

LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

**ATMIŅAS IZMETES PIELIETOŠANA
KAUDZES ATKĻŪDOŠANAS METODES
IZSTRĀDEI**

BAKALAURA DARBS

Autors: **Renata Januškeviča**

Studenta apliecības Nr.: rj10013

Darba vadītājs: Mg. sc. ing. Romāns Taranovs

RĪGA 2014

ANOTĀCIJA

Darbs sastāv no ievada, 6 nodaļām, secinājumiem un 3 pielikumiem. Tajā ir 49. lappuses, 24 attēli, 3 tabulas pamattekstā un 28 nosaukumi literatūras sarakstā.

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

ABSTRACT

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

The work consists of introduction, 6 chapters, conclusions and 3 appendixes. It contains 49. pages, 24 figures, 3 tables and 28 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. Atklūdošana, izmantojot atmiņas izmeti.....	7
1.4.1 Atmiņas izmetes atklūdošana, 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 glibc bibliotēkā.....	13
2.3.1 Atmiņas arēna.....	14
2.3.2 Atbrīvotās atmiņas organizācija.....	15
2.3.3 Atmiņas gabali.....	17
2.4. Secinājumi.....	19
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.	Atklūdošanas metodes apraksts	31
4.1.	Analizatora darbības princips	31
4.2.	Atmiņas noplūdes analizators	32
4.3.	Maksimālās atmiņas izmantošanas problēmas analizators	32
4.4.	Fragmentēšanas analizators	34
4.5.	Secinājumi	34
	Galvenie rezultāti un secinājumi	35
	Pateicības	36
	Izmantotā literatūra un avoti	39
	Pielikumi	40
1.	pielikums. Atmiņas noplūde	40
2.	pielikums. Maksimālās atmiņas izmantošanas problēma	41
3.	pielikums. Fragmentēšana	42
4.	pielikums. Gdb skripts atmiņas noplūdes atklūdošanai	43
5.	pielikums. Gdb skripts pārējo divu problēmu atklūdošanai	45

APZĪMĒJUMU SARAKSTS

Debugging - Atklūdošana - Procedūra pielauto 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 atklūdošanai.

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 pirmkodu 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 priekš 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 paņēmieni, kurš ir pielietots kompilācijas laikā.

Dynamic memory allocation - Dinamiskā atmiņas iedalīšana - atmiņas iedalīšanas paņēmieni, 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 - C valodas bibliotēka, kura nodrošina sistēmas izsaukumus un pamata funkcijas (malloc(), printf(), open()). Bibliotēku izmanto GNU operētājsistēmās.

IEVADS

Daudzas problēmas C un C++ programmās ir saistītas ar dinamisko atmiņu [1]. Problēmas ir grūti atklāt, 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 ierobežoti ar malloc() un free() funkciju izmantošanu. Tāpēc, lai atklātu lietotnes tiek veidotas speciālās uzturēšanas komandas. Atklādošanas 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 atklādošanas procesu un padara darbu neefektīvu.

Šajā bakalaura darbā tiek aprakstīta un piedāvāta pieeja, kura palīdzēs vienkāršot atklādošanas procesu problēmām, kas saistītas ar kaudzi un dinamisko atmiņas iedalīšanu. Piemēram, kaudzes problēmas var būt sekojošas: atmiņas noplūde, datu kaudzes bojāšana, maksimālās atmiņas izmantošanas problēma, fragmentēšana, kļūdas trešās puses bibliotēkās. Darbā ir aprakstīta atklādošanas pieeja, kura balstīta uz atmiņas izmetes analīzi. Šī pieeja ir aprakstīta uz 3 izvēlētiem problēmu piemēriem: atmiņas noplūde, fragmentēšana, maksimālās atmiņas izmantošanas problēma. Atmiņas noplūde ir izvēlēta, tāpēc ka 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 atklādotas ar atmiņas izmetes palīdzību. Pieveja var tikt pielietota gadījumos, kad ir novērojamas problēmu sekas vai, kad ir vēlme pārliecināties par problēmas eksistenci, sasniedzot kādu stāvokli programmā.

Darba sastāv no ievada, 4 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 3 no 5 izvēlētiem problēmām un identificētas to pazīmes atmiņas izmetē, nodaļā iekļauti arī pārējo 2 problēmu apraksti.
- Nodaļā "Atklādošanas 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. Nodaļa ir aprakstīts atklūdošanas process, kas var būt 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 atklūdotājā, tāda 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` atklūdotāja 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.

```
1  #include <signal.h>
2
3  int main () {
4
5      raise(SIGSEGV); /* signāls, kurš tiek nosūtīts, kad ir nederīga norāde */
6
7      return 0;
8  }
```

1.1. att. Atmiņas izmetes ģenerēšana, pārtraucot procesa darbību

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

`abort()`, kā arī var apzināti pieļaut kļūdu kodā. Tāda kļūda kā dalīšana ar nulli nosūta SIGFPE signālu, bet vērsšanās pēc null radītāja - SIGSEGV signālu. Izmantojot funkciju `raise()`, ir iespējams norādīt atmiņas izmeti izraisīto 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`.

```
1  #include <stdlib.h>
2
3  int main () {
4      int child = fork();
5      if (child == 0) {
6          abort(); /* izpilda bērna process */
7      }
8      return 0;
9  }
```

