Gestiunea memoriei

Contents

Gestiunea memoriei

Subsistemul de gestiune a memoriei din cadrul unui sistem de operare este folosit de toate celelalte subsisteme: scheduling, I/O, filesystem, gestiunea proceselor, networking. Memoria este o resursă importantă i sunt necesari algoritmi eficieni de utilizare i gestiune a acesteia.

Rolul subsistemului de gestiune a memoriei este de a ine evidena zonelor de memorie fizică ocupate sau libere, de a oferi proceselor sau celorlalte subsisteme acces la memorie i de a mapa paginile de memorie virtuală ale unui proces (pages) peste paginile fizice (frames).

Nucleul sistemului de operare oferă un set de interfee (apeluri de sistem) care permit alocarea/dezalocarea de memorie, maparea unor regiuni de memorie virtuală peste fiiere, partajarea zonelor de memorie.

Din păcate, nivelul limitat de înelegere a acestor interfee i a aciunilor ce se petrec în spate conduc la o serie de probleme foarte des întâlnite în aplicaiile software: memory leak-uri, accese invalide, suprascrieri, buffer overflow, corupere de zone de memorie.

Este, în consecină, fundamentală cunoaterea contextului în care acionează subsistemul de gestiune a memoriei i înelegerea interfeei pusă la dispoziie de sistemul de operare programatorului.

Spaiul de adresă al unui proces

Spaiul de adresă al unui proces
Spaiul de adrese al unui proces, sau, mai bine spus, spaiul virtual de adresă al unui proces reprezintă zona de memorie virtuală utilizabilă de un proces. Fiecare proces are un spaiu de adresă propriu. Chiar în situaiile în care două procese partajează o zonă de memorie, spaiul virtual este distinct, dar se mapează peste aceeai zonă de memorie fizică.

În figura alăturată este prezentat un spaiu de adresă tipic pentru un proces. În sistemele de operare moderne, în spaiul virtual al fiecărui proces se mapează memoria nucleului, aceasta poate fi mapată fie la începutul fie la sfâritul spaiului de adresă. (Note). În continuare ne vom referi numai la spaiul de adresă din user-space pentru un proces.

Cele 4 zone importante din spaiul de adresă al unui proces sunt zona de date, zona de cod, stiva i heap-ul. După cum se observă i din figură, stiva i heap-ul sunt zonele care pot crete. De fapt, aceste două zone sunt dinamice i au sens doar în contextul unui proces. De partea cealaltă, informaiile din zona de date i din zona de cod sunt descrise în executabil.

Zona de cod

Zona/segmentul de cod (denumit i 'text segment') reprezintă instruciunile programului. Registrul de tip 'instruction pointer' va referi adrese din zona de cod. Se citete instruciunea indicată, se decodifică i se interpretează, după care se incrementează contorul programului i se trece la următoarea instruciune. Zona de cod este, de obicei, o zonă read-only.

Zone de date

Zonele de date conin variabilele globale definite într-un program i variabilele de tipul read-only. În funcie de tipul de date există mai multe sub-tipuri de zone de date.

.data

Zona .data conine variabilele globale iniializate la valori nenule ale unui program. De exemplu:

```
static int a = 3;
char b = 'a';
```

.bss

Zona .bss conine variabilele globale neiniializate sau iniializate la zero ale unui program. De exemplu:

```
static int a;
char b;
```

În general acestea nu vor fi prealocate în executabil ci în momentul creării precesului. Alocare zonei .bss se face peste pagini fizice zero (zeroed frames).

.rodata

Zona .rodata conine informaie care poate fi doar citită, nu i modificată. Aici sunt stocate constantele:

```
const int a;
const char *ptr;
```

i literalii:

```
"Hello, World!"
"En Taro Adun!"
```

Stiva

Stiva este o regiune dinamica în cadrul unui proces. Stiva este folosită pentru a reine "stack frame-urile" (link) în cazul apelurilor de funcii i pentru a stoca variabilele locale. Pe marea majoritate a arhitecturilor moderne

stiva crete în jos i heap-ul crete în sus. Stiva este gestionată automat de compilator. La fiecare revenire din funcie stiva este golită.

În figura de mai jos este prezentată o vedere conceptuală asupra stivei in momentul apelului unei funcții.

Heap-ul

Heap-ul este zona de memorie dedicată alocării dinamice a memoriei. Heap-ul este folosit pentru alocarea de regiuni de memorie a căror dimensiune se află doar la runtime.

La fel ca i stiva, heap-ul este o regiune dinamică i care îi modifică dimensiunea. Spre deosebire de stivă, însă, heap-ul nu este gestionat de compilator. Este de datoria programatorului să tie câtă memorie trebuie să aloce i să reină cât a alocat i când trebuie să dezaloce. Problemele frecvente în majoritatea programelor in de pierderea referinelor la zonele alocate (memory leaks) sau referirea de zone nealocate sau insuficient alocate (accese invalide).

La limbaje precum Java, Lisp, etc. unde nu există "pointer freedom", eliberarea spaiului alocat se face automat prin intermediul unui garbage collector (link). Pe aceste sisteme se previne problema pierderii referinelor, dar încă rămâne activă problema referirii zonelor nealocate.

Alocarea memoriei

Alocarea memoriei este realizată static de compilator sau dinamic, în timpul execuiei. Alocarea statică este realizată în segmentele de date pentru variabilele locale sau pentru literali.

În timpul execuiei, variabilele se alocă pe stivă sau în heap. Alocarea pe stivă se realizează automat de compilator pentru variabilele locale unei functii (mai putin variabilele locale prefixate de identificatorul **static**).

Alocarea dinamică se realizează în heap. Alocarea dinamică are loc atunci când nu se tie în momentul compilării câtă memorie va fi necesară pentru o variabilă, o structură, un vector. Dacă se tie din momentul

compilării cat spaiu va ocupa o varibilă, se recomandă alocarea ei statică, pentru a preveni erorile frecvente apărute în contextul alocării dinamice.

Pentru a fragmenta cât mai puin spaiul de adrese al procesului, ca urmare a alocărilor i dezalocărilor unor zone de dimensiuni variate, alocatorul de memorie va organiza segmentul de date alocate dinamic sub formă de heap, de unde i numele segmentului.

