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

Bakalaura darba mērķis ir izstrādāt kaudzes atkļūdošanas metodi, kura ir balstīta uz atmiņās izmetes analīzi un ļauj bez tiešas piekļuves sistēmai atrast kaudzes problēmas programmā. Pētījuma rezultātā tiek identificētas kaudzes problēmu pazīmes atmiņas izmetē un ir piedāvāta kaudzes atkļūdošanas metode.

Izstrādājamā metode ir balstīta uz GNU C bibliotēkas iedalītāju un ir nodemonstrēta, izmantojot trīs kaudzes problēmu piemērus: atmiņas noplūdi, fragmentēšanu un maksimālās atmiņas izmantošanas problēmu.

Piedāvātā metode tika pārbaudīta, izstrādājot analizatorus, katrai izvēlētai problēmai, kuri ļauj parādīt atkļūdošanas metodi darbībā un liecina par to, ka izstrādājamā metode strādā un var tikt pielietota kaudzes atkļūdošanai.

Darbs sastāv no ievada, 4 nodaļam, secinājumiem un 5 pielikumiem. Tajā ir 50 lappuses, 27 attēli, 3 tabulas pamattekstā un 29 nosaukumi literatūras sarakstā.

Atslēgvārdi: atmiņas izmete, atkļūdošanas metode, kaudze, glibc.

ABSTRACT

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

The purpose of this paper is to develop a heap debugging method which is based on core dump analysis and allows finding heap problems in the program without direct access to the system. The research has identified characteristics of heap memory problems in core dumps, a method for heap debugging is proposed.

The method is based on GNU C library's implementation of heap memory allocator and is demonstrated using three problems of heap: memory leak, fragmentation, peak memory utilization.

The proposed method was verified by implementing analyzers for each of the problems. The analyzers demonstrate that the proposed method is feasible for heap memory debugging.

The work consists of introduction, 4 chapters, conclusions and 5 appendixes. It contains 50 pages, 27 figures, 3 tables and 29 references.

Keywords: core dump, debugging method, heap, glibc.

SATURS

Apzīmējumu saraksts 1
Ievads 3
1. Atmiņas izmete 4
1.1. Atmiņas izmetes ģenerēšana 4
1.1.1 Atmiņas izmetes ģenerēšana no koda 4
1.1.2 Atmiņas izmetes ģenerēšana no gdb 5
1.1.3 Atmiņas izmetes ģenerēšana no komandrindas interpretatora 6
1.2. Atmiņas izmetes ģenerēšanas nosacījumi 6
1.3. Atmiņas izmetes validēšana 6
1.4. Atkļūdošana, izmantojot atmiņas izmeti 7
1.4.1 Atmiņas izmetes atkļūdošana, izmantojot gdb 7
2. Atmiņas iedalīšana, organizācija un pārvaldība 10
2.1. Atmiņas iedalīšanas paņēmieni 10
2.1.1 Statiskā atmiņas iedalīšana 10
2.1.2 Dinamiskā atmiņas iedalīšana 11
2.2. Atmiņas pārvaldība 11
2.2.1 Kodola atmiņas pārvaldība 11
2.2.2 Lietotāja atmiņas pārvaldība 12
2.3. Atmiņas organizācija GNU C bibliotēkā 13
2.3.1 Atmiņas arēna 13
2.3.2 Atbrīvotās atmiņas organizācija 15
2.3.3 Atmiņas gabali 16
3. Kaudzes problēmas 20
3.1. Atmiņas noplūde 20
3.1.1 Atmiņas noplūdes pazīmes atmiņas izmetē 22
3.2. Maksimālās atmiņas izmantošanas problēma 24
3.2.1 Maksimālās atmiņas izmantošanas problēmas pazīmes atmiņas izmetē 25
3.3. Fragmentēšana 26
3.3.1 Fragmentēšanas pazīmes atmiņas izmetē 28
3.4. Datu kaudzes bojāšana 29

3.5. Kļūdas trešās puses bibliotēkās 29
3.6. Secinājumi 30
4. Atkļūdošanas metodes apraksts 31
4.1. Analizatora darbības princips 31
4.2. Atmiņas noplūdes analizators 32
4.3. Maksimālās atmiņas izmantošanas problēmas analizators 34
4.4. Fragmentēšanas analizators 35
Galvenie rezultāti un secinājumi 37
Pateicības 38
Izmantotā literatūra un avoti 41
Pielikumi 42
1. pielikums. Atmiņas noplūde 42
2. pielikums. Maksimālās atmiņas izmantošanas problēma 43
3. pielikums. Fragmentēšana 44
4. pielikums. Gdb skripts atmiņas noplūdes atkļūdošanai 45
5. pielikums. Gdb skripts pārējo divu problēmu atkļūdošanai 47

APZĪMĒJUMU SARAKSTS

Debugging - Atkļūdošana - Procedūra pieļauto kļūdu atrašanai, lokalizēšanai un novēršanai.

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

Heap - Kaudze - globāla datu struktūra, no kuras tiek iedalīta dinamiskā atmiņa procesam.

POSIX (Portable Operating System Interface) - IEEE un ISO standartu kopa, kas reglamentē kā rakstīt pieteikumu pirmkodā tā, lai lietotne būtu pārnēsājama 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.

Segment - Segments - blakusiedalītas atmiņas reģions.

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

Memory allocation - Atmiņas iedalīšana - atmiņas 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.

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

Chunk - Gabals - nepārtraukts atmiņas gabals ar noteikto struktūru.

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

dlmalloc (Doug Lea's Malloc) - atvērtā pirmkoda programmatūra, kura nodrošina lietotāja līmeņa atmiņas pārvaldību, uz kuru balstīta ptmalloc/ptmalloc2/ptmalloc3 realizācijas.

bin - viensaišu vai dubultsaišu saraksts, kurā tiek uzglabāti atbrīvoti atmiņas gabali.

Memory leak - Atmiņas noplūde - ir problēma, kas notiek nepareizās lietotāja atmiņas pārvaldības dēļ, kad atmiņa, kura vairs netiks izmantota programmā, netiek atbrīvota.

GNU C (glibc) - C valodas bibliotēka, kura nodrošina sistēmas izsaukumus un pamata funkcijas (malloc(), printf(), open()). Bibliotēku izmanto GNU operētājsistēmās.

LIFO (last-in-first-out) - Pēdējais iekšā-pirmais ārā - datu apstrādes procedūra, kas paredz kā pirmo apstrādāt pēdējo rindā ienākušo vai datnē ierakstīto informāciju.

FIFO (first-in-first-out) - Pirmais iekšā-pirmais ārā - datu elementu uzglabāšanas un atgūšanas metode, kas paredz, ka pirmais atmiņā ievietotais elements kā pirmais tiek arī nolasīts.

IEVADS

Daudzas problēmas C un C++ programmās ir saistītas ar dinamisko atmiņu [1]. Problēmas ir grūti atkļūdot, jo parasti nav tiešo pazīmju, kas liecinātu par problēmām. Tāpēc, lai saprastu, kas notiek, ir svarīgi zināt, kā realizēta atmiņas gabala iedalīšana un atmiņas pārvaldība no programmas un iedalītāja puses. Taču programmētāju zināšanas bieži ir ierobežotas ar malloc() un free() funkciju izmantošanu. Tāpēc, lai atkļūdotu lietotnes, tiek veidotas speciālās uzturēšanas komandas. Atkļūdošanas darbs nav viegls [2]. Parasti nav piekļuves klientu sistēmām vai piekļuve ir ierobežota, kā arī ļoti bieži sniegtā informācija par problēmu nav pietiekoša, lai varētu viennozīmīgi identificēt problēmu. Var būt grūti atkārtot problēmu pat tad, ja ir aprakstīti scenāriji un ir pielikta konfigurācija. Tas viss sarežģī atkļūdošanas procesu un padara darbu neefektīvu.

Šajā bakalaura darbā tiek aprakstīta un piedāvāta metode, kura palīdzes vienkāršot atkļūdošanas procesu problēmām, kas saistītas ar kaudzi un dinamisko atmiņas iedalīšanu. Piemēram, kaudzes problēmas var būt šādas: atmiņas noplūde, datu kaudzes bojāšana, maksimālās atmiņas izmantošanas problēma, fragmentēšana, kļūdas trešās puses bibliotēkās. Darbā ir aprakstīta atkļūdošanas metode, kura balstīta uz atmiņas izmetes analīzi. Šī metode ir aprakstīta trijos izvēlētos problēmu piemēros: atmiņas noplūde, fragmentēšana, maksimālās atmiņas izmantošanas problēma. Atmiņas noplūde ir izvēlēta, tāpēc ka tā ir viena no bieži sastopamām problēmām [3]. Fragmentēšana un maksimālās atmiņas izmantošanas problēma ir divas problēmas, kuras pēc autores viedokļa, ir tuvas, un var tikt atkļūdotas ar atmiņas izmetes palīdzību. Metode var tikt pielietota gadījumos, kad ir novērojamas problēmu sekas vai, kad ir vēlme pārliecināties par problēmas eksistenci, sasniedzot kādu stāvokli programmā.

Darbs sastāv no ievada, četrām nodaļām un secinājumiem:

- Nodaļā "Atmiņas izmete" ir aplūkots atmiņas izmetes jēdziens, atmiņas izmetes ģenerēšanas un izmantošanas iespējas.
- Nodaļā "Atmiņas iedalīšana, organizācija un pārvaldība" ir aprakstīta atmiņas organizācija un ptmalloc2 realizācija.
- Nodaļā "Kaudzes problēmas" ir pētītas trīs no piecām izvēlētām problēmām un identificētas to pazīmes atmiņas izmetē, nodaļā iekļauti arī pārējo divu problēmu apraksti.
- Nodaļā "Atkļūdošanas metodes apraksts" ir piedāvāta metode, kura balstīta uz iepriekš izklāstītiem jēdzieniem.
- Nodaļā "Galvenie rezultāti un secinājumi" ir apkopoti darba laikā gūtie rezultāti un secinājumi.

1. ATMIŅAS IZMETE

Šajā nodaļā tiek aplūkots atmiņas izmetes jēdziens, atmiņas izmetes ģenerēšanas iespējas un nosacījumi. Nodaļā ir aprakstīts atkļūdošanas process, kas varētu tikt paveikts, izmantojot atmiņas izmeti.

1.1. Atmiņas izmetes ģenerēšana

Sistēmās, kuras atbalsta POSIX standartus, ir signāli [4], 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 [5] 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, tādā kā gdb, lai apskatītu programmas stāvokli uz brīdi, kad atnāca operētājsistēmas signāls [6]. Veicot atmiņas izmetes analīzi, kļūst iespējams atrast un izlabot kļūdas, pat tad, ja nav tiešas piekļuves sistēmai.

Reālajās sistēmās atmiņas izmetes tiek uzģenerētas atmiņas kļūdu dēļ. Dažas no kļūdām sīkāk ir aprakstītas 3. sadaļā. Bet eksistē vairākas iespējas, kā atmiņas izmeti var uzģenerēt patstāvīgi. Tas varētu būt nepieciešams programmas stāvokļa apskatīšanai. Atmiņas izmeti var uzģenerēt 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 darbību pēc signāla nosūtīšanas.

```c
#include <signal.h>

int main () {

    raise(SIGSEGV); /* signāls, kurš tiek nosūtīts, kad ir nederīga norāde */

    return 0;
}
```

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

Ja nav nepieciešams, lai process turpinātu darbību, tad var izmantot funkciju raise()

vai abort(), kā arī var apzināti pieļaut kļūdu kodā. Tāda kļūda kā dalīšana ar nulli nosūta SIGFPE signālu, bet vēršanās pēc nulles radītāja - SIGSEGV signālu. Izmantojot funkciju raise(), ir iespējams norādīt atmiņas izmeti izraisošo signālu. Piemērā (sk. 1.1. attēlu) 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.

```c
#include <stdlib.h>

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

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

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

1.1.2. Atmiņas izmetes ģenerēšana no gdb

Atmiņas izmetes ģenerēšanas nolūkam var izmantot gdb komandas: `generate-core-file [file]`(sk. 1.3. attēlu) 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 un S390 sistēmās [6].

```
(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 komandrindas interpretatoru, ir iespējams izmantot īsinājumtaustiņus signālu nosūtīšanai. Nospiežot Control + \, tiks nosūtīts SIGQUIT signāls procesam, kas pašreiz ir palaists (sk. 1.4. attēlu) [7]. Šajā piemērā ziņojumu - Quit (core dumped), izdruka komandrindas interpretators, kurš noteic, ka sleep procesu pārtrauca SIGQUIT signāls. Pēc šī signāla nosūtīšanās darba mapē tiek uzģenerēta atmiņas izmete.

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

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

1.2. Atmiņas izmetes ģenerēšanas nosacījumi

Lai uzģenerētu atmiņas izmeti, ir jābūt izpildītiem šādiem nosacījumiem [7]:

- ir jānodrošina atļauja procesam rakstīt atmiņas izmeti darba mapē;
- ja datne ar vienādu nosaukumu jau eksistē, tad uz to 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 jābūt lielākai par ģenerējamā faila izmēru, RLIMIT_FSIZE robežai jāļauj procesam izveidot atmiņas izmeti;
- ir jāatļauj lasīt bināro datni, kura tika palaista;
- failu sistēmai, kurā atrodas darba mape, ir jābūt uzmontētai rakstīšanai, tai nav jābūt pilnai un ir jāsatur brīvie indeksa deskriptori;
- binārā datne 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.

1.3. Atmiņas izmetes validēšana

Šis posms nav pietiekoši labi izpētīts literatūrā. Bet, pirms sākt atkļūdošanu, ir nepieciešams pārliecināties, ka atmiņas izmete var tikt pielietota šim nolūkam. Ģenerējot atmiņas

izmeti, ir iespējams, ka datnes izmērs sasniegs vairākus gigabaitus un datne tiks nogriezta. Tas notiek, jo pieejamā brīva vieta ir ierobežota, un, kopējot, pārsūtot vai ģenerējot atmiņas izmeti, var pazaudēt daļu no datiem. Šo gadījumu var pārbaudīt, izdrukajot galveni ar `objdump -p <core>` komandu. Gadījumos, kad atmiņas izmete tiek nogriezta, tad tiks izvadīts brīdinājums par to, ka datnes izmērs neatbilst sagaidāmajam. Nogriezts saturs varētu traucēt analizēt atmiņas izmeti, jo tiek sagaidīts, ka dati ir pilnīgi.

1.4. 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 tā ir speciāli uzģenerēta datne, kura palīdz iegūt atmiņas stāvokli signāla nosūtīšanas brīdī. Atmiņas izmete ir labi piemērota kļūdu meklēšanai, kas saistītas ar nepareizo atmiņas izmantošanu lietotnē.

Atmiņas izmete ir ELF, a.out vai cita formātā binārā datne. ELF formāts ir Linux un Unix standarts izpildāmām datnēm, objektu datnēm, bibliotēkām un atmiņas izmetēm. Lai darbotos ar atmiņas izmetem, ir nepieciešams, lai rīks, kurš tika izvēlēts (bibliotēka, utilītprogramma vai atkļūdotājs), atbalstītu uzģenerētās datnes formātu. GNU gdb ir Linux standarta atkļūdotājs [8], kurš ir plaši pielietojams atmiņas izmešu analīzei. Turpmāk tiek apskatīta atmiņas izmetes analīze ar gdb atkļūdotāja palīdzību.

