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 26. lappuses, 16 attēli, 2 tabulas pamattekstā un 21 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 26. pages, 16 figures, 2 tables and 21 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, kura nodrošina dinamiski iedalītās atmiņas glabāšanu.

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.

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 [16], 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 [12] 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 [11]. Veicot atmiņas izmetes analīzi, kļūst iespējams atrast un izlabot kļūdas, pat tad, ja nav piekļuves sistēmai.

Reālajās sistēmās atmiņas izmetes tiek uzģenerētas atmiņas kļūdu dēļ. Dažas no kļūdām sīkāk ir aprakstītas 3. nodaļā. Taču tas nav vienīgais veids, ka var iegūt atmiņas izmeti, eksistē vairākas iespējas kā to var uzģenerēt patstāvīgi. To var izdarīt 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 darbu 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. Tādas kļūdas kā dalīšana ar nulli nosūta SIGFPE signālu, bet vēršanās pēc rādītāja ar null vērtību - SIGSEGV signālu. Izmantojot funkciju raise(), ir iespējams norādīt atmiņas izmeti izraisošo signālu. Piemērā (sk. 1.1. 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. Funkcijas fork() veiksmīgas izpildes gadījumā, bērnu procesam atgriež 0 vērtību. Pēc abort() funkcijas izpildes, bērns beidz izpildi un uzģenerē atmiņas izmeti. Vecāks process turpina izpildi.

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 and 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) [17]. 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 [17]:

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

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

1.2. Atkļūdošana, izmantojot atmiņas izmeti

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

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

Atmiņas izmete ir ELF 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 izmetēm ir nepieciešams, lai rīks, kurš tika izvēlēts (bibliotēka, utilītprogramma vai atkļūdotājs) atbalstītu uzģenerētā formāta datnes, piemēram, ELF formātu. GNU gdb ir Linux standarta atkļūdotājs [19], kurš ir plaši pielietojams atmiņas izmešu analīzei. Turpmāk tiek apskatīta atmiņas izmetes analīze ar gdb atkļūdotāja piemēru.

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 mainīgo un funkciju datu tipus. Ja atmiņas izmete neiekļauj atkļūdošanas informāciju, tad atmiņas izmete var attēlot 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: izpildes fails un atmiņas izmete. Izpildes 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, kaudze[1], koplietošanas atmiņa. Tā kā segmentu robežas ir zināmas, tad kļūst iespējams izdrukāt

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

atmiņas saturu, kas pieder segmentiem, uzzināt kuram segmentam pieder konkrētā atmiņas adrese un kādas ir segmentu adrešu vērtības.

Lai izdrukātu atmiņas apgabalu ir nepieciešams norādīt atmiņas adresi (`addr`), no kura sākt atmiņas izdruku, formātu (`f`), apgabala lielumu (`n`) un norādīt vienības lielumu (`u`) (sk. 1.7. attēlu). Izmantojot doto piemēru tiks izdrukāts `n` liels atmiņas apgabals, kurš sākas ar adresi `addr`. Formātu un vienības lielumu vajag norādīt saskaņā ar gdb pamācību [21].

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

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

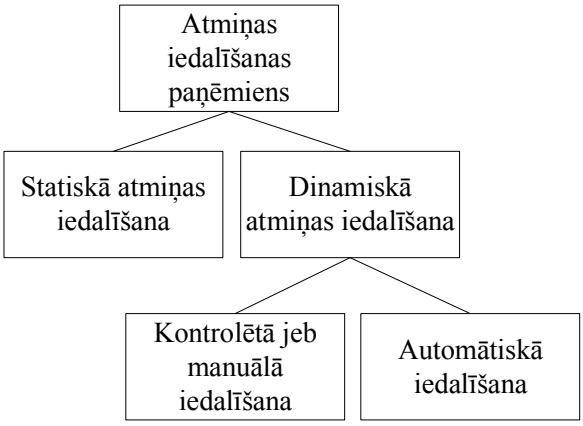
Atmiņas izmetes analīze sākas ar backtrace izdrukāšanu. Bactrace ir pārskats, kurš attēlo kā programma nonāca stāvoklī, kurā pabeidza savu darbību. Tas palīdz ātri atrast instrukciju kura bija izpildīta pēdēja un daudzos gadījumos, tas palīdz ātri identificēt kļūdas cēloni. Katra rindiņa satur rāmi (frame).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

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 divas 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 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 [13].

