LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

ATMIŅAS IZMETES PIELIETOŠANA KAUDZES ATKĻŪDOŠANAS METODES IZSTRĀDEI

BAKALAURA DARBS

Autors: **Renata Januškeviča**
Studenta apliecības Nr.: rj10013
Darba vadītājs: Mg. sc. ing. Romāns Taranovs

RĪGA 2014

ANOTĀCIJA

Darbs sastāv no ievada, 6 nodaļām, secinājumiem un 3 pielikumiem. Tajā ir 21. lappuses, 14 attēli, 0 tabulas pamattekstā un 18 nosaukumi literatūras sarakstā.

Atslēgvārdi:

ABSTRACT

The development of a heap debugging method based on the use of core dumps

The work consists of introduction, 6 chapters, conclusions and 3 appendixes. It contains 21. pages, 14 figures, 0 tables and 18 references.

Keywords:

SATURS

APZĪMĒJUMU SARAKSTS

POSIX (Portable Operating System Interface) - IEEE un ISO standartu kopa, kas reglamentē kā rakstīt pieteikumu pirmkodu tā, lai pieteikumi būtu pārnēsājami starp operētājsistēmām.

IEEE (Institute of Electrical and Electronics Engineers) - Elektrotehnikas un elektronikas inženieru institūts.

ISO (International Organization for Standardization) - Starptautiskā Standartu organizācija.

Hard link - Stingrā saite - rādītājs uz datnes indeksa deskriptoru.

Heap - Kaudze - atmiņas apgabals, kurš tiek izmantots dinamiskajai atmiņas iedalīšanai.

ELF (Executable and Linkable Format) - ELF formāts - bināro datņu formāts, kurš ir Unix un Linux standarts. Šīs formāts var būt izmantots priekš izpildāmam datnēm, objektu datnēm, bibliotēkām un atmiņas izmetēm.

Memory allocation - Atmiņas iedalīšana - atmiņās adreses piesaistīšana instrukcijām un datiem.

Static memory allocation - Statiskā atmiņas iedalīšana - atmiņas iedalīšanas paņēmiens, kurš ir pielietots kompilācijas laikā.

Dynamic memory allocation - Dinamiskā atmiņas iedalīšana - atmiņas iedalīšanas paņēmiens, kurš pielietots programmas izpildes laikā.

Core dump - Atmiņas izmete - visa atmiņas satura vai tā daļas pārrakstīšana citā vidē (parasti - no iekšējās atmiņas ārējā). Izmeti izmanto programmu atkļūdošanai.

Instance of the program - Programmas instance - izpildāmās programmas kopija, kurai ir nepieciešama vieta operatīvajā atmiņā.

Chunk - Gabals - nepārtraukts atmiņas gabals ar īpatnējo struktūru.

ptmalloc2 - atvērtā pirmkoda programmatūra. Realizācija ptmalloc2 ir daļa no GNU C bibliotēkas, kura nodrošina dinamisko atmiņas iedalīšanu, izmantojot malloc(), free(), realloc() funkcijas izsaukumus.

IEVADS

1. JĒDZIENI, UZ KURIEM BALSTĪTA METODE

Šajā nodaļā tiek aplūkots atmiņas izmetes jēdziens, ka arī aprakstītas atmiņas izmetes ģenerēšanas iespējas un nosacījumi. Nodaļā ir aprakstīts atkļūdošanas process, kas var būt paveikts, izmantojot atmiņas izmeti. Uz aprakstītiem pamatjēdzieniem, turpmāk tiks balstīta kaudzes atkļūdošanas metode.

1.1. Atmiņas izmete

Sistēmās, kuras atbalsta POSIX standartus, ir signāli [13], kuri, pēc noklusētās apstrādes, izraisa atmiņas izmetes ģenerēšanu un pārtrauc procesa darbību. Šos signālus var atrast `man 7 signal` komandas izvadā. Signāliem, kuri izraisa izmetes ģenerēšanu, signālu tabulā [11] ir lauks ar vērtību core, kas atrodas ailē ar nosaukumu darbība (Action). Uzģenerētā atmiņas izmete iekļauj sevī procesa atmiņas attēlojumu uz procesa pārtraukšanas brīdi, piemēram, CPU reģistrus un steka vērtības katram pavedienam, globālos un statiskos mainīgos. Atmiņas izmeti var ielādēt atkļūdotājā, tāda kā gdb, lai apskatītu programmas stāvokli uz brīdi, kad atnāca operētājsistēmas signāls [10]. Veicot atmiņas izmetes analīzi, kļūst iespējams atrast un izlabot kļūdas, pat tad, ja nav piekļuves sistēmai.

Eksistē vairākas iespējas kā var uzģenerēt atmiņas izmeti. To var izdarīt no programmas koda, gdb atkļūdotāja vai komandrindas interpretatora. Turpmāk katra no iespējam tiks apskatīta sīkāk.

1.1.1. Atmiņas izmetes ģenerēšana no koda

Ģenerējot atmiņas izmeti no programmas koda, ir divas iespējas: process var turpināties vai beigt savu darbu pēc signāla nosūtīšanas.

```
1 #include <signal.h>
2
3 int main ()
4 {
5    raise(SIGSEGV); /* Signal for Invalid memory reference */
6
7    return 0;
8 }
```

1.1. att. **Atmiņas izmetes ģenerēšana, pārtraucot procesa darbību**

Ja nav nepieciešams, lai process turpinātu darbību, tad var izmantot funkcijas raise(), abort(), kā arī var apzināti pieļaut kļūdu kodā. Tādas kļūdas kā dalīšana ar nulli nosūta SIGFPE signālu, bet

vēršanās pēc rādītāja ar null vērtību - SIGSEGV signālu. Izmantojot funkciju raise(), ir iespējams norādīt atmiņas izmeti izraisošo signālu. Piemērā (sk. 1.1. attēls) ir redzams C kods, kur funkcija raise() nosūta SIGSEGV signālu izpildāmai programmai. Pēc šī izsaukuma izpildes tiek izvadīts ziņojums: Segmentation fault (core dumped). Atmiņas izmeti lietotāju procesiem var atrast darba mapē, jo Linux operētājsistēmā tā ir noklusēta atmiņas izmetes atrašanas vieta, bet noklusētais atmiņas izmetes nosaukums ir core.