1.4.1. Atmiņas izmetes atkļūdošana, izmantojot gdb

Ja atmiņas izmetes analīzei tika izvēlēts GNU gdb atkļūdotājs, tad, pirms sākt analīzi, ir nepieciešams pārliecināties, ka gdb ir pareizi nokonfigurēts procesora arhitektūrai, no kuras bija iegūta atmiņas izmete. To var identificēt uzreiz pēc gdb palaišanas, ar šādas 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, kā arī uzglabā mainīgo un funkciju datu tipus. Ja atmiņas izmete neiekļauj atkļūdošanas informāciju, tad var tikt izdrukāts šāds teksts (sk. 1.5. attēlu):

```
1 (gdb) p main
2 $ 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. attēlu). Atkļūdotājam kā argumenti tiek padoti: izpildāms fails un atmiņas izmete. Izpildāmam 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, un uzzināt, kuram segmentam pieder nezināmā atmiņas adrese.

Lai izdrukātu atmiņas apgabalu, var izmantot instrukciju ar šādu formātu: `x/nfu addr`. Ir nepieciešams norādīt atmiņas adresi (addr), no kuras sākt atmiņas izdruku, formātu (f), apgabala lielumu (n) un norādīt vienības lielumu (u). Izmantojot doto piemēru (sk. 1.7. attēlu), tiks izdrukāti četri elementi, kuri pieder stekam, jo Intel x86 procesoros 32 bitu režīmā uz steku norāda $esp reģistrs. Formātu un vienības lielumu vajag norādīt saskaņā ar gdb pamācību [9]. Dotajā gadījumā atmiņa tiks izdrukāta heksadecimālā formātā (x), un vienības lielums ir vārds (word) jeb 4 baiti.

```
(gdb) x/4wx $esp
```

1.7. att. **Atmiņas apgabala izdrukāšana**

Lai uzzinātu, kuram simbolam (funkcijai, mainīgam vai tipam) pieder adrese, var izmantot šādu gdb instrukciju (sk. 1.8. attēlu) [9]. Instrukcija `print` vai `p` ļauj izdrukāt datus, bet `p/a` izdrukā absolūto adresi un relatīvo jeb adresi ar nobīdi no tuvāka simbola, kuram pieder adrese. Tādā veidā var noteikt, kuram atmiņas apgabalam pieder nezināmā adrese.

```
(gdb) p/a 0x54320
$3 = 0x54320 <_initialize_vx+396>
```

1.8. att. **Noteikšana, kuram simbolam pieder adrese**

Atmiņas izmetes analīze sākas ar backtrace izdrukāšanu. Backtrace ir pārskats, kurš attēlo, kā programma nonāca stāvoklī, kurā pabeidza savu darbību. Tas palīdz ātri atrast instrukciju, kura bija izpildīta pēdēja un daudzos gadījumos, ļauj ātri identificēt kļūdas cēloni.

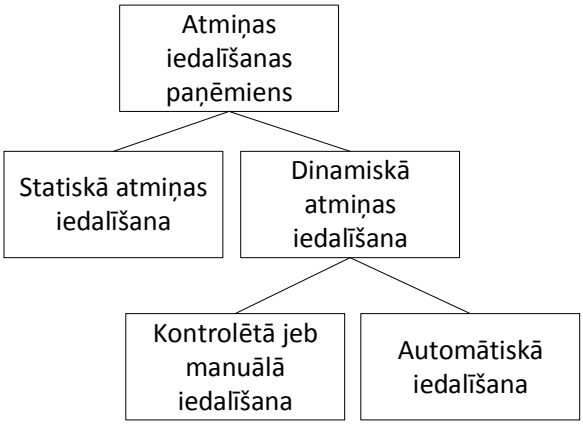
Backtrace nesniedz patieso informāciju par funkciju, ja process tika pabeigts ārējo apstākļu dēļ, nevis tāpēc, ka bija notikusi kļūda programmā. Katra rindiņa satur rāmi (frame). Backtrace izdruka sākas ar rāmi, kurā iekļauta funkcija, kura bija izpildīta pēdēja. Nākamais rāmis iekļauj funkciju, kas izsauca iepriekšējā rāmī iekļauto funkciju. Katrai backtrace rindiņai tiek piešķirts rāmja numurs. Katrs rāmis var iekļaut: funkcijas nosaukumu, pirmkoda datnes nosaukumu, pirmkodam atbilstošo rindiņas numuru un funkcijas argumentus. Backtrace var tikt iegūts, izmantojot gdb komandu `backtrace full` vai `bt f`. Pēc noklusējuma daudzpavedienu lietotnēs gdb rāda backtrace kārtējām pavedienam, bet pastāv iespēja iegūt arī backtrace izdruku citiem pavedieniem. Ja programma bija nokompilēta ar optimizācijas opciju, tad backtrace varētu neiekļaut funkcijas argumentus. Šajā gadījumā funkciju argumenti varētu tikt nodoti caur CPU reģistriem, kuru vērtības ir iespējams iegūt, izmantojot komandu `info registers` vai `i r`. Atmiņas izmetē atrodas pēdējais atmiņas stāvoklis, tāpēc CPU reģistru vērtības visticamāk tiks pārrakstītas. Ja ir nepieciešamība, tad reģistru vērtības ir iespējams atjaunot no steka.

2. ATMIŅAS IEDALĪŠANA, ORGANIZĀCIJA UN PĀRVALDĪBA

Šī nodaļa palīdz saprast kopējo atmiņas organizāciju un kaudzes[1] lomu tajā. Šajā nodaļā ir aprakstīti atmiņas iedalīšanas paņēmieni, ir dots īss ieskats GNU C bibliotēkas kaudzes atmiņas iedalītāja ptmalloc2 realizācijā un aprakstīta atmiņas pārvaldība no lietotāja un no kodola puses.

2.1. Atmiņas iedalīšanas paņēmieni

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



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

2.1.1. Statiskā atmiņas iedalīšana

Statiskā atmiņas iedalīšana ir atmiņas iedalīšana pirms programmas palaišanas, parasti tas notiek kompilācijas laikā. Programmas izpildes laikā atmiņa vairs netiek iedalīta, kā 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 pie dotajiem nosacījumiem vai nē.

[1]Šī termina nozīme atšķiras no datu struktūras "kaudze", kurā elementi tiek izvēlēti saskaņā ar to prioritāti.

2.1.2. Dinamiskā atmiņas iedalīšana

Dinamiskā atmiņas iedalīšana nozīmē, 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 automātiska vai kontrolēta [10].

Automātiskā iedalīšana notiek, kad sākās programmas funkcijas izpilde. Automātiskās atmiņas iedalīšanai, izmanto steku. Šeit viens un tas pats atmiņas apgabals, kurš bija atbrīvots, var tikt izmantots vairākas reizēs. Piemēram, funkcijas argumenti un lokālie mainīgie ir saglabāti stekā un izdzēsti pēc šīs funkcijas izpildes. Pēc tam atbrīvotā atmiņa var būt izmantota atkārtoti. Vērtību izdzēšana vai saglabāšana notiek, nobīdot steka norādi. Visiem funkcijas mainīgiem var piekļūt izmantojot steka norādes nobīdi, kas tiek uzglabāta reģistrā, piemēram, Intel x86 procesoros, 16 bitu režīmā tas ir reģistrs `SP`, 32 bitu režīmā - `ESP` un 64 bitu režīmā - `RSP` [11].

Kontrolētā atmiņas iedalīšana nozīmē, ka programma izvēlas brīvus atmiņas gabalus no pieejama segmenta telpas programmas datiem. Kontrolētā jeb manuālā atmiņas iedalīšana parasti ir nodrošināta ar kaudzes palīdzību. Šeit nav iespējams piekļūt visiem iedalītajiem atmiņas gabaliem, izmantojot vienu norādi un tās nobīdi, piemēram, kā tas tiek nodrošināts stekā. Tagad katram iedalītam atmiņas gabalam var piekļūt tikai tad, ja ir norāde uz šo iedalīto atmiņas gabalu. Gadījumos, kad norādes nav, tad adreses vairāk nav sasniedzamas un kļūst pazaudētas. Turpmāk darbā, termins "dinamiskā atmiņa" apzīmēs atmiņu, kura tiek iedalīta, izmantojot kontrolēto atmiņas iedalīšanu.

2.2. Atmiņas pārvaldība

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

2.2.1. Kodola atmiņas pārvaldība

Operētājsistēmas kodols apstrādā visus atmiņas pieprasījumus, kas attiecas uz programmu vai programmas instancēm. Kad lietotājs sāk programmas izpildi, tad kodols iedala atmiņas apgabalu tekošajam procesam. Šis apgabals, no procesa viedokļa, ir viena lineārā virtuālā adrešu telpa, kura ir sadalīta vairākos segmentos. Iedalīto virtuālo adrešu karti var atrast `/proc/<pid>/maps` datnē. Svarīgākie procesa segmenti [12]:

- Teksta segments - šeit tiek uzglabāti dati, kuri tiek izmantoti tikai lasīšanai. Tās ir nokompilētas koda instrukcijas. Vairākas programmas instances var izmantot šo atmiņas apgabalu.

- Statisko datu segments - apgabals, kurā tiek uzglabāti dati ar iepriekš zināmu izmēru. Tie ir globālie un statiskie mainīgie. Operētājsistēma iedala šī apgabala kopiju katrai programmas instancei atsevišķi.
- Kaudzes segments - apgabals, no kura tiek iedalīta dinamiskā atmiņa. Tajā atrodas dinamiski iedalītā un atbrīvotā atmiņa. Kaudzes segmenta saturs ir sadalīts sīkāk, mazākos atmiņas gabalos un aug no mazākas adreses līdz lielākai. Lai palielinātu kaudzes segmenta izmēru, tiek veikts brk() sistēmas izsaukums, kurš uzstāda jauno beigu robežu kaudzes segmentam [13].
- Steka segments - apgabals, kurā tiek uzglabāti: funkciju izsaukumu stāvoklis, katram funkcijas izsaukumam, kā arī lokālo mainīgo un reģistru vērtības. Steks aug no lielākas adreses līdz mazākai. Steks ir iedalīts katrai programmas instancei atsevišķi.

2.2.2. Lietotāja atmiņas pārvaldība

Atmiņa, kas var tikt dinamiski iedalīta, parasti ir novietota kaudzē. Lietotāja atmiņas pārvaldība ir dinamiskās atmiņas pārvaldība no lietotnes. Lai nodrošinātu lietotāja atmiņas pārvaldību no lietotnes, ir nepieciešams iedalītājs (allocator), kurš veic sistēmas izsaukumus un pārvalda iegūto atmiņu, sadalot to sīkākos gabalos. Iedalītājs ļauj efektīvāk pārvaldīt atmiņu, nekā tas būtu nodrošināts, katru reizi pieprasot atmiņas gabalu ar sistēmas izsaukumiem. Šobrīd eksistē vairāki iedalītāji, piemēram, Hoard memory allocator, ptmalloc2, dlmalloc. Iedalītāju galvenie uzdevumi:

1. sekot atmiņas gabaliem, kuri ir izmantoti;
2. sekot atbrīvotiem atmiņas gabaliem;
3. nodrošināt iespēju atkārtoti izmantot atmiņu.

Dažreiz tiek izveidots individuālā iedalītāja risinājums. Kaut arī daži universālie iedalītāji strādā pietiekoši ātri un fragmentēšanas līmenis ir zems, individuālais risinājums var ņemt vērā lietotnei raksturīgas īpatnības un tādējādi nodrošināt labāko veiktspēju [14].

```
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++**

GNU C bibliotēkā ir iebūvēts ptmalloc2 iedalītājs. Turpmāk tiks apskatīta lietotāja atmiņas pārvaldība no lietotnes, izmantojot GNU C bibliotēkas funkciju palīdzību, ko nodrošina ptmalloc2. C valodā dinamiskā atmiņa tiek pārvaldīta ar malloc(), realloc(), free()

un calloc() funkciju palīdzību [12]. C++ valodā ir izmantots operators new, lai pieprasītu atmiņu. Attēlā 2.2. ir redzama C un C++ sintakse atmiņas pieprasīšanai.

Funkcija malloc() ir definēta malloc.c datnē GNU C bibliotēkā. Funkcijas prototips ir definēts <stdlib.h>. Funkcija malloc() ļauj dinamiski iedalīt atmiņu procesam. Vienīgais arguments malloc() funkcijai ir baitu skaits. C programmai, lai saskaitītu, cik baitu ir nepieciešams pieprasīt, vajag 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 gabalu inicializētu ar 0 vērtībām.

Funkcija free() atbrīvo ar malloc() palīdzību iedalīto atmiņu. Lielāka atšķirība starp free() un delete ir tāda, ka vecajās free() realizācijās netiek nodrošināts atbalsts free() funkcijai, kad arguments ir null [15].

Programmas rakstīšanā nejauc kopā C un C++ stilus, tāpēc C++ programmām izmanto new un delete operatorus (sk. 2.3. attēlu), bet C programmām malloc() un free(). Ja atmiņa pēc izmantošanas netiek nekad atbrīvota, un katru reizi, izpildot vienu un to pašu koda gabalu, iedalīta no jauna, tad no operētājsistēmas pieejams atmiņas daudzums ar laiku samazinās.

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

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

2.3. Atmiņas organizācija GNU C bibliotēkā

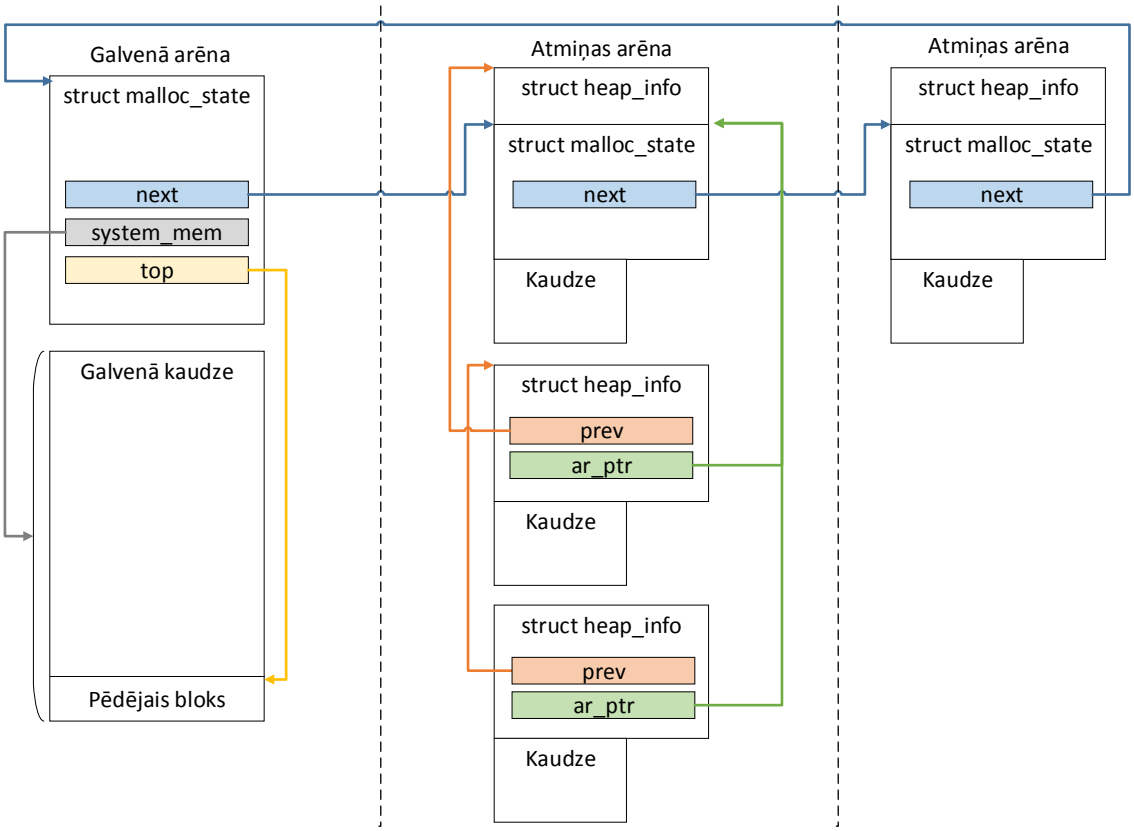
Darbā tiek aplūkota GNU C bibliotēkas (versija 2.3) ptmalloc2 realizācija, kuru izstrādāja Wolfram Gloger, balstoties uz Doug Lea dlmalloc realizāciju. Atmiņas iedalīšana sākas ar malloc() vai līdzīgo funkciju izsaukumiem no programmas koda un tiek nodrošināta ar GNU C bibliotēkas palīdzību.