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].

```
1  (gdb) attach <pid>
2  (gdb) generate-core-file <filename>
3  (gdb) detach
4  (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 -<SIGNAL_NUMBER> <PID>`, signāls ar numuru `SIGNAL_NUMBER` tiks nosūtīts procesam ar norādītu `PID` vērtību. Izmantojot shell komandrindas interpretatoru ir iespējams izmantot īsinājumtaustiņus signālu nosūtīšanai. Nospiežot `Control + \` tiks nosūtīts `SIGQUIT` signāls procesam, kas pašreiz ir palaists (sk. 1.4. attēlu) [7]. Šajā piemēra ziņojumu - `Quit (core dumped)`, izdruka shell. Šis komandrindas interpretators noteic, ka `sleep` procesu (`shell` bērnu) pārtrauca `SIGQUIT` signāls. Pēc šī signāla nosūtīšanās, darba mapē tiek uzģenerēta atmiņas izmete.

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

1.4. att. Atmiņas izmetes ģenerēšana, izmantojot īsinājumtaustiņus

1.2. Atmiņas izmetes ģenerēšanas nosacījumi

Lai uzģenerētu atmiņas izmeti ir jābūt izpildītiem sekojošiem 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` jāpārsniedz ģenerējamā faila izmēru, `RLIMIT_FSIZE` robežai jāļauj procesam izveidot atmiņas izmeti;
- ir jāatļauj lasīt bināro datni, kura ir palaista;
- failu sistēmai, kurā atrodas darba mape, ir jābūt uzmontētai priekš rakstīšanas, tai nav jābūt pilnai un ir jāsaturs brīvie indeksa deskriptori;
- bināro datni jāizpilda lietotājam, kurš ir datnes īpašnieks (`group owner`).

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

1.3. Atmiņas izmetes validēšana

Šis posms nav pietiekoši labi izpētīts literatūrā. Bet pirms sākt atklādošanu 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 vieta ir ierobežota un kopējot, pārsūtot vai ģenerējot atmiņas izmeti var pazaudēt daļu no datiem beigās. Šo gadījumu var pārbaudīt, izdrukājot galveni ar `objdump -p <core>` komandu. Gadījumos, kad atmiņas izmete tiek nogriezta, tad tiks izvadīts brīdinājums par to, ka datnes izmērs neatbilst sagaidāmam. Šī kļūda varētu traucēt analizēt atmiņas izmeti, jo tiek sagaidīts ka dati ir pilnīgi.

1.4. Atklūdošana, izmantojot atmiņas izmeti

Atmiņas izmete satur datus, kuri dod iespēju atrast kļūdas. Tāpēc atmiņas izmete var tikt pielietota, lai veiktu lietotnes atklūdošanu, pēc neparedzētas programmas apstāšanās. Atmiņas izmetes analīze ir efektīvs veids, kā var attālināti atrast un izlabot kļūdas bez iejaukšanās un tiešas piekļuves sistēmai. Daudzos gadījumos, atmiņas izmete ir speciāli uzģenerēta datne, kura palīdz iegūt atmiņas stāvokli uz signāla nosūtīšanas 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 priekš izpildāmām datnēm, objektu datnēm, bibliotēkām un atmiņas izmetēm. Lai darbotos ar atmiņas izmetem ir nepieciešams, lai rīks, kurš tika izvēlēts (bibliotēka, utilitprogramma vai atklūdotājs) atbalstītu uzģenerētās datnes formātu. GNU gdb ir Linux standarta atklūdotājs [8], kurš ir plaši pielietojams atmiņas izmešu analīzei. Turpmāk tiek apskatīta atmiņas izmetes analīze ar gdb atklūdotāja palīdzību.

1.4.1. Atmiņas izmetes atklūdošana, izmantojot gdb

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

```
1 (gdb) p main
2 $ 1 = {<text variable, no debug info>} 0x80483e4 <main>
```

1.5. att. Atmiņas izmete nesatur atklūdošanas informāciju

Kad atmiņas izmete ir uzģenerēta, tad to var apskatīt, izmantojot gdb atklūdotāju

(sk. 1.6. attēlu). Atklūdotājam kā argumenti tiek padoti: izpildāms fails un atmiņas izmete. Izpildāmam failam ir jāatbilst atmiņas izmetei, lai varētu apskatīt korektus, nesabojātus datus.

```
1 $ gdb <path/to/the/binary> <path/to/the/core>
```

1.6. att. Atmiņas izmetes atvēršana, izmantojot gdb atklūdotāju

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: statistiskie 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 sekojošo formātu: `x/nfu addr`. Ir nepieciešams norādīt atmiņas adresi (`addr`), no kuras sākt atmiņas izdrukāšanu, formātu (`f`), apgabala lielumu (`n`) un norādīt vienības lielumu (`u`). Izmantojot doto piemēru (sk. 1.7. attēlu), tiks izdrukāti 4 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.

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

1.7. att. Atmiņas apgabala izdrukāšana

Lai uzzinātu kuram simbolam (funkcijai, mainīgam vai tipam) pieder adrese var izmantot sekojošo 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.

```
1 (gdb) p/a 0x54320
2 $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 pēdējā un daudzos gadījumos, ļauj ātri identificēt kļūdas cēloņi.

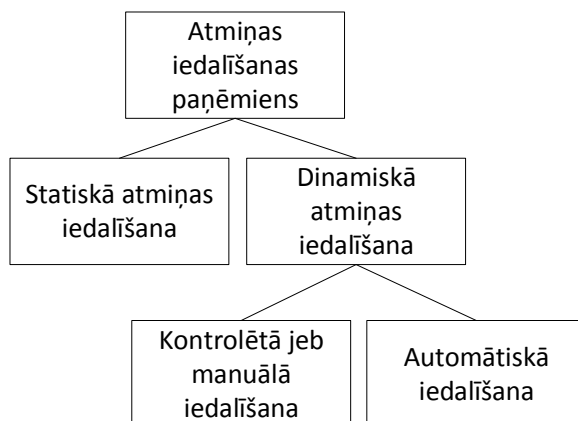
Backtrace nesniedz patieso informāciju par funkciju, ja process tika pabeigts ārējo apstākļu dēļ, nevis tāpēc, ka bija notikusi kļūda programmā. Katra rindiņa satur rāmi (frame). Backtrace izdruka sākas ar rāmi, kurā iekļauta funkcija, kura bija izpildīta pēdēja. Nākamais rāmis iekļauj funkciju, kas 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 priekš 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 parakstītas. 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 kaudzes¹ lomu tajā. Šajā nodaļā ir aprakstīti atmiņas iedalīšanas paņēmieni, ir dots īss ieskāts GNU C bibliotēkas `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 paņēmieni klasifikācija

2.1.1. Statiskā atmiņas iedalīšana

Statiskā atmiņas iedalīšana nozīmē, ka atmiņa tiek iedalīta pirms programmas palaišanas, parasti tas notiek kompilācijas laikā. Programmas izpildēs laikā atmiņa vairs netiek iedalīta, ka arī netiek atbrīvota. Statiskais atmiņas iedalīšanas paņemiens nodrošina to, ka atmiņa tiek iedalīta statiskiem un globāliem mainīgiem, neatkarībā no tā vai mainīgais tiks izmantots programmā pie dotajiem nosacījumiem vai nē.

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

2.1.2. Dinamiskā atmiņas iedalīšana

Dinamiskā atmiņas iedalīšana nozīme, ka atmiņa tiek iedalīta programmas izpildes laikā. Tas var būt nepieciešams, kad atmiņas daudzums nav zināms programmas kompilācijas laikā. Dinamiskā atmiņas iedalīšana, var būt realizēta ar steka vai kaudzes palīdzību un var būt automātiskā vai kontrolētā [10].

