója, a második az elküldendő adat pufferére mutató pointer, a harmadik az elküldendő adat mérete. A flags argumentumban megadott jelzőbitek jelentése:

Jelzőbit	Leírás
`MSG_OOB`	Soron kívüli sürgős adatcsomagot (out-of-band data) küld. Általában jelzések hatására használják.
`MSG_DONTROUTE`	A csomag nem mehet keresztül az útválasztókon, csak közvetlen hálózatra kapcsolódó gép kapja meg.
`MSG_DONTWAIT`	Engedélyezi a nem blokkoló I/O-t, EAGAIN-nel tér vissza, majd a már tárgyalt módon a select() utasítással kaphatunk értesítést a művelet kimeneteléről.

Jelzőbit	Leírás
`MSG_NOSIGNAL`	Adatfolyam alapú kapcsolat esetén a program nem kap SIGPIPE jelzést.

A send párja a

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

int recv(int s, void *buf, size_t len, int flags);
```

függvény, amely egy puffert, annak méretét, valamint egy jelzőbitet vár. A jelzőbitek értékét a következő táblázat foglalja össze.

Jelzőbit	Leírás
`MSG_OOB`	Soronkívüli adat fogadása.
`MSG_PEEK`	Az adat beolvasása anélkül, hogy a beolvasott adatot eltávolítaná a pufferből. A következő recv hívás ugyanazt az adatot még egy-szer kiolvassa.
`MSG_WAITALL`	Addig nem tér vissza, míg a megadott puffer meg nem telik, vagy egyéb rendhagyó dolog nem történik, pl. jelzés érkezik.
`MSG_NOSIGNAL`	Ugyanaz, mint a send függvény esetén

8.4.1.10 Összeköttetés nélküli kommunikáció

Összeköttetés alapú kommunikáció esetén a kommunikációt megbízható adatfolyamként foghatjuk fel. Ez egyfelől kényelmes megoldás, mert nem kell foglalkoznunk azzal, hogy az adat, amelyet elküldtünk, valóban célba ért-e, ugyanis a kommunikációt végrehajtó függvények jelzik a hibát vagy a kapcsolat bontását.

Az adatok szinkronizációja azonban több csomagforgalmat is jelent egyben. Ezért, olyan alkalmazások esetén, ahol „nem túl nagy probléma”, ha elveszik „egy-egy” csomag, ott alkalmazhatunk összeköttetés nélküli megoldást, ami egyben egy gyorsabb kommunikációt eredményez. Ilyenek tipikusan a multimédia-alkalmazások, hiszen, ha egy zene hangmintáit küldjük el, nem jelent számottevő minőségromlást egy-egy keret kimaradása, valamint, ha későn érkezik, nem is tudunk mit kezdeni vele, hiszen már előrébb tartunk a lejátszásban.

Összeköttetés nélküli kapcsolathoz a szállítási rétegben UDP-t (User Datagram Protocol) használunk TCP helyett.

Ebben az esetben azonban nem garantált, hogy

- a csomag célba ér.
- az elküldött adatok ugyanabban a sorrendben érkeznek is meg, melyben küldtük.

Az UDP-csomagok mérete csak legfeljebb 64 kB lehet.

A küldő inicializálása hasonló a TCP-nél látottakhoz:

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

int sd;
struct sockaddr_in addr;
/* datagramsocket létrehozása */
sd = socket(PF_INET, SOCK_DGRAM, 0);

/* A cél címének törlése */
memset(&addr, 0, sizeof(addr));
/* A címcsalád: IP */
addr.sin_family = AF_INET;
/* A cél port száma */
addr.sin_port = htons(10035);
/* A cél címének beállítása */
inet_aton(”152.66.70.16”, &addr.sin_addr);
```

Ezek után elküldjük a kívánt adatot:

```
/* buflen méretű adat a buf változóban elküldése az addr címre */
sendto(sd, buf, buflen,0,&addr, sizeof(addr));
A fentiek alapján összeköttetés nélküli kapcsolatok esetén küldésre a
#include <sys/types.h>
#include <sys/socket.h>

int sendto(int sd, const void *msg, size_t len, int flags, const
struct sockaddr *to, socklen_t tolen);
```

függvényt használjuk, amelynek a címzett socket címét, illetve annak méretét is meg kell adnunk a függvény utolsó két paramétereként. A flags jelzőbitek megegyeznek a send hívásnál részletezettekkel.

A vevő oldalon is a TCP-hez igen hasonlóan járunk el:

```
/* ezen a porton figyelünk */
addr.sin_port = htons(1035);
```

```
/* bárhonnan elfogadjuk a csomagot */
addr.sin_addr.s_addr = INADDR_ANY;
/* hozzárendeljük az socketet a helyi porthoz */
bind(sd, &addr, sizeof(addr));
```

Ezek után fogadjuk a csomagokat:

```
char buf[MAXBUF];
int len = sizeof(addr);
```

/* várunk, ameddig nem érkezik csomag */

```
int bytesread = recvfrom(sd, buf, bufsize, 0, &addr, &len);
```

Ha a megadott puffer rövidebb, mint amekkora hosszúságú adat érkezett, akkor a recvfrom a csomag végét automatikusan levágja.

Összeköttetés nélküli kapcsolat esetén tehát az adatfogadás a

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

int recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);
```

függvénnyel történik. A flags paraméter megegyezik a recv függvénynél tárgyaltakkal. Ameddig nem érkezik adat, a recv függvény várakozik. A blokkolódás elkerülésére bármely TCP esetén alkalmazott technika használható (többszálúság, multiplexing stb.).

9 Kernel modulok fejlesztése

9.1 Kernelmodulok

A Linux kernel monolitikus. Vagyis egyetlen nagy program, amely tartalmaz minden funkciót, és belső adatstruktúrát, amely az operációs rendszerhez tartozik. (Az alternatíva a mikrokernel volna, amely minden funkcionális elemet külön egységekben tartalmaz, és jól meghatározott kommunikáció van az egyes részek között. Ezáltal minden új komponens, funkció hozzáadása a kernelhez nagyon időigényes feladat volna. A korai kernelverzióknál, ha egy új eszközt tettünk a gépbe, akkor ténylegesen újra kellett konfigurálni, és fordítani a kernelt.)

Azonban az 1.2-es verzió óta már teljes támogatást kapunk a futás közben betölthető kernelmodulok használatára. Ez lehetővé teszi, hogy az operációs rendszer egyes komponenseit dinamikusan betöltsük és eltávolítsuk, attól függően, hogy szükségünk van-e rá vagy sem. A Linux-modulok olyan tárgykódú binárisok, amelyeket a rendszer indítása után bármikor dinamikusan hozzákapcsolhatunk a kernelhez, majd lebonthatjuk (unlink) és eltávolíthatjuk a memóriából, ha már nincs rá szükségünk. A legtöbb eszközvezérlő modulként került megvalósításra, és ez a megoldás javasolt az egyéni eszközvezérlők készítésénél is. (Magát a fejlesztést is nagyban könnyíti és gyorsítja.)

9.1.1 Hello modul világ

Indulásként nézzük meg egy egyszerű példán, hogy hogyan is néz ki egy kernelmodul forráskódja. Ez a kód a kernel 2.6.x-es verziója alatt fordul és működik. (A nagyobb kernelverzió-váltások általában a modulok kisebb-nagyobb módosításaival járnak együtt.)

```
/* hellomodule.c - Egyszeru kernel modul. */

#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void)
{
  printk("Hello modul vilag!\n");
  return 0;
}

void cleanup_module(void)
{
  printk("Viszlat modul vilag!\n");
```

```
}

MODULE_LICENSE("GPL");
```

Ez a modul a lehető legegyszerűbb. Minden modul minimum két függvénnyel rendelkezik. Az egyik a modul betöltésekor (`init_module`), a másik az eltávolításkor (`cleanup_module`) hajtódik végre. Ezek formája a 2.6-os kernel esetén a példában látható. Azonban a gyakorlatban nem a függvények beépített neveit használjuk, hanem a saját nevekkel létrehozott függvényeket regisztráljuk be. Ez látható a következő példában:

