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 33. lappuses, 20 attēli, 3 tabulas pamattekstā un 27 nosaukumi literatūras sarakstā.

Atslēgvārdi:

ABSTRACT

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

The work consists of introduction, 6 chapters, conclusions and 3 appendixes. It contains 33. pages, 20 figures, 3 tables and 27 references.

Keywords:

SATURS

APZĪMĒJUMU SARAKSTS

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.

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

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

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

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

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

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

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

Chunk - Gabals - nepārtraukts atmiņas gabals ar noteikto struktūru.

ptmalloc2 - atvērtā pirmkoda programmatūra, kura nodrošina lietotāja līmeņa atmiņas iedalīšanu. 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 iedalīšanu, 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.

IEVADS

1. JĒDZIENI, UZ KURIEM BALSTĪTA METODE

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

1.1. Atmiņas izmete

Sistēmās, kuras atbalsta POSIX standartus, ir signāli [20], 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 [14] ir lauks ar vērtību core, kas atrodas ailē ar nosaukumu darbība (Action). Uzģenerētā atmiņas izmete iekļauj sevī procesa atmiņas attēlojumu uz procesa pārtraukšanas brīdi, piemēram, CPU reģistrus un steka vērtības katram pavedienam, globālos un statiskos mainīgos. Atmiņas izmeti var ielādēt atkļūdotājā, tāda kā gdb, lai apskatītu programmas stāvokli uz brīdi, kad atnāca operētājsistēmas signāls [13]. 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. nodaļā. Bet eksistē vairākas iespējas kā atmiņas izmeti var uzģenerēt patstāvīgi. Tas varētu būt nepieciešams programmas atkļūdošanai. Atmiņās izmeti var uzģenerēt no programmas koda, gdb atkļūdotāja vai komandrindas interpretatora. Turpmāk katra no iespējam tiks uzskatāmi nodemonstrēta un 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    raise(SIGSEGV); /* Signal for Invalid memory reference */
5
6    return 0;
7 }
```

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 vēršanās pēc null 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(); /* Child */
    }
    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 [5].

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

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

1.1.3. Atmiņas izmetes ģenerēšana no komandrindas interpretatora

Trešā iespēja ir nosūtīt signālu, izmantojot komandrindas interpretatoru. Komanda kill var nosūtīt jebkuru signālu procesam. Pēc komandas `kill -<SIGNAL_NUMBER> <PID>`, signāls ar numuru SIGNAL_NUMBER tiks nosūtīts procesam ar norādītu PID vērtību. Izmantojot shell komandrindas interpretatoru ir iespējams izmantot īsinājumtaustiņus signālu nosūtīšanai. Nospiežot Control + \ tiks nosūtīts SIGQUIT signāls procesam, kas pašreiz ir palaists (sk. 1.4. attēlu) [21]. Šajā piemēra ziņojumu - Quit (core dumped), izdruka shell. Šīs komandrindas interpretators noteic, ka sleep procesu (shell bērnu) pārtrauca SIGQUIT signāls. Pēc šī signāla nosūtīšanās, darba mapē tiek uzģenerēta atmiņas izmete.

