Memóriakezelés haladóknak

A GNU C könyvtár néhány hasznos függvényének segítségével nemcsak takarékoskodhatunk az értékes memóriával, de csúnya hibákat is elháríthatunk.

A dinamikus memóriakezelés hagyományosan a C- és C++-programozás legkellemetlenebb része volt. Nem is meglepő, hogy bizonyos könnyebb, vagy legalábbis könnyebbnek tartott nyelvek, például a Java a szemétgyűjtő eljárások révén megszabadította a programozókat ettől a tehertől. A nehézsúlyú C-programozók számára azonban a GNU C könyvtár néhány olyan eszközt kínál, amelyek segítségével a memóriahasználat kézben tartható, ellenőrizhető és követhető.

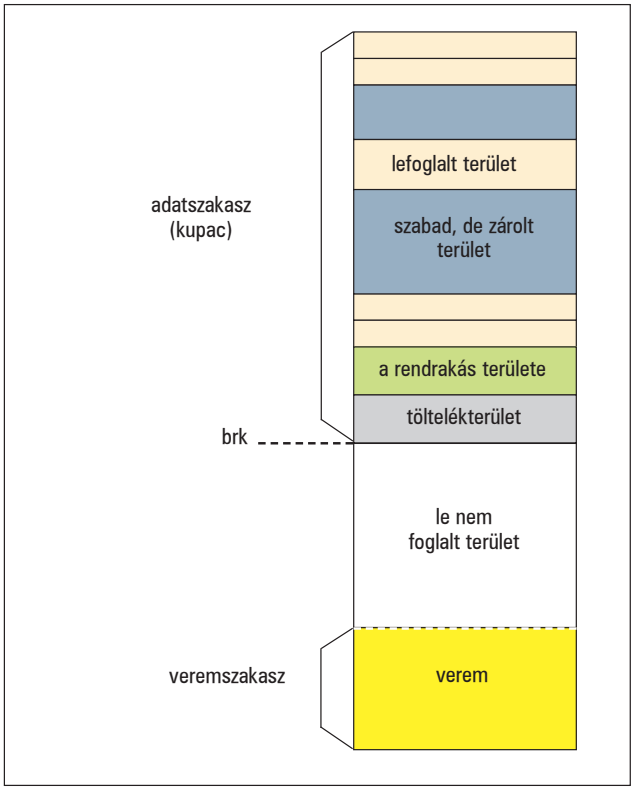
A memóriakezelés alapjai

Az adott folyamathoz tartozó memóriaterület általában lehet statikus, ekkor méretének meghatározására a fordításkor kerül sor, illetve dinamikus, ekkor a szükséges terület nagysága futás közben dől el. Az utóbbi esetben a memória két részre osztható: a kupacra (heap), ami a `malloc()` függvénnyel foglalt területeket öleli fel, illetve a veremre (stack), ami a függvények ideiglenes munkaterületét jelenti. Mint *ábránk* is szemlélteti, a kupac felfelé, míg a verem lefelé növekszik.

Ha egy folyamatnak memóriára van szüksége, akkor a `brk()` vagy az `sbrk()` rendszerhívással lehet kitolni a kupac felső határát. Mivel a rendszerhívások a processzorterhelés szempontjából költségesnek tekinthetők, egyetlen `brk()` hívással inkább egy nagyobb területet érdemes lefoglalni, majd azt – szükség szerint – kisebb darabokra osztva felhasználni. A `malloc()` pontosan ezt teszi: nagyszámú kisebb `malloc()` kérést kevesebb, de nagyobb `brk()` hívásba fog össze. Ezzel a módszerrel számottevő teljesítménynövekedést lehet elérni. A `malloc()` hívás maga is jóval olcsóbb, mint a `brk()`, ugyanis könyvtári (library call), és nem rendszerhívás. Hasonló folyamatok játszódnak le, amikor a folyamat memóriát szabadít fel. A memóriablokkokat a rendszer nem kapja vissza azonnal, hiszen ehhez minden alkalommal – negatív átadott értékkel – egy `brk()` hívást kellene végrehajtani. Ehelyett a C könyvtár addig gyűjtöget, amíg elegendően nagy memóriaterület nem szabadítható fel egyetlen lépésben.