Automātiskā iedalīšana notiek, kad sākas programmas funkcijas izpilde. Priekš automātiskās atmiņas iedalīšanas, izmato steku. Šeit viens un tas pats atmiņas apgabals, kurš bija atbrīvots, var tikt izmantots vairākas reizes. 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 nozīme, ka programma izvēlās brīvus atmiņas gabalus no pieejama segmenta telpas, priekš programmas datiem. Kontrolētā jeb manuālā atmiņas iedalīšana parasti ir nodrošināta ar kaudzes palīdzību. Šeit nav iespējams piekļūt visiem iedalītiem 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 lietotnes 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 tekošam procesam. Šis apgabals, no procesa viedokļa, ir viena lineārā virtuālā adrešu telpa, kura ir sadalīta vairākos segmentos. Svarīgākie procesa segmenti [12]:

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

- Statisko datu segments - apgabals, kurā tiek uzglabāti dati ar iepriekš zināmu izmēru. Tās ir globālie un statiskie mainīgie. Operētājsistēma iedala šī apgabala kopiju katrai programmas instancei atsevišķi.
- Kaudzes segments - apgabals, no kura tiek iedalīta dinamiskā atmiņa. Kaudzes segmentā atrodas dinamiski iedalītā un atbrīvotā atmiņa. Kaudzes segmenta saturs ir sadalīts sīkāk, mazākos atmiņās gabalos. Kaudzes segments aug no mazākas adreses līdz lielākai. Lai palielinātu kaudzes segmenta izmēru, tiek veikts `brk()` sistēmas izsaukums. Izsaukums uzstāda kaudzes segmenta jauno beigu robežu [13].
- Steka segments - apgabals, kurā tiek uzglabāti: funkciju izsaukumu stāvoklis, katram funkcijas izsaukumam, ka arī lokālo mainīgo un reģistru vērtības. Steks aug no lielākas adreses līdz mazākai. Steks ir iedalīts priekš katras programmas instances atsevišķi. Iedalīto virtuālo adresu karti var atrast `/proc/<pid>/maps` datnē.

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 tās nodrošinās labāko veikspēju [14].

```

1 int * ptr1 = new int; // C++
2 int * ptr1 = (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 izmantojot C un C++ kodu.

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, ir nepieciešams zināt cik daudz vietas aizņem viens elements un kāds ir elementu skaits. Funkcija `malloc()` atgriež `void` tipa rādītāju, tāpēc C programmās ir nepieciešams izmantot drošo tipa pārveidotāju (`typecast`). Tas ir nepieciešams, lai saglabātu atgriezto norādi lokālajā mainīgajā. Atmiņas inicializācija C kodā var būt veikta izmantojot arī citas funkcijas, piemēram `calloc()` funkciju, kura atgriež atmiņas 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 priekš C++ programmas izmanto `new` un `delete` operatorus (sk. 2.3. attēlu), bet priekš 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 pieejams no operētājsistēmas atmiņas daudzums ar laiku samazinās.

```
1 delete ptr1; // C++
2
3 if(ptr1 != NULL)
4     free(ptr1); /* C */
```

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

2.3. Atmiņas organizācija glibc 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. Attēlā¹ 2.4. ir parādītas 3 arēnas, kuras ir atdalītas savā starpā ar raustītam līnijām. Atmiņas arēnu vienkāršotā veidā 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 kārtēja kaudze ir izlietotā un tajā nav atmiņas, tad tiek iedalīta jauna kaudze ar fiksēto 64 MB izmēru. Tāda veidā arēnas var tikt paplašinātas, izveidojot jaunās kaudzes un savienojot tās sava starpā. Jaunai kaudzei ir norāde gan uz

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

arēnu, kurai tā pieder, gan uz iepriekšējo kaudzi.

Lai uzlabotu veikspēju 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 veikspēju. Gadījumos, kad pavedieni izmanto atmiņu no vairākām atsevišķām arēnām (kā tas notiek `ptmalloc2` realizācijā), tad vienas arēnas bloķēšana neietekmē atmiņas iedalīšanu pārējās kaudzēs, kuras nepieder nobloķētai arēnai, un atmiņas iedalīšana var notikt paralēli. Lai nodrošinātu labāku veikspēju, 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 mainīga `MALLOC_ARENA_MAX` vērtības. Tā kā pavedienu skaits parasti nepārsniedz divkārtšo kodolu skaitu, tad normālā gadījumā katrs pavediens izmanto atsevišķo arēnu. Darbība ar arēnām notiek saskaņā ar sekojošo algoritmu:

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

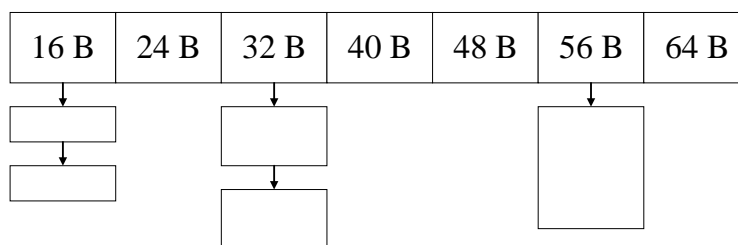
Vispirms atmiņas iedalīšana sākas no galvenās arēnas (`main arena`). GNU C bibliotēkā ir globāls `malloc_state` objekts - galvenā arēna, kura atšķiras no pārējām arēnām ar to, ka tā tiek paplašināta, izmantojot `brk()` nevis `mmap()` sistēmas izsaukumu. Līdz ar galvenās arēnas paplašināšanu, tiek paplašināts arī procesa kaudzes segments. `brk()` sistēmas izsaukumam ir viens arguments, kurš uzstāda procesa kaudzes segmenta jaunas beigas. `mmap()` sistēmas izsaukums paplašina pārējās dinamiskās arēnas daudzpavedienu lietotnēs, ka 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. Priekš 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 saistītiem 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 programma 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 āra (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 izmērā 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 [?]. Tāda 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 mazā izmērā (parasti līdz 64 baitiem) un pieder ātrajām sarakstām, bet parastie gabali - parastajām sarakstām. 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 kaudzēs 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 sarakstām
struct malloc_chunk* bk	Rādītājs uz iepriekšējo atbrīvoto gabalu, ja kārtējais gabals ir atbrīvots un pievienots dubultsaišu bin sarakstām

```
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 izmantoti

divas `INTERNAL_SIZE_T` tipa vērtības un, kad gabals ir atbrīvots, tad dubultsaīšu saraksta uzturēšanai, papildus tiek izmantoti divi rādītāji (`fd` un `bk`) 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 $2 * \text{sizeof}(\text{INTERNAL_SIZE_T})$ reizinājums. Šis skaitlis ir vienāds ar 8 baitu izlīdzinājumu, ja makrodefinīcijas `INTERNAL_SIZE_T` vērtība ir vienāda ar 4 baitiem [16].



