FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Zadání diplomové práce

Název:	Profilování haldy s nízkou latencí
Student:	Bc. Jan Tománek
Vedoucí:	Ing. Daniel Langr, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Systémové programování
Katedra:	Katedra teoretické informatiky
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

Seznamte se s problematikou profilování haldy, tj. operací souvisejících s dynamickou alokací paměti.
Seznamte se s existujícími nástroji pro profilování haldy, jako jsou např. Valgrind či Heaptrack, a prozkoumejte metody, které používají.
Navrhněte a implementujte vlastní nástroj pro profilování haldy pro více-vláknové aplikace se zaměřením na minimální latenci profilování, tj. co nejnižší časovou režii přidanou k běhu profilovaného programu.
Proveďte experimentální vyhodnocení vámi navrženého řešení ve formě jeho porovnání se stávajícími nástroji na různých testovacích programech.

Elektronicky schválil/a doc. Ing. Jan Janoušek, Ph.D. dne 1. května 2020 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Profilování haldy s nízkou latencí

Bc. Jan Tománek

Katedra teoretické informatiky
Vedoucí práce: Ing. Daniel Langr, Ph.D.

6. května 2021

Poděkování

Rád bych tímto poděkoval vedoucímu této práce, panu Ing. Danielu Langrovi, PhD., za jeho ochotu, trpělivost, vstřícnost a cenné rady, které mi při psaní práce poskytoval. Dále bych chtěl poděkovat mé rodině za podporu, které se mi během celého vysokoškolského studia dostávalo.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisu, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programu, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli zpusobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelum). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným zpusobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 6. května 2021 . . . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2021 Jan Tománek. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Tománek, Jan. *Profilování haldy s nízkou latencí*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Četné dynamické alokace z haldy mohou mít zásadní dopad na celkovou dobu běhu programu. Problém se stává ještě více kritickým ve vícevláknových aplikacích, kde přílišná interakce s haldou může přerůst v hlavní zdroj neefektivity programu a negativně se podepsat na jeho škálovatelnosti. Existující profilery haldy přidávají k běhu programu tak velkou režii, že se profilování paměťově náročných vícevláknových aplikací stává téměř nemožné.

Tato práce popisuje obvyklé rozložení paměti Linuxového procesu, nastiňuje interní části paměťového manažeru GNU C, analyzuje existující nástroje pro profilování haldy a provádí čtenáře procesem konstrukce vlastního profileru haldy. Práce dále navrhuje koncept profilování haldy s nízkou latencí, na jehož základě byl vytvořen nový profiler s názvem Heaprof. Ten byl následně porovnán s existujícími profilery. Experimentální vyhodnocení odhalilo, že Heaprof jako jediný ze zkoumaných nástrojů dokázal provádět profilování bez dalšího dopadu na škálovatelnost programu. V osmivláknových benchmarcích došlo až k osminásobnému urychlení celého profilování.

Klíčová slova dynamická alokace paměti, halda, vícevláknové aplikace, profilování, backtrace

Abstract

Frequent heap allocations can have a significant impact on the overall performance of a program. The problem becomes even more critical in multithreaded applications, where too many interactions with heap can grow into the main bottleneck and effectively diminish scalability. Profiling such applications with existing heap profilers is almost impossible due to the enormous runtime overhead they add.

This thesis describes the typical memory layout of a Linux process, briefly shows the internals of the GNU C memory allocator, analyzes available heap profiling tools, and guides the reader through the process of constructing a custom heap profiler. Further, the thesis proposes a concept of low-latency heap profiling. Based on that, a new profiler Heaprof has been developed and compared with other heap profiling tools. Experimental results revealed that Heaprof was the only tool capable of profiling without further compromising the program's scalability. For 8-threaded benchmarks, Heaprof achieved up to 8 times profiling speedup.

Keywords dynamic memory allocation, heap, multithreaded applications, profiling, backtrace

Obsah

Seznam obrázků

Seznam tabulek

Seznam ukázek kódů

Úvod

Není tomu tak dávno, co počítače disponovaly několika kilobyty operační paměti. V těch dobách museli vývojáři velice pečlivě dbát na efektivní využití každého dostupného bytu. Nebyl prostor na žádné plýtvání pamětí.

Doba se ale změnila. V průběhu posledních dekád došlo ve světe hardwaru k mnohanásobnému navýšení kapacit a výkonu na všech myslitelných úrovních. Dnes jsou běžně k dispozici zařízení disponující vícejádrovými procesory, desítkami gigabytů rychlé operační paměti, vnitřním úložištěm v řádu terabytů.

Adekvátně na to zareagoval i svět softwarový. Dnešní programy jsou daleko sofistikovanější a komplexnější, než tomu bylo v minulosti. To s sebou však přineslo neúměrnou náročnost při tvorbě programů. Historicky se proto velmi osvědčil koncept abstrakce. Kdykoliv začne být chování nějakého systému příliš složité, je abstrahováno do snadněji použitelné podoby. Díky tomu je možné systém použít bez nutnosti znát všechny jeho interní detaily.

Příkladem takovéto abstrakce je halda. Ta představuje volný paměťový prostor, ze kterého může běžící program formou dynamických alokací čerpat paměť potřebnou pro své další výpočty. Pro provedení dynamické alokace nemusí vývojář vůbec znát interní mechanismy, které halda používá. O paměť stačí jen definovaným způsobem zažádat.