A nagyon nagy területre irányuló kérések kezeléséhez a `malloc()` az `mmap()` rendszerhívással keres megcímezhető memóriaterületet. Ezzel részben elháríthatók a memóriatöredezés nemkívánatos hatásai, amikor nagyobb memóriablokkokat szabadítunk fel, ám a felhasználásukat akadályozhatják a közéjük ékelődő és a lefoglalt terület végére kerülő, újabban lefoglalt, kisebb méretű darabok. Ilyenkor a `brk()` hívással korábban lefoglalt blokkot hiába szabadítja fel a folyamat, a rendszer továbbra sem tudja használni.

A dinamikus memóriakezeléssel kapcsolatos függvények között a `malloc()` és a `free()` mellett számos továbbit is találunk, ám ezt a kettőt használjuk a legtöbbet. Érdemes megemlíteni a `realloc()` függvényt, amivel már lefoglalt blokkot tudunk átméretezni; a `calloc()` függvényt, ami „kitisztított” területet foglal; a `memalign()`, `posix_memalign()` és `valloc()` függvényt, amelyek egybefüggő terület foglalására alkalmasak.



A kupac és a verem egymással ellentétes irányban növekedik

A memóriaállapot kezelése

A C könyvtár memóriakezelő kódrészlete alapvetően általános jellegű memóriakezelésre való. Ez a megközelítés az esetek túlnyomó részében megfelelő teljesítményt eredményez, ám bizonyos programoknál érdemes módosítani a beállításokon. Először is a `malloc_stats()` vagy a `mallinfo()` könyvtári hívással gyűjtsünk adatokat a memóriahasználatról. Az előbbi tömör összefoglalót küld a program memóriahasználatáról a szabványos hibakimenetre. Az összefoglaló tartalmazza a rendszertől `brk()` hívások által lefoglalt bájtok számát, a `malloc()` hívásokkal ténylegesen használatba vett terület nagyságát, illetve az `mmap()` hívásokkal igényelt memóriaméretet. Példa a hívás kimenetére:

```
Arena 0:
system bytes     =    205892
(rendszerbájtok száma)
in use bytes     =    101188
(használatban lévő bájtok száma)
Total (incl. mmap):
(összes, az mmap hívásokkal együtt)
system bytes     =    205892
(rendszerbájtok száma)
in use bytes     =    101188
(használatban lévő bájtok száma)
```

```
max mmap regions     =    0
max mmap regions     =    0
```

Ha pontosabb adatokat akarunk látni, és nem csak egy egyszerű kiíratást szeretnénk, a `mallinfo()` hívást használhatjuk. A függvény egy `mallinfo` nevű adatszerkezetet ad vissza, ami különféle, a memóriaállapottal kapcsolatos állapotjelzőket tartalmaz. Ezek közül a legfontosabbak leírását a „A legfontosabbak a `mallinfo` által visszaadott adatok közül” című széljegyzet tartalmazza. Az adatszerkezet teljes körű ismertetése a */usr/include/malloc.h* című állományban található meg.
A `malloc_usable_size()` a libc egy másik hasznos függvénye, ami egy korábban lefoglalt memóriaterületből felhasználható bájtok számát adja vissza. A minimális méretekre vonatkozó megszorítások és a területek folytonosságára való törekvés miatt ez a mennyiség az eredetileg kértnél nagyobb is lehet. Ha például lefoglalunk 30 bájtot, a ténylegesen felhasználható terület 36 bájt lesz. Vagyis legfeljebb 36 bájtot írhatunk be a memóriablokkba úgy, hogy más blokkokat ne írjunk felül. Természetesen ennél visszataszítóbb programozási módszert nehéz lenne említeni, és erősen változatfüggő is, tehát semmiképpen ne használjuk. A `malloc_usable_size()` elsősorban mint hibakereső eszköz lehet hasznos. Alkalmas például arra, hogy ellenőrizzük egy kívülről kapott memóriablokk méretét, még mielőtt beleírnánk valamit.