Alocarea memoriei în Linux

În Linux alocarea memoriei pentru procesele utilizator se realizează prin intermediul funciilor de bibliotecă `malloc`, `calloc` i `realloc` iar dezalocarea ei prin intermediul funciei `free`. Aceste funcii reprezintă apeluri de bibliotecă i rezolvă cererile de alocare i dezalocare de memorie pe cât posibil în user space. Aadar, se in nite tabele care specifică zonele de memorie alocate în heap. Dacă există zone libere pe heap, un apel `malloc` care cere o zonă de memorie care poate fi încadrată într-o zonă liberă din heap va fi satisfăcut imediat marcând în tabel zona respectivă ca fiind alocată i întorcând programului apelant un pointer spre ea. Dacă în schimb se cere o zonă care nu încape în nicio zonă liberă din heap, `malloc` va încerca extinderea heap-ului prin apelul de sistem `brk` sau `mmap`.

```
void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
void free(void *ptr);
```

Întotdeauna eliberai (`free`) memoria alocată. Memoria alocată de proces este eliberată automat la terminarea procesului însă în cazul unui proces server, de exemplu, care rulează foarte mult timp i nu eliberează memoria alocată acesta va ajunge să ocupe toată memoria disponibilă în sistem cauzând astfel consecine nefaste. Atenie să nu eliberai de două ori aceeai zonă de memorie întrucât acest lucru va avea drept urmare coruperea tabelelor inute de `malloc` ceea ce va duce din nou la consecine nefaste. Întrucât funcia `free` se întoarce imediat dacă primete ca parametru un pointer `NULL`, este recomandat ca după un apel `free`, pointer-ul să fie resetat la `NULL`.

Câteva exemple de alocare a memoriei sunt prezentate în continuare:

```
int n = atoi(argv[1]);
char *str;

/* de obicei mallocul primete argumentul de spaiu în forma size_elems * num_elems */
str = (char *) malloc((n + 1) * sizeof(char));
if (NULL == str) {
perror("malloc");
exit(EXIT_FAILURE);
}

[...]

free(str);
str = NULL;

---

/* crearea unui vector de siruri ce contine doar argumentele unui program */

char **argv_no_exec;
```

```
/* se aloca spatiu pentru argumente */
argv_no_exec = (char **) malloc((argc - 1) * sizeof(char*));
if (NULL == argv_no_exec) {
perror("malloc");
exit(EXIT_FAILURE);
}

/* se creeaza referinte catre argumente */
for (i = 1; i < argc; i++)
        argv_no_exec[i - 1] = argv[i];

[...]

free(argv_no_exec);
argv_no_exec = NULL;
```

Apelul `realloc` este folosit pentru modificarea spatiului de memorie alocat dupa un apel `malloc`:

```
int *p;

p = (int *) malloc(n * sizeof(int));
if (NULL == p) {
perror("malloc");
exit(EXIT_FAILURE);
}

[...]

p = (int *) realloc(p, (n + extra) * sizeof(int));

[...]

free(p);
p = NULL;
```

Apelul `calloc` este folosit pentru alocarea de zone de memorie al căror coninut este nul (plin de valori de zero). Spre deosebire de `malloc`, apelul va primi două argumente: numărul de elemente i dimensiunea unui element.

```
list_t *list_v; /* list_t poate fi orice tip de date din C (mai putin void) */

list_v = (list_t *) calloc(n, sizeof(list_t));
if (NULL == list_v) {
perror("calloc");
exit(EXIT_FAILURE);
}

[...]

free(p);
p = NULL;
```

Mai multe informaii găsii în manualul bibliotecii standard C i în pagina de manual **man malloc**.

Alocarea memoriei în Windows

În Windows un proces poate să-i creeze mai multe obiecte `Heap` pe lângă `Heap`-ul cu care este creat procesul. Acest lucru este foarte util în momentul în care o aplicaie alocă i dezalocă foarte multe zone de memorie cu câteva dimensiuni fixe. Aplicaia poate să-i creeze câte un `Heap` pentru fiecare dimensiune i, în cadrul fiecărui `Heap`, să aloce zone de aceeai dimensiune reducând astfel la maxim fragmentarea heapului.

Pentru crearea, respectiv distrugerea, unui `Heap` se vor folosi funciile `HeapCreate` i `HeapDestroy`:

```
HANDLE HeapCreate(
        DWORD flOptions,
        SIZE_T dwInitialSize,
        SIZE_T dwMaximumSize
);

BOOL HeapDestroy(
        HANDLE hHeap
);
```

Pentru a obine un descriptor al heapului cu care a fost creat procesul (în cazul în care nu dorim crearea altor heapuri) se va apela funcia `GetProcessHeap`. Pentru a obine descriptorii tuturor heapurilor procesului se va apela `GetProcessHeaps`.

Exista, de asemenea funcii care enumeră toate blocurile alocate într-un heap, validează unul sau toate blocurile alocate într-un heap sau întorc dimensiunea unui bloc pe baza descriptorului de heap i a adresei blocului: HeapWalk, HeapSize, HeapValidate.

Pentru alocarea, dezalocarea, redimensionarea unui bloc de memorie din `Heap`, Windows pune la dispoziia programatorului funciile `HeapAlloc`, `HeapFree`, respectiv `HeapReAlloc`, cu signaturile de mai jos:

```
LPVOID HeapAlloc(
        HANDLE hHeap,
        DWORD dwFlags,
        SIZE_T dwBytes
);

BOOL HeapFree(
        HANDLE hHeap,
        DWORD dwFlags,
        LPVOID lpMem
);

LPVOID HeapReAlloc(
        HANDLE hHeap,
        DWORD dwFlags,
        LPVOID lpMem,
        SIZE_T dwBytes
);
```