Ir iespējams uzģenerēt atmiņas izmeti, nepārtraucot procesa darbību (sk. 1.2. attēls). To var panākt ar fork() funkcijas palīdzību. Funkcija fork() izveido bērna procesu, kas ir vecāka procesa kopija. Funkcijas fork() veiksmīgas izpildes gadījumā, bērnu procesam atgriež 0 vērtību. Pēc abort() funkcijas izpildes, bērns beidz izpildi un uzģenerē atmiņas izmeti. Vecāks process turpina izpildi.

```c
#include <stdlib.h>

int main ()
{
    int child = fork();
    if (child == 0) {
        abort(); /* Child */
    }
    return 0;
}
```

1.2. att. **Atmiņas izmetes ģenerēšana, turpinot procesa darbību**

1.1.2. Atmiņas izmetes ģenerēšana no gdb

Izsaukumi no koda nav vienīga iespēja kā varētu iegūt atmiņas izmeti. Var izmantot gdb komandas: `generate-core-file [file]`(sk. 1.3. attēls) vai `gcore [file]`. Šīs komandas izveido gdb pakļautā procesa atmiņas izmeti. Izmantojot gdb, var uzģenerēt atmiņas izmeti, kura atbilst kādam pārtraukuma punkta stāvoklim. Neobligāts arguments `filename` nosaka atmiņas izmetes nosaukumu. Šī gdb komanda ir realizēta GNU/Linux, FreeBSD, Solaris and S390 sistēmās [3].

```
(gdb) attach <pid>
(gdb) generate-core-file <filename>
(gdb) detach
(gdb) quit
```

1.3. att. **Atmiņas izmetes ģenerēšana, izmantojot gdb**

1.1.3. Atmiņas izmetes ģenerēšana no komandrindas interpretatora

Trešā iespēja ir nosūtīt signālu, izmantojot komandrindas interpretatoru. Komanda kill var nosūtīt jebkuru signālu procesam. Pēc komandas `kill -<SIGNAL_NUMBER> <PID>`, signāls ar numuru SIGNAL_NUMBER tiks nosūtīts procesam ar norādītu PID vērtību. Izmantojot shell komandrindas interpretatoru ir iespējams izmantot īsinājumtaustiņus signālu nosūtīšanai. Nospiežot Control + \ tiks nosūtīts SIGQUIT signāls procesam, kas pašreiz ir palaists (sk. 1.4. attēls) [14]. Šajā piemēra ziņojumu - Quit (core dumped), izdruka shell. Šīs komandrindas interpretators noteic, ka sleep procesu (shell bērnu) pārtrauca SIGQUIT signāls. Pēc šī signāla nosūtīšanās darba mapē tiek uzģenerēta atmiņas izmete.

```
1 $ ulimit -c unlimited
2 $ sleep 30
3 Type Control +\
4 ^\Quit (core dumped)
```

1.4. att. **Atmiņas izmetes ģenerēšana, izmantojot īsinājumtaustiņus**

1.1.4. Atmiņas izmetes ģenerēšanas nosacījumi

Lai uzģenerētu atmiņas izmeti ir jābūt izpildītiem sekojošiem nosacījumiem [14]:

- ir jānodrošina atļauja procesam rakstīt core datni darba mapē,
- ja datne, ar vienādu nosaukumu jau eksistē, tad ir jābūt ne vairāk kā vienai stingrai saitei,
- izvēlētai darba mapei ir jābūt reālai un jāatrodas norādītajā vietā,
- Linux core datnes izmēra robežai RLIMIT_CORE jāpārsniedz ģenerējamā faila izmēru, RLIMIT_FSIZE robežai jāļauj procesam izveidot atmiņas izmeti,
- ir jāatļauj lasīt bināro datni, kura ir palaista,
- failu sistēmai, kurā atrodas darba mape, ir jābūt uzmontētai priekš rakstīšanas, tai nav jābūt pilnai un ir jāsatur brīvie indeksa deskriptori,
- bināro datni jāizpilda lietotājam, kurš ir datnes īpašnieks (group owner).

Pēc noklusējuma atmiņas izmetes ģenerēšanas iespēja ir izslēgta, `ulimit -c unlimited` komanda ļauj ieslēgt atmiņas izmetes ģenerēšanu.

1.2. Atkļūdošana, izmantojot atmiņas izmeti

Atmiņas izmete satur datus, kuri dod iespēju atrast kļūdas. Tāpēc atmiņas izmete var tikt pielietota, lai veiktu lietotnes atkļūdošanu, pēc neparedzētas programmas apstāšanās. Atmiņas izmetes analīze ir efektīvs veids, kā var attālināti atrast un izlabot kļūdas bez iejaukšanās un tiešas piekļuves

sistēmai. Daudzos gadījumos, atmiņas izmete ir speciāli uzģenerēta datne, kura palīdz iegūt atmiņas stāvokli uz signāla nosūtīšanas brīdi. Atmiņas izmete ir labi piemērota kļūdu meklēšanai, kas saistītas ar nepareizo atmiņas izmantošanu lietotnē.