Automātiskā iedalīšana notiek, kad sākās programmas funkcijas izpilde. Šeit viens un tas pats atmiņas apgabals, kurš bija atbrīvots, var tikt izmantots vairākas reizēs. Piemēram, kad tekošās funkcijas argumenti un lokālie mainīgie ir saglabāti stekā un izdzēsti pēc šīs funkcijas izpildes. Vērtību izdzēšana vai saglabāšana notiek, nobīdot steka norādi. Pēc tam atbrīvotā atmiņa var būt izmantota atkārtoti. Priekš automātiskās atmiņas iedalīšanas, izmato steku. Visiem funkcijas mainīgiem var piekļūt izmantojot steka norādes nobīdi, kas tiek uzglabāta reģistrā, piemēram, Intel x86 procesoros, 16 bitu režīmā reģistrs ir `SP`, 32 bitu režīmā - `ESP` un 64 bitu režīmā - `RSP` [18]. Reģistrs uzglabā adresi, kurā atrodas pēdējā uzglabāta vērtība stekā. Steka pārpildīšana var notikt dažādu iemeslu dēļ, piemēram to var izraisīt dziļa rekursija.

Kontrolētā atmiņas iedalīšana nozīme, ka programma var izvēlēties patvaļīgus, brīvus atmiņas apgabalus priekš programmas datiem. Kontrolētā jeb manuālā atmiņas iedalīšana 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.

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 programma pārvalda iedalīto apgabalu, sadalot to vairākos segmentos [15]:

- Teksts - uzglabāti dati, kuri tiek izmantoti tikai lasīšanai. Tās ir koda instrukcijas. Vairākas programmas instances var izmantot šo atmiņas apgabalu.
- Statiskie dati - apgabals, kurā tiek uzglabāti dati ar iepriekš zināmu izmēru. Tās ir globālie un statiskie mainīgie. Operētājsistēma iedala šī apgabala kopiju priekš katras programmas instances atsevišķi.
- Atmiņas arēna - apgabals, kurā tiek uzglabāta dinamiski iedalītā un atbrīvotā atmiņa. Arēna sastāv no sarakstiem ar atbrīvoto atmiņu un vienas vai vairākām kaudzēm.

Kaudze ir apgabals, kurā atrodas visa dinamiski iedalītā atmiņa programmas izpildei.

- Steks - apgabals, kurā tiek uzglabāts funkciju izsaukumu stāvoklis, katram funkcijas izsaukumam. Steks aug no lielākas adreses uz mazāko. Unikāla atmiņas arēna un steks iedalīti priekš katras programmas instances atsevišķi.

Lai palielinātu atmiņas arēnas izmēru, tiek veikts brk() sistēmas izsaukums. Izsaukums uzstāda atmiņas arēnas segmenta jauno beigu robežu. Jā process nepārsniedz savu limitu, tad izsaukums atgriež 0 un arēnas segmenta lielums tiek veiksmīgi izmainīts, pretējā gadījumā tiek atgriezts -1 [6]. Iedalīto adrešu intervālu stekam un atmiņas arēnai var atrast `/proc/<pid>/maps` datnē.

2.2.2. Lietotāja atmiņas pārvaldība

Lietotāja atmiņas pārvaldība ir realizēta veicot sistēmas izsaukumus un pārdalot iegūtos resursus mazākos gabalos. Tas ļauj efektīvāk pārvaldīt no kodola iedalīto atmiņas apgabalu, 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 raksturīgu uzvedību un tās nodrošinās labāko veiktspēju [14]. Turpmāk tiks apskatīta lietotāja atmiņas pārvaldība, izmantojot GNU C bibliotēkas funkciju palīdzību.