2.7. att. Atmiņas gabalu grafiskais struktūras attēlojums

No kreisās pusēs attēlots (sk. 2.7. attēlu) [16] atmiņas gabals, kurš bija iedalīts procesam, no labās, tās, kurš bija atbrīvots. Abos gadījumos rādītājs `chunk` attēlo atmiņas gabalu sākumu. Pēc šī rādītāja 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 $2 * \text{sizeof}(\text{INTERNAL_SIZE_T})$, 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 <code>mmap()</code> sistēmas izsaukumu
P	iepriekšējais atmiņas gabals tiek izmantots

Rādītājs `mem` ir `malloc()` funkcijas atgriežamā vērtība, jeb rādītājs 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 rādītāji uz nākamo un iepriekšējo atbrīvotiem gabaliem, kas atrodas saistītajā sarakstā.

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, ka 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].

2.4. Secinājumi

1. Tiek aplūkota literatūra un GNU C bibliotēkas `malloc` pirmkods.
2. Ir apkopotas zināšanas un uzzīmēta `ptmalloc2` iedalītāja organizācijas reprezentācija (sk. 2.8. attēlu).
3. Galvenie secinājumi par kaudzi:
 - kontrolētā atmiņas iedalīšana ir dinamiskā atmiņas iedalīšana, kura notiek programmas izpildes laikā, un atmiņa parasti tiek iedalīta no kaudzes;
 - kaudzes segments ir procesa adrešu telpas apgabals, kurš var tikt paplašināts;
 - iedalīt atmiņu no kaudzes var no lietotnes ar `malloc()` vai `new` palīdzību;
 - kaudze sastāv no daudziem iedalītiem un atbrīvotiem atmiņas gabaliem. Atbrīvotie gabali var būt savienoti sarakstā, pārrakstot lietotāja datus;
 - vienu arēnu var veidot vairākas kaudzes, kas ir savienotās savā starpā.

3. KAUDZES PROBLĒMAS

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

- nekorekta kaudzes pārvaldība ar `malloc()` vai citām līdzīgām funkcijām;
- nekorekta kaudzes pārvaldība, ko nodrošina iedalītājs.

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 tiek aprakstītas 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. Šīs programmas stāvoklis var būt novērojams 3 iemeslu dēļ [19]:

- pēdēja 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 pakāpeniskā procesa palēnināšana, 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 turpinas 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.

```
1  #include <string>
2  using namespace std;
3
4  int main() {
5      string *str;
6
7      for (int i=0; i<10001; i++) {
8          // 10000*14 baiti tiek pazaudēti
9          str = new string("Hello, World!");
10     }
11     delete str;
12
13     return 0;
14 }
```

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 parakstīts, tad piemēra 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.

Sekojošos 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 citās programmas darbojās visu laiku bez apstāšanās;
- ja portatīvām ierīcēm ir ierobežots atmiņas daudzums;
- ja programmas pieprasa atmiņu uzdevumiem, kuri izpildās ilgstošu laika periodu;
- reālā laikā sistēmās, jo ir svarīgi iegūt rezultātu ierobežotajā laikā.

Atmiņas noplūdes problēmu ir grūti atklūdot, 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 atklūdot atmiņas noplūdes problēmu, tādi ka: Valgrind, Totalview, Purify. Taču tie ne vienmēr sniedz pietiekamu informāciju un bieži netiek izmantoti strādājošās sistēmās, jo piedāvātas atklūdošanas 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 pārmērīgs¹ atmiņas daudzums, kas visu laiku pieaug. Kad 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 brīdī, tad uzģenerētās datnes izmērs, atmiņas noplūdes problēmas ietekmēs 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 vairākas reizes. 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.

¹Šajā kontekstā pārmērīgs nozīme, 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 kuriem malloc() funkcija atgrieza norādes procesam. Ja procesa adresu 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 neieklāut daļu no procesa adresu telpas. Šī pazīme nav raksturīga loģiskajai atmiņas noplūdei.

```

1 class MyClass
2 {
3     virtual SomeVirtualMethod();
4
5     public:
6         void* attribute1;
7         void* attribute2;
8 }

```



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

Loģiskās atmiņas noplūde rezultātā visiem atmiņas gabaliem atbilst norādes procesa adresu telpā, bet tādi gabali patērē visu pieejamo procesam 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 šī 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 trapa virsotnes vai maksimuma punkta.

Traps un maksimums ir shēmas, kuras raksturo programmas uzvedību un ir apkopoti pētījumā [24]. Atmiņas daudzums, kas tiek izmantots programmas izpildes laikā var visu laiku mainīties. Kopumā tiek identificētas trīs svarīgākās atmiņas izmantošanas shēmas: traps (ramps), maksimums (peaks), plato (plateaus). Citas atmiņas izmantošanas shēmas ir iespējamās, bet izpaužas ļoti reti. Ne visām programmām ir raksturīgas visās trīs shēmas, bet vairākiem ir raksturīga viena vai divas no tām. Šīs shēmas tika apkopotas, balstoties uz kvantitatīvo programmu novērtējumu [24].

- Traps. 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.
- Maksimums. Šo veidu var nosaukt par trapu tikai ļoti īsa laika periodā. Daudzām programmām var būt nepieciešams izveidot lielas datu struktūras, kāda uzdevuma izpildīšanai. Pēc šī uzdevuma pabeigšanas gandrīz visā pieprasītā atmiņa var tikt atbrīvota. Grafiks šai shēmai izskatās kā lauztā līnija un atmiņas patēriņš var svārstīties dramatiski.
- Plato. Novērojama, kad programmas ātri uzbūvē datu struktūras un izmanto tās ilgā laika periodā, bieži izmanto līdz programmas izpildes beigām.

