Memóriahiba-keresés Linux alatt

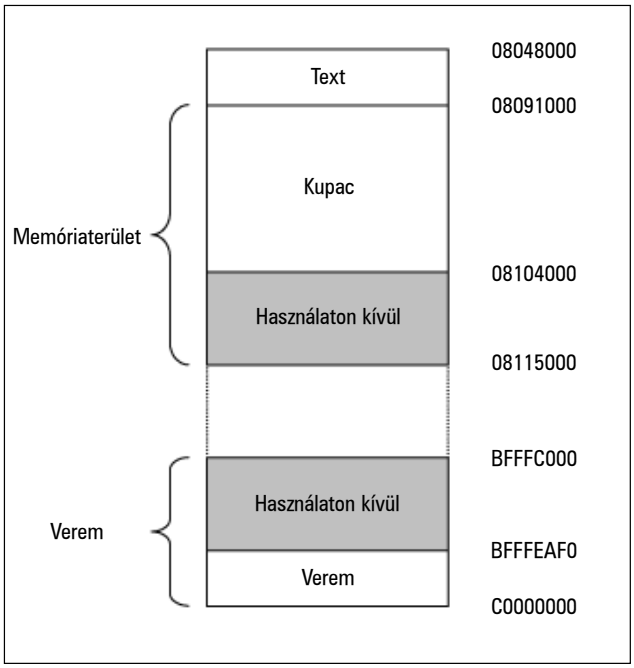
Petertől megtudhatjuk, miként kerülhetik el a programozók a kellemetlen memóriahibákat.

Minden program használ memóriát. Még azok is, amelyek semmit sem csinálnak. A memória helytelen használata pedig meglehetősen gyakori oka az olyan végzetes programhibáknak, mint például a programleállás vagy a váratlan viselkedés.
A memória az adatok tárolására szolgáló eszköz. A program memóriája általában egy bizonyos mennyiségű fizikai memóriához rendelhető, ha azonban éppen nincs használatban, lehet valamilyen háttértárolón is, például a merevlemezen.
A felhasználók szempontjából a memóriát két eszköz használja: maga a rendszermag és az éppen futó program, olyan memóriaszolgáltatásokon keresztül, mint a `malloc()` .

A magmemória

Az adott programhoz vagy annak példányaihoz (az operációs rendszer egy időben több példányt is képes futtatni) szükséges összes memóriát az operációs rendszer magja kezeli. Amikor a felhasználó végrehajt egy programot, a mag bizonyos memóriaterületet foglal le a program számára. Ezután a program a memóriarészt a következő területekre felosztva kezeli:

- Szöveg (Text) – Ez az a terület, ahol a program csak olvasható részei tárolódnak, tulajdonképpen ez a program utasításkódja. Ugyanazon program példányai ezt a memóriaterületet megosztják egymás között.
- Állandó adat (Static Data) – Az előre ismert memória számára lefoglalt terület. Általában a globális változók és statikus C++ osztályelemek kerülnek ide. Az operációs rendszer minden egyes programpéldányhoz külön lefoglal egy ilyen területet.
- Memóriaaréna (Memory Arena, avagy break space) – Ez az a terület, ahol a dinamikus futásidejű memória tárolódik. A memóriaaréna két részből áll: a kupacból és a felhasználatlan memóriából. A felhasználó által lefoglalt memória a kupacban helyezkedik el. A kupac az alacsonyabb memóriacímektől a magasabbak felé növekszik.
- Verem (Stack) – Valahányszor a program függvényhívásokat hajt végre, a pillanatnyi függvényállapotot a verembe kell menteni. A verem lefelé növekszik, a magasabb memóriacímektől az alacsonyabbak felé. Minden egyes programpéldány számára egyedi memóriaaréna és verem foglalódik.



