LINUX PROGRAMOZÁS

v0.1

2002.09.23. 18:27

Bányász Gábor (tomcat@avalon.aut.bme.hu)
Automatizálási és Alkalmazott Inf. tsz.

Tartalomjegyzék

1. BEVEZETÉS ... 4

1.1 A LINUX ... 4
1.2 A SZABAD SZOFTVER ÉS A LINUX TÖRTÉNETE ... 4
1.3 INFORMÁCIÓ FORRÁSOK ... 6
1.3.1 Linux Documentation Project (LDP) ... 6
1.3.2 Linux Software Map (LSM) ... 7
1.3.3 További információforrások ... 7

2. A LINUX KERNEL RÉSZLETEI I. ... 8

2.1 MEMÓRIA MENEDZSMENT ... 8
2.1.1 A Virtuális Memória modell ... 9
2.1.1.1 Igény szerinti lapozás ... 10
2.1.1.2 Swapping ... 11
2.1.1.3 Megosztott virtuális memória ... 11
2.1.1.4 Fizikai és virtuális címzési módok ... 11
2.1.1.5 Hozzáférés vezérlés ... 11
2.1.2 Cache-ek ... 12
2.2 PROCESSZEK ... 13
2.2.1 A Linux processzek ... 14
2.2.2 Azonosítók ... 15
2.2.3 Ütemezés ... 16
2.2.4 Processzek létrehozása ... 17
2.2.5 Idő és időzítők ... 18
2.2.6 A programok futtatása ... 18

3. FEJLESZTŐI ESZKÖZÖK ... 20

3.1 SZÖVEGSZERKESZTŐK ... 20
3.1.1 Emacs ... 20
3.1.2 vi ... 20
3.1.3 pico ... 20
3.1.4 joe ... 21
3.2 FORDÍTÓK ... 22
3.2.1 GNU Compiler Collection ... 22
3.2.2 gcc ... 23
3.3 MAKE ... 26
3.3.1 Kommentek ... 27
3.3.2 Explicit szabályok ... 27
3.3.3 Változódefiníciók ... 28
3.3.4 Direktívák ... 29
3.3.5 Implicit szabályok ... 29
3.4 KDEVELOP ... 31

4. DEBUG ... 33

4.1 GDB ... 33
4.1.1 Példa ... 33
4.1.2 A gdb indítása ... 34
4.1.3 Breakpoint, watchpoint, catchpoint ... 34
4.1.4 xxgdb ... 36
4.1.5 DDD ... 37
4.1.6 KDevelop internal debugger ... 37
4.2 MEMÓRIAKEZELÉSI HIBÁK ... 37
4.2.1 Electric Fence ... 39
4.2.1.1 Az Electric Fence használata ... 39
4.2.1.2 Memory Alignment kapcsoló ... 41
4.2.1.3 Az eléírás ... 41
4.2.1.4 További lehetőségek ... 42
4.2.1.5 Erőforrás igények ... 42

4.2.2 NJAMD (Not Just Another Malloc Debugger) ... *43*
4.2.2.1 Használata ... 43
4.2.2.2 Memory leak detektálás ... 45
4.2.2.3 Az NJAMD kapcsolói ... 45
4.2.2.4 Összegzés ... 46
4.2.3 mpr ... *46*
4.2.4 MemProf ... *47*
4.3 RENDSZERHÍVÁSOK MONITOROZÁSA: STRACE ... 48
4.4 TOVÁBBI HASZNOS SEGÉDESZKÖZÖK ... 48

1. Bevezetés

1.1 A Linux

A **Linux** szónak több jelentése is van. A technikailag pontos definíciója:

> A Linux egy szabadon terjeszthető, Unix-szerű operációs rendszer kernel.

Azonban a legtöbb ember a Linux szó hallatán az egész operációs rendszerre gondol, amely a Linux kernelen alapul. Így általában az alábbi értelemben használjuk:

> A Linux egy szabadon terjeszthető, Unix-szerű operációs rendszer, amely tartalmazza a kernelt, a rendszer eszközöket, programokat, és a teljes fejlesztői környezetet.

Mi is ezt a második értelmét használjuk.

A Linux egy kiváló, ingyenes platformot ad a programok fejlesztéséhez. Az alapvető fejlesztő eszközök a rendszer részét képezik. A Unix-szerűségéből adódóan, programjainkat könnyen portolhatjuk 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 extrémebb problémák, amelyek a kernel módosítását igénylik, megoldására is lehetőségünk nyílik.
- Mivel a stabil és a fejlesztői kernel vonala jól elkülönül, ezért célrendszerek készítésénél szabadon választhatunk, hogy egy stabil, vagy a legújabb fejlesztéseket tartalmazó rendszerre van szükségünk.
- A Linux fejlesztése nem profit orientált 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.

Az előnyei mellett meg kell természetesen említenünk hátrányait is. A decentralizált fejlesztés és a marketing hatások hiányából adódóan a Linux nem rendelkezik olyan egységes, felhasználó barát kezelői felülettel, mint konkurensei. (Bele értendő ebbe a fejlesztői eszközöket is.) Azonban ennek figyelembe vételével is a Linux kiváló lehetőségeket nyújt a fejlesztésekhez, első sorban a nem desktop applikációk területén.

1.2 A szabad szoftver és a Linux története

A számítástechnika hajnalán a cégek kis jelentőséget tulajdonítottak a szoftvereknek. Első sorban a hardwaret szándékozták eladni, a szoftvereket csak járulékosan adták

hozzá, marketing jelentőséget nem tulajdonítottak neki. Ez azt eredményezte, hogy a forráskódok, algoritmusok szabadon terjedtek az akadémiai szférában.

Azonban ez az időszak nem tartott sokáig, a cégek hamar rájöttek a szoftverekben rejlő üzleti lehetőségekre, és ezzel beköszöntött a copyright 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) szervezetet Cambridge-ben. Az FSF célja szabadon fejleszthető szoftverek fejlesztése.

A kifejezésben a *free* nem ingyenességet, hanem szabadságot jelent. Hite szerint a szoftvereknek a hozzájuk tartozó dokumentációval, forrás kóddal együtt szabadon hozzáférhetőnek és terjeszthetőnek kell lenniük. Ennek elősegítésére megalkotta (némi segítséggel) a *General Public License*-t (GPL).

A GPL három fő irányelve:

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

A GPL egyik jellemzője, hogy nem nyilatkozik az árról. Vagyis a GPL-es szoftverek tetszőleges áron eladhatóak a vevőnek. Egyetlen kikötés, hogy a forrás kód ingyenesen jár a szoftverhez. Azonban a vevő szabadon terjesztheti a programot, és a forráskódját. Az Internet elterjedésével ez azt eredményezte, hogy a GPL-es szoftverek ára alacsony (sok esetben ingyenesek), de lehetőség nyílik ugyanakkor arra, hogy a termékhez kapcsolódó szolgáltatásokat, támogatást adjanak el.

Az FSF által támogatott legfőbb mozgalom a GNU's Not Unix (röviden GNU) projekt, amelynek célja, hogy egy 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 gépén. A Minix az operációs rendszerek működését, felépítését volt hivatott bemutatni, ezért egyszerűnek, könnyen értelmezhetőnek kellett maradnia. Ebből következően nem tudta kielégíteni Linus igényeit, ezért egy saját Unix-szerű operációs rendszer fejlesztésébe vágott bele.

Eredetileg a Linux kernelt egy gyenge licensszel látta el, 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 applikációk fejlesztését is a rendszerre. A *gcc*, *bash*, *Emacs* programok portolt változatai gyorsan követték. Így az 1992-es év elején már aránylag könnyen installálható volt a Linux 0.95 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 a rendszer felépítéséhez. A rendszer további jelentős részletei származnak a Kaliforniai Berkley egyetem nyílt Unix forráskódjaiból, illetve az X konzorciumtól.

A Linux fejlődésével 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 viszonylag későn született a társaihoz képest, azonban napjaink egyik legnépszerűbb változata. Céljuk egy stabil, biztonságos, könnyen installálható és használható csomag készítése. Továbbá termékeikhez támogatást, tanfolyamokat, könyveket is nyújtanak. Ennek következtében az üzleti Linux felhasználás egyik vezetőjévé nőtték ki magukat.

A Linux disztribúciók általában a Linux kernel mellett tartalmazzák a fejlesztői könyvtárakat, fordítókat, értelmezőket, *shell*-eket, applikációkat, segéd programokat, konfigurációs eszközöket és még sok más komponenst.

Napjainkban már sok Linux disztribúcióval találkozhatunk. Mindegyiknek megvannak az előnyei, hátrányai, hívői. Azonban mindegyik ugyanazt a Linux kernelt és ugyanazokat a fejlesztői alapkönyvtárakat tartalmazza.

1.3 Információ források

A Linux története során mindig is kötődött az Internethez. Az Internet közösség készítette, fejlesztette, terjesztette, így a dokumentációk is az Interneten találhatóak meg nagyrészt. Ezért a legfőbb információforrásnak a hálózatot tekinthetjük.

1.3.1 Linux Documentation Project (LDP)

A Linux világ egyik legjelentősebb információ forrása a Linux Documentation Project (LDP). Az LDP elsődleges feladata ingyenes, magas szintű dokumentációk fejlesztése a GNU/Linux operációs rendszer számára. Céljuk minden Linux-al kapcsolatos témakör letakarása a megfelelő dokumentumokkal. Ez magában foglalja a *HOWTO*-k és *guide*-ok készítését, a *man* oldalak, *info*, és egyéb dokumentumok összefogását.

- A **HOWTO**-k egy-egy probléma megoldását nyújtják. Elérhetőek számos formátumban: HTML, PostScript, PDF, egyszerű text.
- A **guide** kifejezés az LDP által készített könyveket takarja. Egy-egy témakör bővebb kifejtését tartalmazzák.
- A **man** oldal az általános Unix formátum az elektronikus manuálok tárolására.

Az LDP által elkészített dokumentumok megtalálhatóak a következő címen: http://www.tldp.org/

1.3.2 Linux Software Map (LSM)

Amennyiben problémánk akad egyes Linux-os programok megtalálásával, akkor segíthet a Linux Software Map (LSM). Ez egy nagy adatbázis, amely a Linux-os programok jellemzőit tartalmazza úgy, mint nevét, rövid leírását, verzióját, íróját, fellelhetőségét és licenszét.
Címe: http://lsm.execpc.com/

1.3.3 További információforrások

- Linux system labs: http://www.lsl.com/
- RedHat dokumentumok: http://www.redhat.com/docs/
- Linoleum (a fejlesztésekhez): http://leapster.org/linoleum/
- Linux.hu: http://www.linux.hu/
- Newsgroup-ok, levelezési listák.
- A GPL-es szoftverek mindegyikének a forráskódja elérhető, így kérdéseinkre ott is megtalálhatjuk a választ.
- És még számos hely található az Interneten, ahol a hiányzó információra rálelhetünk.

2. A Linux Kernel részletei I.

Ebben a fejezetben a Linux kernel egyes, általánosan a fejlesztők számára fontos részeivel ismerkedünk meg. Elsősorban a memória menedzsmenttel és a processzekkel kapcsolatos kérdéseket tárgyaljuk. A kernel további részeinek ismertetésére az egyes témakörök kapcsán kerül sor.

2.1 Memória menedzsment

A memória menedzsment alrendszer az operációs rendszerek egyik legfontosabb eleme. A kezdetek óta gyakran felmerül a probléma, hogy több memóriára lenne szükségünk, mint amennyi a gépünkben fizikailag adott. Több stratégia is kialakult az idők során ennek megoldására. Az egyik legsikeresebb a virtuális memóriakezelés. A virtuális memória modellel úgy tűnik, mintha több memória állna rendelkezésünkre az aktuálisnál azáltal, hogy szükség szerint osztja meg a versenyző processzek között. De a virtuális memóriakezelés ennél többet is nyújt.

A memória menedzsment alrendszer feladatai:

Nagy címtartomány

Az operációs rendszer a valódinál nagyobb memória területet mutat a rendszer felé. A virtuális memória a valódi többszöröse is lehet.

Védelem

Minden processz a rendszerben saját virtuális memória területtel rendelkezik. Ezek a címtartományok teljesen elkülönülnek egymástól, így az egyik applikációnak nem lehet hatása a másikra. Továbbá a hardware virtuális memória kezelő mechanizmusa lehetővé teszi, hogy egyes memória területeket teljesen írásvédetté tehessünk. Ez megvédi a programok kód és adat területeit attól, hogy hibás programok beleírhassanak.

Memória leképezés

A memória leképezés lehetővé teszi kép- és adatállományok leképezését a processz memória területére. A memória leképezés során az állományok tartalma közvetlenül bekapcsolódik a processz virtuális memória területére.

Igazságos fizikai memória allokáció

A memória menedzsment alrendszer lehetővé teszi minden futó processz számára, hogy igazságosan részesedjen a fizikai memóriából.