A foglalási stratégia

A memóriakezelő függvények viselkedését bizonyos beállítások módosításával, a `mallopt()` függvény segítségével lehet módosítani (*1.* és *2. lista*).
A függvény prototípusa, illetve négy alapvető, általa módosítható beállítás az SVID/XPG/ANSI szabványban is szerepel.
A jelenlegi GNU C könyvtári megvalósítás (a 2.3.1-es változat a cikk születésének időpontjában) ezek közül csak egyet támogat (`M_MXFAST`), a többi hármat nem. Másrészről a könyvtár további négy, a szabvány által nem említett beállítással vigasztal minket. A `mallopt()` segítségével módosítható beállítások leírása az „A `mallopt()` segítségével módosítható beállítások” című széljegyzetben található.
A foglalás hangolása a `mallopt()` hívásoknak a programba való építése és a program újrafordítása nélkül is megoldható. Ez különösen akkor hasznos, ha bizonyos értékeket gyorsan ki akarunk próbálni, vagy ha nem rendelkezünk a forráskóddal. Mindössze annyit kell tennünk, hogy a program futtatása előtt beállítjuk a megfelelő környezeti változót. *Táblázatunk* a `mallopt()` beállítások és a környezeti változók egymásnak való megfeleltetését tartalmazza, némi plusztájékoztatással kiegészítve. Ha például a „rendrakási” határértéket 64 KB-ra akarjuk állítani, az alkalmazást így kell indítani:

```
MALLOC_TRIM_THRESHOLD=65536 alkalmazás
```

A rendrakásról szót ejtve: a `malloc_trim` (töltelék) hívással lehet elindítani a memória rendbetételét és a használaton kívüli területek rendszernek való visszaadását. A függvény átméretezi az adatszakaszt, legalább ***töltelléknyi*** részt hagyva a végén, és sikertelenül tér vissza, ha nem tudott legalább egy lapnyi területet felszabadítani. A szegmensméret mindig a lapméret – ez i386 processzornál 4096 bájt – többszöröse. A rendrakásra elérhető memória méretét a `mallinfo()` által visszaadott adatszerkezet `keepcost` összetevője tartalmazza. Önműködő rendrakást a `memory_trim()` meghívásával a `free()` függvény is végez, feltéve, hogy a `keepcost` értéke nagyobb az `M_TRIM_THRESHOLD` értékénél, átadott töltelék értékként pedig az `M_TOP_PAD`-et veszi.

A legfontosabbak a „mallinfo()” által visszaadott adatok közül

- `arena`: a kupacból lefoglalt terület teljes mérete (vagyis az azt jelző mérőszám, hogy a határpont mennyivel mozdult el a folyamat indítása óta).
- `uordblks`: a lefoglalt és használatban lévő bájtok száma.
- `fordblks`: a lefoglalt, de használatban nem lévő bájtok száma.
- `keepcost`: a `malloc_trim()` hívása révén felszabadítható terület nagysága.
- `hblks`: az `mmap()` segítségével lefoglalt darabok száma.
- `hblkhd`: az `mmap()` segítségével lefoglalt bájtok száma.

Memóriával kapcsolatos hibakeresés: az éppség ellenőrzése

A memóriakezeléssel kapcsolatos hibák keresése az egyik leginkább időtrabló dolog az összetett alkalmazások fejlesztésekor. Ilyenkor egyrészt a blokkok lefoglalását és felszabadítását kell tisztába tenni, másrészt fel kell deríteni a memóriakezelési hibákat.
A memória sértetlensége akkor bomlik meg, ha olyan helyre írunk, ami ugyan az adatszakaszban található, de az eredetileg használni kívánt memóriablokk határain kívülre esik. Jó példa erre az, amikor egy tömb végén túlra írunk adatot. Természetesen, ha az adatszakaszon kívülre próbálnánk írni, akkor a program szakaszolási (segmention) hibával azonnal leállna, illetve a megfelelő jelzéskezelő meghívására kerülne sor, és így azonosítani lehetne a hibát okozó utasítást. A sértetlenség megbomlása ennél kellemetlenebb, mivel észrevétlenül is lezajlódhat, és később a hibás résztől akár messzeeső programrészekben okozhat rendellenes működést. Minél előbb fedezzük fel tehát a hibát a programban, annál nagyobb eséllyel tudjuk elhárítani a következményeit.
Az adatépség megbomlása más memóriablokkokat is érinthet, és nemcsak az alkalmazás adatainak, de a kupackezelő adatszerkezeteknek a sérülését is okozhatja. Az előbbi esetben a hibára egyedül saját adatszerkezeteink tartalmának az elemzésével deríthetünk fényt. Az utóbbi esetben a GNU libc sértetlenséget ellenőrző eszközeire támaszkodhatunk, amelyek értesítenek minket, ha valamilyen hiba merül fel.
Adott programon belül a memória ellenőrzése önműködően és kézzel egyaránt végezhető. Az előbbi a `MALLOC_CHECK_` környezeti változó értékének megadásával lehetséges:

```
MALLOC_CHECK_=1 alkalmazás
```