2.3.1. Atmiņas arēna

Atmiņas arēnu var nosaukt par loģisko atmiņas kolekciju. 2.4. Attēlā[1] ir parādītas trīs arēnas, kuras ir atdalītas savā starpā ar raustītām līnijām. Atmiņas arēnu vienkāršotā veidā

[1] Attēla izveidošanai tika izmantots GNU C malloc pirmkods [16] un vietnē nopublicēta shēma [17]. Attēls demonstrē atmiņas organizāciju.

var attēlot kā viensaišu saistīto sarakstu, kurš sastāv no vienas vai vairākām atmiņas kaudzēm. Kaudze ir lineārās apgabals, kurš iekļauj sevī iedalītus un atbrīvotus atmiņas gabalus (chunk of memory), kuri ir novietoti blakus viens otram. Atmiņas gabali sīkāk ir aprakstīti 2.3.3. sadaļā. Gadījumos, kad gabals ir iedalīts, tad pašreiz palaists process satur norādi uz



2.4. att. **Atmiņas organizācija GNU C bibliotēkā (versija 2.3)**

iedalīto apgabalu kaudzē. Ja gabals ir atbrīvots, tad tas tiek pievienots vienā no sarakstiem, uz kuriem norāda bin masīvi, kuri atrodas vienā no arēnām. Bin masīvs un bin saraksti sīkāk ir aprakstīti 2.3.2. sadaļā. Katrā arēnā ir rādītājs uz nākamo izveidoto arēnu. Pēdējā izveidotā arēna norāda uz galveno arēnu. Ja kārtējā kaudze ir izlietota un tajā nav atmiņas, tad tiek iedalīta jauna kaudze ar fiksēto 64 MB izmēru. Tādā veidā arēnas var tikt paplašinātas, izveidojot jaunās kaudzes un savienojot tās savā starpā. Jaunai kaudzei ir norāde gan uz arēnu, kurai tā pieder, gan uz iepriekšējo kaudzi.

Lai uzlabotu veiktspēju vairākpavedienu procesiem, GNU C bibliotēkā tiek izmantotas vairākas atmiņas arēnas. Katrs funkcijas malloc() izsaukums bloķē arēnu, no kuras tiek pieprasītā atmiņa. Laikā, kad arēna ir nobloķēta, notiek atmiņas gabala iedalīšana. Kad vairākiem pavedieniem ir nepieciešams vienlaicīgi iedalīt atmiņu no kaudzes un visi pavedieni

mēģina piekļūt vienai un tai pašai arēnai (tas varētu notikt dlmalloc realizācijā), tad arēnas bloķēšana var būtiski samazināt veiktspēju. Gadījumos, kad pavedieni izmanto atmiņu no vairākām atsevišķām arēnām (kā tas notiek ptmalloc2 realizācijā), tad vienas arēnas bloķēšana neietekmē atmiņas iedalīšanu parējās kaudzēs, kuras nepieder nobloķētai arēnai, un atmiņas iedalīšana var notikt paralēli. Lai nodrošinātu labāku veiktspēju, GNU C bibliotēkā tiek izmantots modelis: katram pavedienam - viena arēna. Ja malloc() pirmo reizi izsaukts pavedienā, tad neatkarībā no tā vai kārtējā arēna bija nobloķēta vai nē, tiks izveidota jauna arēna. Arēnu skaits ir ierobežots atkarībā no kodolu skaita, 32 bitu vai 64 bitu arhitektūras un mainīgā MALLOC_ARENA_MAX vērtības. Tā kā pavedienu skaits parasti nepārsniedz divkāršo kodolu skaitu, tad normālā gadījumā katrs pavediens izmanto atsevišķo arēnu. Darbība ar arēnām notiek saskaņā ar šādu algoritmu:

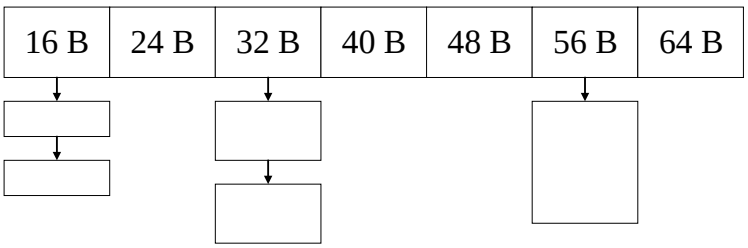
1. malloc() izsaukums vēršas pie arēnas, kurai piekļuva iepriekšējā reizē;
2. ja arēna ir nobloķēta, tad malloc() vēršas pie nākamās izveidotās arēnas;
3. ja nav piekļuves nevienai arēnai, tad tiek izveidota jauna arēna, un malloc() vēršas pie tās.

Vispirms, atmiņas iedalīšana sākas no galvenās arēnas (main arena). GNU C bibliotēkā ir globāls `malloc_state` objekts - galvenā arēna, kura atšķiras no pārējām arēnām ar to, ka tā tiek paplašināta, izmantojot brk() nevis mmap() sistēmas izsaukumu. Līdz ar galvēnās arēnas paplašināšanu tiek paplašināts arī procesa kaudzes segments. brk() sistēmas izsaukumam ir viens arguments, kurš uzstāda kaudzes segmentam jaunas beigas. mmap() sistēmas izsaukums paplašina pārējās dinamiskās arēnas daudzpavedienu lietotnēs, kā arī nodrošina lielu atmiņas bloku iedalīšanu mmap apgabalā. Mazākais gabals, kurš pēc noklusējuma tiks iedalīts ar mmap(), ir vienāds ar 128 kilobaitiem. Sākot ar GNU C bibliotēkas 2.18 versiju, mazāko gabalu, kurš tiks iedalīts ar mmap(), var uzdot ar M_MMAP_THRESHOLD konstanti.

2.3.2. Atbrīvotās atmiņas organizācija

Atbrīvots atmiņas gabals ne vienmēr tiks uzreiz atgriezts operētājsistēmai (sīkāk tas ir aprakstīts 3.2. sadaļā), bet var tikt defragmentēts vai sapludināts ar pārējiem gabaliem un ievietots sarakstā. Realizācijā ptmalloc2 ir masīvi, kuri uzglabā norādes uz bin sarakstiem. Bin saraksti ir struktūras, kuras uzglabā atbrīvotus atmiņas gabalus, līdz brīdim, kad tie tiks iedalīti procesam atkārtoti. Sarakstiem netiek atsevišķi iedalīta atmiņa, bet tiek izmantota kaudzes atmiņa. Tas kļūst iespējams, pārrakstot atmiņas gabalu struktūru. Ja atmiņa bija atbrīvota, tad atmiņas gabali var tikt uzglabāti vienā no bin saistītajiem sarakstiem. Eksistē divi bin saraksta veidi: ātrais (fastbin) un parastais (normal bin).

Ātrais saraksts ir paredzēts bieži izmantotu, mazu atmiņas gabalu glabāšanai. Pēc noklusējuma ātro atmiņas gabalu izmērs nepārsniedz 64 baitus (sk. 2.5. attēlu), bet to var palielināt līdz 80 baitiem [16]. Tas varētu būt nepieciešams, ja programmā ir bieži izmantotas struktūras, kuru izmērs pārsniedz 64 baitus. Atmiņas gabali atrodas viensaišu sarakstā un nav sakārtoti, jo katrā bin sarakstā atrodas elementi, kuriem ir vienāds izmērs. Lai samazinātu fragmentēšanas 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 parastiem gabaliem. Fastbin saraksta elementi ir apstrādāti: pēdējais-iekšā, pirmais-ārā (jeb LIFO) kārtībā [16]. Kad tiek pieprasīta atmiņa no fastbin saraksta, tad jebkurš atmiņas gabals tiek atgriezts konstantā laikā [18].



2.5. att. **Ātrais saraksts**

Kopumā ir 128 parastie saraksti, kurus var sadalīt 3 veidos. Pirmkārt, bin saraksts, kurš 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. Mazā izmēra saraksti iekļauj gabalus ar vienādu izmēru. Lielā izmēra saraksts uzglabā atmiņas gabalus, kuri ir lielāki par 512 baitiem, bet mazāki par 128 kilobaitiem. Lielā izmēra saraksta elementi ir sakārtoti pēc izmēra un ir iedalīti: pirmais-iekšā, pirmais-ārā (jeb FIFO) kārtībā [16]. Tādā veidā vienmēr tiek atgriezts gabals, kurš ir vislabāk piemērots. Tas ir, kad gabalam ir mazāks izmērs no pārējiem saraksta gabaliem, kurš apmierina pieprasījumu pēc atmiņas. Gabali, kuru izmērs ir lielāks par 128 kilobaitiem, netiek uzglabāti bin sarakstos, jo tiek iedalīti, izmantojot mmap().

2.3.3. Atmiņas gabali

Kaudze sastāv no daudziem atmiņas gabaliem. Eksistē divu veidu atmiņas gabali: parastie (normal chunk) un ātrie (fast chunk) gabali. Ātrie gabali ir maza izmēra (parasti līdz 64 baitiem) un pieder ātrajam sarakstam, bet parastie gabali - parastajam sarakstam. Ātrie un parastie gabali, tiek izmantoti, lai nodrošinātu atmiņas iedalīšanu no kaudzes. Atmiņas

gabala fiziska struktūra ir vienāda abu veidu gabaliem, bet ir atkarīga no stāvokļa un var tikt interpretēta dažādi. Atmiņa no kaudzes tiek iedalīta, izmantojot malloc_chunk struktūru (sk. 2.6. attēlu). Sīkāk struktūras malloc_chunk elementi ir aprakstīti tabulā 2.1.

2.1. tabula
Atmiņas gabalu struktūras elementu apraksts

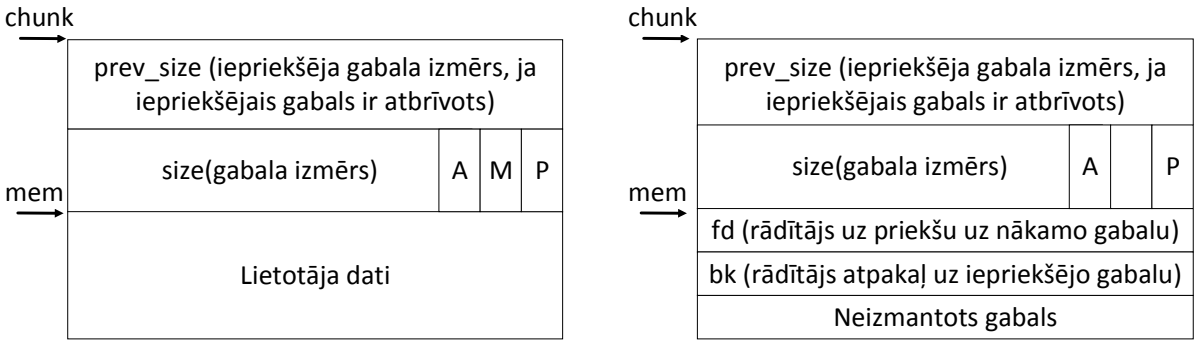
Elements	Nozīme
INTERNAL_SIZE_T prev_size	Iepriekšēja gabala izmērs (baitos), ja tas bija atbrīvots
INTERNAL_SIZE_T size	Kārtējā gabala izmērs (baitos)
struct malloc_chunk* fd	Rādītājs uz nākamo atbrīvoto gabalu, ja kārtējais gabals ir atbrīvots un pievienots dubultsaišu bin sarakstam
struct malloc_chunk* bk	Rādītājs uz iepriekšējo atbrīvoto gabalu, ja kārtējais gabals ir atbrīvots un pievienots dubultsaišu bin sarakstam

```
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  }
```

2.6. att. **Atmiņas gabala struktūra**

Katru reizi ir iedalīts lielāks atmiņas gabals nekā pieprasīts ar malloc() funkciju. Tas ir nepieciešams, lai varētu saglabāt uzturēšanai nepieciešamo informāciju. Iedalītam gabalam uzturēšanas informācija ir divas INTERNAL_SIZE_T tipa vērtības, kas vienādas ar 4*2 vai 8*2 baitiem. Tas ir atkarīgs no tā, kāda vērtība ir piešķirta INTERNAL_SIZE_T makrodefinīcijai, 4 vai 8 baiti. Ar INTERNAL_SIZE_T var uzdot iekšējo vārda izmēru (word-size), kurš pēc noklusējuma ir vienāds ar size_t izmēru. Datoriem ar 64 bitu tehnoloģiju, 4 baitu vērtības piešķiršana makrodefinīcijai var samazināt aizņemtās atmiņas daudzumu, bet ierobežo lielāko iespējamo gabala izmēru. Tā kā 4 baitos nevar saglabāt skaitli, kas ir vienāds vai lielāks par 2^{32}, tad laukā `prev_size` un `size` vērtībai ir jābūt mazākai par šo ierobežojošo vērtību. Kad gabals ir iedalīts, tad uzturēšanas informācijai ir izmantotas divas INTERNAL_SIZE_T tipa vērtības un, kad gabals ir atbrīvots, tad dubultsaišu saraksta uzturēšanai, papildus tiek izmantoti divi rādītāji (bk un fd) uz iepriekšējo un nākamo `malloc_chunk` struktūras objektiem. Kopējais atmiņas gabala uzturēšanai izmantotais datu izmērs var būt 16 baiti (ja INTERNAL_SIZE_T un rādītāja izmērs ir 4 baiti), 24 baiti (ja INTERNAL_SIZE_T ir 4/8 baiti un rādītāja izmērs ir 8/4 baiti) vai 32 baiti (ja INTERNAL_SIZE_T un rādītāja izmērs ir 8 baiti). Otrs iemesls, kāpēc ir iedalīts lielāks atmiņas

daudzums, ir izlīdzināšana skaitlim, kas ir 2*sizeof(INTERNAL_SIZE_T) reizinājums. Šis skaitlis ir vienāds ar 8 baitu izlīdzinājumu, ja makrodefinīcijas INTERNAL_SIZE_T vērtība ir vienāda ar 4 baitiem [16].



(a) Iedalīts atmiņas gabals (b) Atbrīvots atmiņas gabals

2.7. att. **Atmiņas gabalu grafiskais struktūras attēlojums**

No kreisās puses attēlots (sk. 2.7. attēlu) [16] atmiņas gabals, kurš bija iedalīts procesam, no labās, tas, 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 chunk uzglabā iepriekšēja gabala pēdējos baitus no lietotāja datiem. Pēc tam seko kārtēja gabala izmērs un 3 biti ar meta informāciju.

Tā kā notiek izlīdzināšana 2*sizeof(INTERNAL_SIZE_T), kas ir vienāda 8 - ka vai 16 - ka reizinājumam, tad 3 pēdējie biti netiek izmantoti izmēra glabāšanai. Šos bitus izmanto kontroles zīmēm. Katram bitam ir sava nozīme, kura aprakstīta 2.2. tabulā.