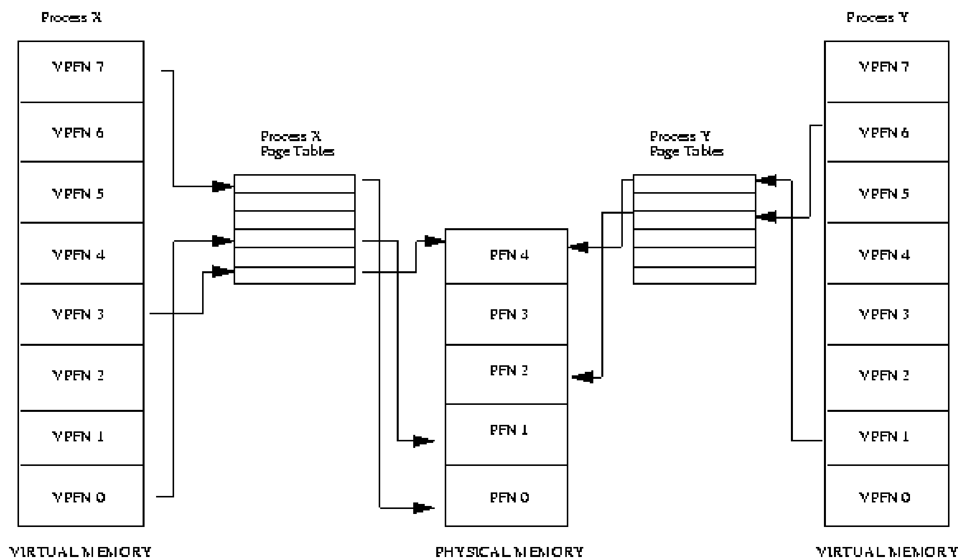
Megosztott virtuális memória

Habár a virtuális memóriakezelés lehetővé teszi a programok számára, hogy egymástól elválasztott címtartományokat kapjanak, időnként szükség van arra, hogy a processzek osztozzanak egy megosztott memória területen.

Említhetjük az esetet, amikor több processz ugyanazt a kódot használja (például többen futtatják a *bash* shellt), ilyenkor ahelyett, hogy bemásolnánk a kódot minden virtuális memória területre sokkal célszerűbb, ha csak egy

példányban tartjuk a fizikai memóriában és a processzek osztoznak rajta. Hasonló az eset a dinamikus könyvtárakkal, ahol szintén a kódot megosztjuk a folyamatok között.
A megosztott memória egy másik alkalmazási területe az *Inter Process Communication (IPC)*. A processzek közötti kommunikáció egyik metódusa, amikor két vagy több folyamat a megosztott memórián keresztül cserél információkat. A Linux támogatja a Unix System V *shared memory IPC* metódusát.

2.1.1 A Virtuális Memória modell



Ábra 2-1 A virtuális memória leképezése fizikai memóriára

A Linux által használt virtuális memóriakezelés tárgyalásaként egy egyszerűsített absztrakt modellt tekintünk át. (A Linux memóriakezelése ennél összetettebb, azonban az ismertetett elméleteken alapszik.)

Amikor a processzor futtat egy programot, kiolvassa a hozzá tartozó parancsokat a memóriából és dekódolja azt. A dekódolás - futtatás során szükség szerint olvashatja és írhatja egyes memória területek tartalmát. Majd tovább lép a következő kód elemre. Ezen a módon a processzor folyamatosan parancsokat olvas és adatokat ír vagy olvas a memóriából.

A virtuális memória használata esetén ezek a címek mind virtuális címek. Ezeket a virtuális címeket a processzor átkonvertálja fizikai címekre az operációs rendszer által karban tartott információs táblák alapján.

Ennek a folyamatnak a megkönnyítésére a virtuális és a fizikai memória egyenlő, 4Kbyte-os (Intel x86) lapokra tagolódik. Minden lap rendelkezik egy saját egyedi azonosító számmal, *page frame number (PFN)*.

Ebben a modellben a virtuális cím két részből tevődik össze. Egy ofszetből és egy lapszámból. Ha a lapok mérete 4Kbyte akkor az első 12 bit adja a ofszetet, a felette levő bitek a lap számát. Minden alkalommal, amikor a processzor egy virtuális címet kap, szétválasztja ezeket a részeket, majd a virtuális lapszámot átkonvertálja fizikaira. Ezek után az ofszet segítségével már megtalálja a memóriában a kérdéses fizikai címet. A leképezéshez a processzor a lap táblákat (*page tables*) használja.

A 2.1-es ábra a virtuális címterület használatát szemlélteti két processzes esetre. Mindkét folyamat saját lap táblával rendelkezik. Ezek a lap táblák az adott processz virtuális lapjait a memória fizikai lapjaira képezik le.
Minden lap tábla bejegyzés az alábbi bejegyzéseket tartalmazza:

- Az adott tábla bejegyzés érvényes-e.
- A fizikai lapszám (PFN).
- Hozzáférési információ. Az adott lap kezelésével kapcsolatos információk, írható-e, kód vagy adat.

2.1.1.1 Igény szerinti lapozás

Ha a fizikai memória jóval kevesebb, mint a virtuális memória, akkor az operációs rendszer csak óvatosan adhat ki területeket, kerülve a fizikai memória ineffektív felhasználását. Az egyik metódus a takarékoskodásra, hogy csak azokat a virtuális lapokat töltjük be a memóriába, amelyeket a futó programok éppen használnak. Például egy adatbázis kezelő applikációnál elég, ha csak azokat az adatokat tartjuk a memóriában, amelyeket éppen kezelünk. Ezt a technikát, amikor csak a használt virtuális lapokat töltjük be a memóriába, igény szerinti lapozásnak hívjuk.

Amikor a processz olyan virtuális címhez próbál hozzáférni, amely éppen nem található meg a memóriában, a processzor nem találja meg a lap tábla bejegyzését. Ilyenkor értesíti az operációs rendszert a problémáról.

Ha a hivatkozott virtuális cím nem érvényes, az azt jelenti, hogy a processz olyan címre hivatkozott, amire nem lett volna szabad. Ilyenkor az operációs rendszer terminálja a folyamatot a többi védelmében.

Ha a hivatkozott virtuális cím érvényes, de a lap nem található éppen a memóriában, akkor az operációs rendszernek be kell hoznia a háttértárolóról. Természetesen ez a folyamat eltart egy ideig, ezért a processzor addig egy másik folyamatot futtat tovább. A beolvasott lap közben beíródik a merevlemezről a fizikai memóriába, és bekerül a megfelelő bejegyzés a lap táblába. Ezek után a folyamat a megállás helyétől fut tovább. Ilyenkor természetesen a processzor már el tudja végezni a leképezést, így folytatódhat a feldolgozás.

A Linux az igény szerinti lapozás módszerét használja a folyamatok kódjának betöltésénél. Amikor a parancsokat lefuttatjuk, a kódot tartalmazó állományt megnyitja rendszer, és a tartalmát leképezi a folyamatok virtuális memória területére. Ezt hívjuk memória leképezésnek (*memory mapping*). Ilyenkor csak a kód eleje kerül be a fizikai memóriába, a többi marad a lemezen. Ahogy a kód fut és generálja a lap hibákat úgy a Linux behozza a kód többi részét is.

2.1.1.2 Swapping

Amikor a folyamatnak újabb virtuális lapok behozására van szüksége, és nincs szabad hely a fizikai memóriában, olyankor az operációs rendszernek helyet kell csinálnia úgy, hogy egyes lapokat eltávolít.

Ha az eltávolítandó lap kódot, vagy olyan adatrészeket tartalmaz, amelyek nem módosultak, olyankor nem szükséges a lapot lementeni. Ilyenkor egyszerűen kidobható, és legközelebb megtalálható az adott kód vagy adat állományban, amikor szükség lesz rá.

Azonban ha a lap módosult, akkor az operációs rendszernek el kell tárolnia a tartalmát, hogy később előhozhassa. Ezeket a lapokat nevezzük *dirty page*-nek, és az állományt, ahova eltávolításkor mentődnek *swap file*-nak. A hozzáférés a swap állományhoz nagyon hosszú ideig tart a rendszer sebességéhez képest, ezért az operációs rendszernek az optimalizáció érdekében mérlegelnie kell.

Ha a swap algoritmus nem elég effektív előfordulhat az, hogy egyes lapok folyamatosan swappelődnek, ezzel elpazarolva a rendszer idejét. Hogy ezt elkerüljük az algoritmusnak azokat a lapokat, amelyeken a folyamatok dolgoznak éppen, lehetőleg a fizikai memóriában kell tartania. Ezeket a lapokat hívjuk *working set*-nek.

A Linux a *Least Recently Used (LRU)* lapozási technikát használja a lapok kiválasztására. Ebben a sémában a rendszer nyilvántartja minden laphoz, hogy hányszor fértek hozzá. Minél régebben fértek hozzá egy laphoz, annál inkább kerül rá a sor a következő swap műveletnél.

2.1.1.3 Megosztott virtuális memória

A virtuális memóriakezelés lehetővé teszi több folyamatnak, hogy egy memória területet megosszanak. Ehhez csak arra van szükség, hogy egy közös fizikai lapra való hivatkozást mindkét processz lap táblájába bejegyezzünk, ezáltal azt a lapot leképezve a virtuális címterületükre. Természetesen ugyanazt a lapot a két virtuális címtartományban két különböző helyre is leképezhetjük.

2.1.1.4 Fizikai és virtuális címzési módok

Nem sok értelme lenne, hogy maga az operációs rendszer a virtuális memóriában fusson. Ha belegondolunk, meglehetősen bonyolult szituáció volna, ha az operációs rendszernek a saját memória lapjait kellene kezelnie. A legtöbb multifunkciós processzor támogatja a fizikai címzést is a virtuális címzés mellett. A fizikai címzési mód nem igényel lap táblákat, a processzor nem végez semmilyen cím leképezést ebben a módban. A Linux kernel is természetesen ebben a fizikai címtartományban fut. (Az Alpha AXP processzor nem támogatja a fizikai címzési módot, ezért ezen a platformon más megoldásokat alkalmaznak a Linux kernelnél.)

2.1.1.5 Hozzáférés vezérlés

A lap tábla bejegyzések az eddig tárgyaltak mellett hozzáférési információkat is tartalmaznak. Amikor a processzor egy bejegyzés alapján átalakítja a virtuális címeket

fizikai címekké, párhuzamosan ellenőrzi ezeket a hozzáférési információkat is, hogy az adott process számára a művelet engedélyezett-e vagy sem.

Több oka is lehet, amiért korlátozzuk a hozzáférést egyes memória területekhez. Egyes területek, mint például a program kód tárolására szolgáló memória rész, csak olvasható lehet, az operációs rendszernek meg kell akadályoznia, hogy a processz adatokat írhasson a kódjába. Ezzel szemben azoknak a lapoknak, amelyek adatokat tartalmaznak, írhatónak kell lenniük, de futtatni nem szabad a memória tartalmát. A futtatásra is a legtöbb processzor két módot támogat: *kernel* és *user* módot. Ennek oka, hogy nem akarjuk, hogy kernel kódot futtasson egy felhasználói program, vagy hogy a kernel adatstruktúrájához hozzáférhessen.

Összességében jellemzően az alábbi jogok szerepelnek egy lap tábla bejegyzésben:

- Érvényesség
- Futtathatóság
- Írhatóság
- Olvashatóság
- Kernel módú program olvashatja-e a lapot.
- User módú program olvashatja-e a lapot.
- Kernel módú program írhatja-e a lapot.
- User módú program írhatja-e a lapot.

Továbbá a Linux által támogatott további jogok:

- Page dirty: A lapot ki kell-e írni a swap állományba.
- Page accessed: A laphoz hozzáfértek-e.

2.1.2 Cache-ek

Az eddigi modell önmagában működik, azonban nem elég effektív. A teljesítmény növelés egyik módszere *cache*-ek használata. A Linux több különböző, memória menedzsmenthez kapcsolódó gyorsító tárat használ.

Buffer Cache

A buffer cache a blokkos eszközök adatait tartalmazza. Blokkos eszköz az összes merev lemez, így ez a cache lényegében lemezgyorsító tár.

Page Cache

Feladata hogy gyorsítsa a lemezen tárolt kódokhoz és adatokhoz való hozzáférést a lapok beolvasása során. Amikor a lapokat beolvassuk a memóriába, a tartalma a page cache-ben is eltárolódik.

Swap Cache

Csak a módosított (*dirty*) lapok kerülnek a swap állományba.
Amikor egy lap nem módosult a legutolsó kiírása óta, olyankor nincs szükség arra, hogy újra kiírjuk a swap állományba. Ilyenkor egyszerűen kidobható. Ezzel a swap kezelése során időt takaríthatunk meg.

Hardware Cache-ek

Az egyik leggyakoribb hardware cache a processzorban van, és a lap tábla bejegyzések tárolására szolgál. Ennek segítségével a processzornak nem kell

mindig közvetlenül a lap táblákat olvasnia, hozzáférhet a gyorsító tárból is az információhoz. Ezek a *Translation Look-aside Buffer*-ek egy vagy több processz lap tábla bejegyzéseinek másolatát tartalmazzák.

Amikor egy virtuális címet kell lefordítania a processzornak, akkor először egy egyező TLB bejegyzést keres. Ha talál, akkor segítségével azonnal lefordíthatja a fizikai címre. Ha nem talált megfelelőt, akkor az operációs rendszerhez fordul segítségért. Ilyenkor az operációs rendszer létrehoz egy új TLB bejegyzést, amely megoldja a problémát, és a processzor folytathatja a munkát.