1. ábra A programpéldányhoz rendelt memória

Felhasználói memória

A felhasználó által foglalható memória a memóriaaréna kupacrészében található. A memóriaaréna kezelését a `malloc()`, `realloc()`, `free()` és `calloc()` eljárások végzik. A felsorolt eljárások a libc részét képezik. Lehetséges azonban, hogy ezeket a függvényeket valamilyen más megvalósítással helyettesítsük, amelyek adott feladat esetén esetleg hatékonyabb megoldást biztosítanak. Az *1. ábrán* ilyen memóriafüggvény-választási lehetőségeket soroltunk fel.
A Linux-rendszereken a programok memóriaaréna-területe előre meghatározott egységekben növekszik, általában egy memórialap-egységenként, avagy a laphatárhoz igazodva. Ha a kupacnak több memóriát igényel, mint amennyi a memóriaarénában rendelkezésre állna, a memóriafüggvények meghívják a `brk()` rendszerhívást, amely további memóriaterületet igényel a magtól. Az igényelt terület méretét a `sbrk()` hívással lehet beállítani.
Bármely folyamat pillanatnyi memóriaaréna- és veremterületét megtekinthetjük, ha megvizsgáljuk az adott folyamathoz tartozó /proc/<pid>/maps fájl tartalmát, ahol a `pid (process id)` a folyamat azonosítója (lásd az *1. listát*).

Szerkezet

Valahányszor a `malloc()` függvénnyel új memóriaterületet foglalunk le, mindig egy kevéssel több memóriát kapunk, mint amennyit kértünk. A memóriaeljárások ezt a további területet használják fel karbantartásra. Ha a felhasználói kérésre lefoglalt memória valós méretére vagyunk kíváncsiak, használjuk a `malloc_usable_space()` függvényt. A valós memóriaszelet ennél általában nyolc bájttal nagyobb.
A memóriaszelet szerkezete a szelet végén található és a szelet korábbi méretét tartalmazza (*2. ábra*). A méretérték egy további bitet is magában foglal: ez mutatja meg, hogy a memóriakezelő rendszer fenntart-e egy másik memóriaszeletet közvetlenül a pillanatnyi előtt.
A GNU libc memóriaeljárásai rekeszeket használnak a hasonló méretű memóriaszeletek tárolására, ilyen módon segítve elő a növekvő teljesítményt, és egyúttal megakadályozzák a széttöredezett memóriaterületek kialakulását, ahol a memóriaaréna belsejében kihasználatlan memórialyukak maradnak. Ezek a memóriaeljárások természetesen szálbiztosak (threadsafe) is. Jóllehet ezek a függvények gyorsak és megbízhatók, van néhány terület, ahol fejleszthetők lennének, például a sebesség és a memórialefedettség terén.

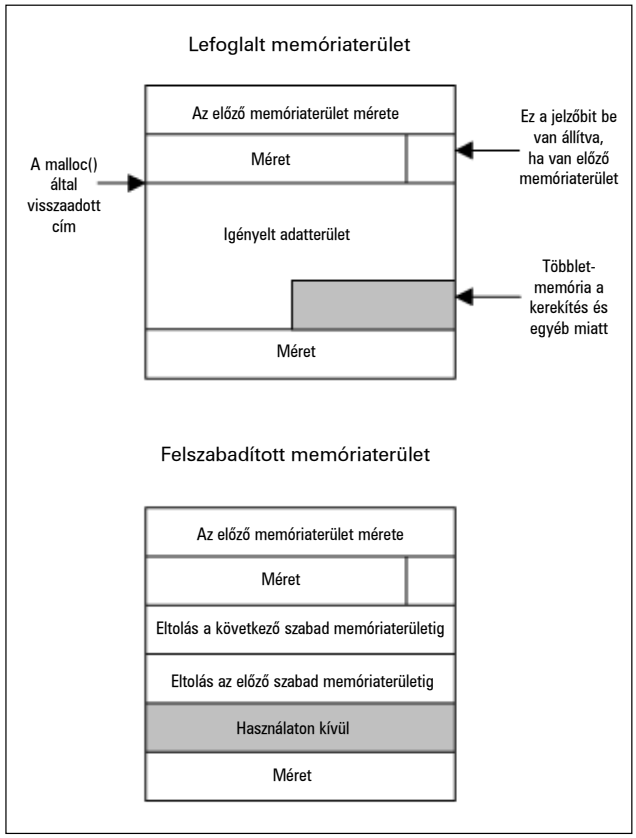
1. lista A /proc/<pid>/maps kimenete

```
<home>$ cat /proc/$$/maps
08048000-08091000 r-xp 00000000 03:03 77807
      ➥/bin/bash
08091000-08097000 rw-p 00048000 03:03 77807
      ➥/bin/bash
08097000-08115000 rwxp 00000000 00:00 0
40000000-40016000 r-xp 00000000 03:03 33122
      ➥/lib/ld-2.2.so
40016000-40017000 rw-p 00015000 03:03 33122
      ➥/lib/ld-2.2.so
40017000-40018000 rwxp 00000000 00:00 0
40018000-4001a000 rw-p 00000000 00:00 0
40023000-40026000 r-xp 00000000 03:03 31161
      ➥/lib/libtermcap.so.2.0.8
40026000-40027000 rw-p 00002000 03:03 31161
      ➥/lib/libtermcap.so.2.0.8
40027000-40148000 r-xp 00000000 03:03 33125
      ➥/lib/libc-2.2.so
40148000-4014e000 rw-p 00120000 03:03 33125
      ➥/lib/libc-2.2.so
4014e000-40152000 rw-p 00000000 00:00 0
40152000-4015c000 r-xp 00000000 03:03 33137
      ➥/lib/libnss_files-2.2.so
4015c000-4015d000 rw-p 00009000 03:03 33137
      ➥/lib/libnss_files-2.2.so
4015d000-40167000 r-xp 00000000 03:03 33140
      ➥/lib/libnss_nisplus-2.2.so
40167000-40169000 rw-p 00009000 03:03 33140
      ➥/lib/libnss_nisplus-2.2.so
40169000-4017c000 r-xp 00000000 03:03 33130
      ➥/lib/libnsl-2.2.so
4017c000-4017d000 rw-p 00012000 03:03 33130
      ➥/lib/libnsl-2.2.so
4017d000-40180000 rw-p 00000000 00:00 0
40180000-4018a000 r-xp 00000000 03:03 33139
      ➥/lib/libnss_nis-2.2.so
4018a000-4018b000 rw-p 00009000 03:03 33139
      ➥/lib/libnss_nis-2.2.so
bfffc000-c0000000 rwxp ffffd000 00:00 0
```