```
/* hellomodule.c - Egyszeru kernel modul. */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

static int __init hello_init(void)
{
  printk("Hello modul vilag!\n");
  return 0;
}

static void __exit hello_exit(void)
{
  printk("Viszlat modul vilag!\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_DESCRIPTION("Hello module");
MODULE_LICENSE("GPL");
```

Regisztrálásukat a `module_init()` és `module_exit()` függvényekkel végezzük. Az üzenetek kiírására a `printk()` függvényt használjuk, amely a `printf()` függvényhez hasonló és a Linux-kernelben van definiálva. A modul betöltésekor a Linux-kernelhez linkelődik, ezáltal ez a függvény is használható lesz. (A `printf()` nem használható, mert az a felhasználói könyvtárban van definiálva.)

Az `__init` attribútum hatására az inicializációs folyamat után az init függvény megszűnik, és a memória felszabadul. Azonban ezt a hatását csak akkor fejti ki, ha a kernelbe fordítjuk a kódot és nem modulként használjuk. Utóbbi esetben nincs hatása. A `__exit` attribútum hatására a cleanup funkciót figyelmen kívül hagyja a fordító, ha a kódot a kernelhez fordítjuk (mivel ilyenkor nincs szükség rá). Ha modulként használjuk, akkor nincs hatása. Így ezekkel a címkékkel csökkenthetjük a kernel memória felhasználását.

A modul végén makrókkal megadhatjuk a modul készítőjét, leírását, és licenszinformációit.

9.1.2 Fordítás

A kernel modulok lefordításához felhasználjuk a kernelforrás `Makefile`-jait. A metódus a következő:

A modul könyvtárában létre kell hoznunk egy `Makefile` állományt. Ebben a modult az alábbi sorral vehetjük fel a fordítási listába:

```
obj-m += hellomodule1.o
```

Ezt követően meg kell hívnunk a `make` parancsot úgy, hogy a kernel forrás fő könyvtárában lévő `Makefile` állományt használja. Ugyanakkor át kell adnunk a SUBS paraméterrel a modul forrásunk könyvtárát. Ezt követően még a „modules” célt kell megadni a fordításhoz. Ezzel a parancssor az alábbi lesz:

```
make -C <kernel könyvtár> SUBDIRS=<modul forrás könyvtár> modules
```

9.1.3 A modulok betöltése és eltávolítása

9.1.3.1 insmod/rmmod

A lefordított modult a következő paranccsal tölthetjük be:

```
insmod hellomodule.ko
```

Ez hozzálinkeli a modult a kernelhez, továbbá végrehajtódik az inicializáló metódusunk. Ezáltal kiírja az üdvözlő szöveget a konzolra. (Amennyiben egyéb terminál programot használunk, akkor a konzol üzenetek nem jelennek meg rajta. Ilyenkor a `dmesg` paranccsal listázhatjuk ki ezeket az üzeneteket.)

A betöltött modulokat az `lsmod` paranccsal ki is listázhatjuk. Ez az aktuálisan betöltött modulokról és a köztük fenn álló kapcsolatról is mutat információkat.

A modul eltávolítására a következő parancs szolgál:

```
rmmod hellomodule
```

Ilyenkor meghívódik a modulunk tisztogató függvénye, amely kiírja a konzolra a búcsú szöveget. Továbbá unlinkelődik a modul és eltávolítódik a memóriából.

9.1.3.2 modprobe

A másik lehetőség a `modprobe` program használata, azonban ehhez először el kell helyeznünk a modult a modulok hivatalos tároló könyvtárába:

```
/lib/modules/<kernel verzió>/
```

Ezt követően le kell futtatni a `depmod -a` parancsot. Innentől a `modprobe` képes betölteni a modulokat és eltávolítani.

A `depmod` parancs valójában a függőségi listát állítja elő a modulokhoz. Így a `modprobe` képes rá, hogy ne csak a megnevezett modult töltse be, hanem a függőségeit is.

A `modprobe` konfigurációs állománya illetve könyvtára:

```
/etc/modprobe.conf
/etc/modprobe.d/
```

A konfigurációs állományban az alábbi parancsokat használhatjuk:

Parancs	Jelentés
`alias`	Beállíthatunk egy másik nevet a modulnak, illetve leggyakrabban
`specifikus`	modulokat rendelünk eszköz nevekhez, hogy azok automatikusan betöltődhessenek.
`options`	Paraméterezést adhatunk meg a modul betöltéséhez.
`install`	A modul betöltése helyett a megadott komplex parancssort hajtja végre.
`remove`	Mint az install, csak eltávolításra.
`include`	Egy állományt vagy egy egész könyvtár tartalmát beilleszthetjük.
`blacklist`	A megadott modulok belső alias beállításait figyelmen kívül hagyja a rendszer. (Ütközések, illetve egyes modulok automatikus betöltése ellen használjuk.)

9.1.4 A modulok és az alkalmazások közti különbség

Mielőtt tovább haladnánk szükséges megtárgyalnunk a kernelmodulok és az alkalmazások közti különbséget.

Először is kernelmodulok készítésénél lehetőleg felejtsünk el más nyelveket, és dolgozzunk C-ben.

Míg az alkalmazások általában egy feladaton dolgoznak az indítástól a végéig, addig a modulok csak regisztrálódnak, hogy később majd feladatokat teljesíthessenek, és az inicializáló függvényük már véget is ér. A modul másik belépési pontja a tisztogató függvény, amely a modul eltávolításakor aktivizálódik, közvetlenül előtte.

Programok fejlesztése során gyakran használunk olyan függvényeket, amelyek fejlesztői könyvtárakban vannak. Ilyenkor maga a program nem tartalmazza a függvényt, hanem a fordítási fázis után a linkelési fázisban oldja fel a rendszer ezeket a külső hivatkozásokat. Ilyen például a `printf()` függvény, amely a libc könyvtárban található. Ezzel szemben a modulok csak a kernellel linkelődnek, ezért csak olyan függvényeket használhatnak, amelyek

a kernelben vannak implementálva. Ezért kellett az előző példánkban is a printk() függvényt használnunk.

Mivel nem használhatunk fejlesztői könyvtárakat, így a hagyományos fejállományok sem szerepelhetnek a kódunkban. Minden header állomány, amely a Linux-kernelhez kapcsolódik a linux és az asm alkönyvtárakban található.

A kernelmodul adatait általában a fizikai memóriában tároljuk, vagyis nem kerülhet ki egy lapcsere folytán a merevlemezre. Természetesen a fizikai memória véges, ezért ne bánjunk vele pazarlóan. Nagy mennyiségű adathoz azonban lehetőségünk van virtuális memóriát is allokálni.

Lebegőpontos számításokat ne használjunk. Ha mégis szükséges, akkor nekünk kell gondoskodnunk az FPU állapotának lementéséről és visszaállításáról.

Az utolsó nagy eltérés a hibák lekezelésénél tapasztalható. Amíg a programok fejlesztése során egy segmentation fault nem veszélyes, és könnyen kiszűrhető, a kernelmodulban elkövetett hasonló hiba komoly következményekkel, akár még a rendszer összeomlásával is járhat. A hibakeresés is nehézkesebb, a korábbi fejezetekben tárgyalt módszerek nem használhatók. Jelen dokumentumban csak az egyszerűbb megoldásokat tárgyaljuk, amelyek kis modulok esetén jól alkalmazhatók.

További különbség a kernel modulok és az alkalmazások között, hogy ebben az esetben a kódunk kernel módban fut.

9.1.5 Felhasználói mód — kernel mód

Mint már volt róla szó, modulunk kernel módban fut, míg a programjaink a felhasználói módban.

Ennek hatása egyrészt a korlátozások hiányában jelentkezik. Mint ahogy az már a korábbi fejezetekben szerepelt, a felhasználói módban futó alkalmazások memóriahasználatát, az eszközökhöz való hozzáférését szabályozza az operációs rendszer. Az illegális hozzáféréseket megakadályozza, és így megvédi a rendszer többi részét. Kernel módban a kódunk számára minden megengedett. (Ezért kernel modulokat csak a root tölthet be.)

9.1.6 Egymásra épülő modulok

A modulok egymásra is épülhetnek. Ilyenkor ki kell exportálnunk a más modulok által használt szimbólumokat. Továbbá betöltéskor gondoskodnunk kell a függőségekről. De a `modprobe` is megoldhatja ezt helyettünk.

Ha egy modulból szimbólumokat akarunk kiexportálni, akkor definiálnunk kell benne az EXPORT_SYMTAB makrót. Ezt követően az exportálandó függvényeket az EXPORT_SYMBOL makró segítségével kell megadnunk.

9.2 Paraméter átadás modul számára

A modulok leggyakrabban egy-egy eszközvezérlőt implementálnak. Az eszközvezérlők sokszor rendelkeznek különféle beállításokkal (például az eszköz I/O címtartománya). Ezeket a beállításokat kontansként „bele is drótozhatjuk” a modulba, amely azt jelenti, hogy ha a

valamelyik beállításon módosítani szeretnénk, akkor újra kell fordítanunk a modult. Ez természetesen nagyon körülményes. A megfelelő megoldás az lenne, ha a modul betöltésekor is megadhatnánk beállításokat. Erre szolgál a Linux kernel paraméterátadási mechanizmusa.

A beállításokat paraméterekben adhatjuk át a modulnak. Ennek módja nem egyezik a felhasználói alkalmazásoknál megszokott parancssori argumentumokkal. Definiálhatunk olyan globális változókat, amelyeket az *insmod* parancs indításkor kitölt. A változó létrehozásakor megadhatjuk inicializációs értékét, amely egyben az alapértelmezett értéke is lesz a paraméternek. Ezt követően a *module_param()* makró valamelyik változatával meg kell adnunk, hogy az adott változó modul paraméternek számít, illetve típustól függően egyéb beállításokat.