Atmiņas izmete ir ELF formāta datne. ELF formāts ir Linux un Unix standarts priekš izpildāmām datnēm, objektu datnēm, bibliotēkām un atmiņas izmetēm. Lai darbotos ar atmiņas izmetēm ir nepieciešams, lai rīks, kurš tika izvēlēts (bibliotēka, utilītprogramma vai atkļūdotājs) atbalstītu ELF formāta failus. GNU gdb ir Linux standarta atkļūdotājs [16], kurš ir plaši pielietojams atmiņas izmešu analīzei.

Ja atmiņas izmetes analīzei tika izvēlēts GNU gdb atkļūdotājs, tad pirms sākt analīzi ir nepieciešams pārliecināties ka gdb ir pareizi nokonfigurēts priekš procesora arhitektūras, no kuras bija iegūta atmiņas izmete. To var identificēt uzreiz pēc gdb palaišanas, ar sekojošās rindiņas palīdzību: `This GDB was configured as i686-linux-gnu`. Lai atmiņas izmete saturētu atkļūdošanas informāciju, ir jānorāda -g opcija kompilācijas laikā. Atkļūdošanas informācija ir uzglabāta objektu datnē un saglabā atbilstību starp izpildāmo datni un pirmkodu, ka arī mainīgo un funkciju datu tipus. Ja atmiņas izmete neiekļauj atkļūdošanas informāciju, tad atmiņas izmete var attēlot sekojošo tekstu (sk. 1.5. attēls).

```
(gdb) p main
$ 1 = {<text variable, no debug info>} 0x80483e4 <main>
```

1.5. att. **Atmiņas izmete nesatur atkļūdošanas informāciju**

Kad atmiņas izmete ir uzģenerēta, tad to var apskatīt, izmantojot gdb atkļūdotāju (sk. 1.6. attēls). Atkļūdotājam kā argumenti tiek padoti: izpildes fails un atmiņas izmete. Izpildes failam ir jāatbilst atmiņas izmetei, lai varētu apskatīt korektus, nesabojātus datus.

```
$ gdb <path/to/the/binary> <path/to/the/core>
```

1.6. att. **Atmiņas izmetes atvēršana, izmantojot gdb atkļūdotāju**

Gdb ļauj iegūt svarīgus datus no atmiņas izmetes. Komanda `info files` ļauj apskatīt procesa segmentus. Katram segmentam ir adrešu apgabals ar nosaukumu. Segmenti, kuru nosaukums ir "loadNNN" pieder procesam, tajos var tikt uzglabāti: statiskie dati, steks, kaudze, koplietošanas atmiņa. Tā kā segmentu robežas ir zināmas, tad kļūst iespējams izdrukāt atmiņas saturu, kas pieder segmentiem, uzzināt kuram segmentam pieder konkrētā atmiņas adrese un kādas ir tās vērtības.

Lai izdrukātu atmiņas apgabalu ir nepieciešams norādīt atmiņas adresi (`addr`), no kura sākt atmiņas izdruku, formātu (`f`), apgabala lielumu (`n`) un norādīt vienības lielumu (`u`) (sk. 1.7. attēls). Izmantojot doto piemēru tiks izdrukāts `n` liels atmiņas apgabals, kurš sākas ar adresi `addr`. Formātu un vienības lielumu vajag norādīt saskaņā ar gdb pamācību [17].

```
(gdb) x/nfu addr
```

1.7. att. **Atmiņas apgabala izdrukāšanas formāts**

Atmiņas izmetes analīze sākas ar backtrace izdrukāšanu. Bactrace ir pārskats, kurš attēlo kā programma nonāca stāvoklī, kurā pabeidza savu darbību. Katra rindiņa satur rāmi (frame). Bactrace izdruka sākas ar rāmi, kurš iekļauj funkciju, kura bija izpildīta pēdēja. Nākamais rāmis iekļauj funkciju, kas izsauca iepriekšējā rāmī iekļauto funkciju. Katrai baktrace rindiņai piešķirts rāmja numurs. Katrs rāmis var iekļaut: funkcijas nosaukumu, pirmkoda datnes nosaukumu, pirmkodam atbilstošo rindiņas numuru un funkcijas argumentus. Bactrace var tikt iegūts izmantojot gdb komandu `bactrace full` vai `bt f`. Pēc noklusējuma, daudzpavedienu lietotnēs gdb rāda bactrace kārtējām pavedienam, bet pastāv iespēja iegūt arī bactrace izdruku priekš citiem pavedieniem. Ja programma bija nokompilēta ar optimizācijas opciju, tad bactrace varētu neiekļaut funkcijas argumentus, tad funkciju argumenti varētu tikt nodoti caur CPU reģistriem. CPU reģistru vērtības ir iespējams iegūt, izmantojot komandu `info registers` vai `i r` (sk. 1.8. attēls). Atmiņas izmetē atrodas pēdējais atmiņas stāvoklis, tāpēc CPU reģistru vērtības ir iespējams atjaunot no steka, ja pēc izjaukšanas (disassembling) ir redzams, ar cik lielu nobīdi tie tika saglabāti stekā.

```
(gdb) i r
eax    0x0     0
ecx    0x2377  9079
edx    x8      8
ebx    0x2377  9079
esp    0xbf97a254   0xbf97a254
ebp    0xbf97a298   0xbf97a298
esi    0x0     0
edi    0xb77abff4   -1216692236
eip    0xb77c2424   0xb77c2424 <__kernel_vsyscall+16>
eflags 0x246   [ PF ZF IF ]
cs     0x73    115
ss     0x7b    123
ds     0x7b    123
es     0x7b    123
fs     0x0     0
gs     0x33    51
```