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

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

1.1.4. Atmiņas izmetes ģenerēšanas nosacījumi

Lai uzģenerētu atmiņas izmeti ir jābūt izpildītiem sekojošiem nosacījumiem [21]:

- 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 jāsatur brīvie indeksa deskriptori,
- bināro datni jāizpilda lietotājam, kurš ir datnes īpašnieks (group owner).

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

1.2. Atkļūdošana, izmantojot atmiņas izmeti

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

iejaukšanās un tiešas piekļuves sistēmai. Daudzos gadījumos, atmiņas izmete ir speciāli uzģenerēta datne, kura palīdz iegūt atmiņas stāvokli uz signāla nosūtīšanas 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, utilītprogramma vai atkļūdotājs) atbalstītu uzģenerētās datnes formātu. GNU gdb ir Linux standarta atkļūdotājs [23], kurš ir plaši pielietojams atmiņas izmešu analīzei. Turpmāk tiek apskatīta atmiņas izmetes analīze ar gdb atkļūdotāja palīdzību.

1.2.1. 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 priekš procesora arhitektūras, no kuras bija iegūta atmiņas izmete. To var identificēt uzreiz pēc gdb palaišanas, ar sekojošās rindiņas palīdzību: `This GDB was configured as i686-linux-gnu`. Lai atmiņas izmete saturētu atkļūdošanas informāciju, ir jānorāda -g opcija kompilācijas laikā. Atkļūdošanas informācija ir uzglabāta objektu datnē un saglabā atbilstību starp izpildāmo datni un pirmkodu, ka arī uzglabā mainīgo un funkciju datu tipus. Ja atmiņas izmete neiekļauj atkļūdošanas 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 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. Izpildamām 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 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, kau-

dze[1], 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 izdruku, 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 [27]. 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) [27]. 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. Bactrace 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 cēloņi. Backrtace nesniedz patieso informāciju par funkciju, ja process tika pabeigts ārējo apstākļu dēļ, nevis tāpēc, ka bija notikusi kļūda programmā. Katra rindiņa satur rāmi (frame). Bactrace izdruka sākas ar rāmi, kurā iekļauta funkcija, kura bija izpildīta pēdēja. Nākamais rāmis iekļauj funkciju, kas izsauca iepriekšējā rāmī iekļauto funkciju. Katrai baktrace rindiņai tiek piešķirts rāmja numurs. Katrs rāmis var iekļaut: funkcijas nosaukumu, pirmkoda datnes nosaukumu, pirmkodam atbilstošo rindiņas numuru un funkcijas argumentus. Bactrace var tikt iegūts izmantojot gdb komandu `bactrace full` vai `bt f`. Pēc noklusējuma, daudzpavedienu lietotnēs gdb rāda bactrace kārtējām pavedienam, bet pastāv iespēja iegūt arī bactrace

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

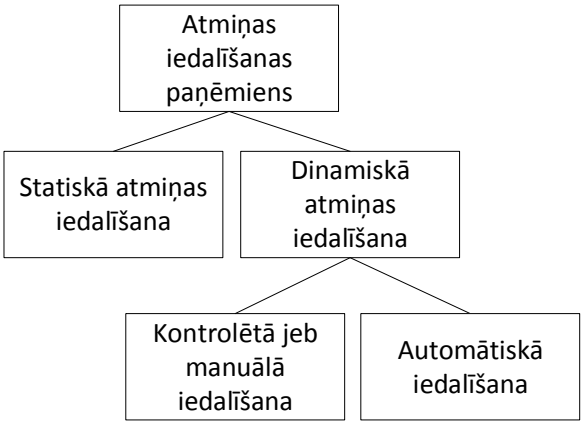
izdruku priekš citiem pavedieniem. Ja programma bija nokompilēta ar optimizācijas opciju, tad bactrace varētu neiekļaut funkcijas argumentus. Šajā gadījumā funkciju argumenti varētu tikt nodoti caur CPU reģistriem, kuru vērtības ir iespējams iegūt, izmantojot komandu `info registers` vai `i r`. Atmiņas 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