2.2. tabula
Atmiņas gabala kontroles zīmes

Kontroles zīme	Nozīme
A	gabals nepieder galvenajai arēnai
M	gabals tiek iedalīts ar mmap() sistēmas izsaukumu
P	iepriekšējais atmiņas gabals tiek izmantots

Rādītājs mem ir malloc() funkcijas atgriežamā vērtība, jeb radītājs uz iedalīto atmiņas apgabalu. 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 atbrīvoto gabalu).

Eksistē divi citi atmiņas gabali (top chunk un last_remainder), kuriem ir īpaša nozīme. Top chunk ir atmiņas gabals, kuram ir kopīga robeža ar procesa kaudzes segmentu. Top gabals

ir izmantots gadījumos, kad nav piemērotu gabalu bin sarakstos, kuri apmierina pieprasījumu vai varētu būt saplūdināti, lai apmierinātu pieprasījumu pēc atmiņas. Sākotnēji atmiņas iedalīšana sākas ar top gabalu, bet top gabals nodrošina arī pēdējo iespēju iedalīt pieprasīto atmiņas daudzumu. Top gabals var mainīt savu izmēru. Tas saraujas, kad atmiņa ir iedalīta un izstiepjas, kad atmiņa ir atbrīvota blakus top gabala objektam. Ja ir pieprasīta atmiņa, kas ir lielāka par pieejamo, tad top gabals var paplašināties ar brk() izsaukuma palīdzību. Top gabals ir līdzīgs jebkuram citam atmiņas apgabalam. Galvenā atšķirība ir lietotāja datu sekcija, kura netiek izmantota, P kontroles zīme, kura vienmēr norāda, ka iepriekšējais gabals ir izmantots, kā arī speciāla top gabala apstrāde, lai nodrošinātu, ka top gabals vienmēr eksistē [16].

Last_remainder ir vēl 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 [16].

3. KAUDZES PROBLĒMAS

Par kaudzes problēmām tiek uzskatītas problēmas, kuras rodas lietotnē nepareizās kaudzes pārvaldības dēļ:

- ja ir nekorekta iedalītāja izmantošana;
- ja ir nekorekta iedalītāja realizācija.

Nodaļā tiek aprakstītas piecas kaudzes problēmas: atmiņas noplūde, fragmentēšana, maksimālās atmiņas izmantošanas problēma, datu kaudzes bojāšana, kļūdas trešās puses bibliotēkās. Izvēlētās pirmās trīs problēmas no piecu problēmu saraksta, tiek pētītas sīkāk. Katrai no trim problēmām ir identificētas pazīmes atmiņas izmetē. Balstoties uz zināšanām par pazīmēm turpmāk ir veidoti analizatori. Pārējās divas problēmas netiek detalizēti pētītas un aplūkotas nodaļas beigās.

3.1. Atmiņas noplūde

Atmiņas noplūde (memory leak) ir viena no bieži sastopamām problēmām C un C++ valodās [3]. Atmiņas noplūde notiek nepareizās lietotāja atmiņas pārvaldības dēļ, kad atmiņa, kura vairs netiks izmantota programmā, netiek atbrīvota.

Atmiņas noplūdes problēmu var sadalīt divos dažādos veidos: fiziskā un loģiskā atmiņas noplūde [19]. Fiziskā atmiņas noplūde ir novērojama, kad atmiņas adreses, kuras tika iedalītas procesam, kļūst nepieejamas, pazaudētas, tas notiek, kad procesa adrešu telpā uz iedalīto atmiņas gabalu kaudzē nenorāda neviens rādītājs. Šis programmas stāvoklis var būt novērojams trīs iemeslu dēļ [19]:

- pēdējā norāde uz atmiņas gabalu ir pārrakstīta vai norāde bija palielināta, piemēram, lai sasniegtu datus ar nobīdi;
- norāde atrodas ārpus darbības lauka (out of scope);
- atmiņas bloks, kurš glabāja norādi, bija atbrīvots.

Loģiskā atmiņas noplūde ir novērojama, kad iekšējā buferī, rindā vai citā datu struktūrā ir uzglabātas norādes uz dinamiski iedalītu atmiņu, bet norāžu skaits pieaug neierobežoti. Loģiskā atmiņas noplūdi bieži nosauc par slēpto atmiņas noplūdi (hidden memory leak) [20], jo atmiņa ir joprojām sasniedzama no programmas, bet nekad netiek atbrīvota.

Abos gadījumos sekas ir vienādas. Sākumā tiks novērota pakāpeniska procesa palēninā-šana, jo daļa no informācijas tiks uzglabāta lapošanas failā (paging file). Kaut kāda brīdī, kad tiks iztērēta visā dinamiskā atmiņa, katrs malloc() funkcijas izsaukums būs neveiksmīgs. Šeit

var notikt kritiskā kļūda, kuras cēlonis ir sliktā programmēšanas prakse. Programmētāji ne vienmēr pārbauda malloc() rezultātu pirms vērsties pēc malloc() funkcijas atgrieztās norādes. Mēģinājums piekļūt null adresei izraisīs Segmentation fault kļūdu. Ja programmā bija paredzēts, ka malloc() var atgriezt null, tad process turpinās izpildi ierobežotā režīmā, jo vairs nav iespējams dinamiski iedalīt atmiņu un izpildīt daudzus uzdevumus. Daudzās sistēmās tas nav pieļaujams un var tikt uzstādīti dažādi ierobežojumi, kuri pēc ierobežojošās vērtības sasniegšanas (izpildes laiks, patērētās atmiņas) automātiski pārtrauks procesa darbību.

```cpp
#include <string>
using namespace std;

int main() {
    string *str;

    for (int i=0; i<10001; i++) {
        // 10000*14 baiti tiek pazaudēti
        str = new string("Hello, World!");
    }
    delete str;

    return 0;
}
```

3.1. att. **Atmiņas noplūde, C++**

Atmiņas noplūdes problēma ir uzskatāmi nodemonstrēta piemērā (sk. 3.1. attēlu). Programma iedala 10001 atmiņas gabalus ar new operatora palīdzību. Rādītājs `str` katru reizi tiek pārrakstīts un norāda uz kārtējo iedalīto atmiņas gabalu, kura izmērs ir 14 baiti. Tā kā atmiņas adreses kļūst pazaudētas un nav iespējas piekļūt iepriekšējiem elementiem, pēc tam, kad `str` radītājs ir pārrakstīts, tad piemērā ir redzama fiziskā atmiņas noplūde. Beigās tiek atbrīvots tikai viens atmiņas gabals, kurš bija iedalīts pēdējais. Programmas darbības laikā kļūst pazaudēti 10000 gabali, kuru kopējais izmērs ir 140000 baiti. Pēc programmas izpildes beigām visā procesam iedalītā atmiņa tiek atgriezta operētājsistēmai.

Šādos gadījumos sistēmas kļūst viegli ievainojamas, ja tajās ir kļūda, kas izraisa atmiņas noplūdi [21]:

- kad operētājsistēma neatbrīvo, lietotnes izpildei izmantoto atmiņu pēc tam, kad lietotne beidz savu darbību, piemēram, AmigaOS;
- ja servera vai citas programmas darbojās visu laiku bez apstāšanās;
- ja portatīvām ierīcēm ir ierobežots atmiņas daudzums;
- ja programmas pieprasa atmiņu uzdevumiem, kuri izpildās ilgstošu laika periodu;
- reālā laika sistēmās, jo ir svarīgi iegūt rezultātu ierobežotajā laikā.

Atmiņas noplūdes problēmu ir grūti atkļūdot, jo nav zināmi nosacījumi, kuriem izpildoties notiek atmiņas noplūde. Ja ir redzamas sekas (ir atmiņas izmete un programma pabeidza savu darbību), bet nav zināms problēmas cēlonis, tad izstrādātājiem ir nepieciešams daudz resursu, lai atkārtotu un izlabotu atmiņas noplūdi. Eksistē vairāki rīki, kuri palīdz atkļūdot atmiņas noplūdes problēmu, tādi kā: Valgrind, Totalview, Purify. Taču tie ne vienmēr sniedz pietiekamu informāciju un bieži netiek izmantoti strādājošās sistēmās, jo piedāvātas atkļūdošanas tehnikas un rīki var palēnināt sistēmas darbību. Piemēram, ieslēdzot memcheck rīku iekš Valgrind instrumentācijas ietvara, programmas izpildes ātrums palēninās aptuveni 20-30 reizes [22].

Reālajās sistēmās problēma var izpausties uzreiz pēc palaišanas, bet var kļūt novērojama tikai pēc dažiem gadiem. Abi gadījumi ir izplatīti [23]. Tā kā atmiņas noplūdes rezultātā atmiņa tiek pazaudēta, tad var periodiski novērot procesa atmiņas patēriņa pieaugumu. Pazīme, kas varētu liecināt par atmiņas noplūdi strādājošā sistēmā ir pārmērīgs[1] atmiņas daudzums, kas visu laiku pieaug. Ja process izmanto pārmērīgo atmiņu un izmantotās atmiņas daudzums nemainās, tad šī pazīme var dot tikai aptuvenu novērtējumu par dotās problēmas esamību, jo eksistē vairākas citas problēmas, piemēram, fragmentēšana, maksimālās atmiņas izmantošanas problēma vai kļūdas trešās puses bibliotēkās, kuras var palielināt izmantotās atmiņas daudzumu.

3.1.1. Atmiņas noplūdes pazīmes atmiņas izmetē

Tā kā atmiņas izmete satur procesa atmiņas attēlojumu uz procesa pārtraukšanas brīdi, tad uzģenerētās datnes izmērs, atmiņas noplūdes problēmas ietekmes rezultātā, var sasniegt vairākus gigabaitus. Turpmāk ir apskatīts piemērs, kurš parāda, kā pazaudētas atmiņas daudzums ietekmē atmiņas izmetes izmēru. Programmai (sk. 1. pielikumu) tika padoti argumenti: 100, 100. Pirmais arguments noteic, cik daudz gabalu nepieciešams iedalīt, otrais - kāds ir katra gabala izmērs. Dotajā programmā atmiņa dinamiski ir iedalīta vairākās reizēs. Sākumā atmiņa ir iedalīta masīvam ar norādēm arr[ ], pēc tam katram masīva elementam ir iedalīts atmiņas gabals norādītajā izmērā. Uz brīdi, kad ir izsaukta abort() funkcija, ir jābūt norādei uz arr[ ] masīvu no procesa adrešu telpas. Turklāt 10000 kilobaiti ir pazaudēti, jo katram elementam masīvā, tika piešķirta NULL vērtība. Kopējais dotās programmas atmiņas izmetes izmērs pēc atmiņas izmetes ģenerēšanas bija 10404 kilobaiti.

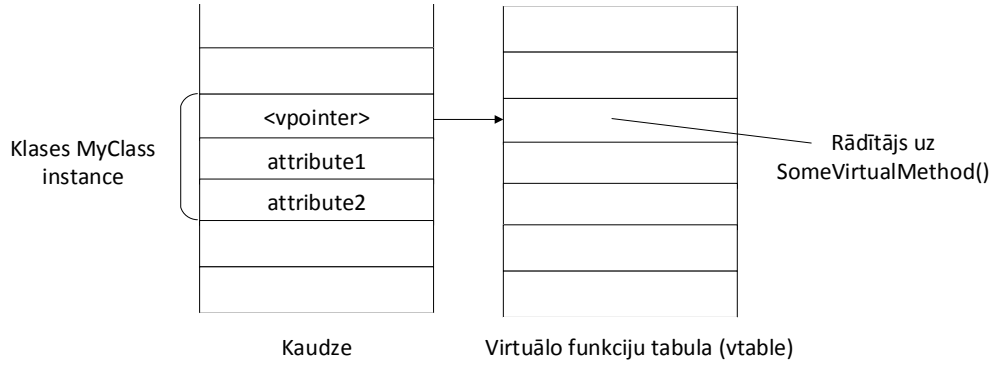
Par fiziskās atmiņas noplūdes pazīmi var uzskatīt stāvokli, kad uz atmiņas gabaliem kaudzē nav norāžu no procesa adrešu telpas. Par šo programmas stāvokli var pārliecināties, veicot atmiņas izmetes analīzi. Atmiņas izmetē atrodas kaudzes saturs visām atmiņas arēnām.

[1] Šajā kontekstā pārmērīgs nozīmē, ka izmērs ir lielāks par to, kuru paredz programmētājs un tas rāda pamatotas šaubas, par atmiņas noplūdes problēmas esamību programmā.

Interpretējot katru kaudzes saturu, kā kopu ar daudziem atmiņas gabaliem, var iegūt adreses, uz kurām malloc() funkcija atgrieza norādes procesam. Ja procesa adrešu telpā nav nevienas norādes uz atrastajām adresēm, tad ar lielu varbūtību var apgalvot, ka programmā ir atmiņas noplūde. Kamēr kļūda nav atrasta kodā, to nevar secināt, jo atmiņas izmete var būt bojāta un var neiekļaut daļu no procesa adrešu telpas. Šī pazīme nav raksturīga loģiskajai atmiņas noplūdei.

```
class MyClass
{
    virtual SomeVirtualMethod();

    public:
        void* attribute1;
        void* attribute2;
}
```



3.2. att. **C++ klases ar virtuālo funkciju izvietojums atmiņā**

Loģiskās atmiņas noplūde rezultātā visiem atmiņas gabaliem atbilst norādes procesa adrešu telpā, bet tādi gabali aizņem visu pieejamo atmiņu. Problēmai ir raksturīgs stāvoklis, kad ir daudzi iedalītie atmiņas gabali, kuru lietotāja datu sekcija satur līdzīgus datus (izmēru, līdzīgas datu shēmas). Turpmāk tiks apskatīts piemērs, kurš paskaidro, kā var izpausties šī pazīme. Piemērā ir aplūkots gadījums, kad programmā ir izmantoti objekti, kuri ir C++ klases instances un klasē ir izmantota virtuālā funkcija. Ja C++ klasē ir virtuālās funkcijas, tad kompilators izveido virtuālo funkciju tabulu (vtable), kura iekļauj rādītājus uz šīs klases virtuālām funkcijām. Katrai klasei ir tikai viena virtuālo funkciju tabula, kuru izmanto visi klases objekti. Ar katru virtuālo funkciju tabulu ir saistīts virtuālo funkciju rādītājs (vpointer). Šis rādītājs norāda uz virtuālo funkciju tabulu un tiek izmantots, lai piekļūtu virtuālajām funkcijām. Klase, kurā ir virtuālā funkcija, atmiņā tiks izvietota sekojoši (sk. 3.2. attēlu). Ja loģiskā atmiņas noplūde notiks, tāpēc ka atmiņā neierobežoti pieaugs MyClass

objektu skaits, tad pēc vpointer norādes atmiņas gabalos var identificēt doto problēmu, bet saprast, kurai klasei pieder objekti, var ar gdb palīdzību. Instrukcijas, kas ļauj apskatīties, kuram apgabalam pieder adrese, jau tika aprakstītas sadaļā 1.4.1.

3.2. Maksimālās atmiņas izmantošanas problēma

Maksimālās atmiņas izmantošanas (peak memory utilization) problēma var notikt, kad iedalītu un atbrīvotu gabalu izmēru summa kaudzē sasniedz maksimumu procesa izpildes laikā. Ir svarīgi pievērst uzmanību gadījumiem, kad var tikt sasniegts maksimums. Piemēram, tas var notikt, kad process tiecās pie rampas virsotnes vai smailes.

Rampa un smaile ir shēmas, kuras raksturo programmas uzvedību un ir apkopotas pētījumā [24]. Atmiņas daudzums, kas tiek izmantots programmas izpildes laikā, var visu laiku mainīties. Kopumā tiek identificētas trīs svarīgākās atmiņas izmantošanas shēmas: rampa (ramp), smaile (peak), līdzenums (plateau). Citas atmiņas izmantošanas shēmas ir iespējamas, bet izpaužas ļoti reti. Ne visām programmām ir raksturīgas visas trīs shēmas, bet vairākumam ir raksturīga viena vai divas no tām. Šīs shēmas tika apkopotas, balstoties uz kvantitatīvo programmu novērtējumu [24].