Koncept haldy bývá univerzální napříč různými programovacími jazyky a programy ji velmi hojně využívají. I relativně malá aplikace může provést miliony dynamických alokací ze stovek různých míst. Provedení jedné dynamické alokace nezabere příliš mnoho času, začne-li jich ale program provádět desítky tisíc za vteřinu, může se to výrazně podepsat na celkové době běhu programu.

Ještě palčivějším se tento problém stává ve vícevláknových aplikacích. Při provádění dynamických alokací se mohou vlákna navzájem blokovat a aplikace nebude škálovat dle představ. To platí i v případě, že každé vlákno jinak pracuje jen se svými vlastními daty.

I přes velké množství operační paměti, která dnes bývá k dispozici, ji může

jediný program velmi snadno vyčerpat. Operační systém pak začne provádět přesuny dat mezi operační pamětí a diskem. To se může negativně projevit nejen na době běhu odpovědného programu, ale i na celkové použitelnosti celého operačního systému.

Identifikace a oprava problémů spjatých s dynamicky alokovanou pamětí nemusí být tak snadná, jak se na první pohled může zdát. Dynamické alokace jsou často schovány za dalšími vrstvami abstrakce. Nevinně vyhlížející funkce z externí knihovny jich může provést relativně velké množství. Pro uživatele takové funkce je ale náročné to zjistit.

Existující profilery haldy dokáží velmi přesně monitorovat její využití a lze pomocí nich odhalit problematická místa v programu. Obecně však trpí dvěma zásadními nedostatky – běh profilovaného programu příliš zpomalují a generují velké výstupní soubory. Zejména u vícevláknových paměťově náročných aplikací to má často za následek, že jimi lze profilovat jen několik minut běhu programu.

Cíl a struktura práce

Nabízí se proto následující otázka: „Nebylo by možné vytvořit profiler haldy s podstatně nižší latencí?“. Cílem práce je tuto otázku zodpovědět. Práce je primárně koncipována do Linuxového prostředí, mnohé ze zmíněných konceptů však mají přesah i do ostatních operačních systémů. U témat, která jsou závislá i na architektuře procesoru, je uvažována architektura x86(_64). Pokud práce hovoří o programu, pak je předpokládán program napsaný v jazyce C/C++. Některé pojmy a postupy však budou platné i pro jiné kompilované programovací jazyky.

Kapitola 1 seznamuje čtenáře se základním rozložením paměti procesu a představuje možnosti, jakými může program za běhu získat dodatečnou paměť. Kapitola 2 představuje implementaci operací souvisejících s dynamickou alokací paměti v jednom z nejrozšířenějších paměťových manažerů GNU C. Kapitola 3 dále analyzuje tradiční nástroje pro profilování haldy.

Kapitola 4 popisuje nezbytné kroky pro získání backtrace. Kapitola 5 pak slouží jako návod pro vytvoření vlastního profileru haldy pro operační systém Linux. A konečně, kapitola 6 popisuje profiler implementovaný v praktické části práce a provádí porovnání s ostatními vybranými profilovacími nástroji.

Kapitola 1

Paměť procesu

1.1 Adresní prostor procesu

Na moderních systémech je běžně využívána technika virtualizace paměti. Každý běžící proces má vlastní *virtuální adresní prostor* (anglicky *virtual address space*, VAS), ve kterém pracuje. Operační systém se ve spolupráci s hardwarem stará o mapování virtuálních adres procesu na adresy fyzické paměti.

VAS je lineární, začíná adresou 0 a pokračuje až do nějaké vysoké maximální adresy. Maximální adresa je dána konkrétní architekturou procesoru, operačním systémem a samotným spouštěným programem. 32bitový proces dokáže adresovat nejvýše 2^{32} bytů paměti (4 GiB), u 64bitových procesů je tato hranice teoreticky až 2^{64} bytů (16 EiB). V praxi však většina procesů využívá pouze malou část svého VAS.

To, že je virtuální adresní prostor lineární neznamená, že je zároveň i souvislý. Ve skutečnosti se v něm vyskytuje spousta adres, které proces nepoužívá a nesmí k nim ani přistoupit.

Stránkování

V soudobých operačních systémech je koncept virtuální paměti obvykle kombinován se stránkováním. Celý VAS procesu je rozdělen na menší části, takzvané *stránky*. Velikost všech stránek je stejná, a to i napříč dalšími procesy, které v systému běží. Velikost stránky je závislá na architektuře procesoru a nastavení operačního systému, obvykle se ale jedná o 4 KiB.

Fyzická paměť je rovněž rozdělena na takto velké oblasti. Ty se nazývají *rámce*. Stránky virtuální paměti jsou ukládány do volných rámců paměti fyzické. Díky stejné velikosti tak obsah jedné stránky odpovídá obsahu jednoho rámce. Dvě adresně sousední stránky nicméně nemusí být nutně uloženy v adresně sousedních rámcích. Pro běžící proces je postačující, jsou-li v hlavní fy-

zické paměti (RAM) alespoň stránky, které aktuálně používá. Ostatní stránky mohou být dočasně odkládány disk. [1, kap. 8]

V této souvislosti se lze dále setkat se dvěma termíny a jejich zkratkami. *Virtual memory size* (VSZ) označuje celkový počet stránek, které proces využívá. *Resident set size* (RSS) udává počet stránek procesu, které jsou aktuálně přítomny v operační paměti (RAM). Zbytek stránek se nachází na disku. Obě hodnoty mohou být udávány i v bytech – velikost stránek je fixní, počet stránek a počet využitých bytů na sebe lze převádět.

