Valgrind

A szoftverek minőségellenőre

– Mester, fagy a program!
– Valóban? Nem értem, hisz nálam működött!

A C, C++ fejlesztőknek gondolom fájdalmasan ismerősek a fenti sorok. Ugyan honnan is tudnánk, hogy hibás a program, ha nálunk rendesen működik? A megoldás: Valgrind.

A *Valgrind* egy program, mely képes megmutatni a C és C++ programok tipikus memória kezelési hibáit (puffertúlírás, döntés hozás ismeretlen tartalmú (inicializálatlan) memóriacella alapján, átlapoló `memcpy`, összekevert `malloc/delete`,...). Használatával és a hibák javításával, jó esélyünk van arra, hogy a programunk nem csak nálunk, hanem más felhasználók gépén is működni fog.
Mint a *Krusader* (kétpaneles fájlkezelő) projekt fejlesztője, igen sokszor találkozom olyan szituációkkal, hogy egyik gépen megy valami, a másikon pedig fagy. A legutolsó ilyen hiba volt a legfájdalmasabb számunkra, amikor kiadtuk az 1.60-as „stabil" változatot. Kezdetben mindenki elégedett volt, aztán egyik napról a másikra egy csomó levelet kaptunk, hogy a program fagy. A legrosszabb az volt, hogy még csak el sem indult, hanem már a bejelentkező ablak megjelenítése előtt elszállt. Mi történhetett? Egyik fejlesztőtársam végül megfejtette a rejtélyt: egy puffertúlírási hiba miatt kizárólag *GCC 3.x* alatt működött a program. Amikor viszont frissen kijött a *GCC 4.0*, az alatta fordított *Krusader* fagyott.
Még a bejelentkező ablakig sem jutott el, így teljesen használhatatlan lett. Szerencsére a felhasználói táborunk igen türelmes volt és sokan még arra is hajlandóak voltak, hogy lefordítsák az instabil változatot (amiben a hiba már javításra került). Persze ilyen lépést csak egyszer lehet meglépni. Amennyiben rendszeressé válnak a fagyások, előbb-utóbb elpártolnak tőlünk a felhasználók. Ez a hiba ráadásul olyan triviális volt, hogy a *Valgrind* azonnal jelezte volna...
A történtek után nekiugrasztottam a *Valgrindet* a *Krusadernek* és még 7-8(!) memóriahibát kijavítottam, ami nálam nem fagyott (remélem máshol sem), de fagyhatott volna. Néhol engem is meglepett, hogy a legstabilabbnak tűnő részeken is talált valamit. Azóta csönd és béke van a projekt körül és egyenlőre nincs komoly panasz rá. Ezután elhatároztam, hogy minden egyes stabil kiadás előtt legalább egyszer tesztelem a *Krusadert Valgrinddel* is, mert megéri.

A Valgrindről

A projekt neve a skandináv mitológiából ered. *Valgrind* a főbejárata *Valhallának* (a kiválasztott halottak termének). A bejáraton túl egy farkas áll őrt, mögötte pedig egy vaddisznó fej van, melyen hatalmas sas ül.
A sas szemével kilenc világ messzi tájait láthatja. Csak azok kelhetnek át *Valgrind* kapuján, akiket az őrök érdemesnek ítélnek meg. A többiek kívül maradnak. Ennyit a mitológiáról.
Maga a *Valgrind* program egy szintetikus x86-os processzor. A tesztelt program utasításait nem a mikroprocesszor értelmezi, hanem a *Valgrind*, mely eközben elvégzi a szükséges cím és memória ellenőrzéseket.
Úgy működik, mint egy emulátor. Ez persze teljesítménycsökkenést eredményez, ezért készüljünk fel arra, hogy a programjaink legalább 10-szer lassabban fognak futni és több, mint kétszer annyi memóriát zabálnak majd, mint nélküle. Mindenképpen megér egy erős, nagy memóriájú gépet használni tesztelésre.
A *Valgrindet* szinte az összes nagy linuxos projekt használta már, vagy használja: *Firefox, OpenOffice, StarOffice, AbiWord, Opera, KDE, GNOME, Qt, libstdc++, MySQL, PostgreSQL, Perl, Python, PHP, Samba, RenderMan, Nasa Mars Lander software, SAS, The GIMP, Ogg Vorbis, Unreal Tournament, Medal of Honour, RenderMan,...*
A *Valgrind* nagyon sok eszközt biztosít a szoftverek tesztelésére, cikkem viszont kizárólag a memóriaellenőrző komponensével (*memcheck*) foglalkozik.