Problēma ir novērojama, kad liels atmiņas daudzums netiek atgriezts operētājsistēmai pēc izmantošanas, 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 trapa virsotnes vai maksimālā punkta sasniegšanas. Turpmāk tiks apskatīts piemērs, kurš demonstrē to kā izpaužas dotā 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 kilobaiti. Tika palaista programma un katrā programmas solī tika noņemti rādītāji (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 rādītāji

Solis	VSZ	RSS
1. Sākums	3228	612
2. Ar <code>new</code> 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 izmēra. 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 sa-
sniegts maksimums (peaks). Atmiņas izmete palīdz saprast, kas notiek atmiņā. Ar komandu

```

1 (gdb) p main_arena
2 $3 = {mutex = 0, flags = 1, fastbinsY = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, ↵
    0x0}, top = 0x97ba4b8, last_remainder = 0x0,
3 bins = {0x8df6198, 0x8df6198, 0xb76cf478, 0xb76cf478, 0xb76cf480, 0xb76cf480, ↵
    0xb76cf488, 0xb76cf488, 0xb76cf490, 0xb76cf490,
4 0xb76cf498, 0xb76cf498, 0xb76cf4a0, 0xb76cf4a0, 0xb76cf4a8, 0xb76cf4a8, 0xb76cf4b0, ↵
    0xb76cf4b0, 0xb76cf4b8, 0xb76cf4b8,
5 0xb76cf4c0, 0xb76cf4c0, 0xb76cf4c8, 0xb76cf4c8, 0xb76cf4d0, 0xb76cf4d0, 0xb76cf4d8, ↵
    0xb76cf4d8, 0xb76cf4e0, 0xb76cf4e0...},
6 binmap = {0, 0, 0, 0}, next = 0xb76cf440, next_free = 0x0, system_mem = 10375168, ↵
    max_system_mem = 10375168}
7 (gdb) x/16wx 0x8df6198
8 0x8df6198: 0x00000000 0x009ab319 0xb76cf470 0xb76cf470
9 0x8df61a8: 0x00000000 0x00000000 0x07070707 0x07070707
10 0x8df61b8: 0x07070707 0x07070707 0x07070707 0x07070707
11 0x8df61c8: 0x07070707 0x07070707 0x07070707 0x07070707

```

3.3. att. bin saraksta izdruka, izmantojot atmiņas izmeti

p main_arena ir izdrukāta galvenās arēnas struktūra un iegūtas bin sarakstu sākuma adreses (sk. 3.3. attēlu). Visi bin saraksti atrodas masīvā bins. 7 rindīnā ir komanda, kura izdrukā 16 adreses no pirmā nesakārtotā bin saraksta, kurā atrodas gabali, kuri bija nesen atbrīvoti. Sākot ar 8 rindīnu 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 $100 \cdot 1024 \cdot 99 = 10137600$. 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 notiek atmiņas gabalu izlīdzināšana 8 - ka vai 16 - reizinājumam. 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 brīdī. 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).

```
1  #include <stdio.h>
2  #include <malloc.h>
3
4  int main () {
5      char * ptr1;
6      int chunk_size;
7
8      ptr1 = (char *)malloc(4);
9
10     /* tiek iegūts atmiņas gabala izmērs (otrais malloc_chunk elements) */
11     chunk_size = *((char *) ptr1 - sizeof(size_t));
12     /* mazākie 3 biti tiek izmantoti meta informācijas glabāšanai */
13     chunk_size = chunk_size - (chunk_size & 7);
14
15     printf("size = %d\n", chunk_size);
16     free(ptr1);
17
18     return 0;
19 }
```

3.4. att. Izmēra noteikšana iedalītām gabalam

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

iedalīts no kaudzes. Piemēra ir redzams, ka tiek iedalīti 4 baiti, bet programma izdruka 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);
3. tiek atņemtas A, M, P kontroles zīmes $111 = 7$ 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 doto problēmu ir nepieciešams zināt cik daudz atmiņas pieprasīja lietotne. Izmantojot atmiņas izmeti, šo informāciju var iegūt tikai no backtrace, taču šī informācija var netikt iekļauta.

No problēmas apraksta seko, ka ārējai fragmentēšanai ir raksturīgs liels mazo¹ atbrīvoto gabalu skaits. Šie gabali var būt saglabāti vienā no bin sarakstiem:

- ātrajos sarakstos. 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;
- parastajos sarakstos. Ja ātrie gabali tika sapludināti vai gabali ir lielāki par 64 baitiem;

```
1 0x8b30198: 0x00000000 0x00000011 0x00000000 0x00000000
2 0x8b301a8: 0x00000000 0x00000021 0x00000000 0x00000000
3 0x8b301b8: 0x00000000 0x00000000 0x00000000 0x00000000
4 0x8b301c8: 0x00000000 0x00000011 0x08b30198 0x00000000
5 0x8b301d8: 0x00000000 0x00000021 0x00000000 0x00000000
6 0x8b301e8: 0x00000000 0x00000000 0x00000000 0x00000000
7 0x8b301f8: 0x00000000 0x00000011 0x08b301c8 0x00000000
8 0x8b30208: 0x00000000 0x00000021 0x00000000 0x00000000
9 0x8b30218: 0x00000000 0x00000000 0x00000000 0x00000000
10 0x8b30228: 0x00000000 0x00000011 0x08b301f8 0x00000000
```

3.5. att. Fragmentētā kaudze

Kaudze būs saskaldīta un katrs iedalītais gabals robežos ar mazāko atbrīvoto gabalu. Piemēra ir izdrukāts kaudzes saturs (sk. 3.5. attēlu). Piemērā izmantotā atmiņas izmete tika uzģenerēta izmantojot iepriekš sagatavoto programmu (sk. 3. pielikumu). Programma

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

ir izveidotā problēmas pazīmju identificēšanai un tāpēc ir vienkāršots problēmas attēlojums, izmantojot mazā izmēra gabalus: 32 un 16 baitiem. Tā kā programmā uzreiz pēc iedalīšanas tiek pārtraukta, tad ātra 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 sava starpā un atrodas ātrajā sarakstā. Tas ir novērojams, jo lietotāju dotos ātrajiem gabaliem ir uzglabāta nākama gabala adrese. Pēdējām gabalam ātrajā sarakstā ir uzglabāta 0x00000000 adrese. Ja kaudzē atrodas brīvie gabali, kuri nav lielāki par 16 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 novērojamās vairākas bojāšanas kļūdas (heap corruption) [27]:

- robežu pārpildīšana (boundary overrun). Notiek, kad programma raksta aiz malloc() funkcijas robežām. Tāda veidā var pārrakstīt nākamo datu struktūru atmiņā;
- rakstīšana pirms malloc() iedalītas sākuma adreses robežām;
- piekļuve neinicializētam atmiņas gabalam. Programma mēģina lasīt datus no gabala, kurš nav inicializēts;
- piekļuve atbrīvotām gabalam. Programma mēģina lasīt vai rakstīt atmiņas gabalā, kurš bija atbrīvots;
- divkārsa atbrīvošana. 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 izstrādājama programma strādā, izmantojot trešās puses bibliotēkas, tad programmas uzticamība un kvalitāte ir atkarīga arī no iedalītāja trešās puses bibliotēkās. Ir iespējams, ka notiek problēma programmā nekorektās iedalītāja realizācijas dēļ. Piemēram, ir iespējams apskatīt GNU C bibliotēkas pieteiktās kļūdas [28]. Ja iedalītais ir uzturēts, tad iespējams, ka var pierēģistrēt problēmas, lai nākamajās versijās tas tiktu izlabotas. Izmantojot individuālo iedalītāja risinājumu problēmas būs jāatklādo 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. Šīs īpašības ir jāņem vērā izstrādājot lietotnes, lai nerastos problēmas, kas saistītas ar trešās puses bibliotēkām. Ja problēma notiek iekšā trešās puses bibliotēkā, tad ir 3 iespējas kā var izvairīties no problēmām lietotnēs: ir jāatrod ceļš ka nepieļaut doto kļūdu, ir jāgaida atjauninājumus, kurā kļūda tiks izlabota, vai ir jānomaina tekošais iedalītājs.

3.6. Secinājumi

Nodaļā tika izpētītas 3 problēmas un identificētas to pazīmes atmiņas izmete.

1. Atmiņas noplūdes pazīmes:

- pārmērīgs atmiņas izmetes izmērs;
- uz gabaliem kaudzē nav norāžu no procesa adresu 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;

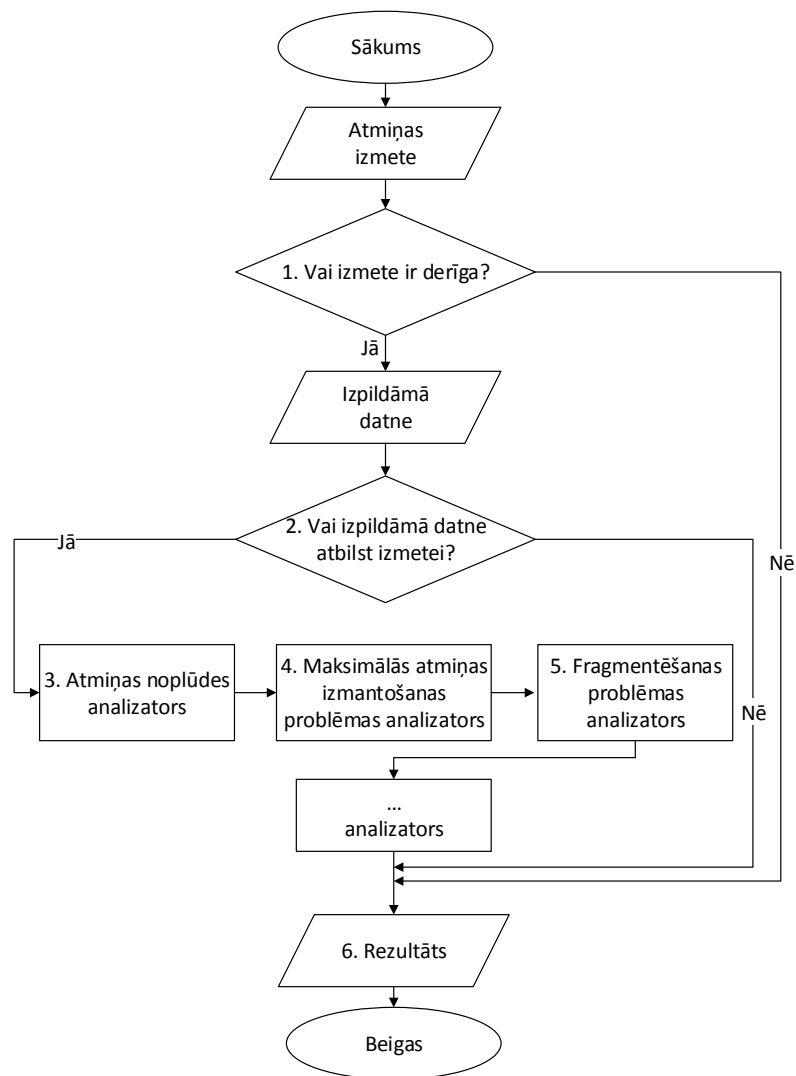
3. Fragmentēšanas pazīmes:

- 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 atklūdošanai. Tāpat ir aplūkots metodes algoritms un 3 analizatoru realizācijas: atmiņas noplūdes analizators, maksimālās atmiņas izmantošanas problēmas analizators, fragmentēšanas problēmas analizators. Analizatori, kuri tika realizēti, nav gatavs risinājums atklūdotājām, bet 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 atklūdošanas algoritms. Šī algoritma

ievaddati ir izpildāmā datne un atmiņas izmete. Atmiņas izmete var būt bojāta, tāpēc pirms sākt atklūdošanas procedūru ir nepieciešams veikt atmiņas izmetes validēšanu (1). Šeit ir jāveic dažas pārbaudes, piemēram, pārbaudi uz neatbilstību formātam vai atmiņas izmetes nogriežto saturu. To, kā var pārbaudīt vai izmetei ir nogriezts saturs ir aprakstīts 1.3. sadaļā. Validēšana palīdz savlaicīgi uzzināt, ka atmiņas izmete nav derīga analīzei un pārtraukt algoritma darbību. Nākamā pārbaude ir veikta pēc izpildāmās datnes nolasīšanas. Ir iespējams, ka izpildāmā datne neatbilst 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 jāpārlicinās, ka var piekļūt galvenās arēnas datiem. 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 ar 3 analizatoru piemēriem (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 par problēmām.