```
module_param(név, típus, sysfs-jogok);
module_param_names(név, változónév, típus, sysfs-jogok);
module_param_array(név, típus, méret mutató, sysfs jogok);
module_param_array_named(név, változónév, típus, méret mutató, sysfs-jogok);
module_param_string(név, változónév, méret, sysfs-jogok);
```

A *név* a változó neve, a *típus* a változó típusa, a *sysfs jogok* a paraméter jogosultság beállításai a *sysfs* állományrendszeren belül. A *méret mutató* a tömb esetén opcionális paraméter. Ha megadunk egy változót illetve annak mutatóját, akkor a rendszer a változóban eltárolja a tömb elemeinek számát a paraméter átadáskor. A méret paraméter a string típus esetén a szöveg maximális hosszát adja meg beleértve a záró karaktert is. (A string típusú paraméternél a *név* ismétlődése nem nyomdahiba, hanem implementációs furcsaság.) A string paraméter másik megadási módja, ha típusnak *charp* értéket adunk meg, ami karakter mutatót jelent. Ekkor azonban nem kell megadnunk a méretet:

```
module_param(név, charp, sysfs-jogok);
```

A paraméterek leírását is célszerű megadnunk a modulban a *MODULE_PARM_DESC()* makró használatával. Erre azért van szükség, hogy a felhasználók a modulból megtudhassák a paraméterek jelentését. A makró formája:

```
MODULE_PARM_DESC(név, leírás);
```

A paramétereket a leírásaikkal a `modinfo` parancs segítségével nézhetjük meg:

```
modinfo helloparam.ko
```

A paramétereket az *insmod* parancs használatakor állíthatjuk be. Nem szükséges minden paramétert megadnunk. Amelyiket nem adjuk meg annak az értéke az lesz, amivel a C kódban inicializáltuk az adott statikus változót.

```
insmod helloparam.ko param1="hello" param2=5
```

Ha a *modprobe* parancsot használjuk a modul betöltésére, akkor a *modprobe.conf* állományban kell megadnunk a paramétereket.

Ha a *sysfs* jogosultságok beállításánál adtunk olvasási vagy írási jogosultságot a paraméterekre, akkor utólag is kiolvashatjuk vagy módosíthatjuk az értéküket. Ehhez a

/sys/module/<modul név>/parameters könyvtárban kell körülnéznünk. Itt minden egyes paraméterhez találhatunk egy virtuális állományt. Ezeket olvashatjuk és írhatjuk.

Természetesen utólagos módosítások esetén számolnunk kell esetleges konkurencia problémákkal. Erre példaként nézzük meg a következő esetet. Egy több processzoros gépen futtatunk két alkalmazást párhuzamosan. Az egyik alkalmazás használja a kernel modul szolgáltatását, míg a másik módosítja a kernel modul paraméterét a *sysfs* állományrendszeren keresztül. Ha a paraméter összetett (szöveg, tömb), akkor több gépi művelettel történik az írása illetve az olvasása. Vagyis a két alkalmazás által kiváltott paraméter kezelési folyamatok átlapolódhatnak, ezáltal kritikus versenyhelyzetet eredményezhet.

9.3 Karakteres eszközvezérlő

Ebben a fejezetben egy **karaktertípusú eszközvezérlő** modulját fogjuk elkészíteni. Azért ezt a típust, mert általában ez illeszkedik az egyszerűbb hardver eszközökhöz, illetve ez a legegyszerűbben érthető eszközvezérlő-fajta.

A Unix-rendszerek az eszközöket a */dev* könyvtárban található speciális állományokként teszik elérhetővé a felhasználók számára, illetve a felhasználói módban futó programok is álllományokként használják azokat. Ezért eszközvezérlő létrehozásakor az első feladatunk, hogy hardver eszközeink funkcióit állománykezelésre képezzük le: megnyitás, írás, olvasás, bezárás.

Az eszközvezérlő modulnak az állománykezeléssel kapcsolatos funkciók teszik ki az egyik részét. A másik részük a modulkezeléshez tartozó inicializáló és tisztogató függvények, amelyeket az előző fejezetben láthattunk. Egy eszközvezérlő esetén ezekben a függvényekben regisztráljuk be a kernel nyilvántartásába az eszközünket, illetve távolítjuk el a bejegyzést.

9.3.1 Fő és mellékazonosító (major és minor number)

Ahogy említettük, az eszközünk eléréséhez szükségünk van speciális állományokra, amelyeket az *mknod* paranccsal vagy rendszerhívással hozhatunk létre. A korábbi fejezetekben láthattuk, hogy a létrehozásukhoz szükséges a típus, egy fő és egy mellékazonosító (**major** és **minor** number) megadása.

A **fő** azonosító azonosítja a speciális állományhoz tartozó modult. A kernel ennek alapján választja ki, hogy mely eszközvezérlő modulhoz forduljon, ha egy program ezt az állományt nyitja meg.

A **mellék**azonosítót az eszközvezérlő program használhatja, amennyiben több állományinterfészt is kezel. Ezt nekünk kell kezelnünk a modulon belül, a kernel ezzel nem foglalkozik[15]. Például a soros interfész fő azonosítója 4, azon belül a mellékazonosítók határozzák meg a konkrét eszközök interfészeit. Például az első soros port (/dev/ttyS0) fő azonosítója 4, mellék azonosítója 64[16].

[15] Itt jegyezzük meg, hogy a kernel kizárólag a fő azonosítót használja a modul azonosítására. A modul neve, megjelenése az állományrendszerben kizárólag a könnyebb kezelhetőséget segíti elő emberi felhasználók számára. A blokkos és karakteres eszközöket teljesen külön kezeli a kernel, ezért egy blokkos és egy karakteres eszköz azonosítója lehet ugyanaz.

[16] Az eddig kiosztott azonosítók a The Linux Assigned Names And Numbers Authority (http://www.lanana.org) oldalán találhatók.

Egy új eszközvezérlő beregisztrálása lényegében azt jelenti, hogy hozzárendeljük egy fő azonosítóhoz a virtuális állománykezelést megvalósító függvényeit. A karakteres eszközvezérlő regisztráló függvényének alakja:

```
int register_chrdev(unsigned int major, const char *name, struct file_operations *fops);
```

A visszatérési érték negatív szám esetén hibát jelent. A *major* argumentum a fő azonosítót jelenti, a *name* argumentum az eszköz neve, amely a */proc/devices* listában megjelenik majd. A *fops* paraméter egy struktúrára mutat, amely az egyes állománykezelési funkciókat megvalósító függvények mutatóit tartalmazza, például, hogy a speciális állományba való íráskor melyik függvény hívódjon meg.

Lehetőségünk van arra, hogy a regisztrálás során ne mi válasszuk meg a fő azonosítót – kockáztatva ezzel az ütközést egy másik meghajtóval – hanem a rendszerre bízzuk ezt a feladatot. Ekkor a kernel automatikusan kiválaszt egy azonosítót és a függvény visszatérési értékében adja vissza.

Miután az eszközvezérlőnket regisztráltuk a kernel könyvelésébe, minden olyan művelet esetén, amelyet az általunk regisztrálttal megegyező fő azonosítójú speciális állományokkal végzünk, a kernel meghívja a művelethez megadott függvényt.

A modulunk eltávolításakor természetesen el kell távolítanunk ezt a bejegyzést, és el kell eresztenünk a fő azonosítót. Erre szolgál a következő függvény:

```
void unregister_chrdev(unsigned int major, const char *name);
```

Az első paraméter a fő azonosító, a második az eszközvezérlő neve. Utóbbi csak biztonsági okokból szerepel. A kernel összehasonlítja a korábban regisztrálttal, és eltérés esetén nem hajtja végre a regisztráció eltávolítását.

Ha netán a modult a regisztrálás törlése nélkül távolítanánk el, az később komoly kellemetlenségeket okozhat. Minden művelet, amely erre az eszközvezérlőre hivatkozik a már említett **Oops** hibaüzenet váltja ki a kernelből.

9.3.2 Az eszközállományok dinamikus létrehozása

Tradicionálisan a */dev* könyvtár tele volt statikusan létrehozott eszközállományokkal. Minden a Linux által támogatott eszközállománynak szerepelnie kellett benne, hátha az adott meghajtót használni szeretnénk. Ez jelentős és felesleges helyfoglalást jelentett. Ezért célszerűnek látszott az eszközállományok dinamikus létrehozása.

Az első próbálkozás *devfs* állományrendszer támogatása volt. A kernel ezen az állomány rendszeren keresztül virtuális állományokkal reprezentálta az eszközállományokat, amikor erre szükség volt. A modern kernelváltozatok ezt a mechanizmust manapság már nem támogatják.

A jelenleg is használatos megoldás az *udev* felhasználói módban futó alkalmazásra épít. A folyamatosan futó *udev daemon* fogadja a kernel *hotplug* üzeneteit és ezek alapján dinamikusan hoz létre és töröl le eszközállományokat egy virtuális, memóriában tárolt állományrendszeren.

A *hotplug* üzenetek generálásához az eszközünknek implementálnia kell a 2.6-os kernelekkel bemutatkozó eszközmodellt (device model). A Linux eszközmodell lényegében egy komplex adatstruktúra, amely tartalmazza, hogy az eszköz milyen típus osztályba tartozik, melyik buszra csatlakozik, és milyen speciális attribútumokkal rendelkezik. Ha az eszközkezelőkben elvégezzük ezeket a regisztrációkat, akkor az eszközvezérlőnk illetve az eszközünk megjelenik a *sysfs* állományrendszerben is a hierarchiának megfelelő helyen. Emellett lehetővé válik az *udev* rendszer használata.

Amennyiben célunk mindössze az eszközállományok automatikus létrehozása, akkor elegendő a *device_create()* függvényt használnunk, amely az *udev* számára kiadja a megfelelő jelzést. A függvény alakja:

```
struct device* device_create(struct class* osztály, struct device*
szülő, dev_t eszköz, const char* név, ...);
```

Az *osztály* egy mutató az osztályra, amelyhez az eszközt rendeljük. Létrehozása a *class_create()* függvénnyel lehetséges. A *szülő* egy mutató a szülőeszközre, ha ilyen létezik. Az *eszköz* a fő és mellék azonosítókat tartalmazó argumentum. Előállítása az *MKDEV()* makróval lehetséges. A *név* és az ezt követő argumentumok az eszköz állománynevének összeállítását teszik lehetővé a *printf()* függvényhez hasonló módon.

A létrehozott eszközt a *device_destroy()* függvénnyel kell megsemmisíteni a modul eltávolítása előtt. A függvény alakja:

```
void device_destroy(struct class* osztály, dev_t eszköz);
```

A függvény paramétereinek értelmezése megegyezik a létrehozásnál használtakkal.

Az eszköz létrehozásához szükséges egy osztály, amelyhez hozzárendelhetjük. Ezt a *class_create()* függvénnyel hozhatjuk létre amelynek alakja a következő:

```
struct class* class_create(struct module* modul, const char* név);
```

A *modul* paraméter annak a modulnak a mutatója, amely az eszközosztályt birtokolja. A *THIS_MODULE* makróval adhatjuk meg a modulunkat. A *név* az osztály szöveges megnevezése.

Az osztály megszüntetését a *class_destroy()* függvénnyel kell elvégeznünk a tisztogatásnál. A függvény szintaxisa:

```
void class_destroy(struct class* osztály);
```

Nézzünk egy példát az eszköz állomány létrehozására:

```
static struct class* hello_class;
hello_class = class_create(THIS_MODULE, "hello");
device_create(hello_class, NULL, MKDEV(120, 0), NULL, "hello");
```

Az eltávolítás:

```
device_destroy(hello_class, MKDEV(120, 0));
class_destroy(hello_class);
```

A kódrészleteket elhelyezve az *init* és *exit* függvényekben az *udev* automatikusan létrehozza, illetve letörli a „hello” eszközállományt. Ha szeretnénk az állomány jogosultságait megadni, akkor az az *udev* konfigurációs állományában lehetséges.

9.3.3 Állományműveletek

Az eszközvezérlő az eszközök elérhetővé tételére, kezelésére az állománykezeléseknél használatos függvényeket kell, hogy implementálja. Ezeknek a függvényeknek a mutatóit tartalmazza a *file_operations* struktúra. Olyan műveletek ezek, mint a megnyitás, olvasás, írás, bezárás stb. A lista tartalmazhat **NULL** pointereket is, ennek jelentése, hogy az adott műveletet az eszközvezérlő nem támogatja.

A struktúra a kernelverziók változásakor folyamatosan nőtt és átalakult az újabb elemek felvételével. Az éppen aktuális leírását a *linux/fs.h* állomány tartalmazza.

A struktúra egyes elemeiből látható, hogy az adott metódust milyen függvénnyel lehet lekezelni. Az implementálni kívánt állománykezelő metódusokat a struktúrában megadott paraméterekkel, és visszatérési értékkel kell definiálni.

Mivel a struktúra az egyes verziókban változhat, ezért az alábbi módon javasolt a használata[17]:

```
static struct file_operations hello_fops =
{
  owner:          THIS_MODULE,
  read:           hello_read,
  write:          hello_write,
  open:           hello_open,
  release:        hello_release
};
```

Az *owner* mező megadásával jelezzük, hogy melyik modulhoz kötődik a leíró struktúra. A *THIS_MODULE* makró az aktuális kernel modul leíró struktúrájának mutatóját adja vissza.

9.3.4 Használatszámláló

Ha egy alkalmazás használja az eszközvezérlőt, a kernelnek nem szabad addig eltávolítania a modult, mígnem az utolsó alkalmazás is lezárja az eszközvezérlőhöz tartozó állományt. Ezt a védelmet a kernel klasszikus referenciaszámlálással implementálja. Nekünk csak annyit kell tennünk, hogy amikor megnyitják az eszközünket, akkor növelünk egy számlálót, amikor pedig lezárják, akkor csökkentjük a számlálót. A kernel nem távolítja el a modult a memóriából, amíg ennek a számlálónak az értéke nagyobb, mint nulla[18].

[17] Ez a definíció egy *gcc* kiterjesztésre épül, amely lehetővé teszi a struktúra tagjainak név szerinti inicializálását (a nem inicializált tagokat a *gcc* nullára inicializálja). A szabványos megoldás a C99-es szabvány úgynevezett név szerinti inicializálás (designated initializer) mechanizmusát használja. Ilyenkor pont után megadjuk az egyes mezők neveit, és expliciten adunk értéket nekik egyenlőségjellel. Azokat a változókat, amelyeket nem sorolunk fel, a fordító nullával inicializálja. Ezt a C-ben is viszonylagosan új szintaxist a C++ nem támogatja. Vagyis ha a tagok nevei elé pontot, a kettőspont helyett egyenlőséget használunk, akkor hordozható megoldáshoz jutunk.

[18] A kernel természetesen számon tartja azt is, hogy ha más modulok is használnak egy adott modult. Ilyenkor szintén növeli a referencia számlálót.

Ennek a mechanizmusnak a használatához kész függvényeket nyújt a rendszer. A számlálót növelni az alábbi függvénnyel tudjuk:

```
try_module_get(THIS_MODULE);
```

Ha már nem használják a modult, akkor csökkentenünk kell a számláló értékét:

```
module_put(THIS_MODULE);
```

A rendszer figyeli, és ha a használatszámláló értéke nagyobb mint 0, akkor nem engedi a modult eltávolítani. Ez ugyanakkor a fejlesztés során gondot is jelenthet, mert ha elrontjuk a könyvelést, akkor nem lehet a modult a rendszer újraindítása nélkül eltávolítani. Ezért a kernelbe belefordítható a kényszerített eltávolítás lehetősége[19].

9.3.5 Adatmozgatás Kernel és User space között

A `read` és a `write` függvények olyan buffereket kapnak paraméternek a felhasználói programoktól, amelyek a User space-ben vannak lefoglalva. Ugyanakkor a kernel modulok által foglalt területek általában a Kernel space-ben vannak. Ezért meg kell oldanunk az adatok mozgatását a két terület között. Erre a `linux/uaccess.h` tartalmaz függvényeket:

```
unsigned long copy_from_user(void *to, const void __user *from, unsigned long n)