1. Lista – Példa memória túlolvasásra (test.c)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5     /* 10 byte memória
➥foglalása */
6     char * mem = (char *)
➥malloc( 10 );
7     /* túlolvasás */
8     printf( "Char: %d\n",
➥mem[ 10 ] );
9  }
```

Tesztelés Valgrinddel

A ***Valgrind a*** program futása közben igen sok dolgot tesztel:

- Inicializálatlan memória használata
- Írás, olvasás felszabadított memóriaterületről
- Lefoglalt memória túlírása, túlolvasása
- A verem (stack) helytelen írása, olvasása
- Memória-szivárgás
- Összekevert `malloc/new/new []`, `free/delete/delete []`
- Átlapoló forrás és cél a `memcpy` függvényben

A tesztelés menete is igen egyszerű: lefordítjuk a programot és ***Valgrinddel*** futtatjuk. Mivel grafikus felület nincs, ezért parancssorból kell használni. Próbaként gépeljük be a ***test.c*** programot *(1. lista)*, mely puffer túlolvasási hibát tartalmaz, majd fordítsuk le és futtassuk ***Valgrinddel.***

```
gcc -g -O test.c -o test
valgrind --tool=memcheck test
```

Nem mindegy, hogy hogyan fordítunk. A `-g` opció `DEBUG` módú fordítást jelent. Fontos, mert a ***Valgrind*** csak így képes a hibás sor számát kiírni. A `-O` opcióval pedig az optimizációt kapcsoljuk ki. Az optimizáció azért problematikus, mert ha inline-ként fordít a fordító egy függvényt és hiba van benne, a problémás sor számát lehetetlen lesz kiírni. `O2` és magasabb optimizációk esetén GCC-vel néha hamis hibaüzenetet is kaphatunk (`conditional jump or move uninitialized depends on uninitialized value(s)`). A projekt írói nem javították ki a hibát, mert sokkal lassabb végrehajtást eredményezne, meg egyébként sincs értelme `O2`-ben ***Valgrindezni***.

A `–tool=memcheck` szintén fontos opció, mert ez állítja be a memóriaellenőrzést. Nélküle nem biztos, hogy megbízható eredményt kapnánk.

A futtatás eredményét a *2. lista* tartalmazza. Amennyiben ettől eltérő jellegűt kapunk, úgy elképzelhető, hogy valami rosszul működik a rendszerben. Ilyenkor töltsük le a legfrissebb ***Valgrindet***, fordítsuk le és használjuk azt.

Vegyük egy kicsit alaposabban szemügyre a végeredményt!

2. Lista – A Valgrindes futtatás végeredménye