Šajā nodaļā ir aprakstīti atmiņas iedalīšanas paņēmieni, ir dots īss ieskāts atmiņas organizācijā un aprakstīta atmiņas pārvaldība, kuru var veikt kodols vai lietotājs. Nodaļā ir iekļauta informācija, kura palīdz saprast kopējo atmiņas organizāciju un tās saistību ar atmiņas pieprasīšanu programmā.

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**

Statiskā atmiņas iedalīšana

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

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 [15].

Automātiskā iedalīšana notiek, kad sākās programmas funkcijas izpilde. Priekš automātiskās atmiņas iedalīšanas, izmato 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` [22]. Steka pārpildīšana var notikt dažādu iemeslu dēļ, piemēram to var izraisīt dziļa rekursija.

Kontrolētā atmiņas iedalīšana nozīme, ka programma var izvēlēties patvaļīgus, brīvus atmiņas apgabalus priekš programmas datiem. Kontrolētā jeb manuālā atmiņas iedalīšana tiek nodrošināta ar atmiņas arēnas un kaudzes palīdzību. Šeit nav iespējams piekļūt datiem izmantojot vienu norādi un tās nobīdi. Tagad katram iedalītam atmiņas apgabalam var piekļūt tikai tad, ja ir norāde uz šo iedalīto atmiņas apgabalu. Gadījumos, kad norādes nav, tad adreses vairāk nav sasniedzamas un kļūst pazaudētas. Turpmāk darbā, termins dinamiskā atmiņas iedalīšana apzīmēs kontrolēto atmiņas iedalīšanu kaudzē.

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 pārvalda visus atmiņas pieprasījumus, kas attiecās uz programmu vai programmas instancēm. Kad lietotājs sāk programmas izpildi, tad kodols iedala atmiņas apgabalu tekošai programmai. Pēc tam process pārvalda iedalīto apgabalu, sadalot to vairākos segmentos [17]:

- Teksts - šeit tiek uzglabāti dati, kuri tiek izmantoti tikai lasīšanai. Tās ir nokompilētas koda instrukcijas. Vairākas programmas instances var izmantot šo atmiņas apgabalu.
- Statiskie dati - apgabals, kurā tiek uzglabāti dati ar iepriekš zināmu izmēru. Tās ir globālie un statiskie mainīgie. Operētājsistēma iedala šī apgabala kopiju katrai programmas instancei atsevišķi.
- Kaudze - apgabals, no kura dinamiski tiek iedalīta atmiņa. Kaudzē atrodas dinamiski iedalītā un atbrīvota atmiņa. Kaudzes saturs parasti ir sadalīts sīkāk, mazākos atmiņās

gabalos. Kaudze aug no mazākas adreses līdz lielākai. Lai palielinātu kaudzes segmenta izmēru, tiek veikts brk() sistēmas izsaukums. Izsaukums uzstāda kaudzes segmenta jauno beigu robežu. Jā process nepārsniedz savu limitu, tad izsaukums atgriež 0 un kaudzes segmenta lielums tiek veiksmīgi izmainīts [7].

- Steks - 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 adrešu intervālu stekam un kaudzei var atrast `/proc/<pid>/maps` datnē.

2.2.2. Lietotāja atmiņas pārvaldība

Lietotāja atmiņas pārvaldība ir dinamiski iedalītās atmiņās pārvaldība, kura realizēta veicot sistēmas izsaukumus un pārvaldot iegūto atmiņu, sadalot to sīkākos gabalos. Pārvaldīt nozīme:

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

Lietotāja atmiņas pārvaldība ļauj efektīvāk pārvaldīt atmiņu, nekā tas būtu nodrošināts, katru reizi pieprasot atmiņas apgabalu ar sistēmas izsaukumiem. Lietotāja atmiņas pārvaldība varētu būt realizēta, izmantojot dažādus atmiņas iedalītājus (allocator), piemēram, Hoard memory allocator, ptmalloc2, dlmalloc. Dažreiz speciāli šīm nolūkam 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 veiktspēju [16].