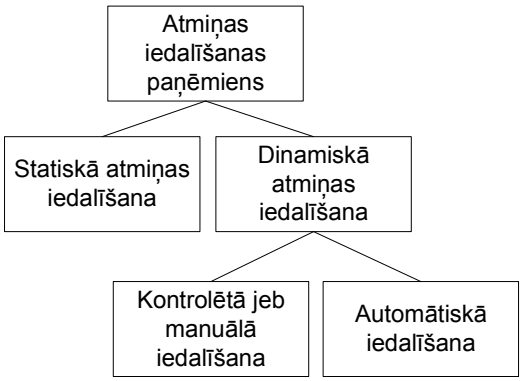
1.8. att. **Atmiņas izmetes CPU reģistri 32 bitu, Intel x86 arhitektūrā**

2. ATMIŅAS IEDALĪŠANA, ORGANIZĀCIJA UN PĀRVALDĪBA

Šajā nodaļā ir aprakstīti atmiņas iedalīšanas paņēmieni, ir dots īss ieskāts atmiņas organizācijā un aprakstīta atmiņas pārvaldība, kuru veic kodols vai lietotājs.

2.1. Atmiņas iedalīšanas paņēmieni

Pirms izpildīt programmu, operētājsistēmai ir nepieciešams iedalīt resursus, tādus kā atmiņas adreses. Eksistē divas atmiņas iedalīšanas paņēmieni: statiskā un dinamiskā atmiņas iedalīšana (sk. 2.1. attēls).



2.1. att. **Atmiņas iedalīšanas paņēmienu klasifikācija**

Statiskā atmiņas iedalīšana

Statiskā atmiņas iedalīšana nozīme, ka atmiņa tiek iedalīta vienu reizi pirms programmas palaišanas, parasti tas notiek kompilācijas laikā. Programmas izpildēs laikā atmiņa vairs netiek iedalīta, ka arī netiek atbrīvota. Statiskais atmiņas iedalīšanas paņēmiens nodrošina to, ka atmiņa tiek iedalīta statiskiem un globāliem mainīgiem, neatkarībā no tā vai mainīgais tiks izmantots programmā vai nē [6].

Dinamiskā atmiņas iedalīšana

Dinamiskā atmiņas iedalīšana nozīme, ka atmiņa tiek iedalīta programmas izpildes laikā. Tas var būt nepieciešams, kad atmiņas daudzums nav zināms programmas kompilācijas laikā. Dinamiskā atmiņas iedalīšana, var būt realizēta ar steka vai kaudzes palīdzību un var būt automātiskā vai kontrolētā [12].

Automātiskā iedalīšana notiek, kad sākās programmas funkcijas izpilde. Šeit viens un tas pats atmiņas apgabals, kurš bija atbrīvots, var tikt izmantots vairākas reizēs. Piemēram, kad tekošās funkcijas argumenti un lokālie mainīgie ir saglabāti stekā un izdzēsti pēc šīs funkcijas izpildes. Vērtību izdzēšana no steka notiek, nobīdot steka norādi uz augšu. Pēc tam atbrīvotā atmiņa var būt izmantota atkārtoti. Priekš automātiskās atmiņas iedalīšanas, izmato steku. Visiem funkcijas mainīgiem var piekļūt izmantojot steka norādes nobīdi, kas tiek uzglabāta reģistrā, piemēram, Intel x86 procesoros, 16 bitu režīmā reģistrs ir `SP`, 32 bitu režīmā - `ESP` un 64 bitu režīmā - `RSP` [15]. Reģistrs uzglabā adresi, kurā atrodas pēdējā uzglabāta vērtība stekā. Steka pārpildīšana var notikt dažādu iemeslu dēļ, piemēram to var izraisīt dziļa rekursija.

Kontrolētā atmiņas iedalīšana nozīme, ka programma var izvēlēties patvaļīgus, brīvus atmiņas apgabalus priekš programmas datiem. Kontrolētā jeb manuālā atmiņas iedalīšana ir realizēta ar kaudzes palīdzību. Šeit nav iespējams piekļūt datiem izmantojot vienu norādi un tās nobīdi. Tagad katram izdalītam atmiņas apgabalam var piekļūt tikai tad, ja ir norāde uz šo iedalīto atmiņas apgabalu. Gadījumos, kad norādes nav, tad adreses vairāk nav sasniedzamas un kļūst pazaudētas.

2.2. Atmiņas pārvaldība

Kad tiek izpildīta jebkura programma, atmiņa tiek pārvaldīta divos veidos: ar kodola palīdzību vai ar lietotnes funkciju izsaukumiem, tādiem kā malloc().

2.2.1. Atmiņas pārvaldība, kuru veic kodols

Operētājsistēmas kodols pārvalda visus atmiņas pieprasījumus, kas attiecās uz programmu vai programmas instancēm. Kad lietotājs sāk programmas izpildi, tad kodols iedala atmiņas apgabalu tekošai programmai. Pēc tam programma pārvalda iedalīto apgabalu, sadalot to vairākos segmentos:

- Teksts - uzglabāti dati, kuri tiek izmantoti tikai lasīšanai. Tās ir koda instrukcijas. Vairākas programmas instances var izmantot šo atmiņas apgabalu.
- Statiskie dati - apgabals, kurā tiek uzglabāti dati ar iepriekš zināmu izmēru. Tās ir globālie un statiskie mainīgie. Operētājsistēma iedala šī apgabala kopiju priekš katras programmas instances atsevišķi.
- Atmiņas arēna - apgabals, kurā tiek uzglabāta dinamiski iedalītā atmiņa. Atmiņas arēna sastāv no kaudzes un neizmantotās atmiņas. Kaudze ir apgabals, kurā atrodas visa iedalītā atmiņa programmas izpildei.
- Steks - apgabals, kurā tiek uzglabāts funkciju izsaukumu stāvoklis, katram funkcijas izsaukumam. Steks aug no lielākas adreses uz mazāko. Unikāla atmiņas arēna un steks iedalīti priekš katras programmas instances atsevišķi.