```
==17199== Memcheck, a memory error detector.
==17199== Copyright (C) 2002-2005, and GNU GPL'd, by Julian
➥Seward et al.
==17199== Using LibVEX rev 1575, a library for dynamic binary
➥translation.
==17199== Copyright (C) 2004-2005, and GNU GPL'd, by OpenWorks
➥LLP.
==17199== Using valgrind-3.1.1, a dynamic binary
➥instrumentation framework.
==17199== Copyright (C) 2000-2005, and GNU GPL'd, by Julian
➥Seward et al.
==17199== For more details, rerun with: -v
==17199==
==17199== Invalid read of size 1
==17199==    at 0x8048402: main (test.c:8)
==17199==  Address 0x415D032 is 0 bytes after a block of size
➥10 alloc'd
==17199==    at 0x401A451: malloc (vg_replace_malloc.c:149)
==17199==    by 0x80483FE: main (test.c:6)
Char: 0
==17199==
==17199== ERROR SUMMARY: 1 errors from 1 contexts (suppressed:
➥13 from 1)
==17199== malloc/free: in use at exit: 10 bytes in 1 blocks.
==17199== malloc/free: 1 allocs, 0 frees, 10 bytes allocated.
==17199== For counts of detected errors, rerun with: -v
==17199== searching for pointers to 1 not-freed blocks.
==17199== checked 57,568 bytes.
==17199==
==17199== LEAK SUMMARY:
==17199==    definitely lost: 10 bytes in 1 blocks.
==17199==      possibly lost: 0 bytes in 0 blocks.
==17199==    still reachable: 0 bytes in 0 blocks.
==17199==         suppressed: 0 bytes in 0 blocks.
==17199== Use --leak-check=full to see details of leaked
➥memory.
```

A ***Valgrind*** alapértelmezés mellett a sztenderd kimenetre írja az üzeneteit. Az „==17199==" a futtatott folyamat száma (***PID***-je). Egy hibát talált a futtatás során, és 10 byte-nyi memóriát elfolyattunk, azaz nem szabadítottunk fel (leak). A tesztprogram 8. sorára ***Invalid read of size 1*** (érvénytelen 1 byte-os olvasás) üzenetet ad, ami jogos, hiszen 1 bájttal valóban túlindexeltük a puffert.

Még két tesztprogramot érdemes átnézni, az egyik a nem inicializált memória felderítését ismerteti *(3. Lista)*, ***a*** másik a felszabadított memória olvasásáról szól *(4. lista)*. Ezek a leggyakoribb ***C*** és ***C++*** hibák.

A Valgrind működése – A- és V-bitek

Ahhoz, hogy megtudjuk, mire alkalmas a program, elengedhetetlen megismerni a működését (az A- és V-biteket), ha pedig tudjuk hogyan működik, a korlátait is megismerhetjük.

A-bit (Valid-address, érvényes cím)

A *Valgrind* minden memóriacellához rendel egy A bitet, mely megmondja hogy a cella tartalma jogosan írható és olvasható-e. Ez nem mond információt a cella tartalmáról (inicializált-e), csak arról, hogy elvileg lehetséges-e írni bele és olvasni.
A cellatartalommal a V bitek foglalkoznak (lásd lejjebb).
Minden esetben, amikor a program egy memóriacellához fordul, a *Valgrind* ellenőrzi, hogy a hozzá tartozó A-bit érvényes-e. Ha nem, hibát jelez. Az írás-olvasás műveletek nem befolyásolják az A-bitek állapotát.

Hogyan állítódnak be az A-bitek?

- Amikor a program elindul, az összes globális adat A-bitje érvényes lesz.
- `malloc`/`new` után a lefoglalt terület A-bitjei érvényesek lesznek (csak azok) és `free`/`delete` esetén érvénytelenné válnak.
- Amikor a veremmutató (stack pointer) föl és le mozog, az A-bitek átállítódnak. A szabály az, hogy a veremmutató feletti A-bitek érvényesek, alatta érvénytelenek. Ennek megfelelően, amikor egy függvény befejeződik, az általa használt memóriaterület érvénytelenné válik, mert a veremmutató feljebb lép. Ez igencsak pontatlan kezelése a dolognak, mert nyugodtan felülírhatnánk mondjuk például a függvény visszatérési címét és még hibát sem kapnánk
- Rendszerhívásoknál a bitek megfelelően állítódnak be (például `mmap`).
- Opcionálisan a programunk is tud szólni a *Valgrindnek*, hogy változtassa át a biteket.

V-bit (Valid-value, érvényes érték)

A szintetikus *Valgrind* processzor szinte teljesen ugyanúgy működik, mint az eredeti, egy dolgot kivéve: minden egyes tárolt és feldolgozott bithez hozzárendel egy másik bitet, hogy a bit tartalma érvényes-e, vagy inicializálatlan. Ennek köszönhetően minden egyes bájthoz a memóriában 8 V-bit és 1 A-bit van rendelve, tehát kicsit több mint duplája lesz a memóriafogyasztás. A processzor regisztereihez is rendel V-biteket. Ez azért fontos, mert csak akkor jelez hibát a program, ha a V bitek alapján feltételes elágazáshoz ér. Addig mindenféle műveletet lehet velük végezni. Az indok

3. Lista – példa nem inicializált memória használatára (test2.c)