Hátránya a cache-eknek hogy kezelésük plusz erőfeszítéseket igényel (idő, memória), továbbá megsérülésük a rendszer összeomlásához vezet.

2.2 Processzek

Ebben a fejezetben megtudjuk, hogy mi az a processz, és a Linux kernel hogyan hozza létre, menedzseli és törli a processzeket a rendszerből.

A processzek egyes feladatokat hajtanak végre az operációs rendszeren belül. A program gépi kódú utasítások, és adatok összessége, amelyeket a lemezeken tárolunk. Így önmagában egy passzív entitás. A processz ez a program működés közben, amikor éppen fut.

Ebből látszik, hogy dinamikus entitás, folyamatosan változik, ahogy egymás után futtatja az egyes utasításokat a processzor. A program kódja és adatai mellett a processz tartalmazza a program számlálót, a CPU regisztereket, továbbá a processz *stack*-jét, amely az átmeneti adatokat tartalmazza (függvény paraméterek, visszatérési címek, elmentett változók). A Linux egy multitaszkos operációs rendszer, a processzek szeparált taszkok saját jogokkal és feladatokkal, amelyeket a Linux párhuzamosan futtatni képes. Egy processz meghibásodása nem okozza a rendszer más processzeinek meghibásodását. Minden különálló processz a saját virtuális címtartományában fut és nem képes más processzekre hatni. Kivételt képeznek a biztonságos, kernel által menedzselt mechanizmusok.

Élete során egy processz számos rendszer erőforrást használhat (CPU, memória, állományok, fizikai eszközök, stb.). A Linux feladata, hogy ezeket a hozzáféréseket könyvelje, menedzselje, és igazságosan elossza a konkuráló processzek között.

A legértékesebb erőforrás a CPU. Általában egy áll rendelkezésre belőle, de több CPU kezelésére is alkalmas a rendszer. A Linux egy multitaszkos rendszer, így egyik lényeges célja, hogy lehetőleg mindig mindegyik CPU-n fusson egy processz a lehető legjobb kihasználás érdekében. Amennyiben több a processzünk, mint a processzorunk (általában ez a helyzet), olyankor a processzeknek meg kell osztozniuk. A megoldás egyszerű: általában egy processz addig fut, amíg várakozásra kényszerül (általában egy rendszer erőforrásra), majd amikor azt megkapta folytathatja a futását. Amikor a processz várakozik, olyankor az operációs rendszer elveszi tőle a CPU-t és átadja egy másik rászoruló processznek. Az ütemező dönti el, hogy melyik processz legyen a következő. A Linux számos stratégiát használ az igazságos döntéshez.

2.2.1 A Linux processzek

A Linux menedzseli a processzeket a rendszerben. Minden processzhez hozzárendel egy leíró adat struktúrát, és bejegyez egy hivatkozást rá a taszk listájába. Ebből következik, hogy a processzek maximális száma függ ennek a listának a méretétől, amely alapértelmezett esetben 512 bejegyzést tartalmazhat.

A normál processzek mellett a Linux támogat real-time processzeket is. Ezeknek a processzeknek nagyon gyorsan kell reagálnia külső eseményekre, ezért a normál folyamatoktól külön kezeli őket az ütemező.

A taszkokat leíró adatstruktúra nagy és komplex, azonban felosztható néhány funkcionális területre:

Állapot

A processz a futása során különböző állapotokba kerülhet a körülmények függvényében:

Running

A processz fut, vagy futásra kész.

Waiting

A processz egy eseményre vagy egy erőforrásra vár. A Linux megkülönböztet megszakítható és nem megszakítható várakozásokat. A megszakítható várakozás esetén egy szignál megszakíthatja, míg nem megszakítható esetén valamilyen hardware eseményre vár és semmilyen körülmények között nem megszakítható a várakozás.

Stopped

A processz megállt, általában valamilyen szignál következtében. A debug-olás alatt lévő processzek lehetnek ilyen állapotban.

Zombie

Ez egy leállított processz, ami valami oknál fogva még mindig foglalja a leíró adat struktúráját. Ahogy a neve is mondja egy halott folyamat.

Ütemezési információ

Az ütemezőnek szüksége van erre az információra, hogy igazságosan dönthessen, hogy melyik processz kerüljön sorra.

Azonosítók

Minden processznek a rendszerben van egy processz azonosítója. A processz azonosító nem egy index a processz táblában, hanem egy egyszerű szám. Továbbá minden processz rendelkezik egy felhasználó és egy csoportazonosítóval. Ezek szabályozzák az állományokhoz és az eszközökhöz való hozzáférést a rendszerben.

Inter-Processz kommunikáció

A Linux támogatja a klasszikus Unix IPC mechanizmusokat (szignál, pipe, szemafor) és a System V IPC mechanizmusokat is (megosztott memória, szemafor, üzenet lista). Ezekre később térünk ki.

Kapcsolatok

A Linuxban egyetlen processz sem független a többitől. Minden processznek, leszámítva az *init* processzt, van egy szülője. Az új processzek nem létrejönnek, hanem a korábbiakból másolódnak, klóónozódnak. Minden processz leíró adatstruktúra tartalmaz hivatkozásokat a szülő processzre és a leszármazottakra. A *pstree* paranccsal megnézhetjük ezt a kapcsolódási fát.

Idő és időzítők

A kernel naplózza a processzek létrehozási idejét, a CPU felhasználásuk idejét. További a Linux támogatja intervallum időzítők használatát a processzekben, amelyeket beállítva szignálokat kaphatunk bizonyos idő elteltével. Ezek lehetnek egyszeri vagy periodikusan ismétlődő értesítések.

File rendszer

A processzek megnyithatnak és bezárhatnak állományokat. A processz leíró adatstruktúra tartalmazza az összes megnyitott állomány leíróját, továbbá a hivatkozást két *VFS inode*-ra. A VFS inode-ok egy állományt vagy egy könyvtárat írhatnak le egy file rendszeren. A két inode-ból az első a processz *home* könyvtárát mutatja, a második az aktuális *working* könyvtárat.
A VFS inode-oknak van egy számláló mezője, amely számolja, hogy hány processz hivatkozik rá. Ez az oka, amiért nem törölhetünk olyan könyvtárakat, amelyeket egy processz használ.

Virtuális memória

A legtöbb processznek van valamennyi virtuális memóriája. Ez alól csak a kernel szálak és daemon-ok kivételek. A korábbiakban láthattuk, ahogy a Linux kernel ezt kezeli.

Processzor specifikus adatok

Amikor a processz fut, olyankor használja a processzor regisztereit, a stacket, stb. Ez a processz környezete, és amikor taszkváltásra kerül sor, ezeket az adatokat le kell menteni a processz leíró adatstruktúrába. Amikor a processz újraindul innen állítódnak vissza.

2.2.2 Azonosítók

A Linux, mint minden más Unix rendszer, felhasználói és csoportazonosítókat használ az állományok hozzáférési jogosultságának ellenőrzésére. A Linux rendszerben minden állománynak van tulajdonosa és jogosultság beállításai. A legegyszerűbb jogok a *read*, *write*, és az *execute*. Ezeket rendeljük hozzá a felhasználók három csoportjához úgy mint tulajdonos, csoport, és a rendszer többi felhasználója. A felhasználók mindhárom osztályára külön beállítható mindhárom jog.

Természetesen ezek a jogok valójában nem a felhasználóra, hanem a felhasználó azonosítójával futó processzekre érvényesek. Ebből következően a processzek az alábbi azonosítókkal rendelkeznek:

uid, gid

A felhasználó *user* és a *group* azonosítói, amelyekkel a processz fut.

effektív uid és gid

Egyes programoknak meg kell változtatniuk az azonosítóikat a sajátjukra (amelyet a VFS inode-ok tárolnak), így nem a futtató processz jogait öröklik tovább. Ezeket a programokat nevezzük *setuid*-os programoknak. Segítségükkel korlátozott hozzáférést nyújthatunk a rendszer egyes védett részeihez. Az effektív uid és gid a program azonosítói, az uid és a gid marad az eredeti, a kernel pedig a jogosultság ellenőrzésnél az effektív azonosítókat használja.

file rendszer uid és gid

Ezek normál esetben megegyeznek az effektív azonosítókkal és a file rendszer hozzáférési jogosultságokat szabályozzák. Elsősorban az NFS file-rendszereknél van rá szükség, amikor a user módú NFS szervernek különböző állományokhoz kell hozzáférnie, mintha az adott felhasználó nevében. Ebben az esetben csak a file-rendszer azonosítók változnak.

saved uid és gid

Ezek az azonosítók a POSIX szabvány teljesítése érdekében lettek implementálva. Olyan programok használják, amelyek a processz azonosítóit rendszerhívások által megváltoztatják. A valódi uid és gid elmentésére szolgálnak, hogy később visszaállíthatóak legyenek.

2.2.3 Ütemezés

Minden processz részben user módban, részben system módban fut. Az, hogy ezeket a módokat hogyan támogatja a hardware, eltérő lehet, azonban mindig van valami biztonságos mechanizmus arra, hogy hogyan kerülhet a processz user módból system módba és vissza. A user módban jóval kevesebb lehetősége van a processznek, mint system módban. Ezért minden alkalommal, amikor a processz egy rendszerhívást hajt végre, a user módból átkapcsol system módba és folytatja a futását. Ezen a ponton a kernel futtatja a processznek azt a részét.

A Linuxban a processzeknek nincs előjoga az éppen futó processzekkel szemben, nem akadályozhatják meg a futását, hogy átvegyék a processzort. Azonban minden processz lemondhat a CPU-ról, amelyiken fut, ha éppen valami rendszer eseményre vár. Például ha egy processznek várnia kell egy karakterre. Ezek a várakoztatások a rendszerhívásokon belül vannak, amikor a processz éppen system módban fut. Ebben az esetben a várakozó processz felfüggesztődik, és egy másik futhat.

A processzek rendszeresen használnak rendszerhívásokat, és gyakran kell várakozniuk. Azonban ha a processzt addig engedjük futni, amíg a következő várakozásra kényszerül, akkor az időnként két rendszerhívás között akár komoly időt is jelenthet. Ezért szükség van még egy megkötésre: egy processzt csak rövid ideig engedhetünk futni, és amikor ez letelik, akkor várakozó állapotba kerül, majd később folytathatja. Ezt a rövid időt nevezzük időszeletnek (*time-slice*).

Az ütemező (*scheduler*) az, aminek a következő processzt ki kell választania a futásra készek közül. Futásra kész az a processz, amely már csak CPU-ra vár, hogy futhasson. A Linux egy egyszerű prioritás alapú ütemező algoritmus használ a választáshoz.

Az ütemező számára a CPU idő elosztásához az alábbi információkat tárolja a rendszer minden processzhez:

policy

Az ütemezési metódus, amelyet a processzhez rendelünk. Két típusa van a Linux processzeknek: normál és valós idejű. A valós idejű processzeknek magasabb prioritásuk van, mint bármely más processznek. Ha egy valós idejű processz futásra kész, akkor mindig őt választja elsőnek a rendszer.
A normál processzeknél csak egy általános, időosztásos metódust választhatunk. A real-time processzeknek azonban kétféle metódusuk lehet: *round robin* vagy *first in, first out*. A FIFO algoritmus egy egyszerű nem időosztásos algoritmus és a statikusan beállított prioritási szint alapján választ a rendszer. A *round robin* a FIFO továbbfejlesztett változata, ahol a processz csak egy bizonyos ideig futhat és a prioritás módosításával minden futásra kész valós idejű processz lehetőséget.

priority

Ez a prioritás, amelyet az ütemező a processznek ad. Módosítható rendszerhívásokkal és a *renice* paranccsal.

rt_priority

A Linuxban használatos real-time processzek számára egy relatív prioritást adhatunk meg.

counter

Az az idő, ameddig a processz futhat. Indulásképpen a prioritás értékre állítódik be, majd az óra ütésekre csökken.

Az ütemezőt több helyen is meghívja a kernel. Lefut, amikor az aktuális processz várakozó állapotba kerül, meghívódhat rendszerhívások végén, vagy amikor a processz visszatér user módba a system módból. Továbbá amikor a processz *counter* értéke eléri a nullát.

2.2.4 Processzek létrehozása

A rendszer induláskor kernel módban fut, és csak egyetlen inicializáló processz létezik. Ez rendelkezik mindazokkal az adatokkal, amelyről a többi processznél beszéltünk, és ugyanúgy egy leíró adatstruktúrában letároljuk, amikor a többi processz létrejön és fut. A rendszer inicializáció végén a processz elindít egy kernel szálat (neve *init*) és ezek után egy idle loop-ban kezd, és a továbbiakban már nincs szerepe. Amikor a rendszernek nincs semmi feladata az ütemező ezt az idle processzt futtatja.