Câteva exemple de folosire a acestor funcii sunt prezentate în continuare:

```
/* alocarea unui vector de intregi */
HANDLE processHeap;
DWORD *data;

processHeap = GetProcessHeap();
if (NULL == processHeap) {
fprintf(stderr, "GetProcessHeap failed with error %ud.\n", GetLastError());
        ExitProcess(1);
```

```
}

data = HeapAlloc(processHeap, HEAP_ZERO_MEMORY, count * sizeof(DWORD));
if (NULL == data) {
fprintf(stderr, "HeapAlloc failed with error %ud.\n", GetLastError());
        ExitProcess(1);
}

[...]

if (HeapFree(processHeap, 0, data) == FALSE) {
fprintf(stderr, "HeapFree failed with error %ud.\n", GetLastError());
        ExitProcess(1);
}

---

/* alocarea unei matrice */
HANDLE processHeap;
DWORD **mat;
INT m, n;
INT i;

processHeap = GetProcessHeap();
if (NULL == processHeap) {
fprintf(stderr, "GetProcessHeap failed with error %ud.\n", GetLastError());
        ExitProcess(1);
}

mat = HeapAlloc(processHeap, 0, m * sizeof(*mat));
if (NULL == mat) {
fprintf(stderr, "HeapAlloc failed with error %ud.\n", GetLastError());
        ExitProcess(1);
}

for (i = 0; i < n; i++) {
   [i] =mat HeapAlloc(processHeap, 0, n * sizeof(**mat));
if (NULL == mat) {
fprintf(stderr, "HeapAlloc failed with error %ud.\n", GetLastError());
goto freeMem;
}
}

[...]
freeMem:
for (j = 0; j < i; j++)
        HeapFree(processHeap, 0, mat[j]);
HeapFree(processHeap, 0, mat);
```

Pe sistemele Windows se pot folosi i funciile bibliotecii standard C pentru gestiunea memoriei: `malloc`, `realloc`, `calloc`, `free`, dar apelurile de sistem specifice Windows oferă funcionalităi suplimentare i nu implică legarea bibliotecii standard C în executabil.

Dezalocarea memoriei

Pentru dezalocarea memoriei, se folosesc funciile `free`, respectiv `HeapFree`. Funciile primesc ca argument un pointer la un spaiu de memorie alocat anterior cu o funcie de alocare.

Dacă se omite dezalocarea unei zone de memorie, aceasta va rămâne alocată pe întreaga durata de rulare a procesului. Ori de câte ori nu mai este nevoie de o zonă de memorie, aceasta trebuie dezalocată pentru eficiena utilizării spaiului de memorie.

Nu trebuie neapărat realizată dezalocarea diverselor zone înainte de un apel `exit` sau `ExitProcess` sau înainte de încheierea programului pentru că acestea sunt automat eliberate de sistemul de operare.

Probleme pot apărea i dacă se încearcă dezalocarea aceleiai regiuni de memorie de două sau mai multe ori i se corup listele

Probleme de lucru cu memoria

Lucrul cu heap-ul este una dintre cauzele principale ale apariiilor problemelor de programare. Lucrul cu pointerii, necesitatea folosirii unor apeluri de sistem/bibliotecă pentru alocare/dezalocare, pot conduce la o serie de probleme care afectează (de multe ori fatal) funcionarea unui program.

Problemele cele mai des întâlnite în lucrul cu memoria sunt:

- accesul invalid la memorie
- leak-urile de memorie

Accesul invalid la memorie prespune accesarea unor zone care nu au fost alocate sau au fost eliberate. Leak-urile de memorie sunt situaiile în care se pierde referina la o zonă alocată anterior. Acea zonă va rămâne ocupată până la încheierea procesului. Ambele probleme i utilitarele care pot fi folosite pentru combaterea acestora vor fi prezentate în continuare.

Acces invalid

De obicei, accesarea unei zone de memorie invalide rezultă într-o eroare de pagină (page fault) i terminarea procesului (în Unix înseamnă trimiterea semnalului SIGSEGV - afiarea mesajului 'Segmentation fault'). Totui, dacă eroarea apare la o adresă invalidă dar într-o pagină validă, hardware-ul i sistemul de operare nu vor putea sesiza aciunea ca fiind invalidă. Acest lucru se datorează faptului că alocarea memoriei se face la nivel de pagină. Pot exista situaii în care să fie folosită doar jumătate din pagină. Dei cealaltă jumătate conine adrese invalide, sistemul de operare nu va putea detecta accesele invalide la acea zonă.

Asemenea accese pot duce la coruperea heap-ului i la pierderea consistenei memoriei alocate. După cum se va vedea în continuare, există utilitare care ajută la detectarea acestor situaii.

Un tip special de acces invalid este buffer overflow. Acest tip de atac presupune referirea unor regiuni valide din spaiul de adresă al unui proces prin intermediul unei variabile care nu ar trebui să poată referenia aceste adrese. De obicei, un atac de tip buffer overflow rezultă în rularea de cod nesigur. Protecia la accese de tip buffer overflow se realizează prin verificarea limitelor unui buffer/vector fie la compilare, fie la rulare.

GDB - Detectarea zonei de acces invalid de tip page fault

Pe lângă facilităi de bază precum urmărirea unei variabile sau configurarea de puncte de oprire (breakpoints), GDB pune la dispoziia utilizatorilor i comenzi avansate, utile în anumite cazuri. Comanda `disassamble` poate fi folosită pentru a afisa codul mainã generat de compilator. Comanda `info reg` afieazã coninutul registrelor. Aceste comenzi sunt folosite rar, atunci când utilizatorul încearcă să depaneze codul generat de compilator, sau când are pări din program scrise direct în asamblare.

O comandă foarte utilă atunci când se depanează programe complexe este `backtrace`. Această comandă afieazã toate apelurile de funcii în curs de execuie.

Exemplu: fibonacci_gdb_test.c

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

 int fibonacci(int no)
 {
if (1 == no || 2 == no)
return 1;
return fibonacci(no-1) + fibonacci(no-2);
 }

 int main()
 {
short int numar, baza=10;
char sir[1];

scanf("%s", sir);
      strtol(sir, NULL, baza);
printf("fibonacci(%d)=%d\n", numar, fibonacci(numar));
return 0;
 }
```

Pentru exemplul de mai sus, vom demonstra utilitatea comenzii `backtrace`:

```
[tavi@dhcp-48 intro]$ gcc -Wall exemplul-7.c -g
[tavi@dhcp-48 intro]$ gdb a.out
(gdb) break 8
Breakpoint 1 at 0x8048482: file exemplul-7.c, line 8.
(gdb) run
Starting program: /home/tavi/cursuri/so/lab/draft/intro/a.out
7