Mechanismus stránkování dává možnost do fyzicky menšího adresního prostoru vtěsnat virtuálně větší adresní prostor. Současně je tím umožněn souběh vícero procesů. Z pohledu jednoho programu je nicméně podstatné, že stránkování je plně řízeno operačním systémem a pro proces samotný je téměř neviditelné. Jinými slovy, běžící program má iluzi, že je celý jeho VAS ve fyzické paměti umístěn a pro přístup k jednotlivým buňkám paměti používá pouze virtuální adresy. Operační systém s podporou hardware už zajistí zbytek.

Z pohledu operačního systému se nicméně jedná o nelehký úkol, který navíc musí být prováděn velmi efektivně. Způsob, jakým k problematice přistupuje Linux, je detailně popsán v [2, kap. 2].

1.2 Paměťové segmenty

VAS je dále organizován do adresně souvislých celků, tzv. *segmentů*. Pojem segment je zde třeba chápat jako interval adres v jednorozměrném VAS, nikoliv jako prostředek pro způsob správy paměti zvaný segmentace.

Některé segmenty na sebe mohou adresně navazovat, je ale obvyklé, že se mezi nimi nachází prázdný prostor adres. Každý ze segmentů obsahuje specifický druh dat. Z bezpečnostních i historických důvodů mohou mít různé segmenty rozdílná oprávnění pro čtení, zápis a spouštění obsažených dat. Příklady segmentů jsou: [3, kap. 7] [1, kap. 8]

Text segment, jenž obsahuje strojový kód programu, řetězcové literály a další konstantní data. Tento segment obvykle není zapisovatelný, program jej tudíž nemůže za běhu modifikovat. Má fixní velikost a je součástí spustitelného souboru, odkud je při spuštění procesu nahrán do paměti.

Data segment obsahující inicializované statické proměnné. Obsah tohoto segmentu může být za běhu modifikován, potřebuje tudíž právo pro čtení a zápis. Velikost je opět fixní, vypočítaná v průběhu kompilace a statického linkování. Iniciální obsah je opět součástí spustitelného souboru.

BSS segment představuje neinicializované statické proměnné. V tomto kontextu je ovšem nutné chápat neinicializované jako inicializované na hodnotu nula. Vyžaduje obdobná práva jako Data segment a má rovněž fixní velikost. Součástí spustitelného souboru je v tomto případě pouze informace

o velikosti BSS segmentu, nikoliv samotný obsah. Ten by totiž obsahoval samé nuly.

Stack je segment určený pro zásobník volání. Velikost tohoto segmentu se při běhu programu mění, proces může data ze segmentu číst, zapisovat a v závislosti na konfiguraci i spouštět.[1] Segment Stack není součástí spustitelného souboru – v paměti je vytvořen až při spuštění procesu. Zásobník volání je podrobněji diskutován v sekci 1.4.

Heap označuje haldový segment představující dynamickou paměť procesu. Co se týče velikosti a práv, platí pro haldový segment stejná tvrzení, jako pro zásobníkový segment. Haldový segment rovněž není součástí spustitelného souboru. Blíže je haldový segment popsán v sekci 1.5.

Takovéto rozložení virtuální paměti je pouze konceptuální, reálné provedení bude záviset na konkrétním operačním systému, překladači či linkeru. Často se lze setkat například s tím, že segmenty Data, BSS a Heap jsou brány jako jeden celek a souhrnně označovány jen jako Data.

Vedle výše uvedených segmentů se v praxi ve VAS nacházejí další oblasti vytvořené *mapováním*. Tímto způsobem jsou do adresního prostoru procesu nahrávány například příslušné části sdílených (dynamických) knihoven. Sdílená knihovna definuje vlastní segmenty typu Text či Data. Mapování je prostředek, jakým je lze do VAS dostat. Obrázek 1.1 ilustruje možné logické rozložení paměti procesu, který sdílené knihovny využívá.

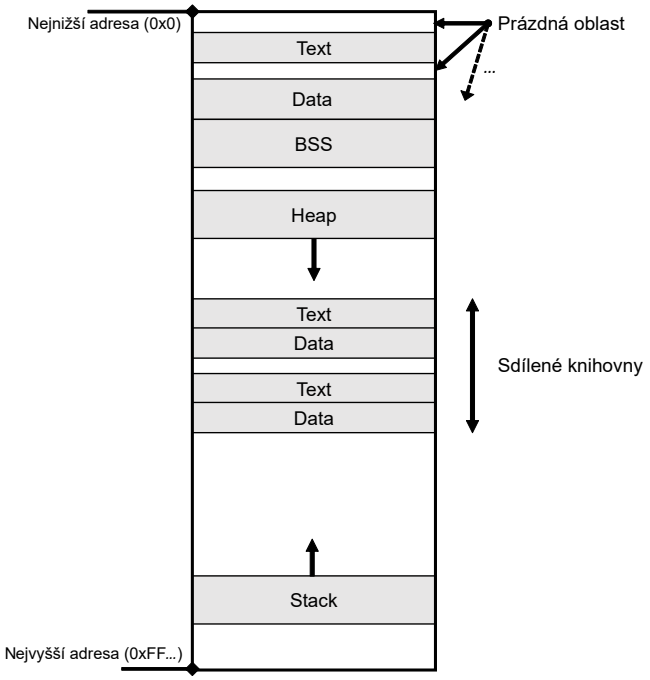
1.3 Alokace paměti

Pojem *alokace paměti* označuje situaci, kdy si program nějakým způsobem nárokuje paměť, kterou chce používat. Opačným procesem je *dealokace* či *uvolnění paměti*. Existují tři základní typy alokací: [4, kap. 3]

Statická alokace je používána pro statické proměnné v programu. Velikost statických proměnných je známa již v době překladu a k alokaci dochází momentem spuštění programu. Staticky alokované proměnné se nacházejí v segmentu Data či BSS a paměť pro ně vyhrazená je dostupná po celou dobu běhu programu.