Lietotājam iedalīta atmiņa atrodas kaudzē, kura tiek novietota atmiņas arēnā. Atmiņās arēna C valodā tiek pārvaldīta ar malloc(), realloc(), free() un calloc() funkciju palīdzību [15]. C++ valodā ir izmantots operators new, lai pieprasītu atmiņu. Attēlā 2.2. ir redzama C un C++ sintakse atmiņas pieprasīšanai izmantojot C un C++ kodu. Vienīgais arguments mal-

```
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++**

loc() 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 mai-

nīgajā. 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 [9].

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.

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

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

Ja atmiņa pēc izmantošanas netiek nekad atbrīvota, un katru reizi, izpildot vienu un to pašu koda gabalu, iedalīta no jauna, tad pieejams operētājsistēmai atmiņas daudzums ar laiku samazinās. Sākumā sistēma paliek arvien lēnāka, pēc tam parasti notiek sistēmas apstāšanās.

2.3. Atmiņas iedalīšana glibc bibliotēkā

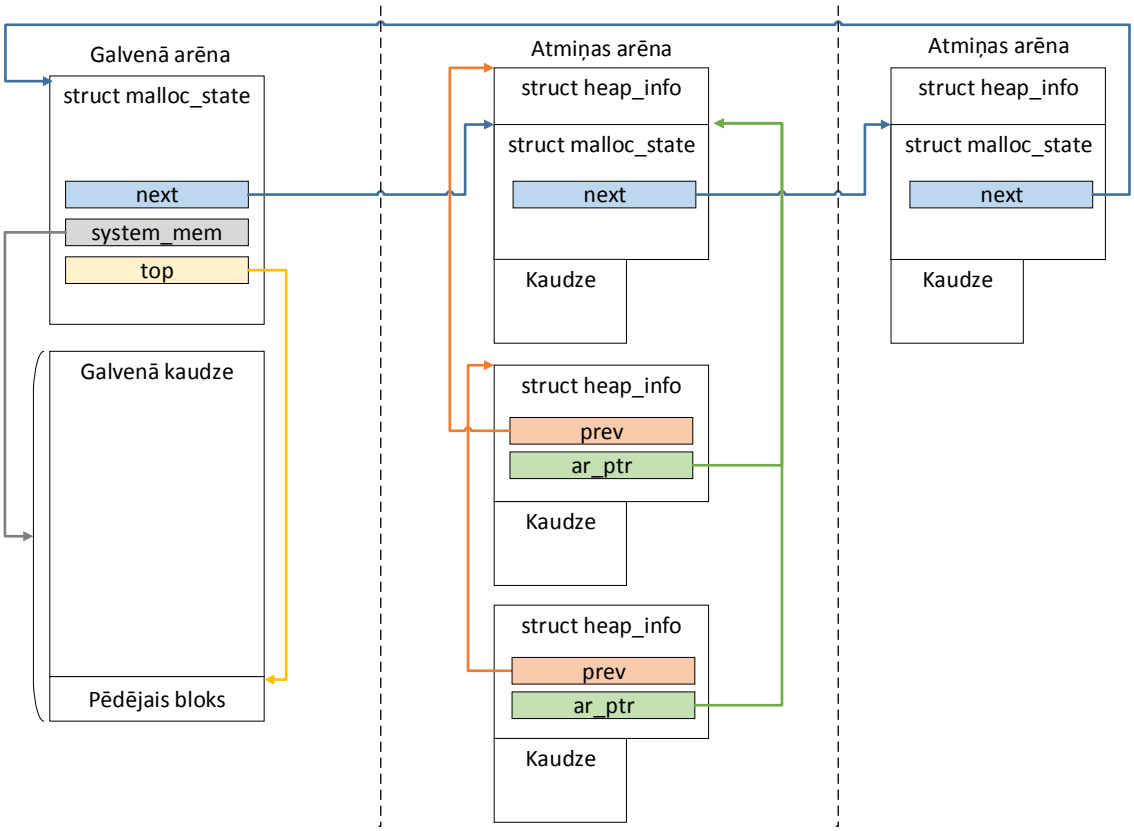
Darbā tiks aplūkota GNU C bibliotēkas (versija 2.3) ptmalloc2 realizācija, kuru izstrādāja Wolfram Gloger, balstoties uz Doug Lea dlmalloc realizāciju. Atmiņas iedalīšana sākas ar malloc() vai līdzīgo funkciju izsaukumiem no programmas koda un tiek nodrošināta ar GNU C bibliotēkas palīdzību.

2.3.1. Atmiņas arēna

Atmiņas arēnu var nosaukt par loģisko atmiņas kolekciju. Attēlā 2.4. ir parādītas 3 arēnas, kuras ir atdalītas savā starpā ar raustītam līnijām[1]. Atmiņas arēnu vienkāršotā veidā 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 iedalīto apgabalu kaudzē. Ja gabals ir atbrīvots, tad tas tiek pievienots vienā no sarakstiem

[1]Attēla izveidošanai tika izmantots GNU C malloc pirmkods [7] un vietnē nopublicēta shēma [1]. Attēls demonstrē atmiņas organizāciju.

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. Tā kā katrai arēnai var būt vairākas kaudzes, tad lai zinātu, kurai arēnai pieder kaudze, katrai kaudzei ir rādītājs uz arēnu. Ja kārtēja kaudze ir izlietotā un tajā nav atmiņas, tad tiek iedalīta jauna kaudze ar fiksēto 64 MB izmēru. Arēnas



2.4. att. **Arēnas GNU C bibliotēkā (versija 2.3)**

šīm nolūkam izmanto `mmap()` sistēmās izsaukumu, un izveido iegūtajā atmiņā jauno kaudzi. 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 realizācija), tad arēnas bloķēšana var būtiski samazināt veiktspēju. Gadījumos, kad pavedieni izmanto atmiņu no vairākām atsevišķām arēnām, piemēram kā tas notiek ptmalloc2 realizācijā, tad vienas

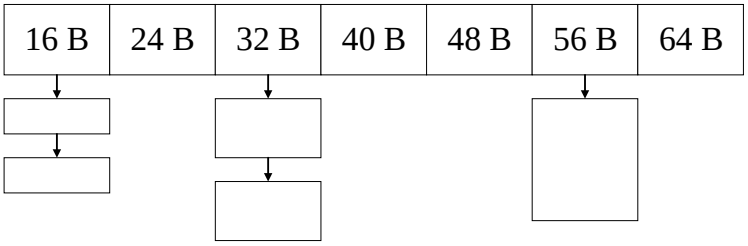