Breakpoint 1, fibonacci (no=2) at exemplul-7.c:8
8                       return 1;
(gdb) bt
#0  fibonacci (no=2) at exemplul-7.c:8
#1  0x0804849d in fibonacci (no=3) at exemplul-7.c:9
#2  0x0804849d in fibonacci (no=4) at exemplul-7.c:9
#3  0x0804849d in fibonacci (no=5) at exemplul-7.c:9
#4  0x0804849d in fibonacci (no=6) at exemplul-7.c:9
#5  0x0804849d in fibonacci (no=7) at exemplul-7.c:9
#6  0x0804851c in main () at exemplul-7.c:20
#7  0x4003d280 in __libc_start_main () from /lib/libc.so.6
(gdb)
```

Se observă că la afiarea apelurilor de funcii se afiează i parametrii cu care a fost apelată funcia. Acest lucru este posibil datorită faptului că atât variabilele locale cât i parametrii acesteia sunt păstrai pe stivă până la ieirea din funcie.

Fiecare funcie are alocată pe stivă un frame, în care sunt plasate variabilele locale funciei, parametrii pasai funciei i adresa de revenire din functie. În momentul în care o funcie este apelată, se creează un nou frame prin alocarea de spaiu pe stivă de către funcia apelată. Astfel, dacă avem apeluri de funcii imbricate, atunci stiva va conine toate frame-urile tuturor funciilor apelate imbricat.

GDB dă posibilitatea utilizatorului să examineze frame-urile prezente în stivă. Astfel, utilizatorul poate alege oricare din frame-urile prezente folosind comanda `frame`. După cum s-a observat, exemplul anterior are un bug ce se manifestă atunci când numărul introdus de la tastatură depăete dimensiunea buffer-ului alocat (static). Acest tip de eroare poartă denumirea de *buffer overflow* i este extrem de gravă. Cele mai multe atacuri de la distană pe un sistem sunt cauzate de acest tip de erori. Din păcate, acest tip de eroare nu este uor de detectat, pentru că în procesul de buffer overrun se pot suprascrie alte variabile, ceea ce duce la detectarea erorii nu imediat când s-a făcut suprascrierea, ci mai târziu, când se va folosi variabila afectat.

```
[tavi@tropaila intro]$ gdb a.out
(gdb) run
Starting program: /home/tavi/cursuri/so/lab/draft/intro/a.out
10

Program received signal SIGSEGV, Segmentation fault.
0x08048497 in fibonacci (no=-299522) at exemplul-7.c:9
9               return fibonacci(no-1) + fibonacci(no-2);
(gdb) bt -5

#299520 0x0804849d in fibonacci (no=-2) at exemplul-7.c:9
#299521 0x0804849d in fibonacci (no=-1) at exemplul-7.c:9
#299522 0x0804849d in fibonacci (no=0) at exemplul-7.c:9
#299523 0x0804851c in main () at exemplul-7.c:20
#299524 0x4003e280 in __libc_start_main () from /lib/libc.so.6
```

Din analiza de mai sus se observă că funcia fibonacci a fost apelată cu valoarea 0. Cum funcia nu testează ca parametrul să fie valid, se va apela recursiv de un număr suficient de ori pentru a cauza umplerea stivei programului. Se pune problema cum s-a apelat funcia cu valoarea 0, când trebuia apelată cu valoarea 10.

```
[tavi@dhcp-48 intro]$ gdb a.out
(gdb) run
Starting program: /home/tavi/cursuri/so/lab/draft/intro/a.out
10

Program received signal SIGSEGV, Segmentation fault.
0x08048497 in fibonacci (no=-299515) at exemplul-7.c:9
9               return fibonacci(no-1) + fibonacci(no-2);
(gdb) bt -2
#299516 0x0804851c in main () at exemplul-7.c:20
#299517 0x4003d280 in __libc_start_main () from /lib/libc.so.6
(gdb) fr 299516
#299516 0x0804851c in main () at exemplul-7.c:20
20              printf("fibonacci(%d)=%d\n", numar, fibonacci(numar));
(gdb) print numar
$1 = 0
(gdb) print baza
$2 = 48
(gdb)
```

Se observă că problema este cauzată de faptul că variabila baza a fost alterată. Pentru a determina când s-a întâmplat acest lucru, se poate folosi comanda `watch`. Această comandă primete ca parametru o expresie i va opri execuia programului de fiecare dată când valoarea expresiei se schimbă.

```
(gdb) quit
[tavi@dhcp-48 intro]$ gdb a.out
(gdb) break main
Breakpoint 1 at 0x80484d6: file exemplul-7.c, line 15.
(gdb) run
Starting program: /home/tavi/cursuri/so/lab/draft/intro/a.out

Breakpoint 1, main () at exemplul-7.c:15
15              short int numar, baza=10;
(gdb) n
18              scanf("%s", sir);
(gdb) watch baza
Hardware watchpoint 2: baza
(gdb) continue
Continuing.
10
Hardware watchpoint 2: baza

Old value = 10
New value = 48
0x40086b41 in _IO_vfscanf () from /lib/libc.so.6
(gdb) bt
#0  0x40086b41 in _IO_vfscanf () from /lib/libc.so.6
#1  0x40087259 in scanf () from /lib/libc.so.6
#2  0x080484ed in main () at exemplul-7.c:18
#3  0x4003d280 in __libc_start_main () from /lib/libc.so.6
(gdb)
```

Din analiza de mai sus se observă că valoarea variabilei este modificată în funcia `_IO_vfscanf`, care la rândul ei este apelată de către functia `scanf`. Dacă se analizează apoi parametrii pasai functiei `scanf` se observ imediat cauza erorii.

Pentru mai multe informaii despre GDB consultai manualul online (alternativ pagina info - `info gdb`) sau folosii comanda `help` din cadrul GDB.

mcheck - verificarea consistenei heap-ului

glibc permite verificarea consistenei heap-ului prin intermediul apelului mcheck definit în `mcheck.h`. Apelul mcheck forează `malloc` să execute diverse verificări de consistenă precum scrierea peste un bloc alocat cu `malloc`.

Alternativ, se poate folosi opiunea `-lmcheck` la legarea programului fără a afecta sursa acestuia.

Varianta cea mai simplă este folosirea variabilei de mediu MALLOC_CHECK_. Dacă un program va fi lansat în execuie cu variabila `MALLOC_CHECK_` configurată, atunci vor fi afiate mesaje de eroare (eventual programul va fi terminat forat - aborted).

Mai jos se găsete un exemplu de cod cu probleme în alocarea i folosirea heap-ului:

Exemplu: mcheck_test.c

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

int main(void)
{
int *v1;

     (int*) malloc(5 * sizeof(*v1));
if (NULL == v1) {
perror("malloc");
exit (EXIT_FAILURE);
}

/* overflow */
  [6] = 100;

free(v1);

/* write after free */
  [6] = 100;

/* reallocate v1 */
     malloc(10 * sizeof(int));
if (NULL == v1) {
perror("malloc");
exit (EXIT_FAILURE);
}

return 0;
}
```

Mai jos programul este compilat i rulat. Mai întâi este rulat fără opiuni de mcheck, după care se definete variabila de mediu MALLOC_CHECK_ la rularea programului. Se observă că dei se depăete spaiul alocat pentru vectorul `v1` i se referă vectorul _după_ eliberarea spaiului, o rulare simplă nu rezultă în afiarea nici unei erori.