Lai palielinātu atmiņas arēnas izmēru, tiek veikts brk() sistēmas izsaukums. Izsaukums uzstāda arēnas segmenta jaunas beigas. Jā process nepārsniedz savu limitu, tad izsaukums atgriež 0 un arēnas segmenta lielums tiek veiksmīgi izmainīts. Iedalīto adrešu intervālu stekam un atmiņas arēnai var atrast `/proc/<pid>/maps` datnē.

2.2.2. Atmiņas pārvaldība, kuru veic lietotājs

Lietotājam iedalīta atmiņa atrodas kaudzē, kura tiek novietota atmiņas arēnā. Atmiņās arēna C valodā tiek pārvaldīta ar malloc(), realloc(), free() un calloc() funkciju palīdzību [18]. C++ valodā ir izmantots operators new, lai pieprasītu atmiņu. Attēlā 2.2. ir redzama C un C++ sintakse atmiņas pieprasīšanai izmantojot C un C++ kodu. Vienīgais arguments malloc() funkcijai ir baitu skaits. C programmai, lai saskaitītu cik baitu ir nepieciešams pieprasīt, ir nepieciešams zināt cik daudz vietas aizņem viens elements un kāds ir elementu skaits. Funkcija malloc() atgriež void tipa rādītāju, tāpēc C programmās ir nepieciešams izmantot drošo tipa pārveidotāju (typecast). Tas ir nepieciešams, lai saglabātu atgriezto norādi lokālajā mainīgajā. Atmiņas inicializācija C kodā var būt veikta izmantojot arī citas funkcijas, piemēram calloc() funkciju, kura atgriež atmiņas apgabalu inicializētu ar 0 vērtībām.

```
1 int * ptrl = new int; // C++
2 int * ptrl = (int *)malloc( sizeof(int) ); /* C */
3
4 char * str = new char[num_elements]; // C++
5 char * str = (char *)malloc( sizeof(char) * num_elements ); /* C */
```

2.2. att. **Dinamiskās atmiņas iedalīšana C un C++**

Funkcija free() atbrīvo ar malloc() palīdzību iedalīto atmiņu. Lielāka atšķirība starp free() un delete ir tāda, ka vecajās free() sistēmās netiek nodrošināts atbalsts free() funkcijai, kad arguments ir null [7].

Programmas rakstīšanā nejauc kopā C un C++ stilus, tāpēc priekš C++ programmas izmanto new un delete operatorus (sk. 2.3. attēls), bet priekš C programmām malloc() un free.

```
1 delete ptrl; // C++
2
3 If( ptrl != NULL )
4       free(ptrl); /* C */
```

2.3. att. **Dinamiskās atmiņas atbrīvošana C un C++**

Ja atmiņa pēc izmantošanas netiek nekad atbrīvota, un katru reizi, izpildot vienu un to pašu koda gabalu, iedalīta no jauna, tad pieejams operētājsistēmai atmiņas daudzums ar laiku samazinās.

Sākumā sistēma paliek arvien lēnāka, pēc tam parasti notiek sistēmas apstāšanās.

2.3. Atmiņas iedalīšana glibc bibliotēkā

Darbā tiks aplūkota GNU C bibliotēkas (versija 2.3) ptmalloc2 realizācija, kuru izstrādāja Wolfram Gloger, balstoties uz Doug Lea dlmalloc realizāciju. Atšķirībā no dlmalloc, ptmalloc2 izmanto atsevišķas arēnas priekš pavedieniem. Tāpēc atmiņas iedalīšana var notikt vienlaicīgi vairākos pavedienos. GNU C bibliotēka iekļauj ptmalloc2 realizāciju sākot ar 2.3 versiju [4].

2.3.1. Atmiņas chunk gabali

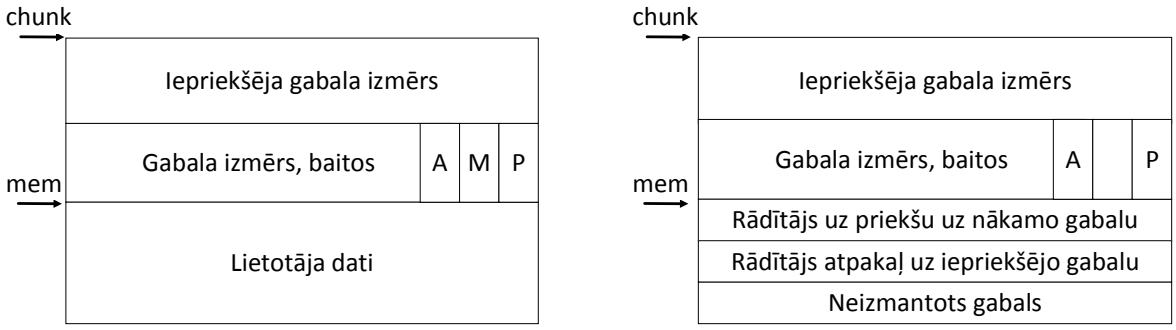
Atmiņa no kaudzēs tiek iedalīta, izmantojot chunk struktūru. Katru reizi ir iedalīts lielāks atmiņās gabals nekā pieprasīts ar malloc() funkciju. Tas ir nepieciešams, lai varētu saglabāt uzturēšanai nepieciešamo informāciju. Iedalītam gabalam uzturēšanas informācija ir vienāda ar 8 vai 16 baitiem, tas ir atkarīgs no tā, cik liela ir `INTERNAL_SIZE_T` makrodefinīcija. `INTERNAL_SIZE_T` ir iekšēji izmantots vārda izmērs (word-size), kurš pēc noklusējuma ir vienāds ar size_t izmēru, bet var būt 32 vai 64 bitu liels. Uzturēšanas informācijai ir izmantoti divas `INTERNAL_SIZE_T` tipa vērtības. Ja gabals ir atbrīvots, tad tiek izmantoti divi rādītāji (fd un bk) uz `malloc_chunk` struktūru (sk. 2.4. attēls).

```
1 struct malloc_chunk {
2              INTERNAL_SIZE_T          prev_size;
3              INTERNAL_SIZE_T          size;
4              struct malloc_chunk*     fd;
5              struct malloc_chunk*     bk;
6 }
```