Pieejā tiek izstrādāta ievērojot sekojošus ierobežojumus:

- GNU C bibliotēkas izmantošana sākot ar 2.3 versiju;
- ELF atmiņas izmetes formāts;
- analizatori nedod pilnīgu secinājumu par problēmas esamību, bet sniedz informāciju par sistēmas stāvokli, kura ļauj izstradātājam secināt par problēmu.

4.2. Atmiņas noplūdes analizators

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. Skriptā ir nodefinēta komanda **analyze**, kura izsauc parējās komandas priekš abiem analizatoriem. Šai komandai no gdb atklūdotāja 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 4.2. 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

```

4.2. att. Gdb skripta palaišana

Šeit tiek aprakstīts maksimālās atmiņas izmantošanas problēmas analizators (sk. 4. 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. 4.3. attēlu) blakus izrēķinātiem rādītājiem, tiek norādīta atmiņas gabala sākuma un beigu adreses. Ja ir nepieciešams, tad var izdrukāt sīkako kaudzes sadalījumu, norādot atbilstošo argumentu **analyze** komandai.

```

1 ----- Maksimālā atmiņas izmantošanas problēma -----
2 Atbrīvoto un iedalīto gabalu attiecība:
3 Apgabals 0x8279000 - 0x82b5238 35%
4 Apgabals 0x82b5238 - 0x82f32e0 31%
5 Apgabals 0x82f32e0 - 0x8333388 35%
6 Apgabals 0x8333388 - 0x8371430 31%
7 Apgabals 0x8371430 - 0x83a54b8 36%

```

4.3. att. Maksimālā atmiņas izmantošanas rādītājs

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 galvenajā arēnā. Kaudzes izmērs ir uzglabāts 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.

4.4. Fragmentēšanas analizators

Šajā sadaļā tiek aprakstīts fragmentēšanas analizators, kurš tika realizēts gdb skriptā (sk. 4. pielikumu). Fragmentēšanas problēmas novērtēšanai analizators izvada 2 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. 4.4. attēlu), ja programma pieprasīs vairāk nekā 8 kilobaitus un atmiņa no operētājsistēmas netiks iedalīta, tad tas varētu izraisīt sistēmas apstāšanos.

```
1 Bin numurs 1: 50 gabals(-li) (8196 - 8196 baiti), kopumā = 409800 baiti
2
3 ----- Fragmentēšana -----
4 Lielākā gabala izmērs: 8196 baiti (8 KiB, 0 MiB),
5 Kopējā atbrīvotā atmiņa bin sarakstos: 409800 baiti (400 KiB, 0 MiB),
```

4.4. att. Fragmentēšanas rādītāji

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 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. 4.4. attēlu). Tas palīdz iegūt detalizētu statistiku par visiem parastajiem sarakstiem, kuri nav tukši. Tas ir: skaits cik ir atbrīvoto gabalu sarakstā, amplitūda (mazākais gabals, lielākais gabals), kopējais gabalu izmērs sarakstā.

4.5. Secinājumi

- 1.

GALVENIE REZULTĀTI UN SECINĀJUMI

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 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. O’Reilly & 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 apskātīts 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 apskātīts 21-Mai-2014].

PIELIKUMI

1. pielikums
Atmiņas noplūde

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <memory.h>
5
6  int main(int argc, char** argv)
7  {
8      int num_allocations = atoi(argv[1]);
9      int alloc_size      = atoi(argv[2]);
10     printf ("Tiks iedalīti %i gabali, %i KiB katrs, %i KiB kopumā.\n",
11            num_allocations, alloc_size, num_allocations*alloc_size);
12
13     char** arr = new char* [num_allocations];
14     for (int i=0; i<num_allocations; i++)
15         arr[i] = new char[alloc_size*1024];
16     printf ("Iedalīšana notika.\n");
17
18     for (int i=0; i<num_allocations; i++)
19         memset (arr[i], 0, alloc_size*1024);
20     printf ("Atmiņa ir aizpildīta.\n");
21
22     for (int i=0; i<num_allocations; i++)
23         arr[i] = NULL;
24
25     printf ("%i KiB pazaudēti.\n",
26            num_allocations*alloc_size);
27
28     abort();
29     delete[] arr;
30 }
```

2. pielikums
Maksimālās atmiņas izmantošanas problēma

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <memory.h>
5
6 int main(int argc, char** argv)
7 {
8     int num_allocations = atoi(argv[1]);
9     int alloc_size      = atoi(argv[2]);
10    printf ("Tiks iedalīti %i gabali, %i KiB katrs, %i KiB kopumā.\n",
11           num_allocations, alloc_size, num_allocations*alloc_size);
12
13    char** arr = new char* [num_allocations];
14    for (int i=0; i<num_allocations; i++)
15        arr[i] = new char[alloc_size*1024];
16    printf ("Iedalīšana notika.\n");
17
18    for (int i=0; i<num_allocations; i++)
19        memset (arr[i], 7, alloc_size*1024);
20    printf ("Atmiņa ir aizpildīta.\n");
21
22    for (int i=0; i<num_allocations-1; i++)
23        delete[] arr[i];
24    printf ("Atmiņa ir atbrīvota izņemot pēdējo gabalu.\n");
25
26    abort();
27    delete[] arr;
28 }
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <memory.h>
5
6  int main(int argc, char** argv)
7  {
8      int num_allocations = atoi(argv[1]);
9      int alloc_size      = atoi(argv[2]);
10     printf ("Tiks iedalīti %i gabali, %i baitu katrs, %i baitu kopumā.\n",
11            num_allocations, alloc_size, num_allocations*alloc_size);
12
13     char** arr = new char* [num_allocations];
14     for (int i=0; i<num_allocations; i++) {
15         if (i%2 == 0) {
16             arr[i] = new char[alloc_size];
17             memset (arr[i], 0, alloc_size);
18         } else {
19             arr[i] = new char[alloc_size+16];
20             memset (arr[i], 0, alloc_size+16);
21         }
22     }
23     printf ("Iedalīšana notika.\n");
24     printf ("Atmiņa ir aizpildīta.\n");
25
26     for (int i=0; i<num_allocations; i = i+2)
27         delete[] arr[i];
28     printf ("Atmiņa uz kuru norāda katra otra norāde ir atbrīvota.\n");
29
30     abort();
31     delete[] arr;
32 }
```

4. pielikums

Gdb skripts atmiņas noplūdes atklādošana