Nyomkövetés

A memóriakezelés hibákat és többnyire váratlan memóriaviselkedést okozhat. Ennek egyik oka az lehet, ha felszabadított memóriát használunk, azaz olyan memóriaszelettel dolgozunk, amelyet a program már felszabadított. Bár ez nem feltétlenül idéz elő azonnali hibát, valamilyen gond mégis biztosan felmerül, ha ugyanerre a területre új memóriafoglalás kerül bejegyzésre. Ennek eredményeképpen – mivel ugyanazt a memóriaterületet két különböző célra használjuk – váratlan értékeket fog eredményezni, sőt könnyen a program leállásához és *core dump*-hoz vezethet, ha memóriaterület-mutatókat vagy eltolási értékeket is tartalmaz.

A másik hibaforrás az lehet, ha a memóriaszelet előtagját írjuk felül. Amennyiben ugyanis a program felülírja a memóriaszelet bevezető részét, a memóriakezelő rendszer szinte bizonyosan hibázni fog vagy váratlan működést mutat, amikor a hibás memóriaszelettel találkozik. Néha a felülírás a szomszédos memóriaszeletben történik, és ez az ottani adatokat károsíthatja. Elképzelhető, hogy a felhasználó csak később veszi észre ezt a hibát, amikor a programvégrehajtás során furcsa értékeket vagy gyanús viselkedést tapasztal.



2. ábra A memóriaszelet szerkezete

Hasonlóképpen, ha a felszabadított memóriaszelet kezelőadatait összezavarjuk, felülírjuk vagy indokolatlanul használjuk, több mint valószínű, hogy a memóriakezelő rendszer maga is hibázni fog.

A memóriaaréna szabad (lefoglalatlan) részeinek használata ugyancsak okozhat hibát. Elképzelhető, hogy a kupacon kívül használunk fel memóriát, de még nem lépünk ki a memóriaarénából. Ez önmagában nem okoz gondot mindaddig, amíg frissen lefoglalt terület nem kerül ugyanerre a helyre. Az ilyesféle hibát igen nehéz felfedezni, mivel a rákövetkező memóriaműveletek (szabályosan) a kupacon belül találhatók.

A legnyilvánvalóbb és közvetlen hiba, amikor a program a memóriaarénán és a programterületen kívülre próbál írni. Ez azonnal SIGSEGV hibát (segmentation violation fault) okoz, következésképpen a program önműködő módon core dumpot készít.