```
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++**

Turpmāk tiks apskatīta lietotāja atmiņas pārvaldība, izmantojot GNU C bibliotēkas funkciju palīdzību. Lietotājam iedalīta atmiņa atrodas kaudzē, kuru var nosaukt par atmiņas arēnas sastāvdaļu. C valodā atmiņas arēna tiek pārvaldīta ar malloc(), realloc(), free() un calloc() funkciju palīdzību [17]. 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 apgabalu 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 [10].

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

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

2.3. Atmiņas iedalīšana glibc bibliotēkā

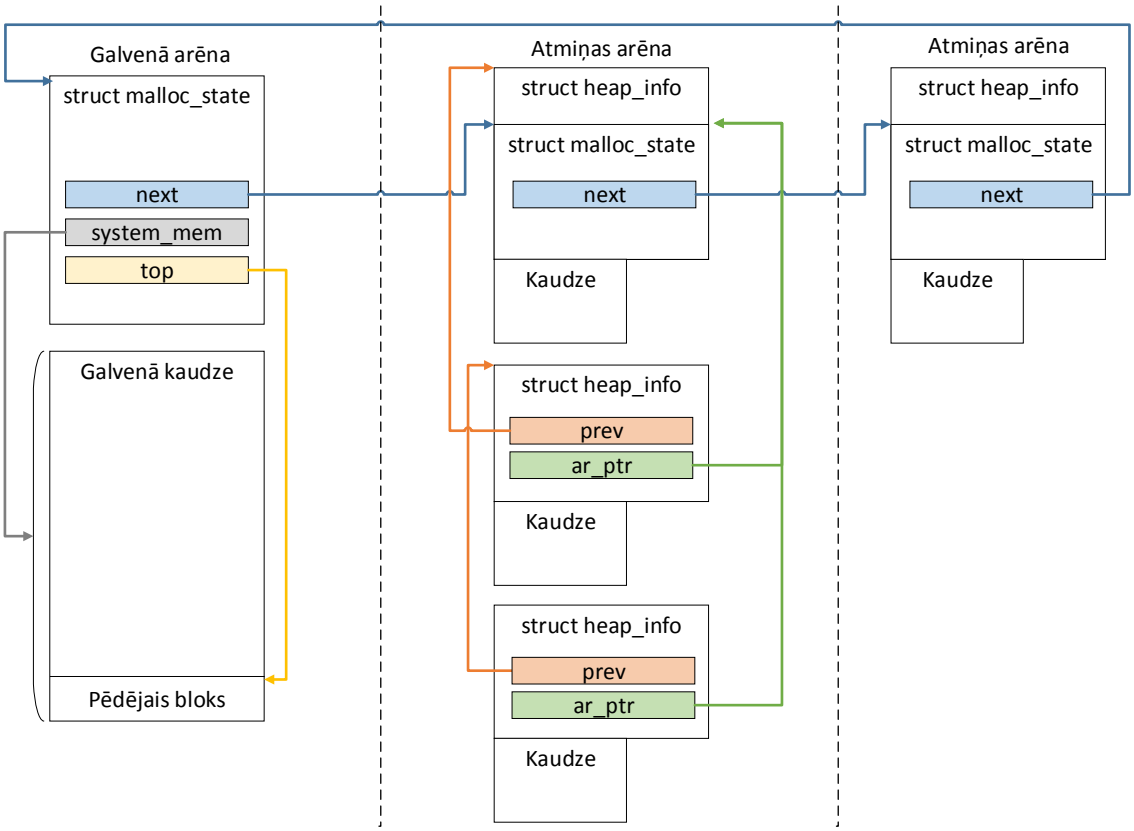
Darbā tiks aplūkota GNU C bibliotēkas (versija 2.3) ptmalloc2 realizācija, kuru izstrādāja Wolfram Gloger, balstoties uz Doug Lea dlmalloc realizāciju. Atmiņas iedalīšana sākas ar malloc() vai līdzīgo funkciju izsaukumiem no programmas koda un tiek nodrošināta ar GNU C bibliotēkas palīdzību.

2.3.1. Atmiņas arēna

Atmiņas arēnu var nosaukt par loģisko atmiņas kolekciju. Attēlā[1] 2.4. ir parādītas 3 arēnas, kuras ir atdalītas savā starpā ar raustītam līnijām. Atmiņas arēnu vienkāršotā veidā

[1]Attēla izveidošanai tika izmantots GNU C malloc pirmkods [8] un vietnē nopublicēta shēma [1]. Attēls demonstrē atmiņas organizāciju.

var attēlot kā viensaišu saistīto sarakstu, kurš sastāv no vienas vai vairākiem atmiņas kaudzēm. Kaudze ir lineārās apgabals, kurš iekļauj sevī iedalītus vai 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. **Arēnas 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 arēna 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 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 apgabala 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, piemēram kā tas notiek ptmalloc2 realizācijā, tad vienas arēnas bloķēšana neietekmē atmiņas iedalīšanu parējās arēnās 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 mainīga 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 sekojošo algoritmu:

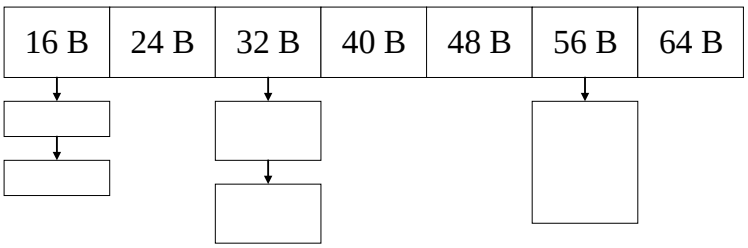
1. malloc() izsaukums vēršas pie arēnas, kurai piekļuva iepriekšējo reizi,
2. ja arēna ir nobloķēta, tad malloc() vēršas pie nākamas izveidotās arēnas,
3. ja nav piekļuves nevienai arēnai, tad tiek izveidota jauna arēna un malloc() vēršas pie tās.