Automatická alokace se využívá pro lokální proměnné a argumenty funkcí. K alokaci dochází automaticky ve chvíli, kdy program vstoupí do bloku složeného příkazu (např. tělo funkce) obsahující deklaraci takové proměnné. Dealokace proběhne rovněž automaticky opuštěním bloku. Velikost automatické alokace bývá[2] opět známa již v době překladu. Konkrétní adresa je

[1] Spustitelný zásobník však dává prostor zranitelnostem typu *buffer overflow*.
[2] Není to pravidlo, viz *Variable-length array*



Obrázek 1.1: Ilustrace konceptuálního rozložení paměti procesu

ale proměnným přiřazována až za běhu a ve dvou různých průchodech stejným blokem se může lišit. K automatickým alokacím se využívá segment Stack.

Dynamická alokace je používána ve chvílích, kdy není možné předem určit velikost požadované paměti či dobu, po kterou bude program potřebovat tuto paměť používat. Programátor musí o paměť explicitně zažádat a po použití ji uvolnit. Dynamicky alokovaná paměť se typicky nachází v haldovém segmentu, nebo v oblastech vytvořených pomocí mapování.

1.4 Zásobník

Segment Stack v sobě uchovává strukturu zvanou *zásobník volání* (zkráceně jen *zásobník*). Zásobník funguje na principu *last-in-first-out* (LIFO), dovoluje ale přistupovat k celému jeho obsahu, nikoliv pouze k položce na jeho vrcholu. Každé programové vlákno pracuje s vlastním zásobníkem.

Velikost zásobníku je obvykle limitována na několik jednotek megabytů. Tento limit je dán operačním systémem a při jeho překročení dochází k chybě zvané přetečení zásobníku (anglicky *stack overflow*), což má pro běžící program obvykle fatální následky.

Položky zásobníku se nazývají *aktivační záznamy* či *zásobníkové rámce*.[3] Aktivační záznam představuje volání jedné funkce programu a nese s sebou kontext tohoto volání. Na konkrétní podobu a obsah zásobníku má opět vliv mnoho faktorů, mezi nimi například architektura procesoru, použitý programovací jazyk či překladač. Obecně lze ale říct, že program může zásobník využívat pro čtyři různé účely: [4, kap. 3]

1. Slouží jako zdroj volné paměti pro automatické alokace. Jsou do něj tedy ukládány obsahy lokálních proměnných, případně argumenty volání funkcí.

2. Umožňuje snadné uložení servisních informací spojených s procesem volání funkce. To zahrnuje například návratovou adresu, ukazatel na předchozí aktivační záznam, zálohy hodnot v registrech, případně také návratovou hodnotu funkce.

3. Je to místo, kam může program dočasně ukládat mezivýsledky svých výpočtů, nevejdou-li se do registrů.

4. Na zásobníku lze provádět dočasné dynamické alokace. V některých rozšířeních C se jedná například o funkci `alloca` [5]. Paměť získaná dočasnou dynamickou alokací je při návratu z funkce automaticky uvolněna. Dynamická alokace na zásobníku nicméně není příliš rozšířená technika.

Na architektuře x86 roste zásobník od vyšších adres směrem k nižším a je implementován pomocí dvojice ukazatelů, jejichž názvy a role jsou následující:

Stack pointer indikuje aktuální vrchol zásobníku a je uložen v registru `esp`. Jakákoliv operace představující vložení nového obsahu na zásobník ukazatel sníží[4] a operace představující odstranění obsahu ze zásobníku jej zvýší.

Base pointer ukazuje na začátek aktuálního zásobníkového rámce. Obvykle jsou lokální proměnné a argumenty funkce ve strojovém kódu adresovány relativně vůči tomuto ukazateli. Je uchováván v registru `ebp`.

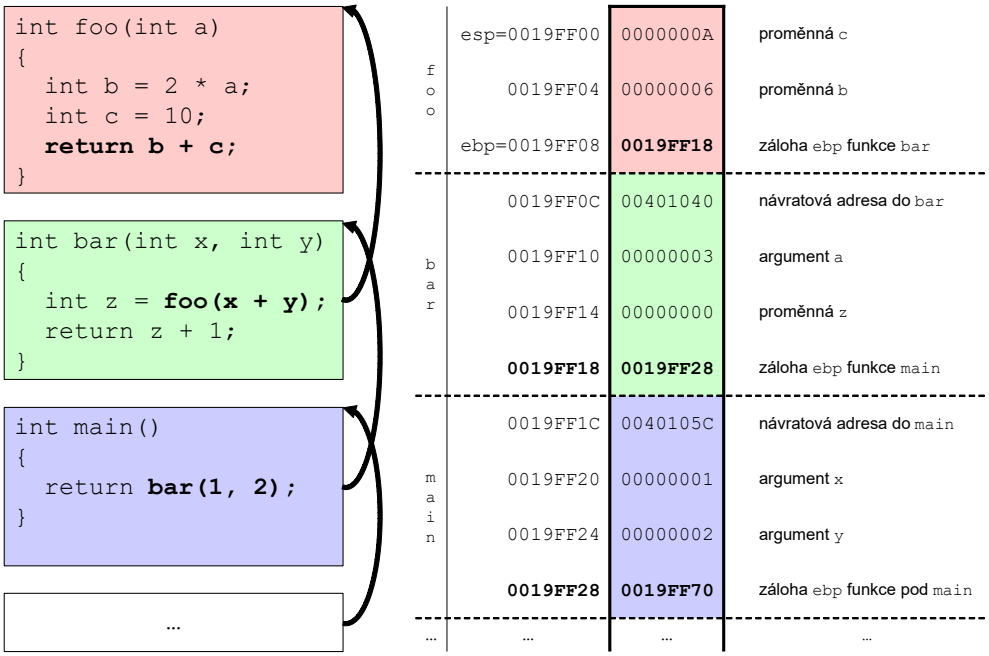
Následující zjednodušený popis volání funkce vychází z [6]. V popisu je předpokládána volací konvence cdecl a překladačem neoptimalizovaný kód.