Az *init* kernel szálnak, vagy processznek a processz azonosítója 1. Ez a rendszer első igazi processze. Elvégez néhány beállítást (feléleszti a rendszer konzolt, *mount*olja a *root* file rendszert), majd lefuttatja a rendszer inicializáló programot (nem keverendő a korábban említett processzel). Ez a program valamelyik a következők közül (az adott disztribúciótól függ): */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 mint az *init* kernel szál leszármazottai.

Újabb processzeket létrehozhatunk egy régebbi processz klóónozásával, vagy ritkábban az aktuális processz klóónozásával. Az újabb folyamat a *fork* (processz) vagy a *clone* (thread) rendszerhívással hozható létre, és maga a klóónozás kernel módban történik. A rendszerhívás végén, pedig egy új processz kerül be a várakozási listába.

2.2.5 Idő és időzítők

A kernel könyveli a processzek létrehozási időpontját és az életük során felhasznált CPU időt. Mértékegysége a *jiffies*. Minden óra ütésre frissíti a *jiffies*-ben mért időt, amit a processz a system és a user módban eltöltött.

Továbbá ezek mellett a könyvelések mellett a Linux támogatja a processzek számára az intervallum időzítőket.

A processz felhasználhatja ezeket az időzítőket, hogy különböző szignálokat küldessen magának, amikor lejárnak. Három féle intervallum időzítőt különböztetünk meg:

Real

A timer valós időben dolgozik, és lejártakor egy SIGALRM szignált küld.

Virtual

A timer csak akkor működik, amikor a processz fut, és lejártakor egy SIGVTALRM szignált küld.

Profile

A timer egyrészt a processz futási idejében működik, másrészt amikor a rendszer a processzhez tartozó műveleteket hajt végre. Lejártakor egy SIGPROF szignált küld. Első sorban arra használják, hogy lemérjék az applikáció mennyi időt tölt a user és a kernel módban.

Egy vagy akár több időzítőt is használhatunk egyszerre. A Linux kezeli az összes szükséges információt hozzá a processz adatstruktúrájában. Rendszerhívásokkal konfigurálhatjuk, indíthatjuk, leállíthatjuk, és olvashatjuk őket.

2.2.6 A programok futtatása

A Linuxban, mint a többi Unix rendszerben, a programokat és a parancsokat általában a parancsértelmező futtatja. A parancsértelmező egy felhasználói processz és a neve *shell*.

A Linuxokban sok shell közül választhatunk (sh, bash, tcsh, stb.). A parancsok, néhány beépített parancs kivételével, általában bináris állományok. Ha egy parancs nevét beadjuk, akkor a shell végigjárja a keresési utakat, és egy futtatható állományt keres a megadott névvel. Ha megtalálja, akkor betölti és lefuttatja. A shell klóónolja magát a fent említett *fork* metódussal és az új leszármazott processzben fut a megtalált állomány. Normál esetben a shell megvárja a gyerek processz futásának végét, és csak

utána adja vissza a promptot. Azonban lehetőség van a processzt a háttérben is futtatni.

A futtatható file többféle bináris vagy script állomány lehet. A scriptet a megadott shell értelmezi. A bináris állományok kódot és adatot tartalmaznak, továbbá információkat a operációs rendszer számára, hogy futtatni tudja őket. A Linux alatt a leggyakrabban használt bináris formátum az *ELF*.

A támogatott bináris formátumokat a kernel fordítása során választhatjuk ki, vagy utólag modulként illeszthetjük be. Az általánosan használt formátumok az *a.out* az *ELF* és néha a Java.

3. Fejlesztői eszközök

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

A Linux disztribúciók számtalan megbízható fejlesztő eszközt 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. Kezelésük megtanulása meg csak egy kis idő kérdése.

3.1 Szövegszerkesztők

Linuxhoz is találhatunk több integrált fejlesztői környezetet (**integrated development environment, IDE**), azonban egyszerűbb esetekben még továbbra is gyakran nyúlunk az egyszerű szövegszerkesztőkhöz. Sok Unix fejlesztő továbbra is ragaszkodik ezekhez a kevésbé barátságos, de a feladathoz sokszor tökéletesen elegendő, eszközökhöz.

A Linux története során az alábbi eszközök terjedtek el:

3.1.1 Emacs

Az eredeti *Emacs* szövegszerkesztőt Richard Stallman, az FSF alapítója, készítette. Évekig a GNU Emacs volt a legnépszerűbb változat. Később a grafikus környezethez készített XEmacs terjedt el.

A felhasználói felülete nem olyan csillogó, mint sok más rendszernél, azonban számos, a fejlesztők számára jól használható funkcióval rendelkezik. Ilyen például a szintaxis ellenőrzés. Vagy ha a fordítót beleintegráljuk képes annak hibaüzeneteit értelmezni, és a hibákat megmutatni. Továbbá lehetővé teszi a debugger integrálását is a környezetbe.

3.1.2 vi

A *vi* egy egyszerű szövegszerkesztő. Kezelését leginkább a gépelni tudók igényeihez alakították. A parancskészletét úgy állították össze, hogy a lehető legkevesebb kézmozgással használható legyen.

Azonban egy tapasztalatlan felhasználó számára kezelése meglehetősen bonyolult lehet, ezért népszerűsége erősen megcsappant.

3.1.3 pico

A *pico* egy egyszerű képernyő orientált szövegszerkesztő. Általában minden Linux rendszeren megtalálható. Alapvetően a *pine* levelező csomag része. Elterjedten

használják a kisebb szövegek gyors módosítására. Komolyabb feladatokra nem ajánlott. Ezekben az esetekben alkalmasabb a következő részben tárgyalt *joe* vagy egy a feladatnak megfelelően specializálódott editor.

A program parancsai folyamatosan láthatóak a képernyő alján. A "^" jel azt jelenti, hogy a billentyűt a ctrl gomb nyomva tartása mellett kell használni. Ha ez esetleg a terminál típusa miatt nem működne, akkor a dupla ESC gombnyomást is alkalmazhatjuk helyette.

Parancsai:

Billentyűkombináció	**Parancs**
<ctrl>-g	Segítség
<ctrl>-x	Kilépés (módosítás esetén rákérdez a mentésre)
<ctrl>-o	Az állomány kiírása
<ctrl>-j	Rendezés
<ctrl>-r	Állomány beolvasása és beillesztése a szerkesztett állományba
<ctrl>-w	Szó keresés
<ctrl>-y	Előző oldal
<ctrl>-v	Következő oldal
<ctrl>-k	Szövegrész kivágása
<ctrl>-u	Az utoljára kivágott rész beillesztése az adott kurzor pozícióra. (Többször is vissza lehet illeszteni, vagyis másolni lehet vele.)
<ctrl>-c	Aktuális kurzor pozíció
<ctrl>-t	Helyesírás ellenőrzés

3.1.4 joe

A *joe* egy elterjedten használt szövegszerkesztő. Komolyabb feladatokra is alkalmas, mint a *pico*. Egyszerre több állományt is megnyithatunk vele különböző opciókkal.

Komolyságát a parancsok számából is láthatjuk, amelyet a <ctrl>-k-h gombokkal hozhatunk elő és tüntethetünk el. Itt is "^" jel azt jelenti, hogy a billentyűket a ctrl gomb nyomva tartása mellett kell használni.

Gyakrabban használt parancsok:

Billentyűkombináció	**Parancs**
^KF	Szöveg keresése
^L	Ismételt keresés
^KB	Blokk elejének kijelölése
^KK	Blokk végének kijelölése
^KM	Blokk mozgatása
^KC	Blokk másolása
^KY	Blokk törlése
^Y	Sor törlése

Billentyűkombináció	Parancs
^_	Változtatás visszaléptetése
^KD	Mentés
^KX	Kilépés mentéssel
^C	Kilépés mentés nélkül

3.2 Fordítók

Linux alatt a programozási nyelvektől függően az alábbi kiterjesztésekkel találkozhatunk:

File utótag	Jelentés
.a	Könyvtár
.c	C nyelvű forrás
.C .cc .cpp .cxx .c++	C++ nyelvű forrás
.f .for	Fortran nyelvű forrás
.h	C/C++ nyelvű header file
.hxx	C++ nyelvű header file
.java	Java nyelvű forrás
.o	Tárgykódú (object) file
.pl	Perl Script
.pm	Perl modul script
.s	Assembly kód
.sa	Statikus programkönyvtár
.so.x.y.z	Megosztott könyvtár
.tcl .tk	Tcl script
.x	RPC interfész file

3.2.1 GNU Compiler Collection

A *GNU Compiler Collection* a C, C++, Objective-C, Ada, Fortran, és a Java nyelvek fordítóit foglalja össze egy csomagba. Ezáltal az ezen nyelveken írt programokat mind lefordíthatjuk a GCC-vel.

A "GCC"-t a GNU Compiler Collection rövidítéseként értjük általában, azonban ez egyben a csomag C fordítójának is a leggyakrabban használt neve (GNU C Compiler).

A C++ forrásokat lefordítására a G++ fordító szolgál. Azonban valójában a fordítók integrációja miatt továbbra is a *gcc* programot használjuk.

Hasonló a helyzet az Ada fordítóval, amelyet GNAT-nak neveznek.

További nyelvekhez (Mercury, Pascal) is léteznek *front end*-ek, azonban ezeknek egy részét még nem integrálták be a rendszerbe.

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

3.2.2 gcc

A *gcc* nem rendelkezik felhasználói felülettel. Parancssorból kell meghívnunk a megfelelő paraméterekkel. Számos paramétere ellenére szerencsére használata egyszerű, mivel általános esetben ezen 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
```

Erre válaszként ilyen sorokat kapunk:

```
Reading specs from /usr/lib/gcc-lib/i386-redhat-linux/2.96/specs
gcc version 2.96 20000731 (Red Hat Linux 7.3 2.96-112)
```

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

A program további, gyakran használt paraméterei:

Paraméter	**Jelentés**
-o *filename*	A kimeneti állománynév megadása. Ha nem adjuk meg, akkor az alapértelmezett file név az "a.out" lesz.
-c	Fordítás, linkelés nélkül. A paraméterként megadott forrás állományból tárgykódú (object) file-t készít.
-D*definíció=x*	A definiálja a *definició* makro-t x értékre.
-I*könyvtárnév*	Hozzáadja a *könyvtárnév* paraméterben meghatározott könyvtárt ahhoz a listához, amelyben a header állományokat keresi.
-L*könyvtárnév*	Hozzáadja a *könyvtárnév* paraméterben meghatározott könyvtárt ahhoz a listához, amelyben a library állományokat keresi.
-l*library*	A programhoz hozzálinkeli a *library* nevű programkönyvtárat.
-static	Az alapértelmezett dinamikus linkelés helyett a fordító a statikus programkönyvtárakat linkeli a programba.
-g, -ggdb	A lefordított állományt ellátja a debuggoláshoz szükséges információkkal. A *-g* opció megadásával a fordító a standard debug információkat helyezi el. A *-ggdb* opció arra utasítja a fordítót, hogy olyan további információkat is elhelyezzen a programban, amelyeket csak a *gdb* debugger értelmez.
-O, -O*n*	Optimalizálja a programot az *n* optimalizációs szintre. Alapértelmezett esetben a gcc csak néhány optimalizációt végez. A legáltalánosabban használt optimalizációs szint a 2-es.
-Wall	Az összes, általában 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, world\n");
  return 0;
}
```

Szeretnénk a jól sikerült programunkat ki is próbálni. Ehhez az alábbi paranccsal fordíthatjuk le:

```
gcc -o Hello Hello.c
```

Ha nem rontottuk el a kódot, akkor a fordító hiba üzenet nélkül lefordítja a forrást, és a programra a futtatási jogot is beállítja. Ezután már csak futtatnunk kell:

```
./Hello
```

A programunk és a konzolon megjelenik a következő felirat:

```
Hello, world
```

Vagyis meghívtuk a *gcc* programot, amely lefordította (compile) a kódot, majd meghívta az *ld* nevű linker programot, amely 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 file jött létre. Ezt természetesen össze kell még linkelnünk:

```
gcc -o Hello Hello.o
```

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

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

```
/* second.c */

#include <stdio.h>

double sinc(double);
```

```
int main()
{
  double x;

  printf("Please input x: ");
  scanf("%lf", &x);
  printf("sinc(x) = %6.4f\n", sinc(x));
  return 0;
}
```

A másik implementálja a $\frac{\sin x}{x}$ függvényt:

```
/* sinc.c */

#include <math.h>

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

Ennek fordítása:

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

Így a *second* futtatható file-hoz jutunk. Ugyanezt megoldhattuk volna több lépésben is:

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

Az *-lm* kapcsolóra azért van szükségünk, hogy hozzálinkeljük a *sin(x)* függvényt tartalmazó matematikai programkönyvtárat a programunkhoz. Általában a *–l* kapcsoló arra szolgál, hogy egy programkönyvtárat hozzáfordítsunk a programunkhoz. A Linux rendszerhez számos szabványosított programkönyvtár tartozik, amelyek a */lib* illetve a */usr/lib* könyvtárban 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 header állományokkal. 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 file 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 header állományainkat.
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 elhelyezhetünk feltételes fordítási direktívákat a debuggolás megkönnyítésére:

```
#ifdef DEBUG
     printf("Thread started successfully.");
#endif
```

A fenti esetben látjuk az új szálak indulását debug módban, viszont ez az információ szükségtelen a felhasználó számára egy már letesztelt program esetén.

3.3 Make

A Unix-os fejlesztések egyik oszlopa a *make* program. Ez az eszköz lehetővé teszi, hogy könnyen leírjuk és automatizáljuk a programunk fordításának menetét. De nem csak nagy programok esetén, hanem akár egy forrásállományból álló programnál is egyszerűsíti a fordítást azáltal, hogy csak egy *make* parancsot kell kiadnunk az aktuális könyvtárban a fordító paraméterezése helyett. Továbbá ha egy komolyabb program sok forrás állományból áll, de csak néhányat módosítunk, akkor az összes állomány újrafordítása helyett csak a szükségeseket frissíti.

Ahhoz, hogy mindezeket a funkciókat megvalósíthassa egy un. *Makefile*-ban le kell írnunk a programunk összes forrás állományát. Erre láthatunk egy példát:

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

- Az **1.** sorban egy **kommentet** láthatunk. A Unix-os tradíciók alapján a kommentet egy # karakterrel jelöljük.
- A **3.** sorban definiáljuk az *OBJS* változót, melynek értéke: *second.o sinc.o*
- a **4.** sorban ugyanezt tesszük a *LIBS* változóval. A későbbiekben ezt fogjuk majd felhasználni linkelési paraméterek beállítására.
- A **6.** sorban egy **szabály** (rule) megadását láthatjuk. Ebben az *second* állomány az *OBJS* változó értékeitől függ. A second állományt hívjuk itt a

szabály **cél**jának (target) és a *$(OBJS)* adja a **függőségi listát**. Megfigyelhetjük a változó használatának módját is.

- A **7.** sor egy parancssor. Azt mondja el, hogy hogyan készíthetjük el a cél objektumot a függőségi lista elemeiből. Itt állhat több parancssor is szükség szerint, azonban minden sort *TAB* karakterrel kell kezdeni.
- A **9.** sor egy érdekes célt tartalmaz. Ebben a szabályban valójában nem egy állomány létrehozása a célunk, hanem az installációs művelet megadása.
- A **10.** sorban végezzük el a programunk installációját az *install* programmal az */usr/local/bin* könyvtárba.
- A **11.** sor egy problémának a megoldását rejti. A 9. sorban a cél nem egy állomány volt. Azonban ha mégis létezik egy *install* nevű állomány, és az frissebb, mint a függőségi listában szereplő *second* állomány, akkor nem fog lefutni a szabályunk. Ezzel természetesen összezavarja és elrontja a *Makefile*-unk működését. Ezt küszöböljük ki ezzel a sorral. A *.PHONY* egy direktíva, amely módosítja a *make* működését. Ebben az esetben megadhatjuk vele, hogy az *install* cél nem egy file neve.

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

- Kommentek
- Explicit szabályok
- Változódefiníciók
- Direktívák
- Implicit szabályok

3.3.1 Kommentek

A kommentek magyarázatul szolgálnak, a *make* segédprogram gyakorlatilag figyelmen kívül hagyja őket. A kommenteknek # karakterrel kell kezdődniük.

3.3.2 Explicit szabályok

Egy **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. Azonban mint láthattuk a cél nem minden esetben állomány. Hogy ezek az állományok létre jöjjenek, általában más állományokra van szükség. Ezek listáját nevezzük **függőségi listá**nak, **feltételek**nek vagy **előkövetelmény**nek.
Például:

```
foo.o: foo.c defs.h       # példa szabályra
gcc -c -g foo.c
```

A fenti példában a szabály tárgya a *foo.o* file, az előkövetelmény a *foo.c* illetve a *foo.h* állományok. Mindez azt jelenti, hogy a *foo.o* file-t akkor kell újrafordítani, ha

- a *foo.o* file nem létezik,
- a *foo.c* időbélyege korábbi, mint a *foo.o* időbélyege,
- a *defs.h* időbélyege korábbi, mint a *foo.o* időbélyege.

Azt, hogy a *foo.o* file-t hogyan kell létrehozni, a második sor adja meg. A *defs.h* nincs a *gcc* paraméterei között: a függőséget egy – a *foo.c* file-ban található - *#include "defs.h"* C nyelvű preprocesszor direktíva jelenti.

A szabály általános formája:

```
TÁRGYAK: ELŐKÖVETELMÉNYEK; PARANCS
PARANCS
...
```

A *TÁRGYAK* szóközzel elválasztott file nevek, akárcsak az *ELŐKÖVETELMÉNYEK*. A file nevek tartalmazhatnak speciális jelentésű karaktereket (wildcard characters), mint például a „.” (aktuális könyvtár), „*” vagy „%” (tetszőleges mennyiségű karaktersorozat), „~” (home könyvtár). A *PARANCS* vagy az előkövetelményekkel egy sorban van pontosvesszővel elválasztva, vagy a következő sorokban, amelyeket ***TAB* karakterrel kell kezdeni**. Mivel a $ jel már foglalt, a valódi $ jelek helyett $$-t kell írnunk.

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

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

Amikor a make segédprogramot paraméterek nélkül futtatjuk, automatikusan az első szabály hajtódik végre, valamint azok a szabályok, amelyektől az a szabály valamilyen módon függ (vagyis tárgya feltétele valamely feldolgozandó szabálynak). A másik lehetőség, hogy a *make* segédprogramot egy szabálynak a nevével paramétereztük. Ilyenkor azt a szabályt hajtódik végre, illetve amelyektől az a szabály függ.

3.3.3 Változódefiníciók

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

Mint láthattuk a változók megadásának szintaxisa:

```
VÁLTOZÓ = ÉRTÉK
```

Erre a változóra később a

```
$(VÁLTOZÓ)
```

szintaxissal hivatkozhatunk.

A változók használata során kerüljük az alábbi megoldásokat:

```
OBJS = first.o
OBJS = $(OBJS) second.o
OBJS = $(OBJS) third.o
```

Azt várnánk, hogy ezek után az *OBJS* értéke *firs.o second.o third.o* lesz, azonban ez nem így van. Valójában az értéke *$(OBJS) third.o* mert a *make* csak akkor értékeli ki a változót, amikor azt használjuk, és nem szekvenciálisan, ahogy vártuk. Ezért a fenti példa egy végtelen ciklust eredményez. Erre a problémára a GNU *make* tartalmaz megoldást, azonban ezt egyedisége miatt kerüljük, hogy ne merüljenek fel portolási problémák.

A változók használata még további lehetőségeket is rejt, azonban ezek ritka használata miatt itt nem térünk ki rá. További információkat az *info make* paranccsal érhetünk el.

3.3.4 Direktívák

A direktívák nagyon hasonlóak a C nyelvben használt preprocesszor direktívákhoz.
Más file-ok futtatása:

```
include FILE_NEVEK...
```

Ahol a FILE_NEVEK egy kifejezés, amely tartalmazhat szóközzel elválasztott file neveket speciális karakterekkel és változókat.
A leggyakrabban használt elemek a feltételes fordítási direktívákhoz kötődnek:

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

A fenti példában, ha a *CC* változó értéke *gcc*, akkor a második sor hajtódik végre, egyébként a negyedik.
Direktívaként változót is megadhatunk, amely tartalmazhat új sor karaktert is:

```
define two-lines
     echo foo
     echo $(bar)
endef
```

További leírás a *make* parancs *info* oldalán található.

3.3.5 Implicit szabályok

Megvizsgálva a C nyelvű programok fordítását, a folyamat két lépésre oszlik.

1. A *.c* végű forrás állományok fordítása *.o* végű tárgykódú állományokká.
2. Az *.o* végű tárgykódú állományok fordítása.

Jó lenne, ha egy *xxx.o* állományra hivatkozva a *make* automatikusan utánanézne, hogy van-e egy *xxx.c* nevű file, majd azt úgy kezelné, mint egy lefuttatandó szabály tárgyát. Ennek a szabálynak a keretében a *gcc*-t felparaméterezve lefordítaná az *xxx.c* állományt *xxx.o* tárgykódú állománnyá. Többek között ezt a funkciót teszik lehetővé az implicit szabályok.

Ha a *make* program egy állományt talál, amely egy feldolgozandó szabály feltételei között szerepel, ugyanakkor nem szerepel egyetlen szabály tárgyaként sem, akkor megpróbál egy alapértelmezett szabályt találni rá, és ha ez sikerül, akkor végre is hajtja. Ezeket az alapértelmezett szabályokat **implicit szabályok**nak nevezzük.

Az implicit szabályok nagy része előre adott, de mi is írhatunk elő. Az implicit szabályok közül a legfontosabbak az ún. **ragozási** (suffix) **szabályok**. A *make* programnak megadható egy suffix lista, amelyet megpróbál eltávolítani az egyes cél nevekből. Az így kapott suffix alapján keres a suffix szabályban. Ilyen suffix lehet például egy file kiterjesztése.

Példaként nézzük egy ilyen szabály megadást:

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

Ez a példa a **.c** kiterjesztésű forrás állományokból a hozzá tartozó **.o** kiterjesztésű állományok előállítására fogalmaz meg egy általános szabályt. (Ez, a C forrásokból a tárgykódú állományok előállításának szabálya, a *make* programban alapértelmezésben adott, így általános esetekben nincs szükség a megadására.)

A suffix szabályok általános alakja:

```
FsCs:
	PARANCS
```

Az **Fs** a forrás suffix, a **Cs** a cél suffix. A forrásból a cél előállításához a *make* a *PARANCS* sorban megadott utasításokat hajtja végre.

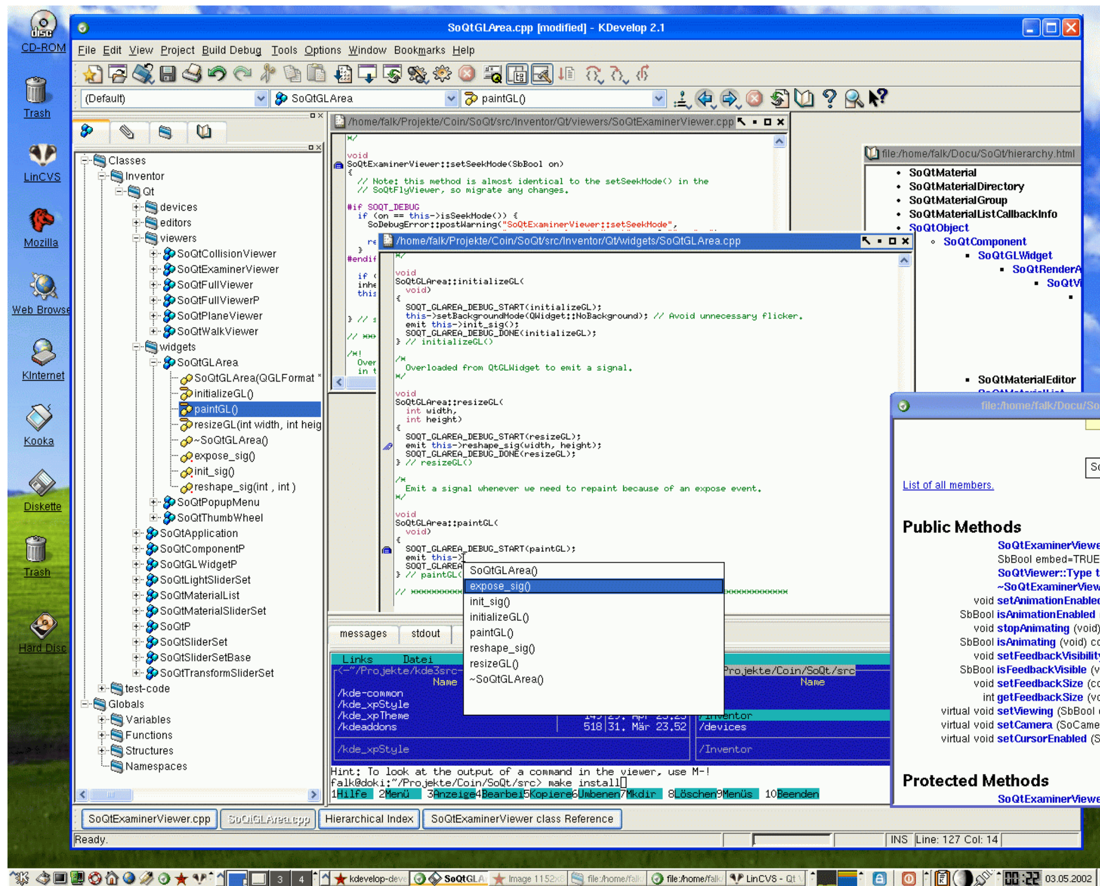
A korábbi példában meg a *make* programnak egy eddig nem látott szolgáltatásával is találkoztunk. A **$@** és a **$<** egy-egy **automatikus változó**, más néven **dinamikus makró**. Az egyértelmű, hogy egy ilyen általánosan megfogalmazott szabálynál szükségünk van egy mechanizmusra, amivel a feltétel és a cél állományok nevét elérhetjük a parancssorban. Erre szolgálnak az automatikus változók.

Az automatikus változók jelentése a következő:

Aut. változó	Jelentés
$@	A cél file neve.
$<	A függőségi lista első elemének neve.
$?	Az összes olyan feltétel file neve (sorban, *space*-el elválasztva), amely frissebb a cél állománynál.
$^	Az össze feltétel file neve *space*-el elválasztva.

Aut. változó	Jelentés
$+	Jelentése nagyrész egyezik az előzővel, azonban duplikált feltétel file neveket többször is tartalmazza úgy, ahogy az előkövetelmények közt szerepel.
$*	A cél file nevének suffix nélküli része.

3.4 KDevelop



Ábra 3-1 KDevelop screenshoot

Manapság a fejlesztők a munkájukhoz kényelmes környezetet szeretnének, amely átveszi tőlük a gyakran ismétlődő egyszerű műveletek egy részét. A Unix világ hagyományos, “fapados” fejlesztői eszközei nem elégítik ki ezt az igényt, ezért született meg 1998-ban a KDevelop projekt. Célja egy könnyen használható C/C++ IDE (Integrated Development Enviroment) létrehozása volt a Unix rendszerekhez. Magába integrálja a GNU általános fejlesztői eszközeit, mint például a g++ fordítót és a gdb debuggert. Ezek hagyományos megbízhatóságát ötvözi a kényelmes, grafikus kezelői felülettel, és néhány, a fejlesztőt segítő automatizmussal. Ezáltal a fejlesztő jobban koncentrálhatja a figyelmét a kódolásra a parancssoros programok kezelése helyett.

A KDevelop elsődleges célja, hogy felgyorsítsa a KDE programok fejlesztését, azonban a C/C++ fejlesztés bármely területén jól használható, ingyenes eszköz.

A KDevelop az alábbi szolgáltatásokkal rendelkezik:

- Minden fejlesztői eszközt integrál, amely a C++ fejlesztéshez szükséges: fordító, linker, *automake* és *autoconf*.
- *KAppWizard*, amely kész applikációkat generál.
- Osztály generátor, amely az új osztályok generálását és a projektbe való integrálását végzi.
- File menedzsment, amely a forrás, header és dokumentációs állományokat kezeli.
- SGML és HTML felhasználói dokumentum generátor.
- Automatikus HTML alapú API dokumentáció generátor, amely kereszthivatkozásokat hoz létre a felhasznált programkönyvtárak dokumentációjához.
- A többnyelvű kezelői felület támogatása a fejlesztett applikációkban.
- WYSIWYG (What you see is what you get) szerkesztői felület a dialógus ablakokhoz.
- CVS *front-end* a projektek verziómenedzsmentjéhez.
- Az applikációk debugolása az integrált debuggerrel.
- Ikon editor.
- Egyéb, a fejlesztésekhez szükséges programok hozzáadása a “Tools” menühöz.

4. Debug

A C az egyik legelterjedtebb nyelv, és egyértelműen a Linux rendszerek általános programozási nyelvének tekinthető, azonban van jó pár olyan jellemzője, amely használata során könnyen vezethet nehezen felderíthető *bug*-okhoz. Ilyen gyakori és nehezen felderíthető hibafajta a *memory leak*, amikor a lefoglalt memória felszabadítása elmarad, vagy a *buffer overrun*, amikor a program túlírja a lefoglalt területet. Ebben a fejezetben ezeknek a problémáknak a megoldására is látunk példákat.

4.1 gdb

A célja egy *debugger*-nek, mint például a *gdb*-nek, az hogy belelássunk egy program működésébe, hogy követhessük a futás során lezajló folyamatokat. Továbbá, hogy megtudjuk mi történt a program összeomlásakor, mi vezetett a hibához.

A *gdb* funkcióit, amelyekkel ezt lehetővé teszi, 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 nyelvek támogatása csak részleges.

4.1.1 Példa

A debuggoláshoz az adott programnak tartalmaznia kell a debug információkat. (A fordító –g kapcsolója.)
Először indítsuk el a programot a