arēnas bloķēšana neietekmē atmiņas iedalīšanu parējās arēnās un atmiņas iedalīšana var notikt paralēli. Lai nodrošinātu labāku veiktspēju, GNU C bibliotēkā tiek izmantots modelis: katram pavedienam - viena arēna. Ja malloc() pirmo reizi izsaukts pavedienā, tad neatkarībā no tā vai kārtējā arēna bija 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 atmiņu no kodola iegūst, izmantojot brk() nevis mmap() sistēmas izsaukumu.

2.3.2. Atbrīvotās atmiņas organizācija

Realizācijā ptmalloc2 ir masīvi, kuri uzglabā norādes uz bin sarakstiem. Bin saraksti ir struktūras, kuras uzglabā atbrīvotus atmiņas gabalus arēnā, līdz brīdim, kad tie tiks iedalīti procesam atkārtoti. Atbrīvots atmiņas gabals, ja tas nebija iedalīts ar mmap() izsaukumu, ne vienmēr tiks uzreiz atgriezts operētājsistēmai, bet ir defragmentēts vai sapludināts ar pārējiem gabaliem un ievietots sarakstā. Ja atmiņa bija atbrīvota, tad atmiņas gabali tiks uzglabāti vienā no bin saistītiem sarakstiem. Eksistē divi bin saraksta veidi: ātrais (fastbin) un parastais (normal bin).



2.5. att. **Ātrais saraksts**

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 [7]. 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 elementi ir ar vienādu izmēru. 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].

Parastie saraksti var būt sadalīti 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 vai vienādi ar 512 baitiem. Vairāki ātrie gabali var būt sapludināti un uzglabāti dotajā sarakstā. Lielā izmēra saraksts uzglabā atmiņas gabalus, kuri ir lielāki par 512 baitiem, bet mazāki par 128 kilobaitiem. Gabali, kuru izmērs ir lielāks par 128 kilobaitiem netiek uzglabāti bin sarakstos, jo tiek iedalīti, izmantojot mmap(). Sākot ar GNU C bibliotēkas 2.18 versiju, gabalu kurš tiek iedalīts ar mmap() var uzdot ar M_MMAP_THRESHOLD konstanti. Lielā izmēra saraksta elementi ir sakārtoti pēc izmēra un ir iedalīti pirmais iekšā, pirmais ārā (jeb FIFO) kārtībā [2].

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 arēnas 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
Chunk struktūras elementu apraksts

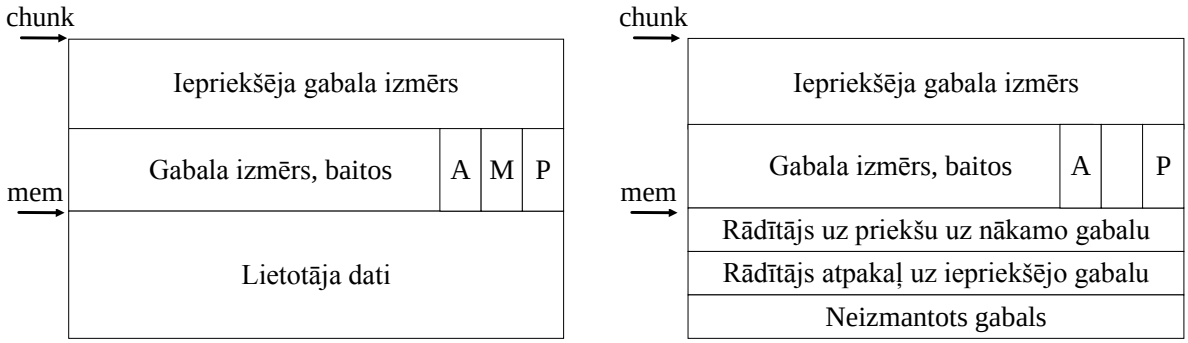
Elements	Nozīme
INTERNAL_SIZE_T prev_size	Iepriekšēja gabala izmērs (baitos), ja tas bija atbrīvots
INTERNAL_SIZE_T size	Kārtējā gabala izmērs (baitos)
struct malloc_chunk* fd	Rādītājs uz nākamo atbrīvoto gabalu, ja kārtējais gabals ir atbrīvots un pievienots dubultsaišu bin sarakstām
struct malloc_chunk* 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 INTERNAL_SIZE_T un rādītāja izmērs ir 8 baiti). Otrs iemesls kāpēc ir iedalīts lielāks atmiņas daudzums ir izlīdzināšana skaitlim, kas ir 2*sizeof(INTERNAL_SIZE_T) reizinājums. Šīs skaitlis ir vienāds ar 8 baitu izlīdzinājumu, ja makrodefinīcijas INTERNAL_SIZE_T vērtība ir vienāda ar 4 baitiem [7].