A legkártékonyabb és legnehezebben nyomon követhető memóriahiba az, amikor a program verme károsodik. A program rengeteg helyi változót, értéket, az előző keretből (frame) származó regisztert, és ami a legfontosabb: visszatérési címet tárol a veremben. Így ha a verem megsérül, a programot hagyományos hibakeresővel szinte lehetetlen nyomon követni, mivel a veremhatárok maguk is használhatatlanná válnak. A veremmemória-hibák felderítésére mindössze néhány nyílt forráskódú (például libsafe) és kereskedelmi termékként kínált memóriahiba-kereső alkalmas, mivel a veremmemória-hibák felderítéséhez a programvégrehajtás menetét kell megváltoztatni vagy fejleszteni.

Létezik néhány módszer, amelyek segítségével meg lehet próbálni felderíteni a hibás memóriahasználatot, közülük néhány sajnos mellékhatásokat is okoz, például lassul a programvégrehajtás, vagy nagyobb lesz a memóriafelhasználás, következésképpen a nagy memóriaigényű programokban nemigen használhatók.

A következőkben bemutatandó hibakeresőkben használt hibás példa-

2. lista mytest00.c példaprogram

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

int
main (int argc, char **argv)
{
        char    *msg = malloc (4);
         // Lefoglalunk 4 bájtot

        strcpy (msg, "hello Linux users");
         // Túlcsordultatjuk a lefoglalt
         // memóriát
        printf ("%s\n", msg);
        free (msg);
         // Felszabadítjuk a foglalt memóriát
        strcpy (msg, "hello again");
         // Ráírunk a felszabadított helyre
        printf ("%s\n", msg);
        free (msg);
         // Felszabadítjuk a felszabadított
         // memóriát
        realloc (msg, 2);
         // Újra lefoglaljuk a felszabadított
         // memóriát
        strcpy (msg, "hello there");
         // A hibás memóriába írunk
        printf ("%s\n", msg);

        return 0;
}
```

3. lista mytest01.c példaprogram

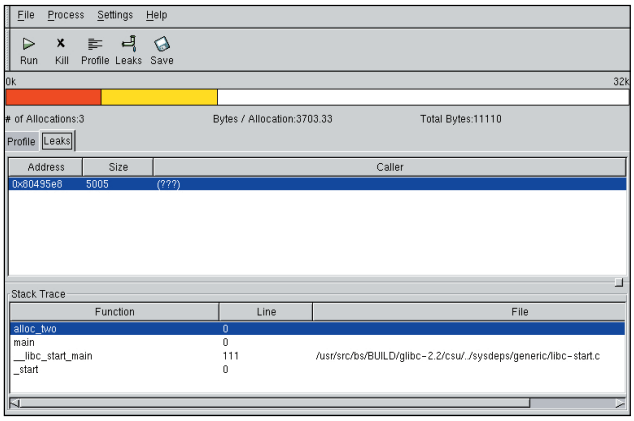
```
#include <stdlib.h>
#include <stdio.h>

int
main (int argc, char **argv)
{
      char    msg[4];
       // 4 bájtot foglalunk a veremben

      strcpy (msg, " hello Linux users 1234");
       // Túlcsordultatjuk a veremhelyet
      printf ("%s\n", msg);

      return 0;
}
```

programok forrása a *2.*, a *3.* és a 4. listában található. (A 4., 5., 6. listák a 15. CD Magazin/memoriahibak könyvtárában találhatók.) Alapértelmezés szerint a MALLOC_CHECK_ környezeti változó egyre állítva minimális hibakeresési célokra használható fel az alapértelmezett *malloc*-függvényhez. Ha a MALLOC_CHECK_-et egyre állítjuk, hibajelentéseket kapunk, ha pedig kettőre, a program bármilyen memóriahibánál azonnal leáll. A kimenet meglehetősen rejtélyes is lehet, mivel a hibakereső mód a hibás területeket olvasható szimbólumok helyett cím formájában jeleníti meg. Ezért nem árt, ha kéznél van egy nyomkövető, amivel megállapíthatjuk, hogy a programban hol is keletkeztek ezek a hibák. A következő példában alapértelmezett memóriahiba-keresést alkalmaztunk:



3. kép Memóriahiba-keresés memproffal