1. táblázat A mallopt() beállítások és a környezeti változók megfeleltetése

mallopt()-beállítás	környezeti változó	alapérték	megjegyzés
M_TRIM_THRESHOLD	MALLOC_TRIM_THRESHOLD_	128 KB	-1U letiltja
M_TOP_PAD	MALLOC_TOP_PAD_	0	
M_MMAP_THRESHOLD	MALLOC_MMAP_THRESHOLD_	128 KB	0 letiltja
M_MMAP_MAX	MALLOC_MMAP_MAX_	64	0 letiltja

1. lista A rendrakás határértékének beállítása a mallopt() függvénnyel

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

int main(int argc, char *argv) {

  int thr;
  char *p1;

  if (argc != 2) {
      printf("Használat: 1_kódrészlet
      ➥<rendrakás határérték " "[KB]>\n");
      exit(0);
  }

  thr = atoi(argv[1])*1024;
  if (!mallopt(M_TRIM_THRESHOLD, thr)) {
      printf("sikertelen mallopt()\n");
  }

  printf("100 k lefoglalása, a rendrakás
  ➥határértéke " " %d bájt\n", thr);
  p1 = malloc(100000);
  malloc_stats();

  printf("\n100 k felszabadítása\n");
  free(p1);
  malloc_stats();
}
```

Ezzel a módszerrel a határtúllépések jelentős része felfedezhető, és egyes esetekben a program összeomlása is megelőzhető.
A hiba felismerésekor végrehajtott művelet a `MALLOC_CHECK_` értékétől függ: 1-es értéknél a rendszer hibaüzenetet küld a `stderr` kimenetre, de magát a programot tovább engedi futni; 2-es értéknél kimenet nélkül leállítja a programot; 3-as értéknél az 1-es és 2-es eset hatása összeadódik.
Az önműködő ellenőrzés csak akkor játszódik le, amikor memóriakezeléssel kapcsolatos függvényeket hívunk meg. Ha tehát egy tömbbe a határain túlnyúlva írunk, akkor csak a következő `malloc()` vagy `free()` hívásnál fogunk hibaüzenetet kapni. A hibák egy része sajnos észrevétlen marad, és a hibaüzenetek tartalma sokszor – finoman szólva – kevésbé hasznos. Egy `free()` hívásnál például tudjuk, hogy melyik mutatót szabadítottuk fel a hiba felismerésekor, de azt nem, hogy pontosan mi vagy ki zavarta össze a kupac tartalmát. Ha lefoglalás közben derül ki valamilyen hiba, a „heap corrupted” (kupac sérült) üzenetet kapjuk.
Egy másik lehetőség az, hogy ellenőrzési pontokat helyezünk el a programban. Ilyenkor még a program futásának megkezdésekor meg kell hívnunk az `mcheck()` függvényt. Segítségével olyan egyedi hibakezelőt vehetünk használatba, amelyet minden egyes kupachiba felismerésekor meg tudunk hívni. Egy alapértelmezett hibakezelő is rendelkezésünkre áll arra az esetre, ha nem írtunk sajátot. Ha az `mcheck()` meghívásán túlestünk, a `MALLOC_CHECK_` révén elérhető, sértetlenséget ellenőrző lehetőségek máris rendelkezésünkre állnak. Emellett az `mprobe()` függvényt bármikor meghívhatjuk, ha a megadott memóriamutatót szeretnénk ellenőrizni. Az `mprobe()`

2. lista Kisebb rendrakási határértéket megadva memóriát takaríthatunk meg

```
branzo@betelgeuse:~/malloc_debug$
➥./1_kodreszlet 128

100 k lefoglalása, a rendrakás határértéke
➥131072 bájt
Arena 0:
system bytes =      103724
in use bytes =      100012
Total (incl. mmap):
system bytes =      103724
in use bytes =      100012
max mmap regions    =           0
max mmap bytes      =           0

100 k felszabadítása
Arena 0:
system bytes =      103724
in use bytes =           4
Total (incl. mmap):
system bytes =      103724
in use bytes =           4
max mmap regions    =           0
max mmap bytes      =           0

branzo@betelgeuse:~/malloc_debug$
./1_kodreszlet 64

100 k lefoglalása, a rendrakás határértéke
➥65536 bájt
Arena 0:
system bytes =      103724
in use bytes =      100012
Total (incl. mmap):
system bytes =      103724
in use bytes =      100012
max mmap regions    =           0
max mmap bytes      =           0