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 norāda uz daļu no iepriekšēja gabala novietotiem lietotāja datiem. Pēc tam seko kārtēja gabala izmērs un 3 biti ar meta informāciju.



2.7. att. **Atmiņas gabalu struktūra**

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. Sistēmas izsaukumu, mmap(), kas ir attēlots 2.2. tabulā, tiek izmantots, lai iegūtu atmiņas apgabalu, kas ir lielāks par 128 kilobaitiem. 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 sarakstā.

2.2. tabula
Chunk 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

Eksistē divi citi atmiņas gabali (top chunk un last_remainder), kuriem ir īpaša nozīme. Top chunk ir atmiņas gabals, kurš ierobežo pieejamās atmiņas daudzumu. Tas ir izmantots gadījumos, kad nav piemērotu gabalu bin sarakstos, kuri apmierina pieprasījumu vai varētu būt saplūdināti, lai apmierinātu pieprasījumu. Top chunk nodrošina pēdējo iespēju iedalīt

pieprasīto atmiņas daudzumu. Top chunk var mainīt savu izmēru. Tas saraujas, kad atmiņa ir iedalīta un izstiepjas, kad atmiņa ir atbrīvota blakus top chunk objektam. Ja ir pieprasīta atmiņa, kas ir lielāka par pieejamo, tad top chunk var paplašināties ar brk() palīdzību. Top chunk ir līdzīgs jebkuram citam atmiņas apgabalam. Galvenā atšķirība ir lietotāja datu sekcija, kura netiek izmantota, ka arī speciāla top chunk apstrāde, lai nodrošinātu, ka top chunk vienmēr eksistē.

Last_remainder ir vel viens atmiņas gabals ar īpašu nozīmi. Tas ir izmantots gadījumos, kad ir pieprasīts mazs atmiņas gabals, kas neatbilst nevienam bin saraksta elementam. Last_remainder ir dalījuma atlikums, kurš izveidojās pēc lielāka gabala sadalīšanas, lai apmierinātu pieprasījumu pēc maza gabala [10].

Zinot atmiņas gabala struktūru, var uzrakstīt kodu C valodā, kurš izdrukās iedalītā gabala izmēru (sk. 2.8. attēlu). Algoritms ir sekojošs:

1. ar malloc() tiek iedalīts atmiņas apgabals,
2. tiek iegūta size elementa adrese (objektam ar malloc_chunk struktūru),
3. tiek atņemtas A, M, P kontroles zīmes 111 = 7,
4. tiek atbrīvota atmiņa,

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

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

    ptrl = malloc(4);

    /* get address of chunk size (the second malloc_chunk element) */
    chunk_size = *((char *) ptrl - sizeof(size_t));
    /* the lower 3-bits are used as metadata */
    chunk_size = chunk_size - (chunk_size & 7);

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

    return 0;
}
```