```
<home>$ MALLOC_CHECK_=1 ./mytest00
malloc: using debugging hooks
hello Linux users
free(): invalid pointer 0x80496d0
hello again
free(): invalid pointer 0x80496d0
realloc(): invalid pointer 0x80496d0
malloc: top chunk is corrupt
hello there
```

A kimenet azt mutatja, hogy a hiba a mytest00.c fájl 8. sorában (2. lista) található, ahol az `strcpy()` függvény túlcsordul és megsérti azt a memóriaszeletet, amelyre az `msg` mutat. A további nyomkövető hibaüzenetek ennek a károsodásnak a következményei.
Létezik néhány kitűnő nyílt forráskódú memóriaeszköz is. Az egyes megvalósítások memóriahiba-lefedettségben, kimenetben és interaktivitásban különböznek.
Az Electric Fence az egyik legkönnyebben használható eszköz. A könyvtár végrehajt néhány memóriaellenőrzést, majd amikor hibát észlel, megállítja a programot. Ez többnyire core dump formájában történik, amelyet aztán a felhasználó hibakereső programmal megvizsgálhat. Az Electric Fence a leghatékonyabban valamilyen nyomkövető programba ágyazva alkalmazható, például a GNU hibakeresőjében, a GDB-ben. Amikor az Electric Fence leállítja a programot, a GDB kapja meg a vezérlést, mégpedig pontosan azon a helyen, ahol a hiba bekövetkezett (lásd az 5. listát).
Ez a példakimenet az Electric Fence könyvtárral fordított ellenőrző-program végrehajtását mutatja GDB alatt. A legelső sérelem a mytest00.c 8. sorában SIGSEGV-et okoz. A GDB által felajánlott veremkövető segítségével a felhasználó azonosíthatja a hiba helyét.
A libsafe-rendszert többféle lehetséges veremhatár-átlépési sérelem ellenőrzésére is fel lehet használni, de csak néhány C könyvtárfüggvény esetében (`strcpy`, `strcat`, `getwd`, `gets`, `scanf`, `vscanf`, `fscanf`, `realpath`, `sprintf` és `vsprintf`) alkalmazható.
A libsafe példa kimenete igen tömör: amint a veremhiba bekövetkezik, a libsafe megjeleníti a hibaüzenetet, majd leállítja a programot. Emellett a libsafe az adott hiba részleteit több levélcímre is el tudja küldeni. Ez a hibajelentésnek kétségtelenül csavaros módja, de a libsafe-et a felhasználók elsődlegesen mégis inkább a puffertúlcsordulást kihasználó biztonsági töréskísérletek felderítésére használják. Egy kis szerkesztéssel a fejlesztő átalakíthatja a libsafe kódját, hogy

valamivel adatdúsabb üzeneteket küldjön. A másik lehetőség, hogy a programot GDB-ben hajtjuk végre és a `_libsafe_die()` címkére töréspontot állítunk be, ami a libsafe által felderített veremsértés esetén lép működésbe. A következő példában a libsafe veremfelülírást észlel, amit a mytest01.c 8. sorában található `strcpy()` hívás okoz (3. lista):

```
<home>$ LD_PRELOAD=/lib/libsafe.so.1.3  ./mytest01
Detected an attempt to write across stack
boundary.
Terminating mytest01.
Null message body; hope that´s ok