```
1  #####
2  # Autore: Renata Januškeviča
3  # 29.05.2014
4  # Lai palaistu skriptu, ir nepieciešams:
5  # 1) ielādēt skriptu gdb atklūdotājā 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 $top_address = (long *) ($arg0 + 48)
17     set $system_mem = (long *) ($arg0 + 1096)
18     set $top_size_address = (long *) ($top_address[0] + 4)
19     set $top_chunk_size = $top_size_address[0] & ~7
20     set $heap_pointer = (long *) ($top_address[0] - $system_mem[0] + $top_chunk_size)
21     # $malloc_pointer ir norāde, kura tiek atgriezta programmai
22     set $malloc_pointer = 0
23     # $ skaitītājs atmiņas gabaliem bez norādēm
24     set $unref = 0
25
26     while ($heap_pointer != $top_address[0])
27         # nākamā atmiņas gabalā atrodas kontroles zīmes, p zīme palīdz noteikt vai ←
            iepriekšējais gabals tiek iedalīts programmai
28         set $next_chunk = $heap_pointer + (($heap_pointer[1] & ~7)/4)
29         # $x mainīgajā tiek saglabāta atrasta simbolu virkne, kas atrasta heksadecimalajā ←
            failā
30         set $x = 0
31         if (($next_chunk[1] & 1) == 1)
32             set $malloc_pointer = $heap_pointer + 2
33
34         # notiek meklēšana
```



```

35     eval "shell cat gdb.core | grep %x > gdb.log", $addr
36     shell echo set \$x=\$(cat gdb.log)\> > gdb.log
37     source gdb.log
38
39     if (sizeof($x) == 1)
40         set $unref = $unref + 1
41         printf "Nav atrasta norāde uz %x adresi.\n", $malloc_pointer
42     end
43     # komanda izdzēs .log datnes saturu
44     shell > gdb.log
45 end
46 # kārtējais gabals ir apstrādāts, ir nepieciešams pārvietot kaudzes norādi
47 set $heap_pointer = $heap_pointer + (($heap_pointer[1] & ~7)/4)
48 end
49 shell rm gdb.log
50 printf "Procesa adrešu telpā nav atrastas %i norādes.", $unref
51 end
52
53 # $arg0: galvenās arēnas adrese
54 # komanda saglabā atmiņas izmeti heksadecimālā formātā un izdzēs atstarpes, jo dati ↵
55     var nebūt izklidzināti
56 define analyze
57     shell od -t x core > gdb_tmp.core
58     shell cat gdb_tmp.core | tr -d ' ' > gdb.core
59     shell rm gdb_tmp.core
60     get_alloc_chunk $arg0
61 end

```

Gdb skripts fragmentēšanas un maksimālās atmiņas izmantošanas problēmas atklādošanai

```

1  #####
2  # Autore: Renata Januškeviča
3  # 23.05.2014
4  # Lai palaistu skriptu, ir nepieciešams:
5  # 1) ielādēt skriptu gdb atklūdotājā ar "source <script.gdb>" komandu,
6  # 2) izsaukt lietotāja definēto komandu analyze ar 2 argumentiem,
7  # $arg0: galvenās arēnas adrese, kuru iespējams iegūt ar komandas "p &main_arena" ←
    palīdzību;
8  # $arg1: skaitlis, kurš norāda cik sīki tiks sadalīta kaudze.
9  # Piemēram: "analyze 0x845e000 5"
10 #####
11
12 # $arg0: galvenās arēnas adrese
13 # $arg1: bin saraksta numurs
14 # komanda savāc datus, kuri nepieciešami statistikai, druka informāciju par sarakstiem
15 define free_chunk_list
16     # bin kārtēja saraksta sākuma adrese
17     set $start_bin = (long *) ($arg0 + 56 + $arg1 * 8)
18     # malloc() funkcija atgriež šo adresi programmai
19     set $free_chunk = (long *) ($start_bin[1] + 8)
20     set $chunk_count = 0
21     set $chunk_max_size = 0
22     set $total_size = 0
23
24     while ($free_chunk != $start_bin)
25         # pēdējie 3 biti netiek izmantoti izmēra glabāšanai
26         set $chunk_size = ($free_chunk[-1] & ~7)
27
28         if ($chunk_count == 0)
29             set $chunk_max_size = $chunk_size
30             set $chunk_min_size = $chunk_size
31         else
32             if ($chunk_min_size > $chunk_size)
33                 set $chunk_min_size = $chunk_size
34             end
35             if ($chunk_max_size < $chunk_size)
36                 set $chunk_max_size = $chunk_size
37             end

```

```

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

```

```

75 define div_stat
76     set $fract_size = $used_and_freed/$arg1
77     set $heap_pointer = (long *) ($top_address[0] - $system_mem[0] + $top_chunk_size)
78     set $chunk_size = 0
79
80     printf "\n----- Maksimālā atmiņas izmantošanas problēma ----- \n"
81     printf "Atbrīvoto un iedalīto gabalu attiecība: \n"
82
83     while ($heap_pointer != $top_address[0])
84         set $free_size = 0
85         set $alloc_size = 0
86         set $fract_fin = $heap_pointer + $fract_size/4
87         set $fract_start = $heap_pointer
88         while (($heap_pointer < $fract_fin) && ($heap_pointer != $top_address[0]))
89             set $next_chunk = $heap_pointer + (($heap_pointer[1] & ~7)/4)
90             set $chunk_size = ($heap_pointer[1] & ~7)
91             if (($next_chunk[1] & 1) == 1)
92                 set $alloc_size = $alloc_size + $chunk_size
93             else
94                 set $free_size = $free_size + $chunk_size
95             end
96             set $heap_pointer = $next_chunk
97         end
98
99         set $relation = ((double) $free_size/($alloc_size + $free_size)) * 100
100         printf "Apgabals 0x%x - 0x%x %i%% \n", $fract_start, $heap_pointer, $relation
101
102     end
103 end
104
105 # $arg0: galvenās arēnas adrese
106 # $arg1: kaudzes sadalījums apgabalos
107 define analyze
108     set $free_in_arena = 0
109     set $bin_number = 0
110     set $biggest_free_size = 0
111     set $count_in_arena = 0
112     set $top_address = (long *) ($arg0 + 48)
113
114     if ($top_address[0] != 0)
115         # Lai savāktu statistiku, ir nepieciešams apstaigāt visus 128 bin sarakstus
116         while ($bin_number < 127)
117             free_chunk_list $arg0 $bin_number

```

```

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

```

Bakalaura darbs „Atmiņas izmetes pielietošana kaudzes atklūdošanas 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): _____