Totui, dacă definim variabila de mediu `MALLOC_CHECK_`, se detectează cele două erori. De observat că o eroare este detectată doar în momentul unui nou apel de memorie interceptat de mcheck.

```
razvan@valhalla:~/school/2007-2008/so/labs/lab4/samples/mcheck$ make
cc -Wall -g    mcheck_test.c   -o mcheck_test
razvan@valhalla:~/school/2007-2008/so/labs/lab4/samples/mcheck$ ./mcheck_test
razvan@valhalla:~/school/2007-2008/so/labs/lab4/samples/mcheck$ MALLOC_CHECK_=1 ./mcheck_test
malloc: using debugging hooks
*** glibc detected *** ./mcheck_test: free(): invalid pointer: 0x0000000000601010 ***
*** glibc detected *** ./mcheck_test: malloc: top chunk is corrupt: 0x0000000000601020 ***
```

mcheck nu este o soluie completă i nu detectează toate erorile ce pot apărea în lucrul cu memoria. Detectează, totui, un număr important de erori i reprezintă o facilitate importantă a glibc.

O descriere completă găsii în pagina asociată din manualul glibc.

Leak-uri de memorie

Un leak de memorie apare în două situaii:

- un program omite să elibereze o zonă de memorie
- un program pierde referina la o zonă de memorie dealocată i, drept consecină, nu o poate elibera

Memory leak-urile au ca efect reducerea cantităii de memorie existentă în sistem. Se poate ajunge, în situaiile extreme, la consumarea întregii memorii a sistemului i la imposibilitatea de funcionare a diverselor aplicaii ale acestuia.

Ca i în cazul problemei accesului invalid la memorie, utilitarul Valgrind este foarte util în detectarea leak-urilor de memorie ale unui program.

mtrace

Un utilitar care poate fi folosit la depanarea erorilor de lucru cu memoria este mtrace. Acest utilitar ajută la identificarea leak-urilor de memorie ale unui program.

Utilitarul mtrace se folosete cu apelurile **mtrace** i **muntrace** implementate în biblioteca standard C:

```
#include <mcheck.h>

void mtrace(void);
void muntrace(void);
```

Utilitarul mtrace introduce handlere pentru apelurile de biblioteca de lucru cu memoria (`malloc`, `realloc`, `free`). Apelurile `mtrace` i `muntrace` activează, respectiv dezactivează monitorizarea apelurilor de bibliotecă de lucru cu memoria.

Jurnalizarea operaiilor efectuate se realizează în fiierul definit de variabile de mediu MALLOC_TRACE. După ce apelurile au fost înregistrate în fiierul specificat, utilizatorul poate să folosească utilitarul `mtrace` pentru analiza acestora.

În exemplul de mai jos este prezentată o situaie în care se alocă memorie fără a fi eliberată:

Exemplu: mtrace_test.c

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

int main(void)
{
/* start memcall monitoring */
      mtrace();

malloc(10);
malloc(20);
malloc(30);

/* stop memcall monitoring */
        muntrace();

return 0;
}
```

În secvena de comenzi de mai jos se compilează fiierul de mai sus, se stabilete fiierul de jurnalizare i se rulează comanda `mtrace` pentru a detecta problemele din codul de mai sus.

```
$ gcc -Wall -g mtrace_test.c -o mtrace_test
$ export MALLOC_TRACE=./mtrace.log
$ ./mtrace_test
$ cat mtrace.log
= Start
@ ./mtrace_test:[0x40054b] + 0x601460 0xa
@ ./mtrace_test:[0x400555] + 0x601480 0x14
@ ./mtrace_test:[0x40055f] + 0x6014a0 0x1e
= End
$ mtrace mtrace_test mtrace.log

Memory not freed:
-----------------
                   Address     Size     Caller
0x0000000000601460      0xa  at /home/razvan/school/2007-2008/so/labs/lab4/samples/mtrace.c:11
0x0000000000601480     0x14  at /home/razvan/school/2007-2008/so/labs/lab4/samples/mtrace.c:12
0x00000000006014a0     0x1e  at /home/razvan/school/2007-2008/so/labs/lab4/samples/mtrace.c:15
```

Mai multe informaii despre detectarea problemelor de alocare folosind mtrace gasiti în pagina asociată din manualul glibc.

Dublă dezalocare

Denumirea "dublă dezalocare" oferă o bună intuiie asupra cauzei: eliberarea de două ori a aceluiai spaiu de memorie. Dubla dezalocare poate avea efecte negative deoarece afectează structurile interne folosite pentru a gestiona memoria ocupată.

În ultimele versiuni ale bibliotecii standard C se detectează automat cazurile de dublă dezalocare. Fie exemplul de mai jos:

```
#include <stdlib.h>

int main(void)
{
char *p;

    mallpc(10);
free(p);
free(p);

return 0;
}
```

Rularea executabilului obinut din programul de mai sus duce la afiarea unui mesaj specific al glibc de eliberare dublă a unei regiuni de memorie i terminarea programului:

```
razvan@valhalla:~/school/2007-2008/so/labs/lab4/samples/dfree$ make
cc -Wall -g    dfree.c   -o dfree
razvan@valhalla:~/school/2007-2008/so/labs/lab4/samples/dfree$ ./dfree
*** glibc detected *** ./dfree: double free or corruption (fasttop): 0x0000000000601010 ***
======= Backtrace: =========
/lib/libc.so.6[0x2b675fdd502a]
```