Při zavolání funkce nejprve volající funkce uloží na zásobník argumenty pro funkci volanou a spustí instrukci `call`. Procesor následně přidá na zásobník návratovou adresu, což je ukazatel na instrukci (anglicky *instruction pointer*, IP) bezprostředně se nacházející za instrukcí `call`.

Poté se začne provádět kód volané funkce. Ta si na jejím začátku uloží na zásobník hodnotu registru `ebp`, který v tu chvíli drží adresu zásobníkového

[3] Což nijak nesouvisí s rámci ve fyzické paměti.

[4] Skutečně sníží – zásobník roste opačným směrem, než může radit intuice.



Obrázek 1.2: Ukázka zásobníku volání

rámce volající funkce. Poté volaná funkce vytvoří svůj zásobníkový rámec tím, že do `ebp` uloží aktuální vrchol zásobníku, tj. registr `esp`. Pro vyhrazení místa pro své lokální proměnné volaná funkce odečte od `esp` hodnotu odpovídající velikosti jejích lokálních proměnných. Tato hodnota je zjištěna již v době kompilace. Tím začne fungovat zmíněné relativní adresování. Na negativních offsetech od `ebp` se budou nacházet lokální proměnné, pozitivní offsety (konkrétně `ebp + 8` a vyšší) dovolí přístup k argumentům funkce.

Na obrázku 1.2 je znázorněn možný obsah zásobníku při zavolání několika funkcí. V levé části obrázku jsou uvedeny definice funkcí a tučně zvýrazněn stav toku programu. Zásobník odpovídá stavu těsně před návratem z funkce `foo`. V pravé části jsou uvedeny adresy, na kterých je zásobník umístěn, hodnoty na zásobníku a jejich význam. Rámce jednotlivých funkcí jsou vizuálně odděleny přerušovanou čarou. Tučně jsou zvýrazněny ukazatele na předchozí zásobníkový rámec a příslušné adresy.

Ze zásobníku je díky jeho struktuře možné vyčíst takzvaný *backtrace*. Ten představuje posloupnost aktuálně zavolaných funkcí v konkrétní moment běhu programu. V základním tvaru má backtrace podobu návratových adres z těchto funkcí. Pro situaci znázorněnou na obrázku 1.2 je backtrace posloupnost začínající adresami `00401040` a `0040105C`. Pro funkci `foo` backtrace přesně identifikuje místo v programu, odkud byla funkce zavolána.

Backtrace v podobě návratových adres je při zapnutém *address space layout randomization* (ASLR) relevantní pouze pro jeden běh programu. Při

dalším spuštění bude kód programu nahrán do jiného místa v paměti, čímž se změní i obsažené návratové adresy.

Za určitých okolností je nicméně možné provést *symbolizaci* backtrace, což představuje transformaci návratových adres do podoby názvů volaných funkcí, případně i přesných lokací (řádků) ve zdrojovém kódu programu, odkud byla funkce zavolána. Celý proces získání backtrace a jeho symbolizaci podrobněji rozebírá kapitola 4.

1.5 Haldový segment a mapování

Haldový segment je jedním z míst, kde se nacházejí dynamicky alokovaná data. Segment začíná na nějaké adrese a v průběhu provádění programu se tato adresa nemění. Adresa konce haldového segmentu se na Unixových systémech nazývá *program break* (někdy též *break point* či *break*).

Proces může požádat operační systém o změnu program break a tím modifikovat velikost haldového segmentu. Potřebuje-li proces zvětšit haldový segment, požádá operační systém o zvýšení program break. Chce-li naopak navrátit operačnímu systému část haldového segmentu, požádá o snížení program break. Haldový segment tedy roste (na rozdíl od zásobníku) směrem od nižších adres k adresám vyšším.

Žádost na změnu program break provádí proces přes systémová volání, ke kterým typicky používá funkce `brk` nebo `sbrk` z `<unistd.h>`: [1, kap. 8], [7]

`int brk(void* addr);`

Funkce `brk` nastaví program break na adresu `addr`. Návratová hodnota 0 indikuje úspěch operace, -1 neúspěch.

`void* sbrk (intptr_t increment);`

Funkce `sbrk` změní aktuální adresu program break o hodnotu `increment` a navrátí původní adresu program break. V případě žádosti o zvýšení program break tak návratová hodnota odpovídá ukazateli na nově získanou oblast. Pro zjištění aktuální adresy program break lze zavolat `sbrk(0)`.