2.4. att. **chunk gabala struktūra**

Otrs iemesls kāpēc ir iedalīts lielāks atmiņas daudzums ir izlīdzināšana skaitlim, kas ir 2*sizeof(INTERNAL_SIZE_T) reizinājums (8 baitu izlīdzinājums ar 4 baitu INTERNAL_SIZE_T) [5].

Atmiņas gabala fiziska struktūra ir vienāda gan normāliem (normal chunk), gan ātriem gabaliem (fast - chunk), bet ir atkarīga no stāvokļa un var tikt interpretēta dažādi (sk. 2.5. attēls). No kreisās pusēs attēlots atmiņas gabals, kurš bija iedalīts procesam, no labās, tās, kurš bija atbrīvots. Abos gadījumos rādītājs chunk attēlo atmiņas gabalu sākumu. Pēc šī radītāja var iegūt iepriekšēja gabala izmēru, ja iepriekšējais gabals bija atbrīvots. Gadījumā, kad iepriekšējais gabals ir iedalīts, tad pēc chunk rādītāja atrodas daļa no iepriekšēja gabala lietotāja datiem. Pēc tam seko kārtēja gabala izmērs un 3 biti ar meta informāciju. Tā kā notiek izlīdzināšana 2*sizeof(INTERNAL_SIZE_T) reizinājumam, tad 3 pēdējie biti netiek izmantoti izmēra glabāšanai. Šos bitus izmanto kontroles zīmēm. Biti palīdz noteikt vai kārtējais gabals nepieder main arēnai, vai apgabals tiek iedalīts ar



2.5. att. **Atmiņas gabalu struktūra**

mmap() sistēmas izsaukumu un vai iepriekšējais atmiņas gabals tiek izmantots (A, M un P atbilstoši). Rādītājs mem ir malloc() funkcijas atgriežamā vērtība. Iedalīts apgabals stiepjas līdz atmiņas gabala struktūras beigām. Pēc šī rādītāja var tikt uzglabāti dati, kad atmiņa ir iedalīta un, ja tā ir atbrīvota, tad šeit tiks uzglabāti divi radītāji uz nākamo un iepriekšējo atbrīvotiem gabaliem, kas atrodas sarakstā.

2.3.2. Bin saraksti ptmalloc2 versijā

Ja atmiņa bija atbrīvota, tad tā tiks uzglabāta saistītajā sarakstā uz kuru norāda bin masīvs. Sarakstā gabali ir sakārtoti, lai varētu ātrāk atrast piemērotu atmiņas gabalu. Atbrīvots atmiņas apgabals netiek atgriezts operētājsistēmai, bet ir defragmentēts vai sapludināts ar pārējiem gabaliem un ievietots sarakstā, lai tiktu vēlreiz iedalīts. Eksistē divi bin saraksti: fastbin un normal bin.

Atmiņas gabali, kuri paredzēti fastbin glabāšanai ir mazi. Pēc noklusējuma atmiņas gabalu izmērs ir 64 baiti, bet to var palielināt līdz 80 baitiem [5]. Atmiņas gabali atrodas vienvirzienu sarakstā un nav sakārtoti. Lai samazinātu fragmentācijas iespējamību, programma, kad pieprasa vai atbrīvo lielus atmiņas gabalus var sapludināt atmiņas gabalus, kuri atrodas fastbin sarakstā. Piekļuve tādiem atmiņas gabaliem ir ātrāka nekā piekļuve normāliem gabaliem. Fastbin saraksta elementi ir apstrādāti pēdējais iekšā pirmais āra (jeb LIFO) kārtībā [9]. Kad tiek pieprasīta atmiņas daudzums no fastbin saraksta, tad jebkurš atmiņas gabals tiek atgriezts konstantā laikā [2].

Normāla izmēra gabali var būt sadalīti 3 veidos. Pirmkārt, bin saraksts uzglabā nesakārtotus gabalus, kuri nesen bija atbrīvoti. Pēc tam tie tiks novietoti vienā no atlikušiem bin sarakstiem: mazā vai lielā izmēra. Mazā izmēra saraksts uzglabā atmiņas gabalus, kuri ir mazāki par 512 baitiem. Vairāki ātrie gabali var būt sapludināti un uzglabāti dotajā sarakstā. Lielā izmēra saraksts uzglabā atmiņas gabalus, kuri ir lielāki par 512 baitiem, bet mazāki par 128 kilobaitiem. Gabali, kuru izmērs ir lielāks par 128 kilobaitiem tiek iedalīti, izmantojot mmap(). Lielā izmēra sarakstā elementi ir sakārtoti pēc izmēra un ir iedalīti pirmais iekšā, pirmais ārā (jeb FIFO) kārtībā [9]. Eksistē divi citi atmiņas gabali (top chunk un last_remainder), kuriem ir īpaša nozīme un tie netiek uzglabāti bin

sarakstos.