```
/lib/libc.so.6(cfree+0x8c)[0x2b675fdd8bbc]
./dfree[0x400510]
/lib/libc.so.6(__libc_start_main+0xf4)[0x2b675fd7f1c4]
./dfree[0x400459]
```

Situaii de dezalocare sunt, de asemenea, detectate de Valgrind.

Valgrind

Valgrind reprezintă o suită de utilitare folosite pentru operaii de debugging i profiling. Cel mai popular este Memcheck, un utilitar care permite detectarea de erori de lucru cu memoria (accese invalide, memory leak-uri etc.). Alte utilitare din suita Valgrind sunt Cachegrind, Callgrind utile pentru profiling sau Helgrind, util pentru depanarea programelor multithreaded.

În continuare ne vom referi doar la utilitarul Memcheck de detectare a erorilor de lucru cu memoria. Mai precis, acest utilitar detectează următoarele tipuri de erori:

- folosirea de memorie neiniializată
- citirea/scrierea din/în memorie după ce regiunea respectivă a fost eliberată
- citirea/scrierea dincolo de sfâritul zonei alocate
- citirea/scrierea pe stivă în zone necorespunzătoare
- memory leak-uri
- folosirea necorespunzătore de apeluri malloc/new i free/delete

Valgrind nu necesită adaptarea codului unui program ci folosete direct executabilul (binarul) asociat unui program. La o rulare obinuită Valgrind va primi argumentul `--tool` pentru a preciza utilitarul folosit i programul care va fi verificat de erori de lucru cu memoria.

În exemplul de rulare de mai jos se folosete programul prezentat la seciunea mcheck:

```
razvan@valhalla:~/school/2007-2008/so/labs/lab4/samples/mcheck$ valgrind --tool=memcheck ./mcheck
==17870== Memcheck, a memory error detector.
==17870== Copyright (C) 2002-2007, and GNU GPL'd, by Julian Seward et al.
==17870== Using LibVEX rev 1804, a library for dynamic binary translation.
==17870== Copyright (C) 2004-2007, and GNU GPL'd, by OpenWorks LLP.
==17870== Using valgrind-3.3.0-Debian, a dynamic binary instrumentation framework.
==17870== Copyright (C) 2000-2007, and GNU GPL'd, by Julian Seward et al.
==17870== For more details, rerun with: -v
==17870==
==17870== Invalid write of size 4
==17870==    at 0x4005B1: main (mcheck_test.c:17)
==17870==  Address 0x5184048 is 4 bytes after a block of size 20 alloc'd
==17870==    at 0x4C21FAB: malloc (vg_replace_malloc.c:207)
==17870==    by 0x400589: main (mcheck_test.c:10)
==17870==
==17870== Invalid write of size 4
==17870==    at 0x4005C8: main (mcheck_test.c:22)
==17870==  Address 0x5184048 is 4 bytes after a block of size 20 free'd
==17870==    at 0x4C21B2E: free (vg_replace_malloc.c:323)
==17870==    by 0x4005BF: main (mcheck_test.c:19)
==17870==
==17870== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 8 from 1)
==17870== malloc/free: in use at exit: 40 bytes in 1 blocks.
==17870== malloc/free: 2 allocs, 1 frees, 60 bytes allocated.
==17870== For counts of detected errors, rerun with: -v
```

```
==17870== searching for pointers to 1 not-freed blocks.
==17870== checked 76,408 bytes.
==17870==
==17870== LEAK SUMMARY:
==17870==    definitely lost: 40 bytes in 1 blocks.
==17870==      possibly lost: 0 bytes in 0 blocks.
==17870==    still reachable: 0 bytes in 0 blocks.
==17870==         suppressed: 0 bytes in 0 blocks.
==17870== Rerun with --leak-check=full to see details of leaked memory.
```

S-a folosit utilitarul `memcheck` pentru obinerea informaiilor de acces la memorie.

Se recomandă folosirea opiunii `-g` la compilarea programului pentru a prezenta în executabil informaii de depanare. În rularea de mai sus, Valgrind a identificat două erori: una apare la linia 17 de cod i este corelată cu linia 10 (`malloc`), iar cealaltă apare la linia 22 i este corelată cu linia 19 (`free`):

```
10     (int *) malloc (5 * sizeof(*v1));
11if (NULL == v1) {
12perror ("malloc");
13exit (EXIT_FAILURE);
14}
15
16/* overflow */
17  [6] = 100;  v1
18
19free(v1);
20
21/* write after free */
22  [6] = 100;  v1
```

Exemplul următor reprezintă un program cu o gamă variată de erori de alocare a memoriei:

Exemplu: valgrind_test.c

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

int main(void)
{
char buf[10];
char *p;

/* no init */
strcat(buf, "al");

/* overflow */
   [11] bufa';

    malloc(70);
 [10] = p;
free(p);

/* write after free */
 [1] = 'p';
    malloc(10);

/* memory leak */
    malloc(10);

/* underrun */
```

```
        p--;
      'a';*p =

return 0;
}
```

În continuare, se prezintă comportamentul executabilului obinut la o rulare obinuită i la o rulare sub Valgrind:

```
razvan@valhalla:~/school/2007-2008/so/labs/lab4/samples/valgrind$ make
cc -Wall -g    valgrind_test.c   -o valgrind_test
razvan@valhalla:~/school/2007-2008/so/labs/lab4/samples/valgrind$ ./valgrind_test
razvan@valhalla:~/school/2007-2008/so/labs/lab4/samples/valgrind$ valgrind --tool=memcheck ./valg
==18663== Memcheck, a memory error detector.
==18663== Copyright (C) 2002-2007, and GNU GPL'd, by Julian Seward et al.
==18663== Using LibVEX rev 1804, a library for dynamic binary translation.
==18663== Copyright (C) 2004-2007, and GNU GPL'd, by OpenWorks LLP.
==18663== Using valgrind-3.3.0-Debian, a dynamic binary instrumentation framework.
==18663== Copyright (C) 2000-2007, and GNU GPL'd, by Julian Seward et al.
==18663== For more details, rerun with: -v
==18663==
==18663== Conditional jump or move depends on uninitialised value(s)
==18663==    at 0x40050D: main (valgrind_test.c:10)
==18663==
==18663== Invalid write of size 1
==18663==    at 0x400554: main (valgrind_test.c:20)
==18663==  Address 0x5184031 is 1 bytes inside a block of size 70 free'd
==18663==    at 0x4C21B2E: free (vg_replace_malloc.c:323)
==18663==    by 0x40054B: main (valgrind_test.c:17)
==18663==
==18663== Invalid write of size 1
==18663==    at 0x40057C: main (valgrind_test.c:28)
==18663==  Address 0x51840e7 is 1 bytes before a block of size 10 alloc'd
==18663==    at 0x4C21FAB: malloc (vg_replace_malloc.c:207)
==18663==    by 0x40056E: main (valgrind_test.c:24)
==18663==
==18663== ERROR SUMMARY: 6 errors from 3 contexts (suppressed: 8 from 1)
==18663== malloc/free: in use at exit: 20 bytes in 2 blocks.
==18663== malloc/free: 3 allocs, 1 frees, 90 bytes allocated.
==18663== For counts of detected errors, rerun with: -v
==18663== searching for pointers to 2 not-freed blocks.
==18663== checked 76,408 bytes.
==18663==
==18663== LEAK SUMMARY:
==18663==    definitely lost: 20 bytes in 2 blocks.
==18663==      possibly lost: 0 bytes in 0 blocks.
==18663==    still reachable: 0 bytes in 0 blocks.
==18663==         suppressed: 0 bytes in 0 blocks.
==18663== Rerun with --leak-check=full to see details of leaked memory.
```