- Rampa. Programma uzkrāj datu struktūras monotoni. Tas varētu notikt, tāpēc ka uzdevuma atrisināšanai ir nepieciešams paveikt daudzas darbības un pakāpeniski uzbūvēt daudzas datu struktūras. Lai atrisinātu uzdevumu, atmiņas patēriņš monotoni aug. Pēc uzdevuma atrisināšanas atmiņas patēriņš strauji samazinās.
- Smaile. Šo veidu var nosaukt par rampu tikai ļoti īsa laika periodā. Daudzām programmām var būt nepieciešams izveidot lielas datu struktūras kāda uzdevuma izpildīšanai. Pēc šī uzdevuma pabeigšanas gandrīz visa pieprasītā atmiņa var tikt atbrīvota. Grafiks šai shēmai izskatās kā lauztā līnija, un atmiņas patēriņš var svārstīties dramatiski.
- Līdzenums. Novērojams, kad programmas ātri uzbūve datu struktūras un izmanto tās ilga laika periodā, bieži izmanto līdz programmas izpildes beigām.

Maksimālās atmiņas izmantošanas problēma ir novērojama, kad liels atmiņas daudzums netiek atgriezts operētājsistēmai pēc smailes vai rampas virsotnes sasniegšanas. Tas notiek, pat tad, ja gandrīz visa atmiņa tika atbrīvota ar free() vai delete palīdzību. Rezultātā process var patērēt pārmērīgo atmiņas daudzumu, kurš nebija paredzēts projektējumā. Šī situācija kļūst iespējama, ja notiek daudzi pieprasījumi pēc atmiņas, kas ir mazāki par 128 kilobaitiem. Pieprasījumi pēc lielākiem atmiņas gabaliem tiks apstrādāti ar mmap() sistēmas izsaukumu un neizraisīs doto problēmu. Pēc mmap() izsaukumiem atmiņu ir iespējams atgriezt operētājsistēmai ar munmap() palīdzību, jo atmiņa neatrodas kaudzē. Izmantojot brk() sistēmas

izsaukumu, kamēr netiks atbrīvots atmiņas gabals, kas atrodas beigās, atmiņa netiks atgriezta operētājsistēmai.

Strādājošā sistēmā problēma ir novērojama kā pārmērīgs atmiņas patēriņš pēc rampas virsotnes vai smailes sasniegšanas. Turpmāk tiks apskatīts piemērs, kurš demonstrē to, kā izpaužas dota problēma. Lai kontrolētu atmiņas patēriņu, procesa izpildes laikā tika izmantota `ps` komanda. Procesam patērēts atmiņas daudzums iegūts no RSS un VSZ rādītājiem. VSZ parāda virtuālo atmiņu, RSS parāda fizisko atmiņu, kuru izmanto process. Rādītāju mērvienība ir kilobaits. Tika palaista programma, un katrā programmas solī tika noņemti radītāji (sk. 3.1. tabulu). Tā kā bija iedalīti 100 gabali, katrs 100 kilobaitu izmērā, tad kaudze bija paplašināta ar brk() sistēmas izsaukumu. Kopēja pieprasīta atmiņa bija vienāda ar 10000 kilobaitiem. Iegūtie rādītāji parāda, ka atmiņa pilnībā tika atbrīvota tikai pēc tam, kad bija atbrīvots pēdējais atmiņas gabals. To var redzēt 4. solī, kur VSZ un RSS rādītāji paliek nemainīgi, salīdzinot ar iepriekšējo soli. Turklāt 5. solī, pēc pēdējā gabala atbrīvošanas, var novērot to, ka atmiņas daudzums samazinās, tas ir izskaidrojams ar to, ka atmiņa tika atgriezta operētājsistēmai.

3.1. tabula **Programmas RSS un VSZ radītāji**

Solis	VSZ	RSS
1. Sākums	3228	612
2. Ar new ir pieprasīti 100 gabali, katrs 100 kilobaitu izmērā	13360	1136
3. Atmiņa ir aizpildīta ar 0	13360	10640
4. Atmiņa tiek atbrīvota, izņemot pēdējo gabalu	13360	10640
5. Atmiņa tiek pilnībā atbrīvota	3360	968

3.2.1. Maksimālās atmiņas izmantošanas problēmas pazīmes atmiņas izmetē

Atmiņas izmetē pazīme, kas varētu liecināt par problēmu, ir lielā izmēra atmiņas gabali, kuri atrodas bin sarakstos un nav vienmērīgi izkliedēti kaudzē. Ja pēc maksimālās atmiņas izmantošanas bija novērojamā fragmentēšana, tad bin sarakstos varētu atrasties liels atmiņas gabalu skaits. Tāds skaits, kurš varētu tikt izveidots, sadalot mazākos gabalos procesa atmiņas patēriņa pieaugumu, kas notika, mēģinot sasniegt maksimumu. Saskaņā ar GNU C realizāciju visi atbrīvotie gabali tiek uzglabāti bin sarakstos. Tā kā ātrie saraksti uzglabā mazus atmiņas gabalus (līdz 64 baitiem) un nav paredzēti ilgstošai atmiņas gabalu glabāšanai, tad daudzi atmiņas gabali tiks uzglabāti parastajos sarakstos.

Tālāk ir aprakstīts piemērs, kurš parāda, kā problēmas pazīme izpaužas atmiņas izmetē. Atmiņas izmete ir uzģenerēta, izmantojot iepriekš sagatavoto programmu (sk. 2. pielikumu). Sākumā tika iedalīti 100 atmiņas gabali, katrs 100 kilobaitu izmērā. Katrā baitā bija ie-

rakstīta vērtība 7. Pēc tam bija atbrīvoti 99 gabali, izņemot pēdējo. Beigās, lai apskatītos pazīmi, tika uzģenerēta atmiņas izmete. Saskaņā ar pētījumā minētām shēmām [24] bija sasniegta smaile (peak). Atmiņas izmete palīdz saprast, kas notiek atmiņā. Ar komandu `p`

```
(gdb) p main_arena
$3 = {mutex = 0, flags = 1, fastbinsY = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, top = 0x97ba4b8, last_remainder = 0x0,
bins = {0x8df6198, 0x8df6198, 0xb76cf478, 0xb76cf478, 0xb76cf480, 0xb76cf480, 0xb76cf488, 0xb76cf488, 0xb76cf490, 0xb76cf490,
0xb76cf498, 0xb76cf498, 0xb76cf4a0, 0xb76cf4a0, 0xb76cf4a8, 0xb76cf4a8, 0xb76cf4b0, 0xb76cf4b0, 0xb76cf4b8, 0xb76cf4b8,
0xb76cf4c0, 0xb76cf4c0, 0xb76cf4c8, 0xb76cf4c8, 0xb76cf4d0, 0xb76cf4d0, 0xb76cf4d8, 0xb76cf4d8, 0xb76cf4e0, 0xb76cf4e0...},
binmap = {0, 0, 0, 0}, next = 0xb76cf440, next_free = 0x0, system_mem = 10375168, max_system_mem = 10375168}
(gdb) x/16wx 0x8df6198
0x8df6198: 0x00000000 0x009ab319 0xb76cf470 0xb76cf470
0x8df61a8: 0x00000000 0x00000000 0x07070707 0x07070707
0x8df61b8: 0x07070707 0x07070707 0x07070707 0x07070707
0x8df61c8: 0x07070707 0x07070707 0x07070707 0x07070707
```

3.3. att. **bin saraksta izdruka, izmantojot atmiņas izmeti**

`main_arena` ir izdrukāta galvenās arēnas struktūra un iegūtas bin sarakstu sākuma adreses (sk. 3.3. attēlu). Visi bin saraksti atrodas masīvā bins. 7. rindiņā ir komanda, kura izdrukā 16 adreses no pirmā nesakārtotā bin saraksta, kurā atrodas gabali, kuri bija nesen atbrīvoti. Sākot ar 8. rindiņu ir redzams atmiņas gabals, kas atrodas bin sarakstā un kura izmērs ir 0x009ab319, kas decimālajā skaitīšanas sistēmā ir vienāds ar 10138393, binārajā skaitīšanas sistēmā 100110101011001100011001. Tā kā 3 mazākie biti netiek izmantoti izmēra glabāšanai, tad gabala izmērs ir 10138393 - 1=10138392. Mūsu 99 atbrīvoto gabalu kopējā izmēru summa ir vienāda ar izmēru kilobaitos, kas ir sareizināta ar atbrīvoto gabalu skaitu, tātad 100*1024*99 = 10137600. Atšķirība starp pirmā gabala izmēru bin sarakstā un atbrīvoto gabalu kopēju izmēru ir vienāda ar 10138392 - 10137600 = 792 un izskaidrojama ar to, ka katram gabalam bija iedalīti 8 baiti uzturēšanas informācijas glabāšanai (prev_size, size). Pārējie 127 bin un ātrie bin saraksti, dotajā piemērā, bija tukši.

3.3. Fragmentēšana

”Ir pierādīts, ka katram atmiņas iedalīšanas algoritmam, vienmēr ir iespējama situācija, ka kāda lietotne pieprasīs un atbrīvos atmiņu tāda veidā, ka tās nojauks iedalītāja stratēģiju un izraisīs lielu fragmentēšanu. Ir pierādīts ne tikai tas, ka nav laba iedalīšanas algoritma, bet arī tas, ka katrs iedalīšanas algoritms var būt slikts dažām lietotnēm” [24]. Tātad, fragmentēšanas problēma var būt aktuālā daudzām C un C++ lietotnēm, kuras pieprasa atmiņu

no kaudzes.

Fragmentēšanas problēmu var iedalīt divos dažādos veidos: iekšējā un ārējā fragmentēšana. Iekšējā fragmentēšana notiek, kad tiek iedalīts lielāks atmiņas gabals nekā tika pieprasīts. Izlīdzināšana ir viens no iekšējās fragmentēšanas cēloņiem. Iekšējo fragmentēšanu ir iespējams paredzēt, jo var izskaitļot, kuram skaitlim tiks noapaļots izmērs. GNU C bibliotēkā atmiņas gabalu izmērs tiek izlidzināts skaitlim, kurš dalās bez atlikuma uz 8 vai 16. Izlīdzināšana samazina atšķirīgu gabalu izmēru skaitu kaudzē. Nodrošinot izlīdzināšanu, ir palielināta iekšējā, turklāt, ir samazināta ārējā fragmentēšana [25]. Ārējā fragmentēšana ir nespēja iedalīt atmiņas gabalu kaudzē, kad kaudzē pietiekoši daudz brīvas atmiņas, lai apmierinātu doto pieprasījumu. Ārējā fragmentēšana var izpausties ar laiku, kad daudzas reizes jau tika iedalīti un atbrīvoti dažāda izmēra atmiņas gabali. Fragmentēšanas rezultātā pārmērīgi tiek izlietoti kaudzes resursi, jo, kad pieprasījums pēc atmiņas nevar tikt apmierināts, tad notiek kaudzes piespiedu paplašināšana.

Ārējo fragmentēšanu mēra procentos (%). Stradajošā sistēmā var būt vairāki veidi, kā var mērīt atmiņas fragmentēšanu [26]. Atmiņas izmetē ir iespējams izrēķināt ārējo fragmentēšanu tikai uz procesa partraukšanas brīdi. Tātad ir iespējams izrēķināt tikai momentāno kaudzes fragmentēšanu. Fragmentēšana var būt izrēķināta kā attiecība starp atmiņas daudzumu kaudzē, ko aizņem iedalītājs pret atmiņas daudzumu, ko izmanto process (neietilpst atbrīvotie atmiņas gabali).

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

int main () {
    char * ptrl;
    int chunk_size;

    ptrl = (char *)malloc(4);

    /* tiek iegūts atmiņas gabala izmērs (otrais malloc_chunk elements) */
    chunk_size = *((char *) ptrl - sizeof(size_t));
    /* mazākie 3 biti tiek izmantoti meta informācijas glabāšanai */
    chunk_size = chunk_size - (chunk_size & 7);

    printf("size = %d\n", chunk_size);
    free(ptrl);

    return 0;
}
```

3.4. att. **Izmēra noteikšana iedalītam gabalam**

Turpmāk ir apskatīts piemērs (sk. 3.4. attēlu), kurš demonstrē iekšējās fragmentēšanas cēloni. Šis C valodā uzrakstītais kods izdruka atmiņas gabala izmēru, kurš īstenībā tiek

iedalīts no kaudzes. Piemērā ir redzams, ka tiek iedalīti 4 baiti, bet programma izdrukā beigu rezultātu - 16 baiti. Dotajā piemērā iekšējā fragmentēšana ir vienāda ar 12 baitiem.

Algoritms ir sekojošs:

1. ar malloc() tiek iedalīts atmiņas apgabals;
2. tiek iegūta size elementa vērtība (objektam ar malloc_chunk struktūru), kur 29 biti ir paredzēti izmēra glabāšanai un 3 ir kontroles zīmes;
3. tiek atņemtas A, M, P kontroles zīmes, kuras ir lielākas par 0, un iegūts iedalītā atmiņas gabala izmērs;
4. tiek atbrīvota atmiņa.

3.3.1. Fragmentēšanas pazīmes atmiņas izmetē

Iekšējai fragmentēšanai ir grūti identificēt pazīmes atmiņas izmetē, kuras varētu liecināt par doto problēmu. Lai atpazītu iekšējo fragmentēšanu, ir nepieciešams zināt, cik daudz atmiņas pieprasīja lietotne. Izmantojot atmiņas izmeti, šo informāciju var iegūt tikai no backtrace, taču šī informācija ne vienmēr tiks iekļauta.

No problēmas apraksta seko, ka ārējai fragmentēšanai ir raksturīgs liels mazo[1] atbrīvoto gabalu skaits. Šie gabali var būt saglabāti vienā no bin sarakstiem:

- Ja izmērs ir līdz 64 baitiem, tad atbrīvotie gabali tiks novietoti ātrajos sarakstos. Ātrajos sarakstos gabals norādīs uz nākamo gabalu un veidos garo sarakstu no visiem atmiņas gabaliem. Ātra saraksta elementi tiek sapludināti, kad nav iespējams iedalīt lielāko gabalu;
- Ja ātrie gabali tika saplūdināti vai gabali ir lielāki par 64 baitiem, tad gabali tiks novietoti parastajos sarakstos;