unsigned long copy_to_user(void __user *to, const void *from, unsigned long n);
```

9.3.6 A mellékazonosító (minor number) használata

A mellékazonosító lehetőséget biztosít az eszközmeghajtó számára, hogy több azonos típusú eszközt tegyen láthatóvá az operációs rendszer számára. Bár a meghajtó szabadon használhat eltérő implementációt a különböző mellékazonosítókhoz, az egységes viselkedés erősen javallott.

Vegyük példának a soros portokat kezelő eszközvezérlőt. Természetesen az eszközvezérlő az egyes portokra megegyezik, mivel pazarlás lenne minden porthoz egy külön eszközvezérlőt betölteni és regisztrálni. Vagyis az eszközvezérlőnk egy példányban van jelen és egy fő azonosítóra regisztrálta az állomány interfészét. Azonban az összes soros porton folyó kommunikációt egyetlen állományon keresztül elérhetővé tenni nehéz lenne és célszerűtlen. Jobban járunk, ha minden porthoz rendelünk egy állományt. Ilyenkor ezek az állományok egy fő azonosítóval rendelkeznek, és a mellékazonosító különbözteti meg őket.

A mellékazonosító kezelése, ahogy korábban már említettük, a mi feladatunk. Két utat választhatunk. Az egyik megoldás szerint az olvasást és írást kezelő függvényeinkbe helyezünk elágazásokat, amelyek a mellékazonosító alapján más-más kód részletet futtatnak. A másik megoldás, ha az előző fejezetben említett módon az állomány megnyitásánál állítunk be eltérő lekezelő függvényeket az egyes minor számok esetében.

---

[19] A kernelfordítás előtti konfigurálás során a modulokra vonatkozó beállításoknál kiválaszthatjuk a kényszerített modul eltávolítás lehetőségét. Később a lefordított kernelt használva lehetőségünk lesz a modul kényszerített eltávolítására. Azonban bár a rendszer engedi az eltávolítást csak szükség esetén használjuk.

9.4 A /proc állományrendszer

A Linux rendelkezik egy további mechanizmussal is, amellyel a kernel, vagy kernel modulok információkat tudnak átadni a felhasználói üzemmódban futó folyamatoknak. Bár a karakteres eszközvezérlőknél megismert módszerrel is megoldhatjuk a kommunikációt egy folyamat és egy kernel modul között, azonban a */proc* állományrendszer használata leegyszerűsíti ezt. Eredeti feladata, hogy könnyen hozzáférhető információt szolgáltasson a processzekhez kapcsolódó adatokról. Manapság a kernel minden olyan része használja, amelyik valami hasznos jelentéssel vagy statisztikával szolgálhat. Továbbá már nemcsak olvashatjuk, hanem írhatjuk is egyes részeit, ezáltal szabályozva a kernel és a modulok működését.

A metódus hasonlít az előzőekben, az eszközvezérlőben használtakhoz. Itt is egy struktúrát kell létrehoznunk, amely tartalmaz minden információt, ami szükséges a */proc* állományunkhoz, beleértve a mutatót a kezelőfüggvényekre. Ha egy csak olvasható állományt szeretnénk létrehozni, akkor csak egy függvényt kell implementálnunk és beállítanunk, amely meghívásra kerül az állomány olvasásakor. A modul inicializálásakor regisztráljuk a leíró struktúránkat, és a tisztogató függvényben eltávolítjuk.

Írhatóvá is tehetjük a virtuális állományunkat. Ehhez szükség van az íráskor meghívott függvény implementációjára. Ez hasonlít a karakteres eszközvezérlőknél megismertekre, beleértve a kernel- és a felhasználói címtartomány közötti adatmozgatást is. Ahhoz, hogy a folyamatok ténylegesen tudják írni az állományt, az írást kezelő függvény regisztrációja mellett arra is szükség van, hogy az írásjogot megadjuk a felhasználóknak.

9.5 Eszközvezérlő modell

A számítógépben található eszközök többnyire különböző buszokon keresztül csatlakoznak a rendszerhez. Ezek a buszok eltérően működnek, a rajtuk folyó kommunikáció más és más.

A 2.4-es és korábbi kernelekben a különböző buszokra csatlakozó eszközök vezérlőkódjaiban eltérő struktúrák tartották nyilván a rendszerbe csatlakoztatott eszközöket, általános könyvelés nem létezett. Így egy eszköz jelenlétéről csak a kernel üzenetek böngészésével szerezhettünk tudomást. Ez a kaotikus helyzet a 2.5-ös kernelben oldódott meg, amely megoldás azután a 2.6-os kernelnek is részévé vált. Egy egységes eszközvezérlő-modell alakult ki a buszok és a rajtuk helyet foglaló eszközök leírására, illetve egy globális könyvelés is megoldottá vált. Az adatstruktúra hozzáférhető és kezelhető, így már pontos képünk van az rendszerben található eszközökről.

9.5.1 A busz

Bár a buszok eltérnek egymástól, rendelkeznek olyan általános jellemzőkkel, amelyek egységesen megtalálhatóak bennük. Ezek adják az alapját az általános buszmodellnek. A buszleíró struktúra az általános attribútumokat és általános műveletekhez tartozó visszahívandó függvények (callback function) mutatóit tartalmazza. Ilyen művelet például az eszközök felderítése (detection), a busz leállítása, tápellátás kezelése.

9.5.2 Eszköz és eszközvezérlő lista

Korábban a buszhoz kapcsolódó eszközök és eszközvezérlők listáját a busz eszközvezérlője tartotta nyilván a saját egyéni módján. Az általános modellben azonban ez a könyvelés is helyet kapott.

Az eszközvezérlőt az alábbi adatstruktúra írja le:

```
struct device_driver {
      const char      *name;
      struct bus_type *bus;
      struct module         *owner;
      …
      int (*probe) (struct device *dev);
      int (*remove) (struct device *dev);
      void (*shutdown) (struct device *dev);
      int (*suspend) (struct device *dev, pm_message_t state);
      int (*resume) (struct device *dev);
      const struct attribute_group **groups;

      const struct dev_pm_ops *pm;

      struct driver_private *p;
};
```

A mezők értelmezése a következő:

Név	Értelmezés
name	Az eszközvezérlő neve.
bus	A busz, amelyhez az eszközvezérlő tartozik.
owner	A kernelmodul, amelyhez tartozik. (THIS_MODULE)
probe	Akkor hívódik meg, ha egy új eszközt csatlakoztatunk a buszhoz. Teszteljük benne, hogy az eszközt kezeli-e az eszközvezérlő, illetve elvégezzük az összerendelést.
remove	Az eszköz eltávolításakor hívódik meg.
shutdown	A rendszer leállításakor hívódik meg.
suspend	Az eszköz elaltatásakor hívódik meg.

resume	Az eszköz felébresztését implementálja.
groups	Alapértelmezett attribútumok.
pm	Tápmenedzsment műveletek.
p	Az eszközvezérlő privát adatai.

Az eszközvezérlőben minimum a *name* és a *bus* mezőket ki kell töltenünk a regisztrációhoz. Ezt követően a regisztrációt az alábbi függvénnyel végezhetjük el.

```
int driver_register(struct device_driver *drv);
```

Ennek a párja, amely megszünteti a regisztrációt:

```
void driver_unregister(struct device_driver *drv);
```

Azonban általában az általános eszközvezérlő struktúra nem elég az összes információ tárolására. Tipikusan további adat szokott lenni az eszközazonosítók táblázata, amelyeket az eszközvezérlő kezel. Ezek busz specifikus adatok így az általános struktúrába nem kerülhettek bele. Ezért a különböző busz típusok saját eszközvezérlő leíró struktúrákat definiálnak, amely egyrészt tartalmazza az általános struktúrát, másrészt kibővíti azt.

Ugyanakkor ezeket a kibővített eszközvezérlő-leíró struktúrákat a korábbi regisztrálófüggvénynek nem adhatjuk át. Ezért tipikusan minden busz típusú eszközvezérlő tartalmaz egy saját eszközvezérlő-regisztráló és egy regisztrálást megszüntető függvényt, amely a kibővített struktúrákkal dolgozik. Ez a speciális regisztráló függvény végzi el a *bus* mező beállítását is az eszközvezérlő leíró struktúrában.

Összegezve, ha egy olyan buszhoz szeretnénk regisztrálni egy eszközvezérlőt, amely nem az általános leírástruktúrát használj –, ilyen a buszok többsége –, akkor a buszspecifikus leíró struktúrából hozunk létre egy példányt és azt töltjük ki, továbbá a buszspecifikus regisztráló és eltávolító függvényeket használjuk.

9.5.3 *sysfs*

A *sysfs* állományrendszerben található egy könyvtár ahol elérhetőek az eddig tárgyalt információk.

```
/sys/bus
```

Ebben a könyvtárban minden regisztrált busz kap egy, a nevével megegyező könyvtárat. A busz könyvtárában megtalálhatóak a busz attribútumait reprezentáló virtuális állományok, továbbá egy *drivers* és egy *devices* alkönyvtár. Ezek tartalmazzák az előző fejezetben említett eszközvezérlő és eszközkönyvelést. Minden eszköz illetve eszközvezérlő további alkönyvtárakat kap amelyben elérhetőek az attribútumaik.

9.5.4 Az eszköz

Az eszközök általános leíró struktúrája a következő:

```
struct device {
```

```
    struct device        *parent;

    struct device_private *p;
    const char      *init_name;
    const struct device_type *type;
    struct mutex         mutex;
    struct bus_type *bus;
    struct device_driver *driver;
    void       *platform_data;
    …
    dev_t           devt;
    …
    struct class         *class;
    const struct attribute_group **groups;

    void  (*release)(struct device *dev);
};
```

A mezők értelmezése a következő:

Név	Értelmezés
parent	Az eszköz szülője, vagyis az az eszköz amire rá van csatlakoztatva. Ez többnyire egy busz, vagy vezérlő.
p	Az eszköz privát adatai.
init_name	Az eszköz neve.
type	Az eszköz típusa.
mutex	Az eszközhöz rendelt mutex az eszközvezérlő műveleteinek szinkronizálásához.
bus	A busz eszköz amire csatlakoztatva van.
driver	A driver amelyik allokálta ezt a struktúrát.

platform_data	Tároló hely, ahova az eszközvezérlő az eszközhöz kapcsolódó információkat helyezheti. Ilyenkor létrehoz egy egyéni adatstruktúrát és ide helyezi a mutatóját.
devt	A sysfs-en belül, illetve udev-el eszköz állomány gyártáshoz.
class	Az eszköz osztálya.
groups	Opcionális csoport attribútumok.
release	Akkor hívódik meg, amikor az eszközleíró struktúrát megsemmisíti a busz eszközvezérlője.

A busz eszközvezérlője érzékeli az eszköz csatlakoztatását. Ennek hatására létrehoz egy példányt a *device* struktúrából és meghívja a megfelelő eszközvezérlő probe() függvényét. (A megfelelő eszközvezérlő kiválasztása a busz típusnak megfelelő azonosítókkal történik.)

9.6 A párhuzamosság kezelése

A kernelfejlesztés során is találkozhatunk olyan helyzetekkel, amikor a párhuzamos végrehajtás miatt szinkronizációra van szükség. Ilyen esetek a következők:

Ha megszakításkezelőben használunk olyan változókat, ami máshol is hozzáférhető a rendszerben.

Ha a kódunk *sleep()* rendszerhívást tartalmaz. Ez tulajdonképpen azt jelenti, hogy a függvényünk futását megállítjuk az adott ponton és más folyamatokra vált át a kernel. (Van több olyan más rendszerhívás is, amely használja a *sleep()* függvényt. Láttuk, hogy a *kmalloc()* is ilyen, ha nem a *GFP_ATOMIC* kapcsolót használjuk, de a *copy_from_user()* is ide tartozik.)

- Ha kernelszálakat használunk.
- Ha többprocesszoros rendszert használunk. (Ide tartoznak a többmagos processzorok és a többszálú processzorok is.)

A párhuzamosság problémáinak kezeléséhez szinkronizációs olyan eszközökre van szükségünk, amelyek a kölcsönös kizárást valósítanak meg. A Linux kernelben több ilyen eszközt is találunk, ezeket tipikus alkalmazási területükkel együtt a következő fejezetekben mutatjuk be.

9.6.1 Atomi műveletek

A legegyszerűbb szinkronizáció, ha nincs szükségünk szinkronizációs eszközre. Ezt úgy érhetjük el, ha a műveletünk atomi, vagyis gépi kód szinten egy utasításban valósul meg. Így

nincs olyan veszély, hogy a művelet végrehajtása közben egy másik processzoron futó másik kód hatással lesz a műveletünkre mivel az egy lépésben valósul meg.

Annak eldöntéséhez, hogy a művelet atomi-e, gépi kód szintjén kell megvizsgálnunk. Ebből adódóan platformfüggő és C fordítófüggő, hogy egy művelet tényleg atomi lesz-e. A kernel platformfüggő, assembly részeiben gondoskodhatunk arról, hogy teljesüljön a műveleteinkre ez a feltétel. Azonban a platform független kernel részek számára is lehetővé kellett tenni, hogy garantáltan atomi műveleteket használhassanak. Ezt a kernel fejlesztői egy atomi típus és a rajta értelmezett függvények definiálásával érték el. Amikor a kernelt egy új platformra ültetik át a fejlesztők, akkor ezeket a függvényeket az adott architektúrának megfelelő módon kell megvalósítaniuk úgy, hogy a végeredmény valóban atomi legyen, vagy megfelelően szinkronizált. Előfordulhat, hogy egy adott platformon az implementáció a hagyományos műveleteknél lassabb. Ezért az atomi műveleteket csak ott használjuk, ahol tényleg szükség van rá.

Az atomi műveletek egyszerűek, különben nem lehetne egy gépi utasításban elvégezni őket.[20] Ezért az atomi függvények halmaza csak egész számokon értelmezett érték növelő/csökkentő/tesztelő és bit műveleteket tartalmaz.

Az egész számot kezelő atomi függvények csak az *atomic_t* adattípuson értelmezettek, amely egy előjeles egész számot tartalmaz. Más, hagyományos típusokkal nem használhatóak. Továbbá az atomi adattípuson nem használhatóak a hagyományos operátorok. Az *atomic_t* típusú változót csak az *ATOMIC_INIT()* makróval inicializálhatunk. Például:

```
static atomic_t szamlalo = ATOMIC_INIT(1);
```

Az atomi műveletek másik csoportját a bitműveletek alkotják. A bitműveletek *unsigned long* típusú értékeket kezelnek, és egy memóriacímmel megadott memóriaterületen végzik el a műveletet annyi bájtra értelmezve, amennyi az *unsigned long* típus mérete az adott architektúrán. A bájtsorrend (little-endian, big-endian) értelmezése szintén az architektúrától függ: annak a bájtsorrendjével egyezik meg. Ez sajnos azt jelenti, hogy a bit műveletek platformfüggőek.

9.6.2 Ciklikus zárolás (spinlock)

A **ciklikus zárolás** (spinlock**)** egy olyan szinkronizációs eszköz, amely folyamatosan, egy CPU-t terhelő ciklusban kísérletezik a zárolás megszerzésével mindaddig, amíg meg nem szerezte. Ezért csak rövid kritikus szakaszok esetén használjuk, mivel egyébként a várakozó szálak/folyamatok számottevő mértékben pazarolnánk a CPU-t. Ugyanakkor a rövid szakaszok esetén hatékonyabb, mint az összetettebb szemafor. További előnye, hogy egyprocesszoros rendszer esetén üres szakasszal helyettesíti a rendszer, mivel ott nincs szükség ilyen jellegű szinkronizálásra.

Ugyanakkor figyelnünk kell arra, hogy a ciklikus zárolással védett kritikus szakasz ne tartalmazzon *sleep()*-et hívó függvényt. Ha ugyanis egy szál/folyamat lefoglalja a zárolást, és *sleep()* miatt egy olyan szál/folyamat kapja meg a vezérlést, amelyik szintén megpróbálja megszerezni a zárolást, akkor CPU-pazarlóan vár rá a végtelenségig. Szélsőséges esetben ez

[20] A mondat nem jelenti azt, hogy minden atomi művelet minden processzortípuson egy gépi utasításra fordul le. Azonban a lista összeállítása során egy-egy processzor utasításkészletéből indultak ki a fejlesztők.

holtponthoz és így a rendszer lefagyásához vezet. Az előző példával analóg eset állhat elő, ha megszakításkezelő függvényben használunk ciklikus zárolást. Ezért megszakításkezelőben erősen ellenjavallt a ciklikus zárolás használata.

Egy ciklikus zárolás létrehozása az alábbi módon történhet:

```
spinlock_t lock;
```

Ezt követően inicializálnunk kell:

```
spin_lock_init(&lock);
```

Azonban a létrehozást és az inicializálást egy lépésben egy makróval is elvégezhetjük, sőt, érdemesebb ezt a megoldást választani:

```
static DEFINE_SPINLOCK(lock);
```

Lefoglalása:

```
spin_lock(&lock);
```

Felszabadítása:

```
spin_unlock(&lock);
```

Előfordulhat, hogy a kritikus szakasznál arról is gondoskodnunk kell, hogy párhuzamosan ne hívódhasson meg egy megszakításkezelő. A megszakításkezelés átmeneti letiltását és engedélyezését az alábbi függvényekkel tehetjük meg:

```
unsigned long flags;
spin_lock_irqsave(&lock, flags);
...
spin_unlock_irqrestore(&lock, flags);
```

Tapasztaltabb programozóknak gyanús lehet, hogy a *spin_lock_irqsave()* második paramétereként nem a *flags* változó mutatója szerepel. Ez nem nyomdahiba. A bemutatott függvények valójában makrók, ezért valójában nem érték szerinti átadást láthatunk, hanem a változót adjuk át.

9.6.3 Szemafor (semaphore)

A szemafor (semaphore) a ciklikus zárolásnál összetettebb mechanizmus, ezért több erőforrást is igényel. Így a nagyon rövid szakaszok esetén inkább a ciklikus zárolást részesítjük előnyben. Ugyanakkor a komolyabb esetekben, illetve ha a ciklikus zárolás a megkötései miatt nem használható, akkor ez a helyes választás.

Figyeljünk arra, hogy a semaphore a várakozáshoz *sleep()* függvényt használ, így nem használhatjuk olyan helyen, ahol a *sleep()*-es függvények tiltottak[21].

A semaphore létrehozása:

```
struct semaphore sem;
```

[21] Nem használhatunk *sleep()*-es függvényeket például a megszakításkezelő függvényekben (hardver és szoftver megszakításkezelők). Emellett ciklikus zárolás által védett szakaszokban sem.

Inicializálása:

```
void sema_init(struct semaphore *sem, int val);
```

A létrehozást és az inicializálást kombináló makró:

```
static DEFINE_SEMPHORE(sem);
```

A *val* paraméter a szemafor kezdőértékét adja meg.

A szemafor lefoglalásakor lehet, hogy várakozásra kényszerül a függvény. Azonban attól függően, hogy ezt a várakozást mi szakíthatja meg több függvény is választható a lefoglalásra.

Egyszerű lefoglalás, amikor csak a szemafor felszabadulása vet véget a várakozásnak:

```
void down(struct semaphore *sem);
```

Ha egy jelzés megszakíthatja a várakozást (ilyenkor -EINTR a visszatérési értéke):

```
int down_interruptible(struct semaphore *sem);
```

Ha kritikus szignál szakíthatja csak meg:

```
int down_killable(struct semaphore *sem);
```

Ha nem akarunk várakozni, hanem csak egy hibajelzést szeretnénk, ha nem lehet lefoglalni:

```
int down_trylock(struct semaphore *sem);
```

Ha időkorlátos várakozást szeretnénk:

```
int down_timeout(struct semaphore *sem,long jiffies);
```

Az időkorlát mértékegysége jiffy. (Lásd 2.3.9 fejezet.)

A szemafor elengedése:

```
void up(struct semaphore *sem);
```

9.6.4 Mutex

A kernelfejlesztők a szemafort többnyire 1 kezdő értékkel mutexként kölcsönös kizárás megvalósítására használják olyan esetekben, ahol a ciklikus zárolás nem használható. Azonban ha nem használjuk ki teljesen a funkcióit, akkor lehetséges egy egyszerűbb megvalósítás is, amivel valamelyest jobb lesz a rendszer teljesítménye. Ilyen megfontolás vezette a fejlesztőket, amikor bevezették a kernel mutex eszközét.

A kernel fejlesztések során a mutex azokban az esetekben használható, ahol egyébként szemafort használnánk kölcsönös kizárásra.

A megkötések nagyrészt a szemafor megkötéseivel azonosak, ugyanakkor vannak új elemek is:

- Csak a lefoglaló szabadíthatja fel.
- Rekurzív foglalás, vagy többszörös felszabadítás nem engedélyezett.
- Nem használható megszakítás kontextusában. (Vagyis sem hardver, sem szoftver megszakítást kezelő függvényben.)

A mutex típusa:

```
struct mutex;
```

Inicializálása:

```
mutex_init(mutex);
```

A létrehozás és az inicializálás a következő makróval történhet:

```
DEFINE_MUTEX(mutex);
```

A mutex lefoglalása:

```
void mutex_lock(struct mutex *lock);
```

A mutex lefoglalása, ha a várakozást jelzéssel megszakíthatóvá szeretnénk tenni:

```
int mutex_lock_interruptible(struct mutex *lock);
```

A mutex lefoglalása, ha sikertelenség esetén nem várakozni akarunk, hanem hiba visszajelzést szeretnénk kapni:

```
int mutex_trylock(struct mutex *lock);
```

A mutex felszabadítása:

```
void mutex_unlock(struct mutex *lock);
```

A mutex foglaltságának ellenőrzése:

```
int mutex_is_locked(struct mutex *lock);
```

9.6.5 Olvasó/író ciklikus zárolás (spinlock) és szemafor (semaphore)

A kritikus szakasznál célszerű megkülönböztetnünk, hogy a szakaszon belül a védett változót írjuk vagy olvassuk. Mivel ha csak olvassuk az értéket, akkor azt párhuzamosan több szál is probléma nélkül megteheti (megosztott zárolás). Míg ha írjuk, akkor sem írni, sem olvasni nem szabad másik szálnak (kizáró zárolás). A két eltérő zárolás használatával növelhetjük a rendszer teljesítményét.

Az olvasó/író spinlock létrehozása és inicializálása:

```
rwlock_t rwlock;
rwlock_init(&rwlock);
```

A létrehozást és az inicializálást kombináló makró:

```
static DEFINE_RWLOCK(rwlock);
```

Olvasási (megosztott) zárolás:

```
read_lock(&rwlock);
...
read_unlock(&rwlock);
```

Írási (kizáró) zárolás:

```
write_lock(&rwlock);
```

```
...
write_unlock(&rwlock);
```

Az olvasó/író semaphore létrehozása:

```
struct rw_semaphore rwsem;
```

Inicializálás:

```
void init_rwsem(struct rw_semaphore *sem);
```

Lefoglalás olvasásra:

```
void down_read(struct rw_semaphore *sem);
int down_read_trylock(struct rw_semaphore *sem);
```

Felszabadítás olvasás esetén:

```
void up_read(struct rw_semaphore *sem);
```

Lefoglalás írásra:

```
void down_write(struct rw_semaphore *sem);
int down_write_trylock(struct rw_semaphore *sem);
```

Felszabadítás írás esetén:

```
void up_write(struct rw_semaphore *sem);
```

Előfordulhat, hogy egy olvasási zárolást felszabadítás nélkül írási zárolásra szeretnénk átalakítani, azonban erre a Linux kernel implementációjában nincs lehetőség: ilyen jellegű próbálkozás holtpontot okoz.

Ugyanakkor az írási zárolást bármikor „visszaléptethetjük” olvasási zárolásra:

```
void downgrade_write(struct rw_semaphore *sem);
```

9.6.6 A Nagy Kernelzárolás

A **Nagy Kernelzárolás** (*Big Kernel lock*) egy globális rekurzív ciklikus zárolás. Használata nem javasolt, mivel értelemszerűen jelentősen korlátozza a kernel működését, és rontja a rendszer valósidejűségét mivel akadályozza a kernel párhuzamos működését.

Ez a zárolásfajta a 2.0 verziótól volt jelent a kernelben, amikor a többprocesszoros rendszerek támogatása belekerült (symmetric multiprocessing, SMP). A lehetséges konkurenciaproblémákat úgy előzték meg a fejlesztők, hogy kernel üzemmódban az egész kernelt zárolták. Ezzel kivédték, hogy egy másik processzor is futtasson kernel módú kódot párhuzamosan. A konkurenciaproblémák kezelésére széles körben alkalmazták, mert segítségével úgymond „biztosra lehetett menni” és nem kellett végig gondolni az adott konkurenciahelyzetek hatásait. Ugyanakkor ez egy átmenetinek szánt megoldás volt, amelynek a helyét később átgondoltabb, kifinomultabb megoldások vették át. Az idők folyamán egyre több helyről sikerült kiszorítani és végül a 2.6.37-es verzióban sikerült végleg eltávolítani. Bár lehetőség van rá, hogy a kernel konfigurációja során visszakapcsoljuk, értelemszerűen ez nem javasolt.

A teljesség kedvéért azonban a Nagy Kernelzárolás lefoglaló és felszabadító függvényeit is megemlítjük:

```
lock_kernel();
...
unlock_kernel();
```

9.7 I/O portok kezelése

Az eddigi fejezetek során bemutattuk, hogy hogyan tudunk az alkalmazások és a kernel modulok között kommunikálni. Azonban egy eszközvezérlőnél ez még csak a feladat egyik fele, hiszen nem elég az alkalmazásokkal kommunikálnia, az általa kezelt hardverrel is tartania kell a kapcsolatot. Ezért a következő néhány fejezetben azt mutatjuk be, hogyan tudunk kernelből különféle eszközöket vezérelni.

Az egyszerűbb hardver eszközökkel a kommunikációt leggyakrabban az I/O portok segítségével folytatja a CPU mégpedig az *in* és *out* parancsok segítségével. A Linux kernelmodulokban is elérhetjük ezeket a parancsokat illetve portokat. Azonban a portok használata előtt illik a tartományt lefoglalni.

A I/O porttartomány lefoglalásának célja, hogy a kernel könyveli, hogy melyik tartományt melyik eszközvezérlő használja. Így egy lefoglalási kérelem során jelezheti, ha az adott tartományt egy másik eszközvezérlő netán már használja. Természetesen az eszközvezérlő eltávolítása során a tartományt is fel kell szabadítanunk, hogy más eszközvezérlő is használhassa.

Porttartomány lefoglalása az alábbi függvénnyel történik:

```
struct resource* request_region(resource_size_t start,
resource_size_t n, const char *name);
```

A *start* a tartomány eleje, az *n* a tartomány mérete. A *name* paraméterre azért van szükség, hogy ha megnézzük a lefoglalt tartományok listáját, akkor láthassuk, hogy melyik eszközvezérlő használja. Ezért ez a paraméter az eszközvezérlő szöveges elnevezését tartalmazza.

A lefoglalhatóságot külön is ellenőrizhetjük:

```
int check_region(resource_size_t start, resource_size_t n);
```

A használat után a terület felszabadítása:

```
void release_region(resource_size_t start, resource_size_t n);
```

A lefoglalt I/O portok listáját felhasználóként is elérhetjük a */proc/ioports* virtuális állomány megtekintésével.

A tartomány lefoglalása után a portokat különböző I/O műveletekkel érhetjük el. Az adat méretétől függően több függvény közül is választhatunk:

8 bites

```
unsigned char inb(int port);
```

```
void outb(unsigned char value, int port);
```

16 bites

```
unsigned short inw(int port);
void outw(unsigned short value, int port);
```

32 bites