100 k felszabadítása
Arena 0:
system bytes =        1324
in use bytes =           4
Total (incl. mmap):
system bytes =        1324
in use bytes =           4
max mmap regions    =           0
max mmap bytes      =           0
```

Az mprobe() visszatérési értékei

- `MCHECK_DISABLED`: sértetlenség ellenőrzése kikapcsolva (az `mtrace()` meghívására nem került sor).
- `MCHECK_OK`: a blokk rendben van.
- `MCHECK_FREE`: a blokkot kétszer szabadítottuk fel.
- `MCHECK_HEAD`: a blokk előtti memóriaterület hibás.
- `MCHECK_TAIL`: a blokk utáni memóriaterület hibás.

A „mallopt()” segítségével módosítható beállítások

- `M_TRIM_THRESHOLD`: az a legkisebb memóriamennyiség, amelynek felszabadítása az adatszakasznak `brk()` hívással történő leszűkítését váltja ki a memóriakezelő rendszerből. A felszabadított adatterület – mint már esett róla szó – a rendszerhívások számának csökkentése érdekében valójában nem kerül vissza azonnal a rendszerhez. Ez az érték adja meg, hogy mennyi memória felszabadítása után kerül sor egy ilyen hívásra. A rendrakási határérték nagyban befolyásolhatja a rendszer teljesítményét. Ha túl magasra állítjuk, akkor több memóriát tartunk vissza, és a rendszer nagyobb valószínűséggel veszi használatba a nagyságrenddel lassabb cseretárhelyet. Ellenben ha túlságosan kis értéket adunk neki, akkor a gyakori `brk()` hívások fogják a teljesítmény romlását okozni. A beállítás főként hosszú ideig futtatott programoknál fontos, ezek ugyanis nagy mennyiségű memóriát szívhatnak el a rendszertől. Azt is fontos megjegyezni, hogy a töredezettség miatt „zárolt” darabok nem adhatók vissza a rendszernek, mégha méretük nagyobb is a határértéknél – éppen ezért van szükség az `mmap()` függvénnyel végzett foglalásra. Az 1. kódrészletben található program a határértéket a neki átadott érték szerintire állítja, majd 100 KB memóriát foglal le és szabadít fel. Az eredmény a 2. kódrészletben látható.
- `M_TOP_PAD`: mennyi tartalékhelyet kell foglalni (meghagyni), amikor egy `brk()` hívással a kupac bővítésére (szűkítésére) kerül sor. Amikor egy `malloc()` hívás hatására a `brk()` veszi át a vezérlést, ennyi pluszterületet igényel a rendszertől. Hasonlóan, amikor egy `free()` hívás hatására negatív `brk()` hívás történik, ennyi pluszhelyet hagy meg a könyvtár jövőbeli használatra. Fontos, hogy az értékét csak megfontoltan változtassuk meg, ha a programunk nagy mennyiségű egymást követő `malloc()` és `free()` hívást hajt végre, nagy számú `brk()` hívást váltva ki ezzel.
- `M_TRIM_THRESHOLD`: az ennél nagyobb méretű memóriaterületre vonatkozó igények kielégítése `brk()` helyett `mmap()` hívással történik. Mint már utaltam rá, az utóbbi eljárás előnye az, hogy az így lefoglalt memóriaterület – a rendrakási határérték elérése nélkül is – felszabadítás után azonnal elérhető lesz a rendszer számára, ellentétben a `brk()` hívással lefoglalt részekkel. Ha kisebb értéket adunk neki, akkor több `mmap()` és kevesebb `brk()` hívás történik, és kevesebb lesz a zárolt memóriatöredék. Hátránya, hogy az `mmap()` általában lassabban intézi a memóriafoglalást, mint a `brk()` – az esetek nagy részében tehát a legjobb az, ha az alapértelmezett értéket hagyjuk meg.
- `M_MMAP_MAX`: az `mmap()` által egyszerre lefoglalható memóriadarabok maximális száma. Olyan rendszereknél van jelentősége, amelyeknél korlátozva van a folyó `mmap()` eljárások száma. Általában az a legjobb, ha nem nyúlunk hozzá.
- `M_MXFAST`: a fastbin memóriakonténerek használatával teljesített kérések által lefoglalható blokkok maximális mérete. Helyszűke miatt nem foglalkozhatunk részletesen a fastbin konténerekkel, annyit viszont érdemes tudni róluk, hogy elsősorban sok kicsi, azonos méretű blokk lefoglalásakor vagy felszabadításakor használhatók hatékonyan. Megjegyzem, mivel a fastbin konténerek soha nem kerülnek összevonásra (vagyis két szomszédos szabad fastbin soha nem egyesül egyetlen nagyobb, szabad blokkba), használatuk növeli a memória töredezettségét, illetve a program összesített memóriaszükségletét. Ebből következik, hogy az érték egyfajta megalkuvás eredménye a sebességnövekedés (kisebb fastbinek) iránti igény és a velük járó töredezés elkerülése (nagyobb fastbinek) iránti szándék között. A kisebb értékek az esetek jelentős részében megfelelők, ugyanis a GNU C könyvtárat úgy optimalizálták, hogy a nagyobb fastbinek jelentős töredezést okoznak, miközben használatuk nem jár érzékelhető sebességnövekedéssel. A fastbin-támogatás a `malloc` eljárások teljes újraírásakor, a 2.3-as változattal jelent meg a GNU C könyvtárban.

által visszaadott értékek ismertetése a „Az `mprobe()` visszatérési értékei” című széljegyzetben található.
Ha a teljes kupacot, és nem csupán egyetlen blokkot kívánunk ellenőrizni, az `mcheck_check_all()` meghívásával az összes aktív blokkon végigmehetünk. Arra is utasíthatjuk a memóriakezelő rutinokat, hogy csak a pillanatnyi blokk ellenőrzése helyett az `mcheck_check_all()` függvényt használják, ehhez induláskor az `mcheck()` helyett az `mcheck_pedantic()` függvényt kell meghívni. Azt azonban ne feledjük, hogy ezzel elég sok erőforrást kötünk le.
A memória-ellenőrzés engedélyezésének harmadik módja az alkalmazás libmcheckkel való fordítása:

```
gcc alkalmazás.c -o alkalmazás -lmcheck
```

Az `mcheck()` függvény így még az első memóriafoglalás végrehajtása előtt önműködően meghívásra kerül – ez különösen akkor érdekes, ha a program már a `main()` függvény elérése előtt is foglal dinamikus memóriablokkokat.

Memóriával kapcsolatos hibakeresés: a blokkok nyomon követése

A memóriablokkok történetét végigkövetve könnyebben felismerhetők a memóriaszivárgásokkal kapcsolatos gondok, illetve a már felszabadított blokkok használata vagy újbóli felszabadítása. A GNU C könyvtár ilyen célokra nyomkövetési lehetőséget kínál, amit az `mtrace()` függvény meghívásával vehetünk igénybe. A hívás végrehajtása után a rendszer minden kupacműveletet egy fájlba naplóz. A fájl nevét a `MALLOC_TRACE` környezeti változóban kell megadni. A naplófájl elemzését egy Perl-parancsfájllal később is el lehet végezni, ami szintén a könyvtár része, és – nem meglepő módon – szintén az ***mtrace*** nevet viseli. A naplózás a `muntrace()` függvény hívásával állítható le, de ne feledjük, ha a programnak csak bizonyos részeire végezzük el a naplózást, akkor az utólagos feldolgozáskor kapott eredmények teljesen értelmetlenek is lehetnek. Nem létező szivárgást fedezhetünk fel például úgy, hogy lefoglalunk egy memóriablokkot, majd a `muntrace()` meghívása után szabadítjuk fel.
Az alábbiakban a nyomkövetésre látható egy példa a *3. listában* szereplő programmal:

```
$ gcc -g 3_kodreszlet.c -o 3_lista
$ MALLOC_TRACE="nyomkovetes.log" ./3_lista
$ mtrace nyomkovetes.log