Top chunk ir atmiņas gabals, kurš ierobežo pieejamās atmiņas daudzumu. Tas ir izmantots gadījumos, kad nav piemērotu gabalu bin sarakstos, kuri apmierina pieprasījumu vai varētu būt saplūdināti, lai apmierinātu pieprasījumu. Top chunk nodrošina pēdējo iespēju iedalīt pieprasīto atmiņas daudzumu. Top chunk var mainīt savu izmēru. Tas saraujas, kad atmiņa ir iedalīta un izstiepjas, kad atmiņa ir atbrīvota blakus top chunk struktūrai. Ja ir pieprasīta atmiņa, kas ir lielāka par pieejamo, tad top chunk var paplašināties ar brk() palīdzību. Top chunk ir līdzīgs jebkuram citam atmiņas apgabalam. Galvenā atšķirība ir lietotāja datu sekcija, kura netiek izmantota, ka arī speciāla top chunk apstrāde, lai nodrošinātu, ka top chunk vienmēr eksistē.

Last_remainder ir vel viens atmiņas gabals ar īpašu nozīmi. Tas ir izmantots gadījumos, kad ir pieprasīts mazs atmiņas gabals, kas neatbilst nevienam bin saraksta elementam. Last_remainder ir dalījuma atlikums, kurš izveidojās pēc lielāka gabala sadalīšanas, lai apmierinātu pieprasījumu pēc maza gabala [8].

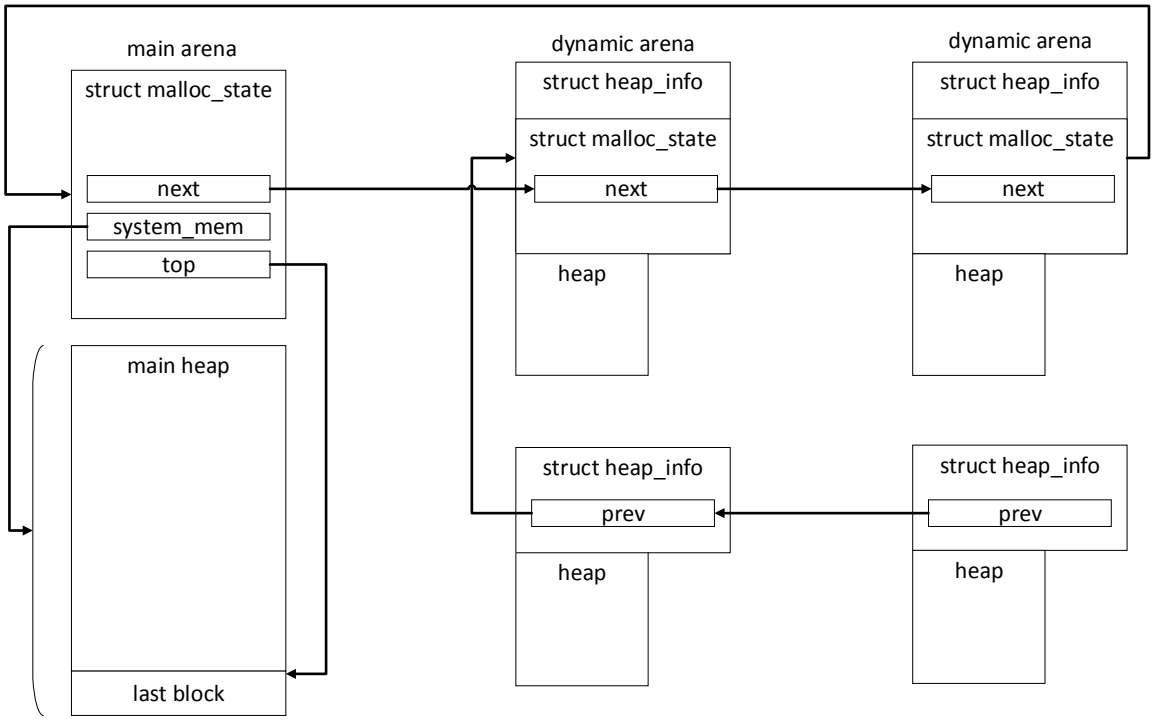
2.3.3. Atmiņas arēna

Lai uzlabotu veiktspēju vairākpavedienu procesiem, GNU C bibliotēkā tiek izmantotas vairākas atmiņas arēnas. Katrs funkcijas malloc() izsaukums bloķē izmantoto arēnu. Laikā, kad arēna ir nobloķēta notiek atmiņas apgabala iedalīšana. Kad vairākiem pavedieniem ir nepieciešams vienlaicīgi iedalīt atmiņu no kaudzes, arēnas bloķēšana var būtiski samazināt veiktspēju. Gadījumos, kad pavedieni izmanto atmiņu no vairākām atsevišķām arēnām, tad vienas arēnas bloķēšana neietekmē atmiņas iedalīšanu parējās arēnās un atmiņas iedalīšana var notikt paralēli. GNU C bibliotēkā darbība ar arēnām notiek saskaņā ar sekojošo algoritmu:

1. malloc() izsaukums vēršas pie arēnas, kurai piekļuva iepriekšējo reizi,
2. ja arēna ir nobloķēta, tad malloc() vēršas pie nākamas izveidotās arēnas,
3. ja nav piekļuves nevienai arēnai, tad tiek izveidota jauna arēna un malloc() vēršas pie tās.