# Email is sent with the following subject
# header

libsafe violation for /tmp/mytest01, pid=27265;
overflow caused by strcpy()
```

A debauch kimenetét szimbólumok helyett kizárólag címek alkotják, emiatt csakis valamilyen hibakeresővel együtt érdemes alkalmazni. A debauch rendelkezik néhány, a felhasználó által előhívható különleges képességgel, amelyeket kifejezetten GDB alatti használatra terveztek. Ezek a képességek jobb memóriafoglalási és felszabadítási nyomkövetést tesznek lehetővé. A debauch meglehetősen alapos, és igen sokfajta memóriahibát képes észlelni, illetve azok megtörténte után helyreállítani (lásd a 6. listát).

A memprof legfőbb jellemzője a grafikus felület, amely könnyen érthetővé teszi, és ezen keresztül világosan látható lesz a memóriahiba kialakulásának helye. Meglehetősen hatékony képességekkel rendelkezik, ugyanis a bináris fájlleíró könyvtár (BFD library) által támogatott függvényeket használja – ugyanazokat, amelyeket a GDB is használ a folyamatok irányítására. *Képünkön* (lásd a 30. oldalon) bemutatjuk, amint a memprof megtalálja a lyukat a mytest02.c program `alloc_two()` függvényében.

A nyílt forráskódú memóriaeszközökön kívül néhány kereskedelmi termék is megvásárolható, ezek többnyire grafikus felülettel rendelkeznek, és még alaposabb ellenőrzést tesznek lehetővé, mint nyílt forráskódú megfelelőik (a kereskedelmi memóriaeszközök felsorolása a *Kapcsolódó címek* részben található.)

Végső megoldásként esetleg saját memóriakezelő-függvények megírása is szóba kerülhet. Ez sokat segíthet a memóriakezelés megértésében, illetve bizonyos egyedi esetekben teljesítménynövekedést eredményezhet (például nagy memóriaterületek gyors foglalása és felszabadítása esetén).

A memóriahibák felderítése nagyon fontos, nemcsak a megbízhatóság, hanem a biztonság szempontjából is. A Linuxhoz számos memóriahibakereső létezik, és mindegyik sajátos képességekkel bír, illetve egyedi működési feltételei vannak. Legjobban tesszük, ha a programot a fenti hibakeresők közül többel is kipróbáljuk valamilyen nyomkövető, például GDB alatt, hiszen így a hibák szélesebb tartományára derülhet fény.

Petr Sorfa
(petrs@sco.com) a Santa Cruz Operation's Development Systems Group tagja, ahol cscope és a Sar3D nyílt forráskódú projektek felelőseként tevékenykedik. Főiskolai végzettséget szerzett a Cape Town és a Rhodes University-n. Érdeklik a nyílt forráskódú fejlesztések, a számítógépes grafika, a fejlesztőrendszerek, továbbá a képregény-rajzolás (sequential art).

Kapcsolódó címek

Választható memóriafüggvények

Boehm Garbage Collector
➲ http://www.hpl.hp.com/personal/Hans_Boehm/gc/

CSRI ➲ ftp://ftp.cs.toronto.edu/pub/moraes/malloc.tar.gz

GNU Malloc
➲ ftp://ftp.cs.colorado.edu/pub/misc/mallocimplementations/

Hoard ➲ http://www.cs.utexas.edu/users/emery/hoard/

Ptmalloc (GNU C Library)
➲ http://www.malloc.de/en/index.html

QuickFit Malloc ➲ ftp://ftp.cs.colorado.edu/pub/misc/qf.c

vmalloc (az ast része)
➲ http://www.research.att.com/sw/download/

Nyílt forráskódú memóriaeszközök

ccmalloc
➲ http://www.inf.ethz.ch/personal/biere/projects/ccmalloc/

Checker
➲ http://www.gnu.org/software/checker/checker.html

dbmalloc ➲ http://dickey.his.com/dbmalloc/dbmalloc.html

DbMalloc ➲ http://www.cs.bris.ac.uk/~mm7323/DbMalloc/

debauch ➲ http://quorum.tamu.edu/jon/gnu/

dmalloc ➲ http://dmalloc.com/

Electric Fence ➲ http://www.perens.com/FreeSoftware/

fda ➲ http://packages.debian.org/unstable/devel/fda.html

leak ➲ http://sources.isc.org/devel/memleak/leak.txt

LeakTracer ➲ http://www.andreasen.org/LeakTracer/

libcw ➲ http://libcw.sourceforge.net/debugging/

libsafe ➲ http://www.bell-labs.com/org/11356/libsafe.html

MCheck
➲ http://www.cs.vu.nl/~rveldema/mcheck/mcheck.html

Memleak (Az X11R6.4 része)
➲ ftp://ftp.x.org/pub/R6.4/xc/util/memleak/

Memdebug
➲ http://www.bss.lu/Memdebug/Memdebug.html

MemProf ➲ http://people.redhat.com/otaylor/memprof/

Memwatch ➲ http://www.link-data.com/sourcecode.html

MM ➲ http://www.engelschall.com/sw/mm/

mpatrol ➲ http://www.cbmamiga.demon.co.uk/mpatrol/

mpr ➲ http://freshmeat.net/projects/mpr/

NJAMD ➲ http://fscked.org/proj/njamd.shtml

YaMa
➲ http://www.geocities.com/ipsgvm/libyama/index.html

YAMD ➲ http://www3.hmc.edu/~neldredge/yamd/

Kereskedelmi memóriaeszközök

Aprobe ➲ http://www.aprobe.com/

Insure++ ➲ http://www.parasoft.com/products/insure/

Purify ➲ http://www.rational.com/products/purify_unix/