Vispirms atmiņas iedalīšana sākas no galvenās arēnas (main arena). GNU C bibliotēkā ir globāls `malloc_state` objekts - galvenā arēna, kura atšķiras no pārējām arēnām ar to, kā tā tiek paplašināta, izmantojot brk() nevis mmap() sistēmas izsaukumu. Līdz ar galvēnās arēnas paplašināšanu, tiek paplašināts arī procesa arēnas 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. apakšnodaļā), 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. 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 [8]. 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 fragmentācijas iespējamību, programma, kad pieprasa vai atbrīvo lielus atmiņas gabalus var sapludināt atmiņas gabalus, kuri atrodas fastbin sarakstā. Piekļuve tādiem atmiņas gabaliem ir ātrāka nekā piekļuve parastiem gabaliem. Fastbin saraksta elementi ir apstrādāti pēdējais iekšā pirmais āra (jeb LIFO) kārtībā [2]. Kad tiek pieprasīta atmiņa no fastbin saraksta, tad jebkurš atmiņas gabals tiek atgriezts konstantā laikā [4].



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 [2]. Tāda veidā vienmēr tiek atgriezts gabals, kurš ir vislabāk piemērots. Tas ir gabalam ir mazākais 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

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 sarakstam. Ātrie un parastie gabali, tiek izmantoti, lai nodrošinātu atmiņas iedalīšanu no kaudzes segmenta. 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
Atmiņās gabalu struktūras elementu apraksts

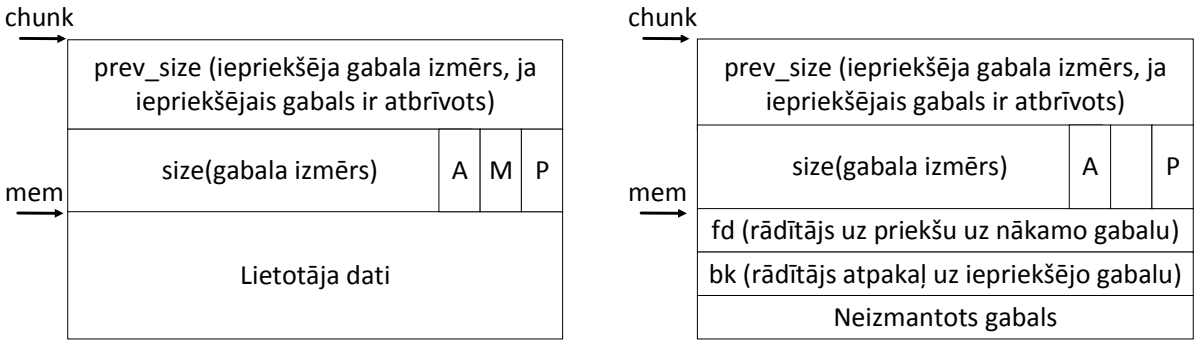
Elements	Nozīme
INTERNAL_SIZE_T prev_size	Iepriekšēja gabala izmērs (baitos), ja tas bija atbrīvots
INTERNAL_SIZE_T size	Kārtējā gabala izmērs (baitos)
struct malloc_chunk* fd	Rādītājs uz nākamo atbrīvoto gabalu, ja kārtējais gabals ir atbrīvots un pievienots dubultsaišu bin sarakstām
struct malloc_chunk*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

NAL_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. Šīs skaitlis ir vienāds ar 8 baitu izlīdzinājumu, ja makrodefinīcijas INTERNAL_SIZE_T vērtība ir vienāda ar 4 baitiem [8].