Se poate observa că la o rulare obinuită programul nu generează nici un fel de eroare. Totui, la rularea Valgrind apar erori în 3 contexte:

1. la apelul `strcat` (linia 10) irul nu a fost iniializat
2. se scrie în memorie după `free` (linia 20: `p[1] = 'a'`)
3. underrun (linia 28)

În plus există leak-uri de memorie datorită noului apel `malloc` care asociază o nouă valoare lui `p` (linia 24).

Valgrind este un utilitar de bază în depanarea programelor. Este facil de folosit (nu este intrusiv, nu necesită modificarea surselor) i permite detectarea unui număr important de erori de programare apărute ca urmare a gestiunii defectuoase a memoriei.

Informaii complete despre moduol de utilizare a Valgrind i a utilitarelor asociate se găsesc în paginile de documentaie Valgrind.

Alte utilitare pentru depanarea problemelor de lucru cu memoria

Utilitarele prezentate mai sus nu sunt singurele folosite pentru detectarea problemelor aparute in lucrul cu memoria. Alte utilitare sunt:

- dmalloc
- mpatrol
- DUMA
- Electric Fence

Exerciii

Prezentare

Pentru a urmări mai uor noiunile expuse la începutul laboratorului urmării această prezentare. odp|pdf

Quiz

Pentru autoevaluare raspundei la întrebările din acest quiz.

Exerciii pre-laborator

Folosii arhiva de pre-sarcini a laboratorului.

Linux

Folosii directorul `lin/` din arhiva de pre-sarcini a laboratorului.

1. Intrai în subdirectorul `my_malloc/`.
   - În fiierul `my_malloc_test.c`, completai funcia `main` alocai spaiu pentru `N_ELEM` elemente întregi în pointerul `elem_p`.
   - Completai două poziii din vectorul obinut (indicat de `elem_p`): poziia `ARRAY_POS_1` i `ARRAY_POS_2`.

- La completare folosii două metode diferite (dereferenciere ca pointer i referire sub formă de vector).
- Eliberai spaiul ocupat de vector.
- Folosii fiierul Makefile din director pentru compilarea programului.
- Rulai programul astfel obinut.
- Testai folosind Valgrind că programul nu conine erori de lucru cu memoria.

2. Rămânei în subdirectorul `my_malloc/`.
   - Configurai macroul `ARRAY_POS_2` pentru a fi egal cu `N_ELEM`.
   - Folosii fiierul Makefile din director pentru compilarea programului.
   - Rulai programul astfel obinut.
   - Rulai programul obinut în Valgrind. Care este cauza apariiei erorii?
3. Intrai în directorul `my_realloc/`.
   - Lucrai peste fiierul `my_realloc_test.c`.
   - În funcia `main` alocai spaiu de `N_ELEM_PHASE1` elemente întregi în pointer-ul `elem_p`.
   - Modificai spaiul ocupat la `N_ELEM_PHASE2`.
   - Omitei (intenionat) eliberarea spaiului ocupat.
   - Folosii fiierul Makefile din director pentru compilarea programului.
   - Rulai programul astfel obinut.
   - Rulai programul obinut în Valgrind. Care este cauza apariiei erorii?
   - De ce, totui, nu este nevoie de eliberare spaiului ocupat?
4. Intrai în directorul `inv_free/`.
   - Investigai fiierul `inv_free.c`.
   - Folosii fiierul Makefile din director pentru compilarea programului.
   - Rulai programul astfel obinut.
   - Care este cauza apariiei erorii?

Windows

Folosii directorul `win/` din arhiva de pre-sarcini a laboratorului.

1. Intrai în subdirectorul `my_heap_alloc/`.
   - În fiierul `my_ha_test.c`, completai funcia `main` alocai spaiu pentru `N_ELEM` elemente întregi în pointerul `elem_p` (folosii `HeapAlloc`).
   - Completai două poziii din vectorul obinut (indicat de `elem_p`): poziia `ARRAY_POS_1` i `ARRAY_POS_2`.
   - La completare folosii două metode diferite (dereferenciere ca pointer i referire sub formă de vector).
   - Eliberai spaiul ocupat de vector. (folosii `HeapFree`)
   - Folosii fiierul Makefile din director pentru compilarea programului.
   - Rulai programul astfel obinut.
2. Rămânei în subdirectorul `my_heap_alloc/`.
   - Configurai macroul `ARRAY_POS_2` pentru a fi egal cu `N_ELEM`.
   - Folosii fiierul Makefile din director pentru compilarea programului.
   - Rulai programul astfel obinut.
3. Intrai în directorul `my_heap_realloc`.
   - Lucrai peste fiierul `my_hra_test.c`.
   - În funcia `main` alocai spaiu de `N_ELEM_PHASE1` elemente întregi în pointer-ul `elem_p`.
   - Modificai spaiul ocupat la `N_ELEM_PHASE2`. (folosii `HeapReAlloc`).
   - Omitei (intenionat) eliberarea spaiului ocupat.

- Folosii fiierul Makefile din director pentru compilarea programului.
- Rulai programul astfel obinut.
- De ce, totui, nu este nevoie de eliberare spaiului ocupat?

Exerciii de laborator

Folosii arhiva de sarcini a laboratorului.

Linux

Folosii directorul `lin/` din arhiva de sarcini a laboratorului.