```
 1  #include <stdio.h>
 2
 3  int main() {
 4    /* 10 byte inicializálatlan memória */
 5    char buf[ 10 ];
 6    /* 5. elem betöltése egy másik változóba */
 7    int num = buf[ 5 ];
 8    /* hiba: elágazás ismeretlen tartalom alapján */
 9    if( num == 0 )
10      printf( "Az 5. elem tartalma 0.\n" );
11    else
12      printf( "Az 5. elem tartalma nem 0.\n" );
13  }

Eredménye:
...
==7904== Conditional jump or move depends on uninitialised
➥value(s)
==7904==     at 0x80483CC: main (test2.c:9)
...
```

4. Lista – példa felszabadított memória olvasására (test3.c)

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3
 4  int main() {
 5    /* 10 byte puffer lefoglalása */
 6    char *buf = malloc( 10 );
 7    /* kezdőérték a 7. elemnek */
 8    buf[ 7 ] = 23;
 9    /* felszabadítjuk a puffert */
10    free( buf );
11    /* hiba: kiolvassuk a tartalmát */
12    if( buf[ 7 ] == 23 )
13      printf( "Rendben.\n" );
14  }
...
==7972== Invalid read of size 1
==7972==     at 0x8048441: main (test3.c:12)
==7972==  Address 0x415D02F is 7 bytes inside a block of size
➥10 free'd
==7972==     at 0x401AF78: free (vg_replace_malloc.c:235)
==7972==     by 0x804843D: main (test3.c:10)
...
```

5. Lista – Példa a Valgrind hibáira (hiba.c)

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3
 4  char buf2[ 10 ];
 5  int  var = 0;
 6
 7  int main() {
 8    /* visszatérési cím olvasása (-1. byte) */
 9    char buf[ 10 ];
10    /* negatív irányba túlolvasás */
11    buf[ -1 ] = 10;
12    printf( "Buf[-1]: %d\n", buf[ -1 ] );
13    /* globális területen felülírok egy tömböt */
14    buf2[ 10 ] = 20;
15    /* a Valgrind egyikért sem szól! */
16  }

Eredmény:
nincs hiba
```

egyszerű: a jól megírt programokban is vannak olyan esetek, hogy inicializálatlan területeket olvasunk és írunk. Képzeljünk el egy struktúrát, amely 2 `int`-ből és 1 `char`-ból áll és a `char` középen van. Mivel a fordító az `int`-eket gyakran 4-8 bájtra igazítja, a középső 1 bájtos `char` után 3-7 bájt üresen maradhat. A struktúra `memcpy`-val történő másolása esetén az üres területek is másolásra kerülnek. Baj csak akkor van, ha a középső üres bájtok tartalma alapján döntést hozunk.

A V-bitek ellenőrzése 3 helyen történik:

- Memóriára hivatkozás esetén
- Feltételes elágazásoknál (`if`)
- Rendszerhívásoknál (például `getcwd`)

Ezekben az esetekben, ha az érték nincs inicializálva, hibát kapunk, majd a problémát okozó V-bitek érvényesre állítódnak, hogy többször ne jelezze ugyanazt a hibát.

A Valgrind hibái

A ***Valgrind*** sajnos korán sem tökéletes. Ahol hibát jelez, akkor ott hiba is van, de ha nem jelez semmit, az még nem jelenti azt, hogy a tesztelt alkalmazás jól működik. A ***hiba.c*** tesztprogram *(5. lista)* szemlélteti azokat a szituációkat, melyeket a ***Valgrind*** rendre elnéz. A legnagyobb hiányossága, hogy tömb indexeléseket nem ellenőrzi rendesen. Ez komoly probléma. Más kereskedelmi programok

6. Lista – Példa felszabadított memória olvasására (test4.c)