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

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

No kreisās pusēs attēlots (sk. 2.7. attēlu) [3] atmiņas gabals, kurš bija iedalīts procesam, no labās, tās, kurš bija atbrīvots. Abos gadījumos rādītājs chunk attēlo atmiņas gabalu sākumu. Pēc šī radītāja var iegūt iepriekšēja gabala izmēru, ja iepriekšējais gabals bija atbrīvots. Gadījumā, kad iepriekšējais gabals ir iedalīts, tad chunk uzglabā daļu no lietotāja datiem, kas dublēti no iepriekšējā gabala. 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 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 atmiņās 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 chunk vienmēr eksistē [11].

Last_remainder ir vel viens atmiņas gabals ar īpašu nozīmi. Tas ir izmantots gadījumos, kad ir pieprasīts mazs atmiņas gabals, kas neatbilst nevienam bin saraksta elementam. Last_remainder ir dalījuma atlikums, kurš izveidojās pēc lielāka gabala sadalīšanas, lai apmierinātu pieprasījumu pēc maza gabala [11].

3. PROBLĒMU APRAKSTS

3.1. Atmiņas noplūde

Atmiņas noplūde (memory leak) ir viena no bieži sastopamām problēmām C un C++ valodās [17]. 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) [26], jo atmiņa ir joprojām sasniedzama no programmas, bet nekad netiek atbrīvota.

Abos gadījumos sekas ir vienādas. Sākumā tiks novērota pakāpeniskā 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 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.

Atmiņās 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, kurā izmērs ir 14 baiti. Tā kā atmiņas adreses kļūst pazaudētas un nav iespējas piekļūt iepriekšējiem elementiem pēc tam

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

int main() {
    string *str;

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

    return 0;
}
```

3.1. att. **Atmiņas noplūde, C++**

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 atmiņās noplūdi [9]:

- 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 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 ka: 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 [12].

3.1.1. Atmiņas noplūdes pazīmes

Reālajās sistēmās problēma var izpausties uzreiz pēc palaišanas, bet var kļūt novērojama tikai pēc dažiem gadiem. Abi gadījumi ir izplatīti [6]. Tā kā atmiņas noplūdes rezultātā atmiņa tiek pazaudēta, tad var periodiski novērot procesa atmiņas patēriņa pieaugumu. Pazīme, kas varētu liecināt par atmiņas noplūdi ir pārmērīgs[1] 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. Tā kā atmiņas izmete satur procesa atmiņas attēlojumu uz procesa pārtraukšanas brīdi, tad uzģenerētās datnes izmērs, atmiņas noplūdes problēmas ietekmēs rezultātā, var sasniegt vairākus gigabaitus.

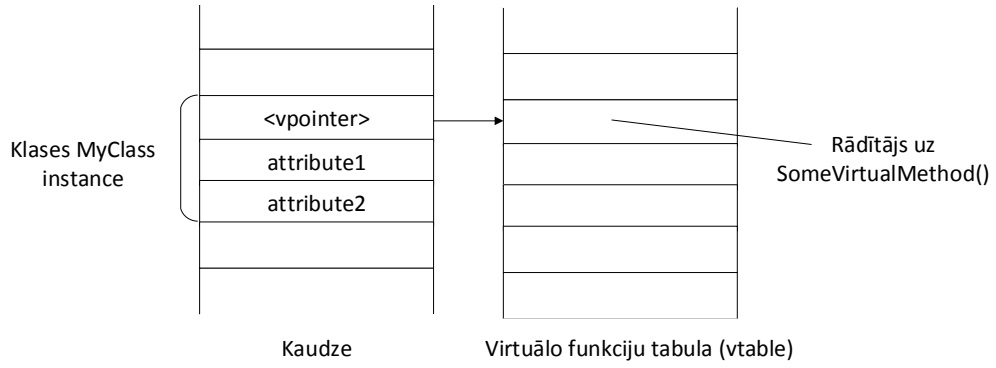
Par fiziskās atmiņas noplūdes pazīmi var uzskatīt stāvokli, kad uz atmiņas gabaliem kaudzē nav norāžu no procesa adrešu telpas. Par šo programmas stāvokli var pārliecināties, veicot atmiņas izmetes analīzi. Atmiņas izmetē atrodas kaudzes saturs visām atmiņas arēnām. Interpretējot katru kaudzes saturu, kā kopu ar daudziem atmiņas gabaliem, var iegūt adreses, uz kuriem 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.

Loģiskās atmiņas noplūde rezultātā visiem atmiņas gabaliem atbilst norādes procesa adrešu telpā. Problēmai ir raksturīgs stāvoklis, kad ir daudzi 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). Šīs 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. Ins-

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

trukcijas, kas ļauj apskatīties, kuram apgabalam pieder adrese jau tika aprakstītas sadaļā 1.2.1.