```
gdb myprogram
```

utasítással, majd állítsuk be a program kimenetének a szélességét a

```
set width=70
```

segítségével.
Helyezzünk el egy **töréspont**ot (breakpoint) a *myfunction* nevű függvénynél:

```
break myfunction
```

Ezután a program futtatásához adjuk ki a

```
run
```

parancsot!

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

Parancs	**Magyarázat**
n	A következő (**n**ext line) sorra ugrás
s	Belépés (**s**tep into) egy függvénybe
p *variable*	Kiírja (**p**rint) a *variable* változó aktuális értékét
bt	A stack frame megjelenítése (**b**ack**t**race)
c	A program folytatása (**c**ontinue)
Ctrl+D	A program leállítása (az EOF jel)

Sikeres debuggolás után a *quit* paranccsal léphetünk ki.

4.1.2 A gdb indítása

A *gdb*-t többféle argumentummal és opcióval indíthatjuk, amellyel befolyásolhatjuk a debuggolási környezetet.
A leggyakrabban egy argumentummal használjuk, amely a vizsgálandó program neve:

```
gdb PROGRAM
```

Azonban a program mellett megadhatjuk második paraméterként a *core* állományt:

```
gdb PROGRAM CORE
```

A *core* file helyett azonban megadhatunk egy *processz 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.

További lehetőségként a *gdb*-t használhatjuk más gépeken futó alkalmazások távoli debuggolására is.

4.1.3 Breakpoint, watchpoint, catchpoint

A ***breakpoint*** megállítja a program futását, amikor az elér a program egy meghatározott pontjára. Minden *breakpoint*-hoz megadhatunk plusz feltételeket is. Beállításuk a **break** paranccsal és paramétereivel történik. Megadhatjuk a program egy sorát, függvény nevet, vagy az egzakt címet a programon belül:

```
break FUNCTION
```

A forrásállomány *FUNCTION* függvényének 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 LINENUM
```

Az aktuális forrásállomány *LINENUM* sorában helyezi el a töréspontot.

```
break FILENAME:LINENUM
```

A *FILENAME* forrás állomány *LINENUM* sorában helyezi el a töréspontot.

```
break *ADDRESS
```

Az *ADDRESS* 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 *breakpoint*, 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 EXPR
```

A *gdb* megállítja a programot, amikor az *EXPR* kifejezés értékét a program módosítja, írja.

```
rwatch EXPR
```

A *watchpoint* akkor állítja meg a futást, amikor az *EXPR* kifejezést a program olvassa.

```
awatch EXPR
```

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

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

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

```
catch EVENT
```

Ahol az *EVENT* kifejezés az alábbiak közül valamelyik:

EVENT	Jelentés

EVENT	Jelentés
throw	Egy C++ exception keletkezésekor.
catch	Egy C++ exception lekezelésekor.
exec	Az "exec" függvény meghívásakor.
fork	Az "fork" függvény meghívásakor.
vfork	Az "vfork" függvény meghívásakor.
load [LIBNAME]	A *LIBNAME* könyvtár betöltésekor.
unload [LIBNAME]	A *LIBNAME* könyvtár eltávolításakor.

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

4.1.4 xxgdb

A *gdb* nagyon hasznos segédprogram, azonban parancssoros felületét nem mindenki kedveli. Ezért kezelésének megkönnyítéséhez készült több grafikus *front-end* is. Ezek egyike, talán a legegyszerűbb, az ***xxgdb***. Az alábbi ábrán láthatunk egy képet a felületéről.

```
/home/doppler/elf/Xt/apps/xmotd/xmotd.c                146

    exit(0);
  }

  main(argc, argv)
  int argc;
  char **argv;
  {
    Pixmap icon_pixmap;
    Widget form, paned, save, abort, text, logo, mlabel, hline;
    struct stat motdstat;

    if(argc<2) exit(0);

Ready for execution

run     cont     next       step    stop at   stop in   delete
where   up       down       print   print *   func      file
status  display  undispla   dump    search    quit

(xdbx) alias c cont
(xdbx) alias d display
(xdbx) alias del delete
(xdbx) alias h help
(xdbx) alias m make
(xdbx) alias n next
(xdbx) alias p print
(xdbx) alias q quit
(xdbx) alias s step
(xdbx) alias sa stop at
(xdbx) alias ud undisplay
(xdbx) stop at 146
(2) stop at "/home/doppler/elf/Xt/apps/xmotd/xmotd.c":146
```

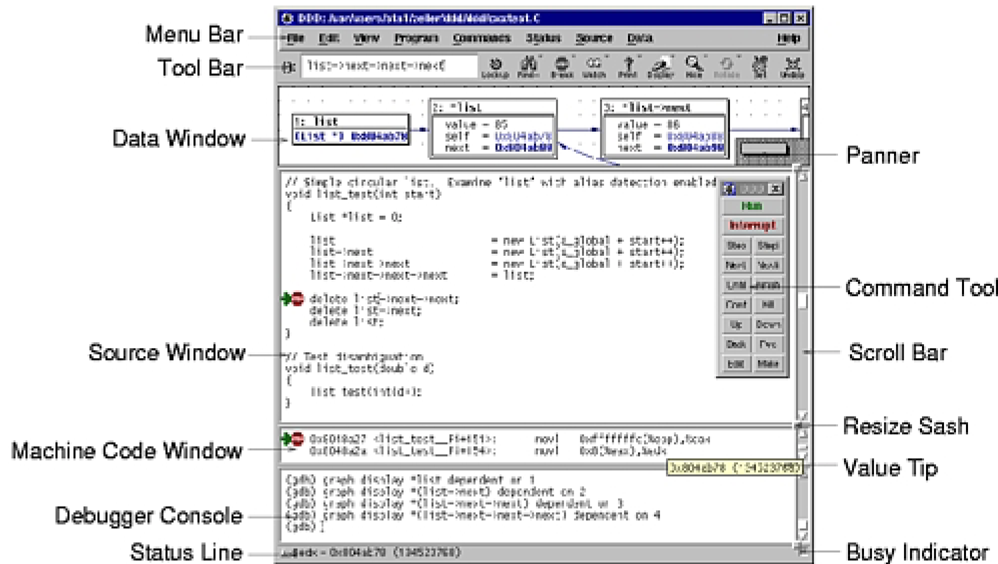
Ábra 4-1 xxgdb

Mint látható a felületén lényegében a *gdb* parancsokat gombokkal érhetjük el. A másik plusz, amit a *gdb*-vel szemben nyújt, hogy a forrás állományt párhuzamosan figyelemmel kísérhetjük.

4.1.5 DDD

A ***DDD*** egy GNU projekt, melynek célja, hogy grafikus *fron-end*ként szolgáljon a parancssoros *debugger*ek számára, mint például a GDB, DBX, WDB, Ladebug, JDB, XDB, a Perl debugger, vagy a Python debugger. A szokásos *front-end* funkciók mellett, mint például a forrás állományok megtekintése, több jelentős szolgáltatással is rendelkezik. Ilyenek az interaktív grafikus adatábrázolások, ahol az adatstruktúrák gráfként vannak megjelenítve.

A *DDD* felületét mutatja a következő kép:



Ábra 4-2 A DDD felülete

A *Data Window* az aktuális megfigyelt program adatait mutatja.
A *Source Window* a program forráskódját mutatja.
A *Debugger Console* fogadja a *debugger* parancsokat és mutatja az üzeneteit.
A *Command Tool* a leggyakrabban használt parancsok gombjait tartalmazza.
A *Machine Code Window* az aktuális gépi kódú programot mutatja.
Az *Execution Window* a vizsgált program bemeneteit és kimeneteit tartalmazza.

4.1.6 KDevelop internal debugger

A KDevelop is rendelkezik egy beépített *gdb* grafikus *front-end*del. Ez funkcióiban valamelyest elmarad a *DDD* mögött, azonban egy jól használható barátságos felületet nyújt. Ezáltal lehetővé teszi a fejlesztett projektünk debuggolását a fejlesztői környezeten belül. Használata során lehetőségünk nyílik a processz adatainak vizsgálatára, *breakpoint*ok használatára, a *framestack* megfigyelésére.

4.2 Memóriakezelési hibák

A fejezet elején már volt arról szó, hogy a C és C++ nyelv használata során számos memóriakezelési hibát ejthetünk, amelyeknek felderítése nehéz feladat. Ennek megoldására számos eszköz született. Most ezekből ismerhetünk meg néhányat.

A vizsgálataink előtt állítsunk elő egy hibás programot, amely az “állatorvosi ló” szerepét fogja betölteni. Íme a memóriakezelési hibáktól hemzsegő kódunk:

```
 1 /*
 2  * Buggy code
 3  */
 4
 5 #include <stdlib.h>
 6 #include <stdio.h>
 7 #include <string.h>
 8
 9 char global[5];
10
11 int main(void)
12 {
13   char* dyn;
14   char local[5];
15
16   // 1. tuliras (kicsit)
17   dyn=malloc(5);
18   strcpy(dyn, "12345");
19   printf("1: %s\n", dyn);
20   free(dyn);
21
22   // 2. felszabaditott terulet hasznalata
23   strcpy(dyn, "12345");
24   printf("2: %s\n", dyn);
25
26   // 3. tuliras (nagyon)
27   dyn=malloc(5);
28   strcpy(dyn, "12345678");
29   printf("3: %s\n", dyn);
30
31   // 4. ele iras
32   *(dyn-1)='\0';
33   printf("4: %s\n", dyn);
34
35   // memory leak !!!
36
37   // 5. lokalis valtozo tuliras
38   strcpy(local, "12345");
39   printf("5: %s\n", local);
40
41   // 6. lokalis valtozo eleiras
42   local[-1]='\0';
43   printf("6: %s\n", local);
44
45   // 7. globalis valtozo tuliras
46   strcpy(global, "12345");
47   printf("7: %s\n", global);
48
49   // 8. globalis valtozo eleiras
50   global[-1]='\0';
51   printf("8: %s\n", global);
52
53   return 0;
54 }
```

Ez a kód háromféle memóriát allokál, és ezekkel követ el hibákat. Az első típus a dinamikusan, *malloc*-al allokált memória, a második lokális változó, amely a program stackjében helyezkedik el, a harmadik pedig globális változó, amely egy külön részen helyezkedik el. Mindhárom esetben túlírjuk a lefoglalt tartományt, illetve egy-egy byte-ot elé írunk. Ezek mellett a kód tartalmaz egy hozzáférést egy már felszabadított memória részhez és egy *memory leak*-et is.

Habár ezt a programot bőven elláttuk hibákkal, mégis általában probléma nélkül lefut. De ez nem jelenti azt, hogy ezek a hibák lényegtelenek. A túlírások idővel a program váratlan, és elsőre megmagyarázhatatlannak tűnő összeomlásához vezethetnek. A *memory leak*-ek pedig idővel feleszik a számítógép memóriáját.

Elsőre a kódot lefordítva, és lefuttatva az alábbit látjuk:

```
$ gcc -ggdb -Wall -o buggy buggy.c
$ ./buggy
1: 12345
2: 12345
3: 12345678
4: 12345678
5: 12345
6: 12345
7: 12345
8: 12345
```

A következőkben megismerhetünk néhány eszközt, amely ennél többet mutat számunkra. Ezek egy része megtalálható az alábbi címen sok más eszköz társaságában:
http://www.ibiblio.org/pub/Linux/devel/lang/c/

4.2.1 Electric Fence

Az első eszközt, amit megismerünk, ***Electric Fence***-nek hívják. Bár a *memory leak*-ek vizsgálatára ez az eszköz nem használható, ugyanakkor a programozókat segítheti a buffer túlírások és a már felszabadított memória használatának felderítésében. Más *malloc* debuggerrel ellentétben az *Electric Fence* a hibás olvasásokat is detektálja.

Az *Electric Fence* a számítógép virtuális memória kezelő hardware-t használja a hibák felderítésére azáltal, hogy letiltott memória lapokat helyez közvetlenül a lefoglalt memória területek után (illetve elé a felhasználó beállításától függően). Amikor a program ezeket a tiltott területeket írja, vagy olvassa, a hardware egy *segmentation fault* hibát vált ki, amely a program leállásához vezet. Ilyenkor a debugerrel megvizsgálhatjuk a hiba okát. Hasonló képen a felszabadított memória területeket is letiltja, így azok utólagos használata szintén hibához és leálláshoz vezet.

4.2.1.1 Az Electric Fence használata

Az *Electric Fence* a C library normál *malloc* függvényét cseréli le a saját speciális allokációs rutinjára. Így egyetlen feladatunk, hogy a *libefence.a* vagy *libefence.so* könyvtár állomány hozzálinkeljük a programunkhoz. Ezt kétféle képen tehetjük.

Az *–lefence* argumentummal fordításkor hozzálinkelhetjük a programunkhoz a könyvtár állományt. Ez a példa programunknál:

```
gcc -ggdb -Wall -o buggy buggy.c -lefence
```

A másik lehetőség dinamikus linkelés esetén az *LD_PRELOAD* környezeti változó használata. Ezzel megadhatjuk, hogy a program futása előtt a könyvtár állomány betöltődjön.

```
export LD_PRELOAD=libefence.so.0.0
```

vagy

```
LD_PRELOAD=libefence.so.0.0 ./buggy
```

(Az utóbbi a javasolt.)

Ha ezek után lefuttatjuk a programunkat, akkor már nem garázdálkodhat szabadon. Az *Electric Fence* védelmi algoritmusai detektálják a hibát.

```
$ ./buggy

  Electric Fence 2.2.0 Copyright (C) 1987-1999 Bruce Perens
1: 12345
Segmentation fault
```

Mint látható az *Electric Fence* nem mondja meg hol a hiba, viszont a hibát sokkal érzékelhetőbbé teszi, és ez által egy debuggerrel, mint például a *gdb*, megvizsgálhatjuk. (Ehhez természetesen a programnak rendelkeznie kell a debug információkkal is.)

Esetünkben ez az alábbiak szerint alakul:

```
$ gdb ./buggy
GNU gdb Red Hat Linux (5.2-2)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License,
and you are welcome to change it and/or distribute copies of it
under certain conditions.
This GDB was configured as "i386-redhat-linux"...
(gdb)
```

Ha dinamikusan szeretnénk hozzáadni a könyvtár állományt, akkor az alábbiakra is szükségünk van:

```
(gdb) set env LD_PRELOAD=libefence.so.0.0
```

Ezek után megkezdhetjük a program vizsgálatát.

```
(gdb) r
Starting program: /home/tomcat/buggy
[New Thread 1024 (LWP 26727)]

  Electric Fence 2.2.0 Copyright (C) 1987-1999 Bruce Perens
1: 12345

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 1024 (LWP 26727)]
0x420807a6 in strcpy () from /lib/i686/libc.so.6
```

```
(gdb) where
#0  0x420807a6 in strcpy () from /lib/i686/libc.so.6
#1  0x080484fc in main () at buggy.c:23
#2  0x42017589 in __libc_start_main () from /lib/i686/libc.so.6
(gdb)
```

Mint látható a 23. sorban máris elkapott a rendszer egy hibát.

```
23   strcpy(dyn, "12345");
```

Ez az a hely, ahol a felszabadított memória területre próbáltunk írni.

Ha ezt a sort kikommentezzük, akkor a 24. sorban, a felszabadított terület olvasásánál érzékeli a rendszer a hibát.

Így tovább haladva a következő hiba a 28. sorban található, ahol jelentősen túlírjuk a lefoglalt memóriát.

Ezt is eltávolítva további hibát nem érzékel, pedig mint látható maradt még bőven.

4.2.1.2 Memory Alignment kapcsoló

Mint észrevehettük az *Electric Fence* átszaladt az első túlíráson, ahol a tartományt csak egy byte-al írtuk túl. A probléma forrása a memóriaillesztés. A modern processzorok a memóriát több byte-os részekre osztják. Ez a 32 bites processzorok esetén 4 byte. A *malloc* általános implementációja is így kerekítve foglalja le a memóriát. Alapértelmezett beállítások mellett az *Electric Fence* is ezt a módszert alkalmazza, ezért nem vette észre az 1 byte-os differenciát.

Ahhoz, hogy ezt a hibát megtaláljuk, az *Electric Fence*-nek meg kell adnunk, hogy 1 byte-os illesztést használjon. Ezt az *EF_ALIGNMENT* környezeti változó 1-re állításával tehetjük meg:

```
(gdb) set env EF_ALIGNMENT=1
(gdb) r
Starting program: /home/tomcat/buggy
[New Thread 1024 (LWP 26810)]

  Electric Fence 2.2.0 Copyright (C) 1987-1999 Bruce Perens

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 1024 (LWP 26810)]
0x420807a6 in strcpy () from /lib/i686/libc.so.6
(gdb) where
#0  0x420807a6 in strcpy () from /lib/i686/libc.so.6
#1  0x080485f8 in main () at buggy.c:18
#2  0x42017589 in __libc_start_main () from /lib/i686/libc.so.6
(gdb)
```

Így már megtalálta a 18. sorban található kis túlírásunkat is.

4.2.1.3 Az eléírás

Mint látható idáig nem sikerült detektálnunk a *buffer underrun* esetét, vagyis amikor a lefoglalt terület elé írtunk egy byte-ot. Az *EF_PROTECT_BELOW* változó 1-re

állításával azt kérhetjük az *Electric Fence*-től, hogy a letiltott memória lapokat a lefoglalt terület elé helyezze, így ezt a hiba típust szűrje ki. Természetesen ilyenkor a *buffer overrun*-t vagyis a túlírást nem tudja érzékelni.

```
(gdb) set env EF_PROTECT_BELOW=1
(gdb) r
Starting program: /home/tomcat/buggy
[New Thread 1024 (LWP 26868)]

  Electric Fence 2.2.0 Copyright (C) 1987-1999 Bruce Perens
1: 12345
3: 12345678

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 1024 (LWP 26868)]
0x08048658 in main () at buggy.c:32
32        *(dyn-1)='\0';
(gdb)
```

4.2.1.4 További lehetőségek

Még az alábbi kapcsolókkal találkozhatunk az eszköz használata során:

Változó	Jelentés
EF_PROTECT_FREE	Ha értéke 1, akkor a felszabadított memória területek nem kerülnek újra kiosztásra, hanem letiltja őket a rendszer.
EF_ALLOW_MALLOC_0	Ha értéke 1, akkor engedélyezi a 0 méretű területek allokálását.
EF_FILL	Amikor 0 és 255 között van az értéke, akkor a lefoglalt terület minden byte-ját feltölti a rendszer ezzel az értékkel. Ezáltal segíti az inicializációs problémák kiszűrését.

4.2.1.5 Erőforrás igények

Habár az *Electric Fence* hasznos eszköz, könnyen kezelhető és gyors (mivel minden hozzáférés ellenőrzést a hardware végez), ennek megvan az ára. A legtöbb processzor a hozzáférés vezérlést csak egy-egy lapra engedi állítani. Egy lap mérete például egy Intel x86-os processzornál 4KB. Mivel az *Electric Fence* allokáló rutinja két különböző területet foglal le minden alkalommal (egyet amihez hozzáférhet a program, és egyet, aminél letiltja), ezért minden egyes allokálás lefoglal egy 4KB-os lapot. Ha a tesztelt kódban sok kisméretű memória allokáció van, akkor az *Electric Fence* használatával több nagyságrenddel megnőhet a memória felhasználás. Természetesen a helyzet tovább romlik, ha a felszabadított területeket védjük, mert ilyenkor valójában nem szabadul fel memória.

Ezért ha azt tapasztaljuk, hogy használatakor a rendszernek nagyon megfogyatkozik a szabad memória kapacitása, akkor ajánlatos a swap területet megnövelni.

4.2.2 NJAMD (Not Just Another Malloc Debugger)

Az NJAMD egy teljes funkcionalitású *malloc debugger*. Véd a *buffer overflow, underflow* és a felszabadított memória írásától-olvasásától, mint az *Electric Fence*, azonban alkalmas a *memory leak* felfedezésére is.

A könyvtár állomány mellett rendelkezik egy, egyenlőre még fejlesztésre szoruló, *front-end*el is, de használható az *Electric Fence*-hez hasonlóan a *gdb*-ből, vagy más debuggerből.

4.2.2.1 Használata

Hasonlóan használhatjuk, mint az *Electric Fence*-t. Hozzálinkelhetjük az *–lnjamd* kapcsolóval a megfelelő könyvtárat a programunkhoz:

```
gcc -ggdb -Wall -o buggy buggy.c -lnjamd
```

Vagy használhatjuk az *LD_PRELOAD* környezeti változót:

```
LD_PRELOAD=libnjamd.so
```

Ezek mellet egy további lehetőség az *njamd* segédprogram - *front-end* használata. Ez tulajdonképpen a *gdb*-t hívja meg, és az *LD_PRELOAD* környezeti változóval használja a könyvtár állományt:

```
$ njamd -e ./buggy
NJAMD - Not Just Another Malloc Debugger
------------------------------------------------------------------
njamd> start
njamd>



------------------------------------------------------------------
welcome to change it and/or distribute copies of it under certain
conditions.
This GDB was configured as "i386-redhat-linux"...
(gdb) set env LD_PRELOAD=libnjamd.so
(gdb) set env NJAMD_PROT=over
(gdb) set env NJAMD_ALIGN=1
(gdb) set env NJAMD_DUMP_LEAKS_ON_EXIT=1
(gdb) set env NJAMD_ALLOW_READ=1
(gdb) set env NJAMD_FE_PORT=33452
(gdb) run
Starting program: /home/tomcat/buggy

Program received signal SIGSEGV, Segmentation fault.
0x420807a6 in strcpy () from /lib/i686/libc.so.6
(gdb)
```

A programunkkal tesztelve az első futtatásra az alábbi eredményt kapjuk:

```
(gdb) set env LD_PRELOAD=libnjamd.so
(gdb) r
Starting program: /home/tomcat/buggy
```

```
Program received signal SIGSEGV, Segmentation fault.
0x420807a6 in strcpy () from /lib/i686/libc.so.6
(gdb) where
#0  0x420807a6 in strcpy () from /lib/i686/libc.so.6
#1  0x080484c8 in main () at buggy.c:18
#2  0x42017589 in __libc_start_main () from /lib/i686/libc.so.6
(gdb)
```

Ebből látható, hogy a kis túlírást elsőre megtalálta a 18. sorban. Tehát az alapértelmezett memória illesztési beállítása olyan, hogy az 1 byte-os hibákat is érzékeli.

Tovább tesztelve megtalálja a 23. sorban a felszabadított terület írását, a 24-ben az olvasását, és a 28. sorban a nagy túlírást.

Az alulírást is detektálhatjuk a 32. sorban, ha az *NJAMD_PROT* környezeti változót *strict* értékre állítjuk.