```
valgrind --tool=memcheck --leak-check=full test4

 1  #include <stdio.h>
 2  #include <stdlib.h>
 3
 4  char * notLeaked = 0;
 5  char * possiblyLeaked = 0;
 6
 7  int main() {
 8    char * pointer;
 9    /* 10 byte lefoglalása és eltárolása */
10    /* nem veszett el, mert mutat rá
➥pointer */
11    notLeaked = (char *)malloc( 10 );
12    /* másik 10 byte lefoglalása */
13    /* mutató csak a terület közepére mutat,
14      tehát valószínűleg elveszett */
15    possiblyLeaked = (char *)malloc( 10 ) + 5;
16    /* 10 byte lefoglalása és felszabadítása */
17    pointer = (char *)malloc( 10 );
18    free( pointer );
19    /* 10 byte lefoglalása, semmi nem mutat rá
➥*/
20    malloc( 10 );
21    /* 10 byte lefoglalása, csak lokális
➥változó
22       mutat rá, ami kilépéskor eltűnik.
➥A memóriában
23       viszont szemétként az értéke megmarad
24       (érvénytelen A-bittel). A Valgrind nem
➥jelez
25       hibát, tehát itt rosszul működik! */
26    pointer = (char *)malloc( 10 );
27  }

==7709== ERROR SUMMARY: 0 errors from 0 contexts
➥(suppressed: 13 from 1)
==7709== malloc/free: in use at exit: 40 bytes
➥in 4 blocks.
==7709== malloc/free: 5 allocs, 1 frees, 50
➥bytes allocated.
==7709== For counts of detected errors, rerun
➥with: -v
==7709== searching for pointers to 4 not-freed
➥blocks.
==7709== checked 57,568 bytes.
==7709==
==7709== 10 bytes in 1 blocks are definitely
➥lost in loss record 2 of 4
```

6. Lista – folytatás