Standard jazyka C zmíněné funkce nedefinuje a z POSIX byly odstraněny [8], na mnoha Unixových systémech jsou ale stále dostupné.

Vedle posunu program break může na Unixových systémech proces získat od operačního systému dodatečnou paměť pomocí mapování. Mapováním se ve VAS vytvoří nová souvislá oblast paměti. Ta je přístupná až do konce běhu programu, nebo do jejího explicitního odmapování. Namapovaná oblast je svým začátkem i koncem zarovnána na velikost stránky.

Mapování může být *anonymní*, nebo lze pro jeho vytvoření použít existující soubor. Anonymní mapování vytvoří oblast přednastavenou na hodnotu 0.

Konceptuálně jej lze chápat jako vytvoření nového haldového segmentu uvnitř VAS procesu.

Při mapování ze souboru se do paměti procesu nahraje jeho obsah. Je navíc možné specifikovat, zda se změna obsahu ve získané oblasti má propsat až namapovaného souboru, nebo má zůstat lokální pro spuštěný proces. V prvním případě se hovoří o sdíleném mapování, ve druhém případě o soukromém mapování.

Pro vytvoření nového mapování se používá funkce `mmap`, pro odstranění stávajícího mapování naopak funkce `munmap`. Jejich deklarace se nacházejí v hlavičkovém souboru `<sys/mman.h>`. Pro popis parametrů, návratových hodnot a přesného chování viz [8]. Na Linuxu existuje ve stejném hlavičkovém souboru dále funkce `mremap` [9], kterou lze použít k rozšíření nebo zmenšení namapované oblasti.

1.6 Halda

Přímé použití systémových volání typu `brk` nebo `mmap` není z pohledu běžného vývojáře uživatelských aplikací dostatečně flexibilním prostředkem k dynamickému získání paměti. To má hned několik důvodů:

- Systémová volání jsou obecně pomalá. Proces může potřebovat provádět velké množství dynamických alokací. Pokud by každá z nich měla znamenat jedno systémové volání, drasticky by se to podepsalo na době běhu programu.

- Posun program break implicitně znamená LIFO pořadí provádění alokací a uvolnění. Nelze dealokovat paměť uprostřed haldového segmentu. Anonymní mapování tento problém částečně řeší – bylo by možné vytvořit při každé alokaci nové mapování, získanou paměť použít a při uvolnění mapování zrušit. Při vytváření mapování by se operační systém postaral o nalezení dostatečně velkého volného místa v rámci VAS. Velikost nově přiděleného místa by však byla násobkem velikosti stránky paměti. Pro malé alokace by to znamenalo značné plýtvání místem.

- Dalším aspektem je přenositelnost kódu. Například zdrojový kód zkompilovatelný pod Linux by nešel bez dalších změn přeložit v prostředí Windows.

Proto se zavádí abstrakce dynamického paměťového prostoru s názvem *halda*. Halda představuje (teoreticky) neomezenou zásobárnu volného místa, odkud lze paměť alokovat a uvolňovat bez ohledu na velikost a pořadí. K tomu se využívá jazykem definovaná sada funkcí.

V případě jazyka C se jedná o funkce `malloc`, `free` a podobné. Tyto funkce jsou implementovány *paměťovým manažerem* (též *alokátor*), který haldu spravuje. Paměťový manažer je součástí GNU C knihovny a pojednává o něm kapitola 2. Lze ale využít i jinou implementaci, mezi nimi například TCMalloc [10, 11], jemalloc [12], nebo lockfree-malloc [13].

Paměťový manažer typicky využívá haldový segment nebo vytváří větší paměťové oblasti pomocí anonymního mapování. Tyto oblasti dále rozděluje na menší paměťové bloky, kterými jsou obslouženy jednotlivé dynamické alokace z programu. Paměť alokovaná na haldě zůstává procesu přístupná až do chvíle, dokud ji sám explicitně neuvolní.

Některé vyšší programovací jazyky používají tzv. *garbage collection*, což je mechanismus, který přenáší zodpovědnost za uvolňování paměti z programátora na běhové prostředí programu. To jednou za čas automaticky detekuje všechny dynamicky alokované bloky paměti, které jsou v běžícím programu v daný moment dostupné, a zbytek uvolní.

Jazyk C tuto metodu nevyužívá. V jazyce C++ pro ni existuje od standardu C++11 podpora [14, kap. 23.10.4], prakticky však nebyla dodnes implementována a nelze vyloučit, že v budoucích verzích jazyka nebude odstraněna. [15]

Může se stát, že program nepracuje s dynamicky alokovanou pamětí korektně a dojde k některé z následujích chyb:

Únik paměti angl. *memory leak*. Program dynamicky alokuje paměť, kterou po jejím použití neuvolní. Ta tak zůstane alokována až do konce běhu programu.

Přístup do uvolněné paměti angl. *memory corruption*. Program se snaží provést zápis nebo čtení z paměti, kterou již uvolnil.

Dvojí uvolnění angl. *double-free*. Speciálním druhem memory corruption je dvojí uvolnění. Jedná se o situaci, kdy se program snaží uvolnit paměť, která již jednou uvolněna byla.

Dynamické alokace v C

K práci s dynamicky alokovanou pamětí nabízí jazyk C následující funkce deklarované v hlavičkovém souboru `<stdlib.h>`: [16, 1]

`void* malloc(size_t size);`

Funkce `malloc` alokuje na haldě `size` bytů a vrátí ukazatel na začátek alokovaného místa. Obsah alokovaného bloku paměti není definovaný a je potřeba jej dále inicializovat. Pokud funkce neuspěje, navrátí hodnotu `NULL`.