```
0x8b30198: 0x00000000 0x00000011 0x00000000 0x00000000
0x8b301a8: 0x00000000 0x00000021 0x00000000 0x00000000
0x8b301b8: 0x00000000 0x00000000 0x00000000 0x00000000
0x8b301c8: 0x00000000 0x00000011 0x08b30198 0x00000000
0x8b301d8: 0x00000000 0x00000021 0x00000000 0x00000000
0x8b301e8: 0x00000000 0x00000000 0x00000000 0x00000000
0x8b301f8: 0x00000000 0x00000011 0x08b301c8 0x00000000
0x8b30208: 0x00000000 0x00000021 0x00000000 0x00000000
0x8b30218: 0x00000000 0x00000000 0x00000000 0x00000000
0x8b30228: 0x00000000 0x00000011 0x08b301f8 0x00000000
```

3.5. att. **Fragmentētā kaudze**

[1]Mazs nozīme tāds, kurš nevar apmierināt turpmākos pieprasījumus pēc atmiņas.

Kaudze būs saskaldīta, un katrs iedalītais gabals robežos ar mazāko atbrīvoto gabalu. Piemērā ir izdrukāts kaudzes saturs (sk. 3.5. attēlu). Piemērā izmantotā atmiņās izmete tika uzģenerēta, izmantojot iepriekš sagatavoto programmu (sk. 3. pielikumu). Programma ir izveidota problēmas pazīmju identificēšanai, un tāpēc ir vienkāršots problēmas attēlojums, izmantojot mazā izmēra gabalus: 32 un 16 baitu izmērā. Tā kā programmā uzreiz pēc iedalīšanas tiek pārtraukta, tad ātrā saraksta gabali netiek saplūdināti. Programmai no komandrindas tika padoti argumenti: 100, 8. Kaudzē atrodas iedalītie gabali 32 baitu (0x21) izmērā un atbrīvotie gabali 16 baitu (0x11) izmērā. Atbrīvotie gabali ir saistīti savā starpā un atrodas ātrajā sarakstā. Tas ir novērojams, jo lietotāju datos ātrajiem gabaliem ir uzglabāta nākamā gabala adrese. Pēdējam gabalam ātrajā sarakstā ir uzglabāta 0x00000000 adrese. Ja kaudzē atrodas brīvie gabali, kuri nav lielāki par 16 baitiem un top gabala izmērs ir mazāks par 32 baitiem, tad maksimālais gabals, kurš varētu tikt iedalīts, ir 16 baiti un pieprasījums pēc 32 baitiem paplašinās kaudzes segmenta izmēru vai izraisīs kļūdu.

3.4. Datu kaudzes bojāšana

Viena kļūda programmā var sabojāt kaudzes datus. Šo kļūdu ir grūti atrast, jo sekas ir novērojamas nevis tad, kad tiek pārrakstīti dati kaudzē, bet kad ir nākamais mēģinājums piekļūt pārrakstītiem datiem. Kaudzē ir novērojamas vairākas bojāšanas kļūdas (heap corruption) [27]:

- robežu pārpildīšana (boundary overrun). Notiek, kad programma raksta aiz malloc() funkcijas iedalītā gabala robežām. Tādā veidā var pārrakstīt nākamo datu struktūru atmiņā;
- rakstīšana pirms malloc() iedalītā gabala sākuma (buffer underrun);
- piekļuve neinicializētam atmiņas gabalam. Programma mēģina lasīt datus no gabala, kurš nav inicializēts;
- piekļuve atbrīvotam gabalam (use after free). Programma mēģina lasīt vai rakstīt atmiņas gabalā, kurš bija atbrīvots;
- divkārša atbrīvošana (double free). Programma atbrīvo datu struktūras, kuras jau tika atbrīvotas;
- programma free() funkcijai padod adresi, kura nebija atgriezta ar malloc().

3.5. Kļūdas trešās puses bibliotēkās

Tā kā izstrādājamā programma strādā, izmantojot trešās puses bibliotēkas, tad programmas uzticamība un kvalitāte ir atkarīga arī no iedalītāja realizācijas trešās puses bibliotēkās. Ir iespējams, ka notiek problēma programmā nekorektas iedalītāja realizācijas dēļ.

Piemēram, ir iespējams apskatīt GNU C bibliotēkas pieteiktās kļūdas [28]. Ja iedalītājs ir uzturēts, tad iespējams, ka var piereģistrēt problēmas, lai nākamajās versijās tas tiktu izlabotas. Izmantojot individuālo iedalītāja risinājumu, problēmas būs jāatkļūdo patstāvīgi. Daži iedalītāji var būt izstrādāti ar iepriekš zināmiem ierobežojumiem, piemēram, ja iedalītāja algoritms ir efektīvs, tad tas var izmantot pārmērīgo atmiņas daudzumu. Cits ierobežojums ir kad tiek izmantots mazs atmiņas daudzums turklāt algoritms nav efektīvs. Šis īpašības ir jāņem vērā izstrādājot lietotnes, lai nerastos problēmas, kas saistītas ar trešās puses bibliotēkām. Ja problēma notiek trešās puses bibliotēkās, tad ir trīs iespējas kā var izvairīties no problēmām lietotnēs: ir jāatrod ceļš, kā nepieļaut doto kļūdu, ir jāgaida atjauninājumi, kuros kļūda tiks izlabota, vai ir jānomaina iedalītājs.

3.6. Secinājumi

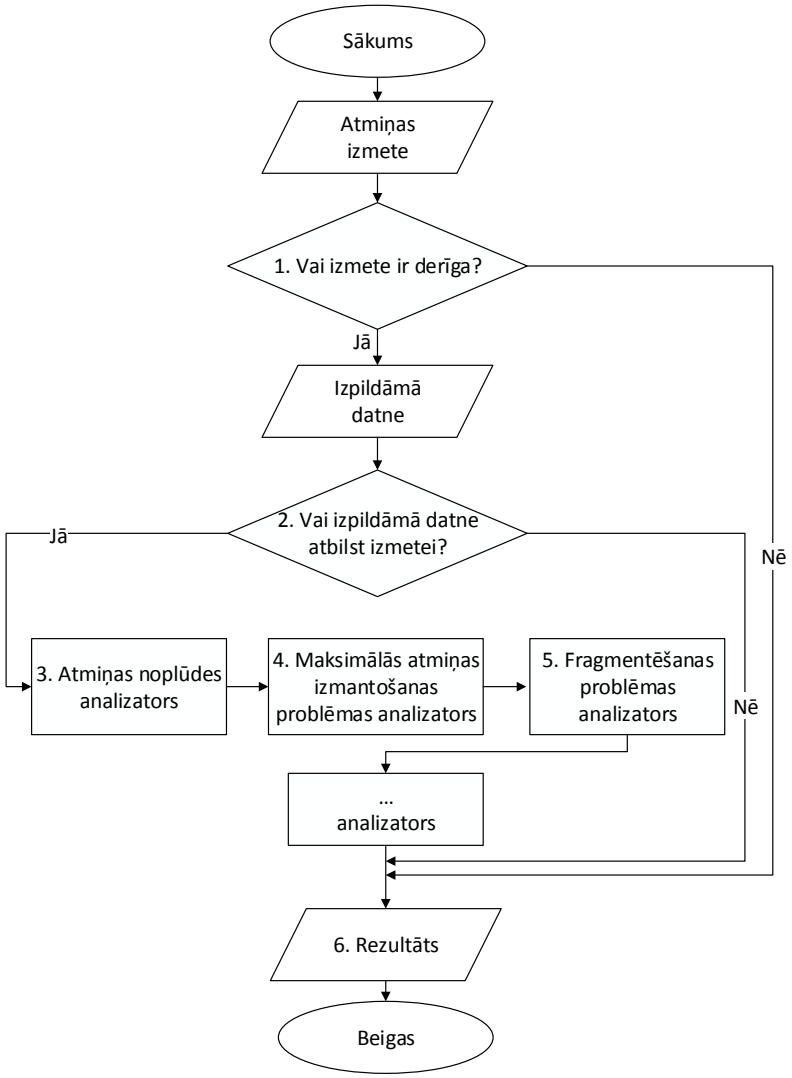
Nodaļā tika izpētītas trīs problēmas un identificētas to pazīmes atmiņas izmetē.

1. Atmiņas noplūdes pazīmes:
    - pārmērīgs atmiņas izmetes izmērs;
    - uz gabaliem kaudzē nav norāžu no procesa adrešu telpas;
    - daudz gabalu ar vienādu izmēru un līdzīgiem datiem;
2. Maksimālās atmiņas izmantošanas problēmas pazīmes:
    - atbrīvotie gabali nav vienmērīgi izkliedēti kaudzē;
    - pārmērīga atbrīvoto gabalu kopējā izmēru summa;
3. Fragmentēšanas pazīmes:
    - mazs maksimālais atbrīvotais gabals bin sarakstos, kad kopējā gabalu izmēru summa ir pietiekoša.

4. ATKĻŪDOŠANAS METODES APRAKSTS

Šajā nodaļā ir aprakstīta metode, kura varētu tikt pielietota kaudzes atkļūdošanai. Šeit ir aplūkots metodes algoritms un trīs analizatoru realizācijas: atmiņas noplūdes analizators, maksimālās atmiņas izmantošanas problēmas analizators, fragmentēšanas problēmas analizators. Realizētie analizatori ir uzskatāms demonstrējums, ka metode strādā un var tikt pielietota.

4.1. Analizatora darbības princips



4.1. att. **Uz metodi balstītā algoritma blok-shēma**

Blok-shēmā (sk. 4.1. attēlu) ir attēlots kaudzes atkļūdošanas algoritms. Šī algoritma

ievaddati ir izpildāmā datne un atmiņas izmete. Atmiņas izmete var būt bojāta, tāpēc pirms sākt atkļūdošanas procedūru ir nepieciešams veikt atmiņas izmetes validēšanu (1). Šeit ir jāveic dažas pārbaudes, piemēram, pārbaudi uz neatbilstību formātam vai atmiņas izmetes nogriezto saturu. To, kā var pārbaudīt vai atmiņas izmetei ir nogriezts saturs ir aprakstīts 1.3. sadaļā. Nākamā pārbaude ir veikta pēc izpildāmās datnes nolasīšanas. Ir iespējams, ka izpildāmā datne neatbilstīs atmiņas izmetei (2), tas var notikt divos gadījumos. Pirmkārt, kad tiek izmantota cita izpildāmā datne atmiņas izmetes ģenerēšanai. Otrkārt, kad atmiņas izmetei neatbilst izpildāmās datnes versija. Ir nepieciešams pārbaudīt atbilstību un jāpārliecinās, ka var piekļūt galvenās arēnas datiem. Visās iepriekš aprakstītās pārbaudes palīdz savlaicīgi uzzināt, ka atmiņas izmete nav derīga analīzei un pārtraukt algoritma darbību. Pēc dotām pārbaudēm var sākt atmiņas izmetes analīzi. Blok-shēmā ir parādīts, ka ir nepieciešams darbināt analizatorus, kuri pārbauda problēmu pazīmes atmiņas izmetē. Daudzkodolu procesoriem ir iespējams realizēt algoritmu, kur analizatori strādās paralēli, vienkodolā visi analizatori izpildīsies secīgi. Analizatoru skaits nav ierobežots, taču bakalaura darbā algoritms tiks nodemonstrēts trijos analizatoru piemēros (3. atmiņas noplūdes analizators, 4. maksimālās atmiņas izmantošanas problēmas analizators, 5. fragmentēšanas problēmas analizators). Katrs analizators pārbauda noteiktās kaudzes problēmas pazīmes. Pēc tam tiek izvadīts kopējais rezultāts (6). Ja ir zināma programmai raksturīga uzvedība, tad pēc rezultāta izvadīšanas var secināt pār problēmām.

Pieeja tiek izstrādāta, ievērojot šādus ierobežojumus:

- GNU C bibliotēkas iedalitāja izmantošana;
- ELF atmiņas izmetes formāts;
- ir nepieciešams divreiz lielāks brīvas atmiņas apjoms par izpildāmās programmas pieprasīto atmiņas apjomu;
- analizātori nedod pilnīgu secinājumu par problēmas esamību, bet sniedz informāciju par sistēmas stāvokli, kura ļauj izstradatājam secināt par problēmu.

4.2. Atmiņas noplūdes analizators

Autore realizēja atmiņas noplūdes analizatoru gdb skriptā (sk. 4. pielikumu). Skriptā ir nodefinēta komanda `analyze`, kura izsauc parējās komandas. Šai komandai no gdb atkļūdotāja ir nepieciešams padot vienu argumentu: galvenās arēnas adresi. Atmiņas izmetei ir jābūt ar nosaukumu core un ir jāatrodas darba mapē. Skripts darbosies uz 32 bitu datoru arhitektūras, kurā vismazāk nozīmīgie baiti tiek uzglabāti sākumā (little endian). Skriptu ir iespējams pielāgot arī 64 bitu arhitektūrai un arhitektūrām ar visvairāk nozīmīgākiem baitiem sākumā (big endian). Analizators ir izstrādāts vienpavedienu lietotnei ar vienu galveno arēnu.

Skripts atrod visus atmiņas gabalus kaudzē. Katrs atmiņas gabals tiek pārbaudīts vai tās tiek iedalīts programmai, vai atbrīvots. Šī informācija tiek iegūta nolasot pēdējo P kontroles bitu nākamajām atmiņas gabalam. Ja tekošais gabals ir atbrīvots, tad tas jau tika pievienots vienam no bin sarakstiem un to nevajag apstrādāt. Pēc tam, tiek nobīdīta norāde un pārbaudīts nākamais atmiņas gabals. Tādā veidā tiek apstrādāti visi atmiņas gabali, kuri ir novietoti pirms top gabala. Katram iedalītām gabalam kaudzē tiek izrēķināta adrese, kurā sākas lietotāju dati. Tā kā adrese ir iedalīta, tad procesa adrešu telpā ir jābūt norādei uz šo vietu atmiņā. Gadījumos, kad norādes nav, tad dotais apgabals ir pazaudēts un tas nozīme, ka ir atrasta atmiņas noplūdes problēma. Programmai iedalītās adreses tiek meklētas ar grep utilītprogrammas palīdzību. Lai nodrošinātu adrešu meklēšanu visā procesa adrešu telpā, atmiņas izmetes saturs tiek saglabāts heksadecimalajā formātā atsevišķajā datnē. Šīm nolūkam ir izmantota `od -t x` komanda, jo, atšķirībā no `xxd`, komanda ļauj bez baitu apgriešanas Intel x86 arhitektūrā iegūt korektas adreses ar vismazāk nozīmīgākiem baitiem sākumā [29]. Atmiņas izmetes heksadecimalajā saturā tiek meklētas iedalītas adreses. Kad adrese ir atrasta, notiek nākamas adrese meklēšana, ja nav atrasta tad tiek pieskaitīta pazaudēto gabalu statistika un izdrukāta dotā atmiņas adrese. Programmas izvads satur pazaudēto norāžu skaitu un pazaudēto atmiņas gabalu adreses (sk. 4.2. attēlu).

```
---Type <return> to continue, or q <return> to quit---
Nav atrasta norāde uz adresi: 9c63498
Nav atrasta norāde uz adresi: 9c7c4a0
Nav atrasta norāde uz adresi: 9c954a8
Nav atrasta norāde uz adresi: 9cae4b0
Nav atrasta norāde uz adresi: 9cc74b8
Procesa adrešu telpā tiek pazaudēts(i): 100 gabals(i).
```

4.2. att. **Atmiņas noplūdes atrašana, gdb skripta izvads**

Skripts var kļūdaini atrast pazaudētas adreses, ja gabaliem nodrošināta piekļuve ar nobīdi. Tas notiek, kad tiek iedalīts gabals, bet no programmas nav norādes uz gabala sākumu. Piemērā ir redzams, ka str norādei ir piešķirta atmiņas gabala adrese ar nobīdi (sk. 4.3. attēlu). Tas ir viens no speciāliem gadījumiem, kurš netiks apstrādāts skriptā. Turklāt ir iespējams iegūt adresi un apskatīties datus, kuri tiek uzglabāti atmiņā.