Vispirms atmiņas iedalīšana sākas no galvenās arēnas (main arena).

GNU C bibliotēkā ir globāls `malloc_state` objekts - globāla arēna, kura atšķiras ar to, kā atmiņu no kodola iegūst, izmantojot brk() (sk. 2.6. attēls) [1]. Pārējas arēnas šīm nolūkam izmanto `mmap()` izsaukumu. Ja kārtēja kaudze ir izlietota, tad tiek iedalīta jauna kaudze ar fiksēto 64 MB izmēru. Tāda veidā arēnas var tikt paplašinātas, pievienojot jaunās kaudzes un savienojot tās sava starpā. Lai nodrošinātu labāku veiktspēju, tiek izmantots modelis: katram pavedienam - viena arēna. Ja malloc() pirmo reizi izsaukts pavedienā, tad neatkarībā no tā vai arēna bija nobloķēta vai nē, tiks izveidota jauna arēna. Arēnu skaits ir ierobežots atkarībā no kodolu skaita, 32 bitu vai 64 bitu arhitektūras un mainīga MALLOC_ARENA_MAX vērtības. Tā kā pavedienu skaits parasti nepārsniedz divkāršo kodolu skaitu, tad normālā gadījumā katrs pavediens izmanto atsevišķo arēnu.



2.6. att. **Arēnas GNU C bibliotēkā**

3. PROBLĒMU APRAKSTS

3.1. Pētīšanas metodes

3.2. Izmantojamie rīki

4. ATKĻŪŠANAS METODES APRAKSTS

4.1. Metodes pamatprincipi

4.2. Detalizēts metodes apraksts

4.3. Salīdzināšana ar eksistējošām metodēm

5. METODES REALIZĀCIJAS APRAKSTS

5.1. Sistēmas apraksts

5.2. Projektējums

5.3. Iegūtais rezultāts

GALVENIE REZULTĀTI UN SECINĀJUMI

LITERATŪRA

[1] Anatomy of memory managers. http://core-analyzer.sourceforge.net/index_files/Page525.html. [Online; resurss apskatīts 28-Apr-2014].

[2] A comprehensive complexity analysis of user-level memory allocator algorithms. file:///D:/Downloads/104923_1.pdf. [Online; resurss apskatīts 29-Apr-2014].

[3] How to produce a core file from your program. http://sourceware.org/gdb/onlinedocs/gdb/Core-File-Generation.html. [Online; resurss apskatīts 22-Mar-2014].

[4] Implementations. http://en.wikibooks.org/wiki/C_Programming/C_Reference/stdlib.h/malloc. [Online; resurss apskatīts 24-Apr-2014].

[5] malloc() realizācija. http://code.woboq.org/userspace/glibc/malloc/malloc.c.html. [Online; resurss apskatīts 26-Apr-2014].

[6] Memory allocation in c programs. http://courses.cs.vt.edu/cs5204/archive/Fall2000/Terriberry.pdf. [Online; resurss apskatīts 23-Mar-2014].

[7] Portability of c functions. http://www.hep.by/gnu/autoconf/Function-Portability.html. [Online; resurss apskatīts 5-Apr-2014].

[8] Understanding the heap by breaking it. http://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf. [Online; resurss apskatīts 26-Apr-2014].

[9] Understanding the heap by breaking it. binning. http://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf. [Online; resurss apskatīts 26-Apr-2014].

[10] by The Free Software Foundation. Invoking gdb. http://www.delorie.com/gnu/docs/gdb/gdb_7.html. [Online; resurss apskatīts 22-Mar-2014].

[11] Christias, P. Standard signals. http://man7.org/linux/man-pages/man7/signal.7.html. [Online; resurss apskatīts 21-Mar-2014].

[12] Dhamdhere, D. M. *Systems Programming and Operating Systems*. Tata McGraw-Hill Publishing Company Limited, 2009, pp. 166--168.

[13] Kay A. Robbins, S. R. *UNIX SYSTEMS Programming*. Prentice Hall Professional, 2003, p. 257.

[14] Kerrisk, M. *The Linux Programming Interface*. No Starch Press, 2010, pp. 448--449.

[15] Leiterman, J. C. *32/64-BIT 80x86 Assembly Language Architecture*. Wordware Publishing, Inc., 2005, p. 44.

[16] Matt Welsh, Matthias Kalle Dalheimer, T. D. L. K. *Running Linux, Fourth Edition*. OŕReilly & Associates, Inc, 2003, p. 485.

[17] Richard Stallman, Roland Pesch, S. S. *Debugging with gdb*. Free Software Foundation, 2009, pp. 89--90.

[18] Sorfa, P. Debugging Memory on Linux. *Linux Journal* (2001).

Bakalaura darbs ,,Atmiņas izmetes pielietošana kaudzes atkļūdošanas metodes izstrādei" izstrādāts LU Datorikas fakultātē.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: _____ Renata Januškeviča

Rekomendēju/nerekomendēju darbu aizstāvēšanai

Vadītājs: Mg. sc. ing. Romāns Taranovs _____ 02.06.2014.

Recenzents: docents Dr.poniz. Jālis Bērziņš

Darbs iesniegts Datorikas fakultātē 02. 06. 2014.

Dekāna pilnvarotā persona: vecākā metodiķe Ārija Sproģe _____

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē

_____. prot. Nr. _____.

Komisijas sekretār__: lektore Anda Kooiņa _____