1. (**1 punct**) Intrai în directorul `01-alloc/`.
   - Rezolvați, în fiierul `alloc.c`, toate problemele marcate cu `TODO` astfel:
     - Alocați memorie pentru un vector care să stocheze `no` şiruri de caractere.
     - Alocați memorie astfel încât fiecare şir să stocheze `crt_len+1` caractere.Nu uitați să verificați rezultatul întors de `malloc`.
     - Afişati vectorul de şiruri de caractere, fiecare şir pe o linie nouă.
     - Eliberați memoria alocată.

       Hints:
       - Citiți secțiunea Alocarea memoriei în Linux din laborator.
       - Dezalocati mai intai memoria alocata pentru fiecare sir de caractere , iar apoi memoria alocata pentru vectorul de siruri de caractere.
2. (**1 punct**) Intrai în directorul `02-struct/`.
   - Rezolvați, in fisierul `struct.c` , toate problemele marcate cu `TODO` astfel:
     - În funcția `allocate_flowers` alocați memorie pentru no elemente de tip `flower_info`.
     - În funcția `free_flowers` eliberați memoria alocată in funcția `allocate_flowers`.
     - Folosiți Valgrind pentru a descoperi eventualele probleme de lucru cu memoria şi corectați-le.

       Hints:
       - Folosiți opțiunea --tool=memcheck pentru valgrind.
       - Citiți secțiunea Valgrind din laborator.
3. (**2.5 puncte**) Intrai în directorul `03-stack/`.
   - Inspectati fisierul `stack_param.c`.
     - Compilati si rulati programul.
     - Ce observați?

       Hints:
     - Citiți secțiunea Stiva din laborator.
   - Inspectați programul din fişierul `README` şi incercati sa descrieți conținutul stivei la momentul indicat în program.
   - Folosiți programul ajutator `stack.c` dupa ce rezolvati problemele indicate de TODO, astfel:
     - în funcţia `show_snapshot` iterați pe toata lungimea stivei şi afişați adresa şi valoarea de la adresa curentă
     - în funcţia `take_snapshot` salvați in structura de date ce retine imaginea stivei campurile adresă şi valoare.

- Rulați executabilul `stack` şi identificați adresele si valorile asociate cu variabilele din program.
  Hints:
  - ◊ Citiți comentariile din codul sursa `stack.c` pentru a înțelege cum se salvează imaginea stivei.
  - ◊ Citiți secțiunea Stiva din laborator.

4. (**1 punct**) Intrați in directorul `04-overflow`
   - Completați problemele marcate de `TODO` la fel ca la exercițiul precedent.
   - De data aceasta functia `f2` pune pe stivă un vector de 3 întregi. În ce ordine sunt puse elementele vectorului pe stivă?
   - Care este adresa de revenire din funcția `f2?`.
   - Folosindu-vă de vectorul v fortați execuția funcției `show_message` fară a o apela explicit. Astfel dupa apelul funcției f2 fluxul programului nu se va mai întoarce în funcția f1 ci va executa `show_message`.
5. (**0.5 puncte**) Intrați in directorul `05-trim`.
   - Analizați programul trim.c şi rulați executabilul trim.
   - Folositi mcheck pentru a detecta problema si corectati-o.
     Hints:
     - ◊ Rulati programul folosind MALLOC_CHECK_=1 ./trim
     - ◊ Citiți secțiunea mcheck din laborator.

Windows

Folosii directorul `win/` din arhiva de sarcini a laboratorului.

1. (**1 punct**) Intrai în directorul `01-util/`.
   - Inspectai cele două fiiere existente: `util.c` i `util.h`.
   - Completai fiierul `util.c` cu definiia funciilor `xmalloc`, i a macrodefiniiei `xfree` după cum urmează:
     - ◊ În cazul `xmalloc` se alocă spaiu folosind `HeapAlloc`; dacă alocarea euează, programul se încheie cu `abort`.
     - ◊ `xfree` este un macro care primete ca argument pointer-ul de eliberat; se apelează `HeapFree` i pointer-ul este resetat la `NULL`
     - ◊ De ce este mai dificil să se realizeze o funcie `xfree` care să realizeze aceleai operaii?
2. (**1 punct**) Intrai în directorul `02-xtest/`.
   - Inspectai fiierul `x_test.c`.
   - Completai fiierul `Makefile` cu informaiile necesare pentru a compila fiierele `01-util/util.c` i `x_test.c` i pentru a le lega în executabilul `x_test`.
   - Modulul obiect asociat lui `util.c` trebuie să se găsească tot în directorul `01-util/`.
     Hints:
     - ◊ Citiți secțiunea Alocarea memoriei în Windows din laborator.
     - ◊ Folosii variabila Makefile standard `CFLAGS` i opiunea de preprocesare `/I`.
     - ◊ Target-urile i cerinele dintr-o regulă Makefile pot fi precizate sub formă de căi în sistemul de fiiere.
   - Rulai executabilul obinut.
3. (**2 puncte**) Intrai în directorul `03-tensor/`.
   - Analizai fiierul `tensor.c`.
   - Implementai funciile `tensor_alloc`, respectiv `tensor_free` care alocă/dezalocă un

vector tridimensional (tensor).

Hints:

◊ Citiți secțiunea Alocarea memoriei în Windows din laborator.

◊ Folosiți funcțiile `xmalloc` si `xfree` definite la punctul 1.

4. (**1 punct**) Intrați în directorul `04-bad_stack/`
   - ♦ Analizați fişierul `bad_stack.c`.
   - ♦ Compilați programul şi rulați executabilul astfel obținut.
   - ♦ Cum se explică rezultatul afişat. Corectați programul astfel încât rezultatul afişat să fie corect

Soluii

- Soluii exerciii laborator 4

Resurse utile

- Linux System Programming - Chapter 8 - Memory Management
- Windows System Programming - Chapter 5 - Memory Management (Win32 and Win64 Memory Management Architecture, Heaps, Managing Heap Memory
- Linux Application Programming - Chapter 7 - Memory Debugging Tools
- Windows Memory Management
- Virtual Memory Allocation and Paging
- GDB manual
- Valgrind Home
- Using Valgrind to Find Memory Leaks
- The Memory Management Reference
- IBM trial download: Rational Purify 7.0
- Using Purify
- Memory Management Software
- Smashing the Stack for Fun and Profit