Memory not freed: (Fel nem szabadított memória)
-----------------
```

```
Address (Cím)    Size (Méret)     Caller (Hívó)
0x08049718       0xa  at
malloc_debug/3_lista.c:9
```

A memória nyomkövetése nem befolyásolja a hibák kialakulását, így attól még, hogy az `mtrace()` meghívása megtörtént, a program akár össze is omolhat. Ennél csak az a rosszabb, ha a program szakaszolási hibát követ el, hiszen ekkor a nyomkövetés naplófájlja befejezetlen marad, és akár ellentmondásos adatokat is tartalmazhat. Ez ellen úgy védekezhetünk, hogy egy SIGSEGV jelzéskezelőt készítünk, ami a `muntrace()` függvényt hívja meg, és ezáltal még a program leállása előtt gondoskodik a naplófájl lezárásáról (*4. lista*).
A memória nyomkövetéséről a libc infooldalán található további tájékoztatást.

Mélyebbre merülve

Megeshet, hogy a GNU C könyvtár normál hibakeresési lehetőségei egy program esetében elégtelennek bizonyulnak. Ebben az esetben külső eszközt is használhatunk a memória-kezelési hibák felderítésére (lásd a *listákon*), de mélyebbre is beáshatjuk magunkat a könyvtár rejtelmeibe. Ehhez csupán három függvényt kell megírnunk, majd rá kell aggatni őket a következő előre megadott változókra:

- `__malloc_hook` – arra a függvényre mutat, ami akkor jut szerephez, ha a felhasználó a `malloc()` függvényt hívja meg. Ebben tetszőleges ellenőrzéseket és számlázást (accounting) végezhetünk, majd a valódi `malloc()` meghívásával lefoglalhatjuk a kért memóriaterületet.
- `__free_hook` – a normál `free()` helyett meghívandó függvényt adja meg.
- `__malloc_initialize_hook` – a memóriakezelő rendszer üzembe helyezésekor meghívott függvényre mutat. Így mód nyílik arra, hogy bizonyos műveleteket hajtsunk végre, például még a többi memóriakezelő művelet elindulása előtt megadhatjuk az előbbi két változó értékét.

3. lista Nyomkövetés az mtrace() segítségével

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

int main() {
  char *ptr;

  mtrace();
  ptr = malloc(10);
  /*  free(ptr); */

}
```

4. lista A SIGSEGV kezelőben meg kell hívni a muntrace() függvényt

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

void handler(int s) {
  muntrace();
  abort();
}

int main() {
  char *ptr;

  signal(SIGSEGV, handler);
  mtrace();
  ptr = malloc(10);
  free(ptr);
  free(ptr);

}
```

Hasonló módosítási lehetőségünk van a többi memóriakezelő függvény esetében is, mint a `realloc()`, `calloc()` stb. Ügyeljünk arra, hogy mentsük a változók korábbi értékeit, majd a `malloc()` vagy `free()` fenti eljárásainkban történő meghívása előtt visszaállítsuk. Ha ezt elmulasztjuk, a program végtelen önhívó ciklusba esik, és használhatatlan lesz. Érdemes egy pillantást vetni a libc infooldalán található memória-hibakeresési példában található kódra, az összes fontos részlet kiderül belőle.
Utolsó megjegyzésként annyit még megemlítenék, hogy a fenti változókat az mcheck- és az mtrace-eszközök is használják. Ha tehát mindet egyszerre akarjuk igénybe venni, nagyon nagy körültekintésre lesz szükségünk.

Összegzés

A GNU C könyvtár számos bővítménnyel rendelkezik, amelyek roppant hasznosnak bizonyulnak a memóriakezeléssel kapcsolatos kérdések megoldásában. Ha finomhangolni szeretnénk egy alkalmazás memóriahasználatát, vagy a saját igényeinkre szabott memória hibakereső megoldást akarunk készíteni, akkor minden bizonnyal jól tudjuk használni ezeket az eszközöket, és megfelelő kiindulópontként szolgálnak saját eljárásaink kidolgozásához.

Linux Journal 2003 május, 109. szám

Gianluca Insolvibile (g.insolvibile@cpr.it)
A 0.99pl4-es rendszermag óta Linux-rajongó. Jelenleg hálózati és digitális videokutatással és -fejlesztéssel foglalkozik.

KAPCSOLÓDÓ CÍMEK

„Debugging Memory on Linux”, írta Petr Sorfa, Linux Journal, 2001. július; elérhető a ➲ http://www.linuxjournal.com/article/4681 címen.
Electric Fence ➲ ftp.perens.com/pub/ElectricFence
A GNU C könyvtár kézikönyve ➲ http://www.gnu.org
mpatrol ➲ http://www.cbmamiga.demon.co.uk/mpatrol
NJAMDB ➲ http://sourceforge.net/projects/njamd