```
(gdb) set env NJAMD_PROT=strict
(gdb) r
Starting program: /home/tomcat/buggy
1:
3:

Program received signal SIGSEGV, Segmentation fault.
0x08048622 in main () at buggy.c:32
32        *(dyn-1)='\0';
(gdb)
```

Azonban az *Electric Fence*-el ellentétben az *NJAMD* ilyenkor is bizonyos mértékig érzékeli a túlírást, amikor felszabadítjuk a lefoglalt területet.

```
(gdb) set env NJAMD_PROT=strict
(gdb) r
Starting program: /home/tomcat/buggy
1: 12345
NJAMD/free: heap corruption. Try using the overflow option to
pinpoint source of error

...

Program received signal SIGSEGV, Segmentation fault.
0x42029331 in kill () from /lib/i686/libc.so.6
(gdb) where
#0  0x42029331 in kill () from /lib/i686/libc.so.6
#1  0x4202911a in raise () from /lib/i686/libc.so.6
#2  0x40029b12 in nj_free_init () from /usr/lib/libnjamd.so.0
#3  0x40029ff1 in __nj_sunderflow_free () from
/usr/lib/libnjamd.so.0
#4  0x4002ce2c in free () from /usr/lib/libnjamd.so.0
#5  0x08048609 in main () at buggy.c:20
#6  0x42017589 in __libc_start_main () from /lib/i686/libc.so.6
(gdb)
```

4.2.2.2 Memory leak detektálás

Az *NJAMD* képes a *memory leak* problémák detektálására is. Ehhez az *NJAMD_DUMP_LEAKS_ON_EXIT* környezeti változó használatára van szükség. Ebben a változóban megadhatjuk a visszakeresési szintet is. Az alapértelmezett maximum 3.

Egy példa a használatára:

```
$ LD_PRELOAD=libnjamd.so NJAMD_PROT=none
NJAMD_DUMP_LEAKS_ON_EXIT=3 ./buggy
1:
3:
4:
5: 12345
6: 12345
7: 12345
8: 12345

0x4213b000-0x4213d000: Aligned len 5
   Allocation callstack:
        called from ./buggy[0x8048603]
        called from ./buggy(__libc_start_main+0x95)[0x42017589]
        called from ./buggy(free+0x41)[0x80484e1]
   Not Freed

NJAMD totals:

        Allocation totals:              2 total, 1 leaked
        Leaked User Memory:             5 bytes
        Peak User Memory:               5 bytes
        NJAMD Overhead at peak:         3.995 kB
        Peak NJAMD Overhead:            3.995 kB
        Average NJAMD Overhead:         3.995 kB per alloc
        Address space used:             16.000 kB
        NJAMD Overhead at exit:         3.995 kB
```

Mint látható megtalálta az 1 darab 5 byte-os memóriaszivárgásunkat. Emellett további statisztikai információkat is kapunk.

4.2.2.3 Az NJAMD kapcsolói

Az *NJAMD* az alábbi környezeti változókat használja, mint a működését befolyásoló kapcsolókat:

Változó	**Érték**	**Jelentés**
NJAMD_PROT	overflow	Alapértelmezett érték. A túlírások ellen véd.
	strict	Az összes alulírás ellen véd.
	underflow	A nagyobb méretű alulírások ellen véd.
	none	Csak *memory leak* ellenőrzés van.

<table>
<tr><th>Változó</th><th>Érték</th><th>Jelentés</th></tr>
<tr><td rowspan="4">NJAMD_CHK_FREE</td><td>segv</td><td>Az alapértelmezett metódus a felszabadított memória védelmére. Segmentation fault hibajelzés nélkül.</td></tr>
<tr><td>error</td><td>Hibajelzést is ad.</td></tr>
<tr><td>none</td><td>Kikapcsolja a felszabadított memória védelmét.</td></tr>
<tr><td>nofree</td><td>Kikapcsolja a memória felszabadítást.</td></tr>
<tr><td>NJAMD_NO_FREE_INFO</td><td colspan="2">Ha 1, akkor kikapcsolja a felszabadított területekről az információtárolást.</td></tr>
<tr><td>NJAMD_ALIGN</td><td colspan="2">Memória illesztés beállítása.</td></tr>
<tr><td>NJAMD_DUMP_LEAKS_ON_EXIT</td><td colspan="2">Memory leak információk megjelenítése. Értéke a visszakeresés mértéke. Alapértelmezett maximum: 3</td></tr>
<tr><td>NJAMD_DUMP_STATS_ON_EXIT</td><td colspan="2">Ha 1, akkor a program futásának végén statisztikai információkat közöl.</td></tr>
<tr><td rowspan="2">NJAMD_DUMP_CORE</td><td>hard</td><td>Ha korrekt és teljes core dump állományt szeretnénk.</td></tr>
<tr><td>soft</td><td>Ha a core file mellett statisztikai információkat is szeretnénk.</td></tr>
<tr><td>NJAMD_PERSISTENT_HEAP</td><td colspan="2">Ha 1, akkor lementi a heap tartalmát egy állományba.</td></tr>
<tr><td>NJAMD_ALLOW_MALLOC_0</td><td colspan="2" rowspan="2">Ha 1, akkor engedélyezi a 0 méretű allokálást és felszabadítást.</td></tr>
<tr><td>NJAMD_ALLOW_FREE_0</td></tr>
</table>

4.2.2.4 Összegzés

Mint láthattuk az *NJAMD* rendelkezik mindazokkal a funkciókkal, mint az *Electric Fence*, és azon felül még további szolgáltatásokkal is. Ezért használata célszerűbb. Azonban a *memory leak* információk nehezen kezelhetőek, és ez sem jelent megoldást a lokális és globális változók hibáinak detektálására.

A vizsgált jellemzők állításával hangolhatjuk a rendszer sebességét, teljesítményét.

4.2.3 mpr

Az *mpr* egy másik *malloc* eszköz a *memory leak*-ek megtalálására. Funkcionalitását tekintve egy memória allokáció *profiler* a C/C++ programok számára. Egyszerű, *brute force* megoldást alkalmaz a memória szivárgások felderítésére. A futás közben logolja az összes *malloc* és *free* hívást, hogy megtalálja a nem felszabadított részeket.

Alapja egy könyvtár állomány, amelyet hasonlóan a korábbiakhoz hozzálinkelhetünk a programunkhoz az *–lmpr* kapcsolóval

```
gcc -ggdb -Wall -o buggy buggy.c -lmpr
```

vagy az *LD_PRELOAD* változóval:

```
LD_PRELOAD=libmpr.so
```

Azonban ezzel szemben a használata az ***mpr*** segédprogrammal javasolt:

```
mpr ./buggy
```

Ilyenkor a futás közbeni eseményekről egy log állományt készít. Ennek neve *log.<pid>.gz*. Azonban az *MPRFI* környezeti változóval megváltoztathatjuk a log tárolásának módját.

Az ***mprmap*** program segítségével megjeleníthetjük az állomány tartalmát olvasható formában. Ilyenkor megtekinthetjük az össze memóriafoglalást és felszabadítást.

```
$ mprmap -l ./buggy log.27391.gz
m:__register_frame():__libc_global_ctors():init():_dl_init_interna
l():24:134518768
m:main(buggy.c,17):__libc_start_main():5:134518800
f:main(buggy.c,20):__libc_start_main():134518800
m:main(buggy.c,27):__libc_start_main():5:134518800
f:__deregister_frame():_fini():_dl_fini():exit():__libc_start_main
():134518768
```

Az ***mprsize*** statisztikát jelenít meg a memóriafoglalásról:

```
$ mprsize log.27391.gz
5               2               10              29.41%
24              1               24              70.59%
```

Az ***mprleak*** kiszűri a logból a memóriaszivárgásokat. Ezt az *mprmap* programmal olvasható formába alakíthatjuk:

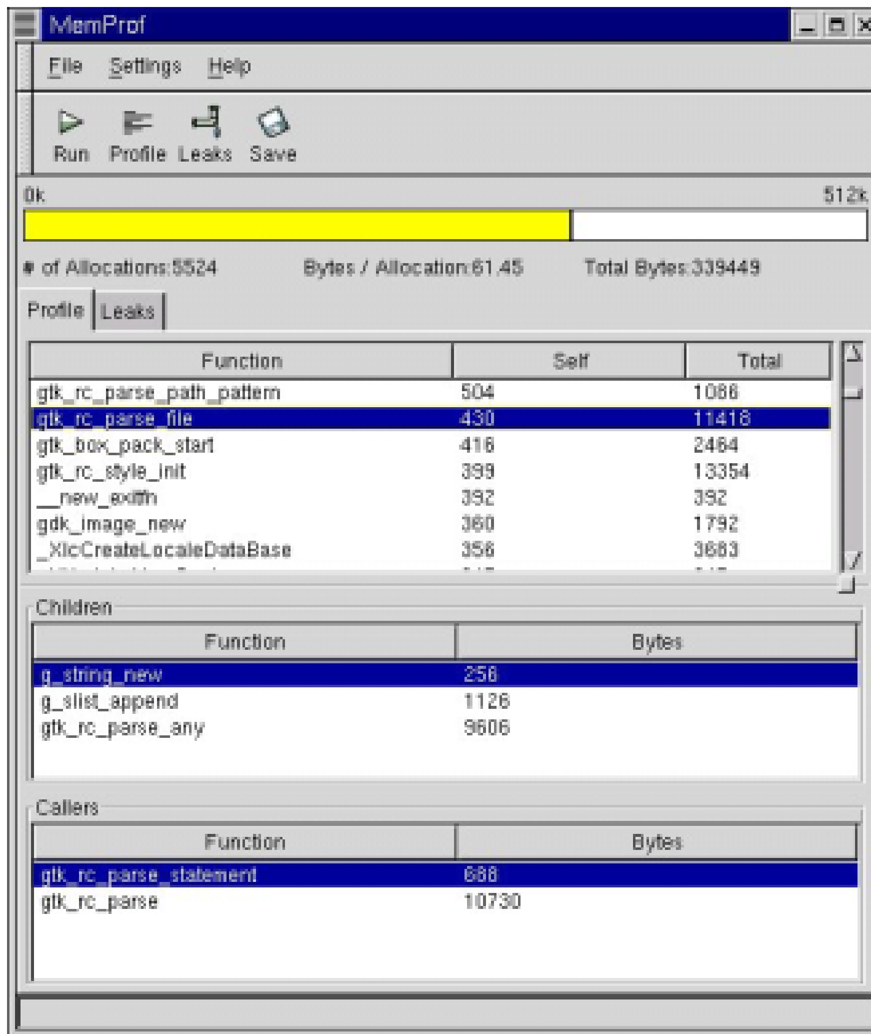
```
$ mprleak log.27391.gz | mprmap -l ./buggy
m:main(buggy.c,27):__libc_start_main():5:134518800
```

Mint láthatjuk meg is találta a programunk hibáját.

4.2.4 MemProf

A *MemProf* szintén egy memória használatot monitorozó eszköz, azonban az *mpr*-el szemben több előnyt is tartalmaz. Egyrészt a grafikus kezelői felületén a program futása közben követhetővé teszi az egyes függvények memória használatát. Másrészt folyamatosan ellenőrzi a memóriát, hogy talál-e olyan blokkokat, amelyekre már nincs hivatkozás. Ez az *memory leak*-ek egy tipikus esete, amikor elfelejtjük felszabadítani a memóriát, és a hivatkozást is töröljük, módisítjuk. Ezzel a programmal ezek a problémák menet közben is megfigyelhetőek, így egy elég komoly segítséget jelent a hibakeresésben.

Kezelői felülete a következő ábrán látható:



Ábra 4-3 MemProf

4.3 Rendszerhívások monitorozása: strace

Az *strace* egy hasznos diagnosztikai, debuggolási eszköz. Általános esetben az *strace* lefuttatja a paraméterként megadott programot, és monitorozza a processz rendszerhívásait és a szignálokat, amelyeket kap. Az összes rendszerhívást a paramétereivel együtt a standard hibakimenetre, vagy a megadott kimeneti állományba írja.

Használható a rendszer működésének vizsgálatához, megismeréséhez is.

4.4 További hasznos segédeszközök

A **lint** segédprogram a hibakeresésben lehet segítségünkre. Használata:

```
lint myprogram.c
```

A lint szolgáltatásai között megemlítjük a típusellenőrzést, paraméterátadást, lehetséges memóriahibák felderítését.

Az **indent** segédprogram olvashatóvá teszi a C forráskódot szóköz, tabulátor és hasonló jellegű karakterek beiktatásával.
Használata:

```
indent myfile.c
```

Bináris file-ok analízisénél jól jöhet, ha hexadecimálisan is meg tudunk jeleníteni egy file-t. Ezt a

```
hexdump filename
```

segédprogrammal tehetjük meg.