2.8. att. **Izmēra noteikšana iedalītām gabalam**

3. PROBLĒMU APRAKSTS

3.1. Atmiņas noplūde

Atmiņas noplūde ir viena no bieži sastopamām problēmām [15]. 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. Pirmkārt, šī problēma ir novērojama, kad procesam iedalītās atmiņas adreses kļūst nepieejamas, pazaudētas. Atmiņa var kļūt nepieejama gadījumos, kad procesa adrešu telpā uz iedalīto atmiņas gabalu kaudzē nenorāda neviens rādītājs. Otrs problēmas veids ir novērojams, kad iekšējā buferī, rindā vai cita datu struktūrā dinamiski iedalītu elementu skaits pieaug neierobežoti. Tas atšķiras no pirmā veida ar to, ka visi elementi joprojām ir pieejami un procesa adrešu telpā ir rādītājs uz dinamiski iedalītu elementu. Abos gadījumos dinamiskā atmiņas iedalīšana programmai turpināsies līdz brīdim, kad tiks sasniegts RLIMIT limits. Šos limitus var atrast `man setrlimit` komandas izvadā. Limitiem, aprakstā ir nodarīts, kas ir jādara, kad limits ir pārsniegts. Daži no limitiem šajā gadījumā nosūta signālus procesiem, kuri izraisa atmiņas izmetes ģenerēšanu.

```
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 bytes are lost
9          str = new string("Hello, World!");
10     }
11     delete str;
12
13     return 0;
14 }
```

3.1. att. **Atmiņas noplūde, C++**

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. Beigās tiek atbrīvots tikai viens atmiņas gabals, kurš bija iedalīts pēdējais. Programmas darbības laikā kļūst pazaudēti 10000 gabali, kuru kopējais izmērs ir 140000 baiti.

Sekojošos gadījumos sistēmas kļūst viegli ievainojamas, ja tajās ir kļūda, kas izraisa atmiņās noplūdi [8]:

- Operētājsistēma neatbrīvo, lietotnes izpildei izmantoto atmiņu, kad lietotne beidz savu darbību, piemēram, AmigaOS,
- Ja servera vai citas programmas darbojās visu laiku bez apstājas,
- Ja iegultās sistēmas strādā gadiem ilgi,
- Ja portatīvām ierīcēm ir ierobežots atmiņas daudzums,
- Ja programmas pieprasa atmiņu uzdevumiem, kuri izpildās ilgstošu laika periodu,
- Reālā laikā sistēmām, jo ir svarīgi iegūt rezultātu ierobežotajā laikā.

Atmiņas noplūdes problēmu ir grūti atkļūdot, jo nav zināmi nosacījumi, kuriem izpildoties notiek atmiņas noplūde. Ja pazaudētās atmiņas daudzums nav liels, tad izstrādātājiem ir grūti atklāt un izlabot atmiņas noplūdi. Eksistē vairāki rīki, kuri palīdz atkļūdot atmiņas noplūdes problēmu, tādi ka: Valgrind, Totalview, Purify.

3.1.1. Atmiņas noplūdes sekas

3.1.2. Atmiņas noplūdes pazīmes

Rakstā [20] ir minēts, ka atmiņas noplūdes problēmu ir grūti atkļūdot, jo tai nav tiešas pazīmes, kas ļauj ātri identificēt problēmu. Tā kā atmiņas noplūdes laikā atmiņa tiek pazaudēta, tad var periodiski novērot procesa atmiņas patēriņa pieaugumu, kura dēļ, daļa no informācijas tiks uzglabāta lapošanas failā (paging file). Pēc tam notiks pakāpeniskā procesa palēninašana, jo procesa izpildei pietrūks brīvpiekļuves atmiņas un virtuālās atmiņas.

un atmiņas patēriņa pieaugums.

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/Page525.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] Linux programmer's manual. http://cf.ccmr.cornell.edu/cgi-bin/w3mman2html.cgi?brk(2). [Online; resurss apskatīts 3-Mai-2014].

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

[8] 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].

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

[10] 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].

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

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

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

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

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

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

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

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

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

[20] Reed Hastings, B. J. Fast Detection of Memory Leaks and Access Errors.

[21] 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 _____ 02.06.2014.

Recenzents: docents Dr.poniz. Jālis Bērziņš

Darbs iesniegts Datorikas fakultātē 02. 06. 2014.

Dekāna pilnvarotā persona: vecākā metodiķe Ārija Sproģe _____

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē

_____. prot. Nr. _____.

Komisijas sekretār___: lektore Anda Kooiņa _____