```
unsigned long inl(int port);
void outl(unsigned long value, int port);
```

Ugyanakkor mindegyik függvénynek létezik egy várakozó („pause”) változata, amely egy kis várakozást is tartalmaz. Erre a lassabb buszok/kártyák használata esetén lehet szükségünk. Ezen függvények alakja megegyezik az előzőekben felsoroltakkal, azonban a függvények nevének a végére egy *_p* utótagot kell illeszteni.

9.8 Megszakítás kezelés

Ha az eszközvezérlőnkben szeretnénk lekezelni egy megszakítást, akkor az alábbi műveleteket kell elvégeznünk:

1. Létre kell hoznunk egy megszakítás kezelő függvényt.
2. Regisztrálnunk kell a megszakítás kezelő függvényt az adott megszakításhoz.
3. A driver eltávolításakor a megszakítás kezelő regisztrációját is el kell távolítanunk.

A megszakítás kezelő függvény alakja:

```
typedef irqreturn_t (*irq_handler_t)(int irq, void *devid);
```

A megszakítás kezelőnket az alábbi függvénnyel regisztrálhatjuk:

```
int request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags, const char *devname, void *devid);
```

Paraméterei a megszakítás száma (**`irq`**), a megszakítás kezelő függvény (**`handler`**), az opciók (**`flags`**), az eszközvezérlő neve (**`devname`**), megosztott megszakítás esetén az egyedi azonosító (**`devid`**).

A **`flags`** mező értékei az alábbiak lehetnek:

- IRQF_DISABLED (SA_INTERRUPT): A gyors interrupt jelzése. Az implementációt gyorsra kell készítenünk, mert a megszakítás kezelő futása alatt a megszakítás le van tiltva.
- IRQF_SHARED (SA_SHIRQ): Annak jelzése, hogy az interruptot megosztjuk más megszakítás kezelőkkel. Ilyenkor általában több hardver használja ugyanazt a megszakítás vonalat.
- IRQF_SAMPLE_RANDOM (SA_SAMPLE_RANDOM): A megszakítás felhasználható a véletlen szám generáláshoz.
- IRQF_TIMER: A megszakítás timer interrupt.

A megszakítás kezelő függvény implementációjánál az alábbi értékekkel térhetünk vissza:

- IRQ_HANDLED: A megszakítást lekezelte a függvény.
- IRQ_NONE: A megszakítást nem kezelte a függvény.

A megszakítás kezelő regisztrációjának törlése a következő függvénnyel történik:

```
void free_irq(unsigned int irq, void *devid);
```

Amennyiben megosztottan használtuk a megszakítás vonalat más megszakítás kezelőkkel, akkor ügyeljünk arra, hogy a felszabadításnál ugyanazt a `devid` értéket adjuk meg, amit a regisztrációnál. A rendszer nem akadályozza meg, hogy más eszköz vezérlő regisztrációját töröljük, ezért különösen oda kell figyelnünk rá.

9.8.1 Megszakítások megosztása

Mivel a megszakítás vonalak száma többnyire véges, ezért a hardver készítők gyakran kényszerülnek arra, hogy több eszköz között osszanak meg egy megszakítás vonalat. Ebben az esetben a megszakítás kezelő függvény regisztrációjánál jeleznünk kell az `IRQF_SHARED` flaggel a megosztott használatot. Emellett a `devid` paraméternek meg kell adnunk egy egyedi értéket, amellyel később a regisztrációnkat azonosíthatjuk.

A megszakítás kezelő függvény implementációjánál az `IRQ_NONE` visszatérési értékkel jelezhetjük, ha a megszakítás nem az általunk kezelt eszköznek szólt.

Természetesen vigyázzunk a megszakítás esetleges letiltásával és engedélyezésével, mivel ebben az esetben más eszközök kezelésére is hatással lehet.

9.8.2 Megszakítás kezelő függvények megkötései

A megszakítás kezelő függvény implementációja során több szabályt is be kell tartanunk.

- Nem használhatunk `sleep()`-es függvényt, mivel beláthatatlan következményekkel járna egy taszkváltás a megszakítás kezelő kontextusban.
- A `kmalloc()`-os allokációt is csak a `GFP_ATOMIC` flaggel végezhetjük, mivel egyéb esetben `sleep()` függvényt használ az allokáció során.
- A kernel és a user space között nem mozgathatunk adatokat.
- Törekedjünk a gyorsaságra. A nagy számításokat lehetőleg máshol végezzük.
- Mivel nem processz hívja meg a függvény, ezért a processz specifikus adatok nem elérhetőek a megszakítás kezelőből.

9.8.3 A megszakítás tiltása és engedélyezése

Amennyiben egy megszakítást szeretnénk letiltani, az alábbi két függvénnyel tehetjük meg:

```
void disable_irq(unsigned int irq);
void disable_irq_nosync(unsigned int irq);
```

A „nosync” függvény abban különbözik a társától, hogy míg a normál változat az esetleg futó megszakítás kezelő függvény végét megvárja a visszatérés előtt, a „nosync” változat azonnal visszatér.

A megszakítás ismételt engedélyezése:

```
void enable_irq(unsigned int irq);
```

Amennyiben szükségünk van az összes megszakítás letiltására, illetve engedélyezésére, akkor az alábbi függvényekkel tehetjük meg:

```
void local_irq_enable();
void local_irq_disable();
```

Azonban többnyire szeretnénk a korábbi beállításokat lementeni, illetve az engedélyezéskor visszaállítani:

```
void local_irq_save(unsigned long flags);
void local_irq_restore(unsigned long flags);
```

9.8.4 A BH mechanizmus

A megszakítás-kezelőben sokszor komolyabb adatfeldolgozást is el kell végeznünk, ekkor azonban nem lesz gyors az implementáció, amely a megszakításkezelő egyik legfontosabb alapkövetelménye. Erre a problémára nyújt megoldást az alsó rész (Bottom half, BH) mechanizmus, amely a megszakításkezelőt egy felső részre (Top half) és egy alsó részre (Bottom half) választja szét. A felső rész a tényleges megszakításkezelő rutin. Ennek feladata csak az adatok gyors letárolása a későbbi feldolgozáshoz, illetve a feldolgozó rutin futtatásának kérvényezése. Az alsó rész rutin már nem megszakításidőben fut, ezért a futás idejére nem érvényesek a szigorú megkötések.

A Bottom half rész implementációjára két mechanizmust használhatunk:

- Tasklet
- Workqueue

9.8.5 Taskletek tulajdonságai

A taskletek implementációja során figyelembe kell vennünk néhány megkötést:

- A tasklet futtatását többször is kérhetjük, mielőtt lefutna. Azonban ilyenkor csak egyszer fog lefutni.
- Ha a tasklet fut már egy CPU-n a kérvényezéskor, akkor később ismét le fog futni.
- A tasklet azon a CPU-n fog lefutni, ahol először kérvényezték.
- Egyszerre a tasklet csak egy példányban futhat még SMP rendszerben is.
- Különböző taskletek viszont futhatnak párhuzamosan SMP rendszerben.
- A tasklet nem indul el, amíg a megszakítás kezelő be nem fejeződik.
- A tasklet futása során meghívódhat a megszakítás kezelő.

- `Sleep()`-es függvényt nem használhatunk tasklet-ben sem.

9.8.6 A workqueue

A workqueue mechanizmus tágabb lehetőségeket ad, mint a tasklet használata. Mivel a függvény ebben az esetben egy külön kernel szálban hívódik meg, ezért egyrészt használhatunk `sleep()`-es függvényeket is, másrészt az implementáció hosszára nincs megkötés mivel közben lehetséges taszkváltás. Ugyanakkor továbbra sem érhetjük el más processzek címterét, azonban ez nem szokott komoly problémát jelenteni.

A workqueue mechanizmus használatához létre kel hoznunk egy lekezelő függvényt, amelyet a workqueue meghív. Ennek alakja:

```
void (*work_func_t)(struct work_struct *work);
```

Továbbá létre kell hoznunk magát a workqueue-t.

```
struct workqueue_struct * create_workqueue(const char *name);
```

Ugyanakkor a kernel modul eltávolításakor nem szabad elfelejtenünk megsemmisíteni:

```
void destroy_workqueue(struct workqueue_struct *wq);
```

Ezt követően a megszakítás kezelő rutinban a feldolgozó függvényünkből egy `work_struct` típusú elemet kell létrehoznunk. Az első alkalommal az alábbi makróval tehetjük ezt meg:

```
INIT_WORK(struct work_struct *work, void (*function)(void *));
```

Azonban a későbbi meghívások során elegendő a következő makrót használnunk:

```
PREPARE_WORK(struct work_struct *work, void (*function)(void *));
```

Majd még szintén a megszakítás kezelő rutinban el kell helyeznünk a `work_struct` elemmel reprezentált feladatot a workqueue-ba:

```
int queue_work(struct workqueue_struct *wq, struct work_struct
*work);
```