```
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**

3.2. Maksimālās atmiņas izmantošanas problēma

Atmiņas daudzums, kas tiek izmantots programmas izpildes laikā var visu laiku mainīties. Pētījumā [24] tiek apkopoti trīs svarīgākas atmiņas izmantošanas shēmas: traps (ramps), maksimums (peaks), plato (plateaus). Citas atmiņas izmantošanas shēmas ir iespējamas, bet izpaužas ļoti reti. Ne visām programmām ir raksturīgas visās trīs shēmas, bet vairākumam 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 atbrī-

vota. Grafiks šai shēmai izskatās kā lauztā līnija un atmiņas patēriņš var svārstīties dramatiski.

- Plato. Novērojama, kad programmas ātri uzbūve datu struktūras un izmanto tās ilgā laika periodā, bieži izmanto līdz programmas izpildes beigām.

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. Problēma ir novērojama, kad liels atmiņas daudzums netiek atgriezts operētājsistēmai pēc izmantošanas, pat tad, ja atmiņa tiek 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. Izmantojot brk() sistēmas izsaukumu, kamēr netiks atbrīvots atmiņas gabals, kas atrodas beigās, atmiņa netiks atgriezta operētājsistēmai.

Turpmāk tiks apskatīts piemērs, kurš demonstrē doto problēmu. 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 atmiņās 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 pēdejo gabalu	13360	10640
5. Atmiņa tiek pilnībā atbrīvota	3360	968

3.2.1. Maksimālās atmiņas izmantošanas problēmas pazīmes

Pazīme, kas varētu liecināt par problēmas esamību ir pārmērīgs atmiņas patēriņš pēc trapa virsotnes vai maksimālā punkta sasniegšanas.

Atmiņas izmetē pazīme, kas varētu liecināt par problēmu, ir lielā izmēra atmiņas gabali, kuri atrodas bin sarakstos. 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 atmiņas gabali tiks uzglabāti parastajos sarakstos.

Tālāk ir aprakstīts piemērs, kurš parāda, ka problēmas pazīme izpaužas atmiņas izmetē. Sākumā tika iedalīti 100 atmiņas gabali, katrs 100 kilobaitu izmēra. Katrā baitā bija ierakstīta vērtība 7. Pēc tam bija atbrīvoti 99 gabali izņemot pēdējo. Beigās, lai apskatīties pazīmi, tika uzģenerēta atmiņas izmete. Saskaņā ar pētījumā minētām shēmām [24] bija sasniegts maksimums (peaks). Atmiņas izmete palīdz saprast, kas notiek atmiņā. Ar `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 kopējais izmērs ir vienāds ar izmēru kilobaitos, kas ir sareizināts 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 notiek atmiņas gabalu izlidināšana 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.

Fragmentēšanu mēra procentos (%). Stradajošā sistēmā var būt vairāki veidi kā var mērīt atmiņas fragmentēšanu [18]. Atmiņas izmetē ir iespējams izrēķināt ārējo fragmentēšanu tikai uz procesa partraukšanas brīdī. Tātad ir iespējams izrēkināt 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 atmiņās gabali).

Turpmāk ir apskatīts piemērs, kurš demonstrē iekšējās fragmentēšanas cēloņi. Zinot atmiņas gabala struktūru, var uzrakstīt kodu C valodā, kurš izdrukās atmiņas gabala izmēru, kurš īstenībā tika iedalīts no kaudzes (sk. 3.4. attēlu). 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.

```
1  #include <stdio.h>
2  #include <malloc.h>
3
4  int main () {
5      char * ptrl;
6      int chunk_size;
7
8      ptrl = (char *)malloc(4);
9
10     /* get value of chunk size (the second malloc_chunk element) */
11     chunk_size = *((char *) ptrl - sizeof(size_t));
12     /* the lower 3-bits are used as metadata */
13     chunk_size = chunk_size - (chunk_size & 7);
14
15     printf("size = %d\n", chunk_size);
16     free(ptrl);
17
18     return 0;
19 }
```