```c
char * str = (char *)malloc(sizeof(char) * num_elements) + 16; /* C */
```

4.3. att. **Speciālgadījums, no procesa adrešu telpā nav norādes uz gabala sākumu**

Atmiņas noplūdes analizators sameklē pazaudētus gabalus un ļauj atrast problēmu atmiņas izmetē. Parauga programmai (sk. 1. pielikumu) ar diviem argumentiem 100, 100 skripts

atgrieza rezultātu, ka tiek pazaudēti 100 atmiņas gabali. Tā kā sniegtā izdruka atbilst programmas stāvoklim,tad tiek uzskatīts, ka dotās problēmas atkļūdošanai var izmantot atmiņas izmeti.

4.3. Maksimālās atmiņas izmantošanas problēmas analizators

Tā kā maksimālās atmiņas izmantošanas problēma ir tuva fragmentēšanai, tad autore nolēma realizēt divus analizatorus vienā gdb skriptā (sk. 5. pielikumu). Skriptā ir nodefinēta komanda `analyze`, kura izsauc parējās komandas priekš fragmentēšanas un maksimālās atmiņas izmantošanas problēmas analizatoriem. Šai komandai no gdb atkļūdotāja ir nepieciešams padot divus argumentus: galvenās arēnas adresi un skaitli, kurš norāda cik daļās būtu jāsadala kaudze. Palaist skriptu var, izmantojot komandas, kuras parādītas attēlā 4.4. Gdb skripts strādā uz 32 bitu datoru arhitektūras, taču to iespējams pielāgot arī 64 bitu arhitektūrai. Skripts ir izstrādāts vienpavedienu lietotnei ar vienu galveno arēnu.

```
1 (gdb) source fragmentation.gdb
2 (gdb) p &main_arena
3 $1 = (struct malloc_state *) 0xb75ed440
4 (gdb) analyze 0xb75ed440 5
```

4.4. att. **Gdb skripta palaišana**

Šeit tiek aprakstīts maksimālās atmiņas izmantošanas problēmas analizators (sk. 5. pielikumu). Atkarībā no otrā argumenta, kaudze tiks sadalīta vienā vai vairākos atsevišķos apgabalos. Turpmāk darbā, viens kaudzes sadalījuma rezultāts tiks nosaukts par kaudzes apgabalu. Katram kaudzes apgabalam tiks izrēķināta attiecība: kopējais atbrīvoto gabalu izmērs pret kopējo atbrīvoto un iedalīto gabalu izmēru. Šī attiecība ļauj iegūt daļu, kuru aizņem atbrīvotie gabali katra kaudzes apgabalā. Jā kaudzes apgabalam, kas atrodas blakus top, rādītājs ir lielāks par rādītāju pārējos kaudzes apgabalos, tad tās var liecināt par maksimālās atmiņas izmantošanas problēmu, kuras laikā atmiņa netiek atgriezta operētājsistēmai pēc maksimuma sasniegšanas. Izdrukā (sk. 4.5. attēlu) blakus izrēķinātiem rādītājiem, tiek norādīta atmiņas gabala sākuma un beigu adreses. Ja ir nepieciešams, tad var izdrūkat sīkāko kaudzes sadalījumu, norādot atbilstošo argumentu `analyze` komandai. Piemēra redzamai programmai nav raksturīga maksimālās atmiņas izmantošanas problēma, jo atšķirība pēdējiem gabaliem nav lielāka par 5%.

Algoritma realizācijai ir nepieciešams apstaigāt kaudzi un sakrāt datus par katru kaudzes apgabalu. Sākumā tiek iegūta kaudzes sākuma adrese. Lai to iegūtu, ir nepieciešams iegūt top gabala adresi, kaudzes izmēru un top gabala izmēru. Top gabala adrese var tikt iegūta no top rādītāja, kurš atrodas galvenās arēnas struktūrā. Kaudzes izmērs ir uzglabāts

```
------------ Maksimālā atmiņas izmantošanas problēma --------
Atbrīvoto un iedalīto gabalu attiecība:
Apgabals 0x8279000 - 0x82b5238 35%
Apgabals 0x82b5238 - 0x82f32e0 31%
Apgabals 0x82f32e0 - 0x8333388 35%
Apgabals 0x8333388 - 0x8371430 31%
Apgabals 0x8371430 - 0x83a54b8 36%
```

4.5. att. **Maksimālā atmiņas izmantošanas rādītājs**

galvenās arēnas `system_mem` elementā. Top izmērs iegūstams piekļūstot top gabalam un nolasot `size` lauku. Kaudze aug no mazākas adreses uz lielāko un top atrodas beigās. Top gabala izmērs iekļauts kopējā kaudzes izmērā, bet šī atmiņa netiek izmantota programmā. Tāpēc no top adreses vajag atņemt kaudzes izmēru un pieskaitīt top gabala izmēru. Rezultātā tiek iegūta kaudzes sākuma adrese. Sākot ar šo adresi tiek apstaigāti visi atmiņas gabali kaudzē. Gabalu apstaigāšana notiek pieskaitot kaudzes adresei kārtēja gabala izmēru un saglabājot vajadzīgos datus. Beigās tiek izdrukāts iegūtais rezultāts, kas izrēķināts karam kaudzes apgabalam. Algoritms ir nodrošināts ar `div_stat` komandas palīdzību.

Maksimālās atmiņas izmantošanas analizators sniedz kaudzes apgabalu statistiku, kura palīdz atrast problēmu. Parauga programmai (sk. 2. pielikumu), kura tika palaista ar diviem argumentiem: 100, 100 skripts atgrieza rezultātu, ka pēdējā kaudzes apgabalā atbrīvoto gabalu daļa no apgabala satura ir 99%, ja kaudze tiek sadalītā 2 daļās. Tā kā sniegtā izdruka atbilst programmas stāvoklim,tad tiek uzskatīts, ka dotās problēmas atkļūdošanai var izmantot atmiņas izmeti.

4.4. Fragmentēšanas analizators

Šajā sadaļā tiek aprakstīts fragmentēšanas analizators, kurš tika realizēts gdb skriptā (sk. 5. pielikumu). Fragmentēšanas problēmas novērtēšanai analizators izvada divus svarīgākus rādītājus: lielāko un kopējo atbrīvoto gabalu izmēru arēnā. Problēma būs novērojama, kad tiks iegūts tāds maksimālais atbrīvotais gabals, kurš nevar apmierināt pieprasījumu pēc atmiņas, ja kopējais izmērs ir pietiekošs. Piemēra redzamai programmai (sk. 4.6. attēlu), ja programma pieprasīs vairāk nekā 8 kilobaitus un atmiņa netiks iedalīta no operētājsistēmas, tad tas varētu izraisīt sistēmas apstāšanos.

Lai iegūtu abus šos rādītājus ir nepieciešams apstaigāt 128 bin sarakstus un no ikviena saraksta iegūt katra atmiņas gabala izmēru. Parastie bin saraksti atrodas galvenajā arēnā, tāpēc piekļūt tiem var, ja ir zināma arēnas struktūra un tās sākuma adrese. Gdb skriptā katram sarakstam ir numurs no 0 līdz 127 un apstaigāšana notiek, izmantojot nobīdes no galvenās arēnas sākuma un ņemot vērā kārtējā saraksta numuru. Katrs gabals norāda uz

```
Bin numurs 1: 50 gabals(-li) (8196 - 8196 baiti), kopumā = 409800 baiti

------------------------ Fragmentēšana -----------------------
Lielākā gabala izmērs: 8196 baiti (8 KiB, 0 MiB),
Kopējā atbrīvotā atmiņa bin sarakstos: 409800 baiti (400 KiB, 0 MiB),
```

4.6. att. **Fragmentēšanas rādītāji**

nākamo un iepriekšējo atmiņas gabalu, tāpēc visus saraksta gabalus var apstaigāt, pārvietojoties pa sarakstu. Pēdējais atmiņas gabals norāda uz kārtēja saraksta sākumu. Saraksta apstaigāšana ir jābeidz, kad ir iegūta apstrādājamā saraksta sākuma adrese. Katram gabalam tiek pārbaudīts vai tekošais gabala izmērs nav lielāks par maksimālo gabala izmēru sarakstā un tiek atjaunināta kopējā saraksta izmēru summa. Apstrādājot iegūtās vērtības sarakstiem, tiek iegūti dotie rādītāji galvenajai arēnai. Gabalu apstaigāšana sarakstā ir nodrošināta ar free_chunk_list komandu.

Pirms izvadīt lielāko un kopējo atbrīvoto gabalu izmēru rādītājus, tiek izdrukāta statistika par katru no bin sarakstiem (sk. 4.6. attēlu). Tas palīdz iegūt detalizētu statistiku par visiem parastajiem sarakstiem, kuri nav tukši. Statistika, kura tiek izdrukāta: skaits cik ir atbrīvoto gabalu sarakstā, amplitūda (mazākais gabals, lielākais gabals), kopējais gabalu izmērs sarakstā.

```
---------------------- Kopējā statistika ----------------------
Kaudzes segmenta izmērs: 1314816 baiti (1284 KiB, 1 MiB),
Kopējais programmai iedalītais atmiņas daudzums: 820208 baiti (800 KiB, 0 MiB),
Top gabala izmērs: 84808 baiti (82 KiB, 0 MiB),
Atbrīvoto gabalu skaits arēnā: 50,
```

4.7. att. **Kopējā statistika**

Skripts uzkrāj kopējo statistiku par kaudzi (sk. 4.7. attēlu). Izdrukas piemēra ir redzams, ka kopumā tika iedalīti 50 atmiņas gabali. Neizmantotās atmiņas daudzums (top gabala izmērs) ir 82 kilobaiti. 2/3 no iedalītās atmiņas kaudzē tiek iedalītas programmai ar malloc() vai līdzīgam funkcijām.

Fragmentēšanas analizators sniedz statistiku, kura palīdz atrast problēmu. Parauga fragmentēšanas programmai (sk. 3. pielikumu) tika padoti 2 argumenti 100, 100. Skripts atgrieza rezultātu, ka lielāka gabala izmērs ir 104 baiti, bet kopējā atbrīvotā atmiņa bin sarakstos ir vienāda ar 5200 baitiem. Tā kā sniegtā izdruka atbilst programmas stāvoklim,tad tiek uzskatīts, ka dotās problēmas atkļūdošanai var izmantot atmiņas izmeti.

GALVENIE REZULTĀTI UN SECINĀJUMI

Bakalaura darba mērķis bija izstrādāt kaudzes atkļūdošanas metodi, kura ir balstīta uz atmiņās izmetes analīzi un ļauj bez tiešas piekļuves sistēmai atrast kaudzes problēmas programmā.

Darbam ir šādi galvenie rezultāti:

- Pētījuma rezultātā tika izstrādāta kaudzes atkļūdošanas metode, kura tika nodemonstrēta darbībā, izmantojot trīs analizatoru piemērus. Pēc katras analizatoru sniegtās izdrukas bija iespējams identificēt vienu no trim pētāmām kaudzes problēmām.
- Lai atrastu problēmām raksturīgas pazīmes atmiņas izmetē, katrai pētāmai problēmai tika uzģenerēta atmiņas izmete. Katrs atmiņas izmetes saturs tika izpētīts sīkāk ar gdb atkļūdotāja palīdzību, rezultātā tika identificētas izvēlēto problēmu pazīmes.
- Tika izstrādāti un aprakstīti trīs analizatoru piemēri: atmiņas noplūdes analizators, maksimālās atmiņas izmantošanas problēmas analizators un fragmentēšanas problēmas analizators. Analizatori tika izstrādāti divos gdb skriptos un veic savu galveno uzdevumu: sniedz izdruku par identificētām problēmu pazīmēm atmiņas izmetē.
- Darbā katrai pētāmai kaudzes problēmai aprakstīti problēmu veidi, cēloni un izpausme strādājošā sistēmā. Izmantojot pirmkodu un citus literatūras avotus ir apkopota informācija par GNU C bibliotēkas ptmalloc2 realizāciju un uzzīmēts atmiņas organizācijas attēls.
- No autores pieredzes ir aprakstīta atmiņas izmetes validēšana un atkļūdošanas procedūra, kas varētu tikt veikta, izmantojot atmiņas izmeti.

Balstoties uz galvenajiem darba rezultātiem var secināt, ka metode strādā un var tikt izmantota plašāk, piemēram, citu problēmu atkļūdošanai. Viecot pētījumu autore ieguva zināšanas par kaudzes problēmām, to pazīmēm un iedalītāja realizāciju. Zināšanas turpmāk tiks pielietotas ikdienas darbā un atvieglos programmas atkļūdošanas procedūru problēmām, kas saistītas ar kaudzi un dinamisko atmiņas iedalīšanu.

Turpmāk iespējams turpināt darbu dotajā virzienā, jo bakalaura darbā nav izpētītas visas zināmas kaudzes problēmas un nav identificētas to pazīmes atmiņas izmetē. Ir iespējams izveidot analizatoru, kurš sniegs kopējo statistiku pār visām zināmam kaudzes problēmām un strādās neatkarīgi no atkļūdotāja. Tā kā kaudzes problēmām var nebūt tiešo pazīmju, tad šīm analizatoram būs savs pielietojums, jo šobrīd nav zināms līdzīgs analizatora risinājums.

PATEICĪBAS

Autore pateicās darba vadītājam, Romānam Taranovam, par sniegtajiem ieteikumiem un sadarbību bakalaura darba izstrādes laikā.

Autore izsaka pateicību Ērikam Ezeriņam par atbalstu un palīdzību aizraujošās tēmas izvēlē.

LITERATŪRA

[1] C. Laird, "Techniques for memory debugging," http://www.ibm.com/developerworks/aix/library/au-memorytechniques.html#categories, [Online; resurss apskatīts 20-Mai-2014].

[2] E. Roberts, "Debugging C++," 2013. [Online]. Available: http://www.stanford.edu/class/archive/cs/cs106b/cs106b.1134/handouts/10-DebuggingC++.pdf

[3] G. Novark, E. D. Berger, and B. G. Zorn, "Efficiently and precisely locating memory leaks and bloat," *SIGPLAN Not.*, vol. 44, no. 6, pp. 397–407, Jun. 2009. [Online]. Available: http://doi.acm.org/10.1145/1543135.1542521

[4] K. A. Robbins and S. Robbins, *UNIX Systems Programming*. Prentice Hall Professional, 2003, p. 257.

[5] The Linux man-pages project, "Signal(7)," http://man7.org/linux/man-pages/man7/signal.7.html, [Online; resurss apskatīts 21-Mar-2014].

[6] R. Stallman, R. Pesch, and S. Shebs, *Debugging with GDB: The GNU Source-Level Debugger*, 9th ed. Free Software Foundation, 2009, pp. 26–30, 105.

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

[8] M. Welsh, M. K. Dalheimer, T. Dawson, and L. Kaufman, *Running Linux*, 4th ed. OŔeilly & Associates, Inc., 2003, p. 485.

[9] R. Stallman, R. Pesch, and S. Shebs, *Debugging with GDB: The GNU Source-Level Debugger*, 9th ed. Free Software Foundation, 2009, pp. 89–90.

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

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

[12] P. Sorfa, "Debugging memory on linux," *Linux J.*, vol. 2001, no. 87, pp. 2–, Jul. 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=509446.509448

[13] The Linux man-pages project, "Brk(2)," http://man7.org/linux/man-pages/man2/brk.2.html, [Online; resurss apskatīts 25-Mai-2014].

[14] E. D. Berger, B. G. Zorn, and K. S. McKinley, "Composing high-performance memory allocators," *SIGPLAN Not.*, vol. 36, no. 5, pp. 114–124, May 2001. [Online]. Available: http://doi.acm.org/10.1145/381694.378821

[15] Free Software Foundation, "Autoconf, portability of c functions," http://www.gnu.org/savannah-checkouts/gnu/autoconf/manual/autoconf-2.69/html_node/Function-Portability.html#Function-Portability, [Online; resurss apskatīts 25-Mai-2014].

[16] ——, "malloc() realizācija," https://sourceware.org/git/?p=glibc.git;a=blob;f=malloc/malloc.c;h=1120d4df8487b78a9f1ceb5394968d6ab651986e;hb=refs/heads/master, [Online; resurss apskatīts 25-Mai-2014].

[17] M. Yan, "Anatomy of memory managers," http://core-analyzer.sourceforge.net/index_files/Page335.html, [Online; resurss apskatīts 28-Apr-2014].

[18] T. Ferreira, M. Fernandes, and R. Matias, "A comprehensive complexity analysis of user-level memory allocator algorithms," in *Computing System Engineering (SBESC), 2012 Brazilian Symposium on*, Nov 2012, pp. 99–104.

[19] G. Xu and A. Rountev, "Precise memory leak detection for java software using container profiling," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 151–160. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368110

[20] R. Reese, *Understanding and Using C Pointers*. O'Reilly Media, Inc., 2013, p. 38.

[21] "Memory leak detection using electric fence and valgrind," http://rts.lab.asu.edu/web_438/project_final/CSE_598_Memory_leak_detection.pdf, [Online; resurss apskatīts 5-Mai-2014].

[22] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247362

[23] Microsoft, "How to troubleshoot a memory leak or an out-of-memory exception in the biztalk server process," http://support.microsoft.com/kb/918643, [Online; resurss apskatīts 6-Mai-2014].

[24] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *Proceedings of the International Workshop on Memory Management*, ser. IWMM '95. London, UK, UK: Springer-Verlag, 1995, pp. 1–116. [Online]. Available: http://dl.acm.org/citation.cfm?id=645647.664690

[25] B. Randell, "A note on storage fragmentation and program segmentation," *Commun. ACM*, vol. 12, no. 7, pp. 365–ff., Jul. 1969. [Online]. Available: http://doi.acm.org/10.1145/363156.363158

[26] M. S. Johnstone and P. R. Wilson, "The memory fragmentation problem: Solved?" *SIGPLAN Not.*, vol. 34, no. 3, pp. 26–36, Oct. 1998. [Online]. Available: http://doi.acm.org/10.1145/301589.286864

[27] Silicon Graphics, "Prodev workshop: Debugger user's guide, detecting heap corruption," http://menehune.opt.wfu.edu/Kokua/More_SGI/007-2579-009/sgi_html/ch09.html, [Online; resurss apskatīts 21-Mai-2014].

[28] "Sourceware bugzilla – bug list," https://sourceware.org/bugzilla/buglist.cgi?bug_status=___open___&content=malloc&no_redirect=1&order=relevance%20desc&product=glibc&query_format=specific, [Online; resurss apskatīts 21-Mai-2014].

[29] J. Fusco, *Linux Programmer's Toolbox*. Pearson Education, Inc., 2007, pp. 312–314.

PIELIKUMI

1. pielikums

Atmiņas noplūde

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

int main(int argc, char** argv)
{
  int num_allocations = atoi(argv[1]);
  int alloc_size    = atoi(argv[2]);
  printf ("Tiks iedalīti %i gabali, %i KiB katrs, %i KiB kopumā.\n",
          num_allocations, alloc_size, num_allocations*alloc_size);

  char** arr = new char* [num_allocations];
  for (int i=0; i<num_allocations; i++)
    arr[i] = new char[alloc_size*1024];
  printf ("Iedalīšana notika.\n");

  for (int i=0; i<num_allocations; i++)
    arr[i] = NULL;

  printf ("%i KiB pazaudēti.\n",
          num_allocations*alloc_size);

  abort();
  delete[] arr;
}
```