```
==7709==    at 0x401A451: malloc
➥(vg_replace_malloc.c:149)
==7709==    by 0x8048437: main (test4.c:20)
==7709==
==7709==
==7709== 10 bytes in 1 blocks are possibly lost
➥in loss record 3 of 4
==7709==    at 0x401A451: malloc
➥(vg_replace_malloc.c:149)
==7709==    by 0x804840F: main (test4.c:15)
==7709==
==7709== LEAK SUMMARY:
==7709==    definitely lost: 10 bytes in 1
➥blocks.
==7709==      possibly lost: 10 bytes in 1
➥blocks.
==7709==    still reachable: 20 bytes in 2
➥blocks.
==7709==         suppressed: 0 bytes in 0
➥blocks.
==7709== Reachable blocks (those to which
➥a pointer was found) are not shown.
==7709== To see them, rerun with:
➥--show-reachable=yes
```

úgy csinálják, hogy fordítási időben extra sorokat raknak a programba indexek ellenőrzésére. Ez sajnos nagyon hiányzik a ***Valgrindből***, de még enélkül is rengeteg hiba felderítésére alkalmas. Ha van pár ezer eurónk megvásárolhatunk olyan termékeket, melyek sokkal jobb eredményt adnak, de amennyiben nincs erre lehetőség, meg kell elégedni ezzel a tudással. A munkahelyemen van ***Rational Purify*** licenszünk, így módomban állt a ***Valgrindet*** a ***Purify***-jal összehasonlítani. A meglepő az volt, hogy néhol a ***Valgrind***, néhol a ***Purify*** adott jobb eredményt, de nem mondhatnám egyértelműen, hogy a kereskedelmi ***Purify*** (ami szintén jó szoftver), jobban teljesített volna.
Másik hibája a ***Valgrindnek***, hogy kizárólag ***x86***-os 32 bites ***Linuxon*** fut, sem ***Windows***, sem ***Solaris***, sem egyéb operációs rendszereken nem megy.
A ***Valgrind*** gyorsan fejlődő projekt. Gyakran jelennek meg új verziók és a személyes véleményem az, hogy továbbra is fejleszteni fogják, mivel egyre több nyílt forrású projektnek lesz szüksége minőség-ellenőrzésre. Ebben a tekintetben pedig jelenleg egyedülálló. Remélhetőleg a következő verziókban már alaposabban fog tesztelni.

Memóriahasználat ellenőrzése

Miután a programunk összes fagyást okozó hibáját eltüntettük, tovább is javíthatunk a minőségen. Mert ugye jó az, ha megbízhatóan fut, de vajon miért is van 150 MB memóriára szüksége? A memóriahasználat helyrerakása persze már kevésbé fontos, mint a fagyások kijavítása. Nem minden programnak van szüksége komoly memória-ellenőrzésre.
Ha például egy ***ICQ*** kliens minden üzenetnél elpazarolna 1k-t, akkor is 1000 üzenet kellene ahhoz, hogy 1 megát elszivárogtasson. 1 mega pedig a mai gépek mellett nem jelentős memória. Szerver alkalmazások esetén már más a helyzet. Ott 10 bájt elherdálása is végzetes lehet, amennyiben ez sokszor (több milliószor!) történik. Általánosságban azokat a részeket kell leellenőriznünk, amelyeken a program rengetegszer végigfut.
A ***Valgrind*** sokat segíthet az eltűnt memória felkutatásában, mert pontosan megmondja, hogy hol foglalódott le és hogyan.
Háromféle memória-szivárgást különböztet meg:

- Biztosan elveszett (***definitely lost***), akkor ha a memória nem lett felszabadítva és rá mutató pointert sem talált sehol a memóriában.
- Valószínűleg elveszett (***possibly lost***), ha ugyan talál mutatót a memóriablokkra, csak az nem a blokk elejére, hanem máshova mutat
- Még elérhető (***still reachable***), ha talál a memóriában mutatót az adott blokk elejére

Amennyiben szeretnénk megtudni, hogy hol tűnt el a memória, akkor indítás előtt adjuk meg a `--leak-check=full` opciót is.
Példát a memória-szivárgásra a *6. lista* mutat.

Végszó

A ***Valgrind*** szerintem nagyon jól program és ha lehetőség van rá, érdemes használnunk. Bár minden előforduló hibát nem mutat meg, de azért nagyon sok helyen figyelmeztet, ha baj van. A hibák javításával szoftvereink minőségét jelentősen megnövelhetjük. Sajnos vannak olyan hibák is – nem is kevés – amelyeket nem mi okozunk, hanem más rendszerkomponens (***KDE***, ***X***, ***QT***,...). Ezekkel a hibákkal nem sok mindent tudunk kezdeni, legfeljebb örülhetünk, ha nem fagy miatta a program. A lényeges az, hogy amit mi csinálunk, azt ellenőrizzük le és javítsuk, ha rossz.
A ***Valgrind*** rengeteg más hasznos funkcióval is rendelkezik (heap ellenőrzés, holtpont vizsgálat, processzor cache elemzés,...), melyeket hely hiányában nem fejtettem ki. Akit komolyabban érdekel, elolvashatja a 184 oldalas dokumentációját. Egyszer legalább érdemes átfutni rajta, mert nagyon sok érdekes apró részletet elárul. Sok sikert kívánok a ***Valgrind*** használatához és még több jó minőségű szoftvert ***Linux*** alá!

Karai Csaba
(cskarai@freemail.hu)

Informatikus vagyok egy mobiltelefonokkal foglalkozó vállalatnál, szabadidőmet legszívesebben feleségemmel töltöm, de szeretek focizni, táncolni, biciklizni és görkorizni is.