3.4. att. **Izmēra noteikšana iedalītām gabalam**

3.3.1. Fragmentēšanas pazīmes

Iekšejai fragmentēšanai nav pazīmju atmiņas izmetē, kuras varētu liecināt par doto problēmu. Lai atpazītu doto problēmu ir nepieciešams zināt cik daudz atmiņas pieprasīja lietotne. Neizmantojot kodu šo informāciju iegūt nevar.

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 un ne vienu reizi netika veikta ātra saraksta saplūdināšana, tad atbrīvotie gabali tiks novietoti ātrajos sarakstos. Ātrajos sarakstos gabals norādīs uz nākamo gabalu un veidos garo sarakstu no visiem atmiņas gabaliem.

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

- Ja ātrie gabali tika saplūdināti vai gabali ir lielāki par 64 baitiem, tad tie tiks novietoti parastajos sarakstos.

Kaudze būs saskaldīta un katrs iedalītais gabals robežos ar mazāko atbrīvoto gabalu. Piemēra ir izdrukāts kaudzes saturs (sk. 3.5. attēlu). Kaudzē atrodas iedalītie gabali 32 baitu (0x21) izmērā un atbrīvotie gabali 16 baitu (0x11) izmērā. Atbrīvotie gabali ir saistīti sava starpā un atrodās ātrajā sarakstā. Tas ir novērojams, jo lietotāju datos ātrajiem gabaliem ir uzglabāta nākama gabala adrese. Pēdejām gabalam ātrajā sarakstā ir uzglabāta 0x00000000 adrese. Dotajam piemēram fragmentēšana ir vienāda ar 50%.

```
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**

3.4. Problēmas glibc bibliotēkā

3.4.1. glibc problēmu pazīmes

4. ATKĻŪŠANAS METODES APRAKSTS

4.1. Metodes pamatprincipi

4.2. Detalizēts metodes apraksts

4.3. Salīdzināšana ar eksistējošām metodēm

5. REALIZĀCIJAS APRAKSTS

5.1. Sistēmas apraksts

5.2. Projektējums

5.3. Iegūtais rezultāts

GALVENIE REZULTĀTI UN SECINĀJUMI

LITERATŪRA

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

[2] Binning. http://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf. [Online; resurss apskatīts 26-Apr-2014].

[3] Chunks of memory. http://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf. [Online; resurss apskatīts 26-Apr-2014].

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

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

[6] 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].

[7] Linux programmer's manual. http://cf.ccmr.cornell.edu/cgi-bin/w3mman2html.cgi?brk(2). [Online; resurss apskatīts 3-Mai-2014].

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

[9] 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].

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

[11] Top chunk, last_remainder. http://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf. [Online; resurss apskatīts 26-Apr-2014].

[12] Using valgrind to detect undened value errors with bit-precision. http://www.cs.columbia.edu/~junfeng/10fa-e6998/papers/memcheck.pdf. [Online; resurss apskatīts 8-Mai-2014].

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

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

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

[16] Emery D. Berger, Benjamin G. Zorn, K. S. M. Composing High-Performance Memory Allocators.

[17] Gene Novark, Emery D. Berger, B. G. Z. Efficiently and Precisely Locating Memory Leaks and Bloat.

[18] Johnstone, M. S., and Wilson, P. R. The Memory Fragmentation Problem: Solved?

[19] Jonas Maebe, Michiel Ronsse, K. D. B. Precise detection of memory leaks.

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

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

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

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

[24] Paul R, Wilson Mark S, J. M. N., and Boles, D. Dynamic Storage Allocation, A Survey and Critical Review.

[25] Randell, B. A Note on Storage Fragmentation and Program Segmentation.

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

[27] Richard Stallman, Roland Pesch, S. S. *Debugging with gdb, Ninth Edition, for GDB version 6.8.50.20090216*. Free Software Foundation, 2009, pp. 89–90.

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: docents Dr.poniz. Jālis Bērziņš

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 sekretār___: _____