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

int main(int argc, char** argv)
{
  int num_allocations = atoi(argv[1]);
  int alloc_size   = atoi(argv[2]);
  printf ("Tiks iedalīti %i gabali, %i KiB katrs, %i KiB kopumā.\n",
          num_allocations, alloc_size, num_allocations*alloc_size);

  char** arr = new char* [num_allocations];
  for (int i=0; i<num_allocations; i++)
    arr[i] = new char[alloc_size*1024];
  printf ("Iedalīšana notika.\n");

  for (int i=0; i<num_allocations; i++)
    memset (arr[i], 7, alloc_size*1024);
  printf ("Atmiņa ir aizpildīta.\n");

  for (int i=0; i<num_allocations-1; i++)
    delete[] arr[i];
  printf ("Atmiņa ir atbrīvota izņemot pēdējo gabalu.\n");

  abort();
  delete[] arr;
}
```

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

int main(int argc, char** argv)
{
  int num_allocations = atoi(argv[1]);
  int alloc_size   = atoi(argv[2]);
  printf ("Tiks iedalīti %i gabali, %i baitu katrs, %i baitu kopumā.\n",
          num_allocations, alloc_size, num_allocations*alloc_size);

  char** arr = new char* [num_allocations];
  for (int i=0; i<num_allocations; i++) {
    if (i%2 == 0) {
      arr[i] = new char[alloc_size];
      memset (arr[i], 0, alloc_size);
    } else {
      arr[i] = new char[alloc_size+16];
      memset (arr[i], 0, alloc_size+16);
    }
  }
  printf ("Iedalīšana notika.\n");
  printf ("Atmiņa ir aizpildīta.\n");

  for (int i=0; i<num_allocations; i = i+2)
    delete[] arr[i];
  printf ("Atmiņa uz kuru norāda katra otra norāde ir atbrīvota.\n");

  abort();
  delete[] arr;
}
```

```
1  #*************************************************#
2  # Autore: Renata Januškeviča
3  # 29.05.2014
4  # Lai palaistu skriptu, ir nepieciešams:
5  # 1) ielādēt skriptu gdb atkļūdotājā ar "source <script.gdb>" komandu,
6  # 2) izsaukt lietotāja definēto komandu analyze ar 1 argumentu,
7  # $arg0: galvenās arēnas adrese, kuru iespējams iegūt ar komandas "p &main_arena" ↩
       palīdzību;
8  # Pirms sāks skripta izpildi ir nepieciešams novietot atmiņas izmeti darbā mapē un ↩
       jāpaliecinās, ka atmiņas izmetes nosaukums ir core
9  # Piemēram: "analyze 0x845e000"
10 #*************************************************#
11
12 # $arg0: galvenās arēnas adrese
13 # komanda apstaiga kaudzi, saskaita gabalu skaitu bez norādēm un izdruka rezultātu
14 define get_alloc_chunk
15   # mainīgie, kas palīdz iegūt norādi uz kaudzi
16   set $system_mem = (long *) ($arg0 + 1096)
17   set $top_size_address = (long *) ($top_address[0] + 4)
18   set $top_chunk_size = $top_size_address[0] & ~7
19   set $heap_pointer = (long *) ($top_address[0] - $system_mem[0] + $top_chunk_size)
20   # $malloc_pointer ir norāde, kura tiek atgriezta programmai
21   set $malloc_pointer = 0
22   # $unref skaitītājs atmiņas gabaliem bez norādēm
23   set $unref = 0
24
25   while ($heap_pointer != $top_address[0])
26     # nākamā atmiņas gabalā atrodas kontroles zīmes, p zīme palīdz noteikt vai ↩
         iepriekšējais gabals tiek iedalīts programmai
27     set $next_chunk = $heap_pointer + (($heap_pointer[1] & ~7)/4)
28     # $x mainīgajā tiek saglabāta atrasta simbolu virkne no datnes, kurā tiek ↩
         uzglabāta atmiņas izmete heksadecimālājā formātā
29     set $x = 0
30     if (($next_chunk[1] & 1) == 1)
31       set $malloc_pointer = $heap_pointer + 2
32
33       # tiek meklēta norāde
34       eval "shell cat gdb.core | grep %x > gdb.log", $malloc_pointer
```

```
35      shell echo set \$x=\"$(cat gdb.log)\" > gdb.log
36      # tiek nolasīta $x vērtība "
37      source gdb.log
38
39      if (sizeof($x) == 1)
40        set $unref = $unref + 1
41        printf "Nav atrasta norāde uz adresi: %x\n", $malloc_pointer
42      end
43    end
44    # kārtējais gabals ir apstrādāts, ir nepieciešams pārvietot kaudzes norādi
45    set $heap_pointer = $heap_pointer + (($heap_pointer[1] & ~7)/4)
46  end
47  shell rm gdb.log
48  shell rm gdb.core
49  printf "Procesa adrešu telpā tiek pazaudēts(i): %i gabals(i).\n", $unref
50 end
51
52 # $arg0: galvenās arēnas adrese
53 # komanda saglabā atmiņas izmeti heksadecimālā formātā un izdzēš atstarpes, jo dati ↩
       var netikt izlīdzināti
54 define analyze
55   set $top_address = (long *) ($arg0 + 48)
56
57   if ($top_address[0] != 0)
58     shell od -t x core > gdb_tmp.core
59     shell cat gdb_tmp.core | tr -d ' ' > gdb.core
60     shell rm gdb_tmp.core
61     printf "\n------------ Atmiņas noplūde --------\n"
62     get_alloc_chunk $arg0
63   else
64     printf "Atmiņa programmā netiek dinamiski iedalīta."
65   end
66 end
```

Gdb skripts fragmentēšanas un maksimālās atmiņas izmantošanas problēmas atkļūdošanai

```
#*************************************************#
# Autore: Renata Januškeviča
# 23.05.2014
# Lai palaistu skriptu, ir nepieciešams:
# 1) ielādēt skriptu gdb atkļūdotājā ar "source <script.gdb>" komandu,
# 2) izsaukt lietotāja definēto komandu analyze ar 2 argumentiem,
# $arg0: galvenās arēnas adrese, kuru iespējams iegūt ar komandas "p &main_arena" ↩
    palīdzību;
# $arg1: skaitlis, kurš norāda cik sīki tiks sadalīta kaudze.
# Piemēram: "analyze 0x845e000 5"
#*************************************************#

# $arg0: galvenās arēnas adrese
# $arg1: bin saraksta numurs
# komanda savāc datus, kuri nepieciešami statistikai, druka informāciju par sarakstiem
define free_chunk_list
  # bin kārtēja saraksta sākuma adrese
  set $start_bin = (long *) ($arg0 + 56 + $arg1 * 8)
  # malloc() funkcija atgriež šo norādi programmai
  set $free_chunk = (long *) ($start_bin[1] + 8)
  set $chunk_count = 0
  set $chunk_max_size = 0
  set $total_size = 0

  while ($free_chunk != $start_bin)
    # pēdējie 3 biti netiek izmantoti izmēra glabāšanai
    set $chunk_size = ($free_chunk[-1] & ~7)

    if ($chunk_count == 0)
      # pirmajā iterācija maksimāls un minimāls izmērs ir pirmā gabala izmērs
      set $chunk_max_size = $chunk_size
      set $chunk_min_size = $chunk_size
    else
      if ($chunk_min_size > $chunk_size)
        set $chunk_min_size = $chunk_size
      end
      if ($chunk_max_size < $chunk_size)
        set $chunk_max_size = $chunk_size
```

```
38        end
39      end
40
41      set $chunk_count = $chunk_count + 1
42      set $total_size = $total_size + $chunk_size
43      set $free_chunk = (long *) ($free_chunk[1] + 8)
44    end
45
46    # dati par katru sarakstu, ja tās nav tukšs
47    if ($chunk_count != 0)
48      printf "Bin numurs %i: %i gabals(-li) (%i - %i baiti), kopumā = %i baiti\n", $arg1 ↩
          + 1, $chunk_count, $chunk_min_size, $chunk_max_size, $total_size
49    end
50  end
51
52  # $arg0: galvenās arēnas adrese
53  # komanda druka kopējo statistiku
54  define print_stat
55    set $system_mem = (long *) ($arg0 + 1096)
56    set $top_size_address = (long *) ($top_address[0] + 4)
57    # pēdējie 3 biti netiek izmantoti izmēra glabāšanai
58    set $top_chunk_size = $top_size_address[0] & ~7
59    set $alloc_memory = $system_mem[0] - $free_in_arena - $top_chunk_size
60    set $used_and_freed = $system_mem[0] - $top_chunk_size
61
62    printf "\n----------------------- Fragmentēšana -----------------------\n"
63    printf "Lielākā gabala izmērs: %i baiti (%i KiB, %i MiB),\n", $biggest_free_size, ↩
          $biggest_free_size/1024, $biggest_free_size/1024/1024
64    printf "Kopējā atbrīvotā atmiņa bin sarakstos: %i baiti (%i KiB, %i MiB),\n", ↩
          $free_in_arena, $free_in_arena/1024, $free_in_arena/1024/1024
65
66    printf "\n---------------------- Kopējā statistika ----------------------\n"
67    printf "Kaudzes segmenta izmērs: %i baiti (%i KiB, %i MiB),\n", $system_mem[0], ↩
          $system_mem[0]/1024, $system_mem[0]/1024/1024
68    printf "Kopējais programmai iedalītais atmiņas daudzums: %i baiti (%i KiB, %i ↩
          MiB),\n", $alloc_memory, $alloc_memory/1024, $alloc_memory/1024/1024
69    printf "Top gabala izmērs: %i baiti (%i KiB, %i MiB),\n", $top_chunk_size, ↩
          $top_chunk_size/1024, $top_chunk_size/1024/1024
70    printf "Atbrīvoto gabalu skaits arēnā: %i,\n", $count_in_arena
71  end
72
73  # $arg0: galvenās arēnas adrese
74  # $arg1: kaudzes sadalījums apgabalos
```

```
# komanda izvada atbrīvoto un iedalīto gabalu attiecību apgabalā
define div_stat
  set $fract_size = $used_and_freed/$arg1
  set $heap_pointer = (long *) ($top_address[0] - $system_mem[0] + $top_chunk_size)
  set $chunk_size = 0

  printf "\n------------- Maksimālā atmiņas izmantošanas problēma --------\n"
  printf "Atbrīvoto un iedalīto gabalu attiecība:\n"

  while ($heap_pointer != $top_address[0])
    set $free_size = 0
    set $alloc_size = 0
    set $fract_fin = $heap_pointer + $fract_size/4
    set $fract_start = $heap_pointer

    while (($heap_pointer < $fract_fin) && ($heap_pointer != $top_address[0]))
      set $next_chunk = $heap_pointer + (($heap_pointer[1] & ~7)/4)
      set $chunk_size = ($heap_pointer[1] & ~7)

      if (($next_chunk[1] & 1) == 1)
        set $alloc_size = $alloc_size + $chunk_size
      else
        set $free_size = $free_size + $chunk_size
      end
      set $heap_pointer = $next_chunk
    end

    set $relation = ((double) $free_size/($alloc_size + $free_size)) * 100
    printf "Apgabals 0x%x - 0x%x %i%%\n", $fract_start, $heap_pointer, $relation
  end
end

# $arg0: galvenās arēnas adrese
# $arg1: kaudzes sadalījums apgabalos
define analyze
  set $free_in_arena = 0
  set $bin_number = 0
  set $biggest_free_size = 0
  set $count_in_arena = 0
  set $top_address = (long *) ($arg0 + 48)

  if ($top_address[0] != 0)
    # Lai savāktu statistiku, ir nepieciešams apstaigāt visus 128 bin sarakstus
```

```
118     while ($bin_number < 127)
119       free_chunk_list $arg0 $bin_number
120       if ($biggest_free_size < $chunk_max_size)
121         set $biggest_free_size = $chunk_max_size
122       end
123       set $bin_number = $bin_number + 1
124       set $free_in_arena = $free_in_arena + $total_size
125       set $count_in_arena = $count_in_arena + $chunk_count
126     end
127
128     print_stat $arg0
129     if ($arg1 != 0)
130       div_stat $arg0 $arg1
131     end
132   else
133     printf "Atmiņa programmā netiek dinamiski iedalīta."
134   end
135 end
```

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 _____ ___.06.2014.

Recenzents: Dr.sc.comp. Vineta Arnicāne

Darbs iesniegts Datorikas fakultātē ___.06.2014.

Dekāna pilnvarotā persona: vecākā metodiķe Ārija Sproģe _____

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē

___.06.2014. prot. Nr. _____.

Komisijas sekretārs(-e): _____