`void* calloc(size_t nmemb, size_t size);`
: Funkce `calloc` alokuje na haldě blok paměti velikosti dostatečné pro uložení pole `nmemb` prvků, každý o velikosti `size`. Návratovou hodnotou je ukazatel na první byte alokovaného bloku.

  Na rozdíl od `malloc` je celý navrácený blok inicializovaný na hodnotu nula. Použití `calloc` je preferované před kombinací `malloc` a `memset`, protože systém může poskytnout již vynulovanou paměť, čímž se ušetří část práce. Pokud volání neuspěje, funkce navrátí `NULL`.

`void* realloc(void* ptr, size_t size);`
: Funkce `realloc` dovoluje změnit velikost dříve alokovaného a doposud neuvolněného bloku paměti odkazovaného v `ptr` na novou velikost `size`. Funkce vrací ukazatel na alokovanou oblast velikosti `size`, v případně neúspěchu vrací `NULL`.

  Realokaci je za určitých okolností možné provést „na místě“. V takovém případě funkce pouze interně rozšíří či zmenší stávající blok paměti na požadovanou velikost a funkce navrátí stejný ukazatel, jako je hodnota `ptr`. Není-li toto možné, funkce alokuje novou oblast velikosti `size`, překopíruje do ní obsah původní oblasti a tu uvolní. Obsah nově vzniklé oblasti není definován.

  Pokud je `ptr` roven hodnotě `NULL`, funkce má stejný efekt jako volání `malloc(size)`. Pokud je `size` rovno nule, výsledek závisí na implementaci `realloc`. Obvyklé je, že funkce v takovém případě provede ekvivalent `free(ptr)`, viz [17] a [18].

`void free(void* ptr);`
: Funkce `free` uvolní paměť dříve alokovanou pomocí `malloc` nebo podobné funkce. Pokud je `ptr` roven `NULL`, volání nemá žádný efekt.

`void* aligned_alloc(size_t alignment, size_t size);`
: Funkce `aligned_alloc` byla do jazyka přidána standardem C11. Umožňuje alokovat zarovnanou paměť o velikosti `size` bytů. Zarovnání je specifikováno parametrem `alignment`, přičemž se musí jednat o validní zarovnání podporované danou implementací. Parametr `size` musí být násobkem `alignment`.

Kromě výše uvedených funkcí, které definuje přímo standard jazyka C, mohou v různých rozšířeních existovat ještě další funkce umožňující interakci s haldou. Příklady takových funkcí jsou `reallocarray` [17], `posix_memalign`, `memalign`, `valloc` a `pvalloc` [19]. Z prostředí Windows pak například funkce `_expand` [20].

Dynamické alokace v C++

V jazyce C++ jsou funkce typu `malloc` rovněž k dispozici, není však doporučováno je používat [21]. Namísto toho je ve standardní knihovně jazyka

`operator new`, `operator new[]`, `operator delete` a `operator delete[]`.

Každý z těchto operátorů je k dispozici v několika přetížených variantách [14, kap. 21.6.2] a uživatel si může definovat i přetížení vlastní. Standard C++ zároveň dovoluje definovat si vlastní verze těchto operátorů, které pak budou použity jako náhrada za výchozí implementace v knihovně.

```
// operator new
// Allocate 100 bytes.
void* a = ::operator new(100);

// new-expression
// Create a dynamically allocated integer initialized to 100.
int* b = new int(100);
```

Kód 1.1: Použití `operator new` a výraz `new`

Je třeba rozlišovat mezi operátorem a výrazem – `operator new` provádí pouze dynamickou alokaci paměti. Vedle toho výraz `new` slouží k vytvoření a inicializaci dynamicky alokovaného objektu požadovaného typu. Pro alokaci potřebného místa výraz `new` volá `operator new`. Rozdíl mezi operátorem a výrazem ilustruje kód 1.1.

Funkce `malloc` a `operator new` si jsou nabízenou funkcionalitou velmi podobné. Jeden z rozdílů je, že `operator new` indikuje neúspěch operace vyhozením výjimky `std::bad_alloc`. Existuje však i přetížená varianta, která chybu indikuje návratem `nullptr`. Konceptuálně provádí `operator new` alokace z jiné haldy, než funkce `malloc`. Standard C++ u `operator new` zmiňuje: [14, kap. 21.6.2.1]

„*Whether the attempt involves a call to the C standard library functions* `malloc` *or* `aligned_alloc` *is unspecified.*“

Rozhodnutí, zda dynamickou alokaci provést přes `malloc` či nikoliv, je tedy ponecháno na tvůrce konkrétní implementace standardní knihovny jazyka C++. Dvě nejrozšířenější z nich — `libstdc++` (GCC) a `libc++` (LLVM) — funkce z knihovny C interně volají. Pro budoucí profiler je toto důležité zjištění. Ke sledování dynamických alokací v C++ programech totiž bude stačit odchytit paměťové funkce z jazyka C. [22]

1.7 Ukázka rozložení paměti procesu

Pojďme si nyní problematiku alokací a rozložení VAS demonstrovat prakticky. Program `memory_regions.c` (kód 1.2) v jazyce C zavádí několik proměnných. Proměnné se od sebe kromě názvu a typu liší i místem jejich definice:

`g` – globální proměnná typu `int`

`c` – globální konstanta typu `const int`

`z` – neinicializované globální pole typu `char[]`

`l` – lokální proměnná typu `int`

`d` – ukazatel na dynamicky alokovanou oblast

`m` – ukazatel na anonymně namapovanou oblast

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

#define PAGE_SIZE 0x1000

int g; // global variable
const int c = 1234; // global constant
char z[PAGE_SIZE]; // zero initialized array

int main()
{
    int l = 1234; // local variable
    void* d = malloc(100); // dynamically allocated data

    // private anonymous mapping
    void* m = mmap(0, 2 * PAGE_SIZE, PROT_READ | PROT_WRITE,
                   MAP_ANON | MAP_PRIVATE, -1, 0);

    printf("g:    %p-%p\n", &g, &g + sizeof(g));
    printf("c:    %p-%p\n", &c, &c + sizeof(c));
    printf("z:    %p-%p\n",  z,  z + sizeof(z));
    printf("l:    %p-%p\n", &l, &l + sizeof(l));
    printf("d:    %p\n",     d);
    printf("m:    %p\n",     m);
    printf("brk:  %p\n",     sbrk(0));
    printf("main: %p\n",     main);
    fflush(stdout);

    while (1);
}
```

Kód 1.2: Výpis adres z různých oblastí paměti

Úkolem programu je vypsat adresy uvedených proměnných, v případě ukazatelů adresy odkazovaných oblastí. Program dále vypíše program break a adresu funkce `main`. Pro další zkoumání necháme program po výpisu zacyklit. Pro ukázku je použit překladač GCC verze 8.4.0 na 64bitovém Linuxovém systému s vypnutým ASLR a velikostí stránky 4 KiB.

Program zkompilujeme, spustíme a jakmile provede výstup, pozastavíme jej. Dále si zjistíme, jaké paměťové oblasti má proces namapované. To lze provést několika způsoby, nejpřímočařejší z nich je využít Linuxový `proc` filesystém a vypsat obsah souboru `/proc/[pid]/maps` [23]. Alternativou je např. `pmap` [24], nebo připojení se k procesu přes GDB a následně spuštění příkazu `info proc mappings` [25, kap. 21.1.2].

```
$ gcc memory_regions.c; ./a.out
g:    0x8201010-0x8201020
c:    0x8000964-0x8000974
z:    0x8201040-0x8202040
l:    0x7fffffffee0d4-0x7fffffffee0e4
d:    0x8403260
m:    0x7fffff7e0000
brk:  0x8424000
main: 0x8000785
^Z
[5]+  Stopped                 ./a.out
$ cat /proc/`pidof a.out`/maps
08000000-08001000 r-xp 00000000 00:00 24459                      a.out
08200000-08201000 r--p 00000000 00:00 24459                      a.out
08201000-08202000 rw-p 00001000 00:00 24459                      a.out
08202000-08203000 rw-p 00000000 00:00 0
08403000-08424000 rw-p 00000000 00:00 0                          [heap]
7fffff000000-7fffff1e7000 r-xp 00000000 00:00 329960             libc-2.27.so
7fffff1e7000-7fffff1f0000 ---p 001e7000 00:00 329960             libc-2.27.so
7fffff1f0000-7fffff3e7000 ---p 001f0000 00:00 329960             libc-2.27.so
7fffff3e7000-7fffff3eb000 r--p 001e7000 00:00 329960             libc-2.27.so
7fffff3eb000-7fffff3ed000 rw-p 001eb000 00:00 329960             libc-2.27.so
7fffff3ed000-7fffff3f1000 rw-p 00000000 00:00 0
7fffff400000-7fffff428000 r-xp 00000000 00:00 329921             ld-2.27.so
7fffff428000-7fffff429000 r-xp 00028000 00:00 329921             ld-2.27.so
7fffff629000-7fffff62a000 r--p 00029000 00:00 329921             ld-2.27.so
7fffff62a000-7fffff62b000 rw-p 0002a000 00:00 329921             ld-2.27.so
7fffff62b000-7fffff62c000 rw-p 00000000 00:00 0
7ffffff7d0000-7ffffff7d2000 rw-p 00000000 00:00 0
7ffffff7e0000-7ffffff7e2000 rw-p 00000000 00:00 0
7ffffff7ef000-7ffffffef000 rw-p 00000000 00:00 0                 [stack]
7ffffffef000-7fffffff0000 r-xp 00000000 00:00 0                  [vdso]
```

Každý řádek souboru `maps` popisuje jednu oblast paměti v procesu. Pro naše účely je nejpodstatnější první a poslední sloupec. V nich nalezneme rozsah adres dané oblasti, respektive název souboru, ze kterého byla oblast do paměti namapována. Standardně se v posledním sloupci nachází absolutní cesta k namapovanému souboru. Pro větší přehlednost je ale v ukázce uvedeno pouze jeho jméno. Pojďme nyní porovnat výstup programu s výpisem mapovaných oblastí:

- Do oblasti `08000000-08001000` náleží funkce `main` a globální konstanta `c`. Tato oblast představuje segment Text, což koresponduje i s právy pro čtení a spouštění dat.
- V oblasti `08201000-08202000` se nachází segment Data. Je zde umístěna globální proměnná `g`. Stejně jako předchozí oblast, je i tato mapována ze souboru. Stojí za povšimnutí, že zde začíná a z velké části leží pole `z`.
- Oblast `08202000-08203000` představuje segment BSS a pole `z` zde končí. Tato oblast už ze souboru mapována není. Linux umisťuje segment BSS

na konec segmentu Data. Při startu programu je do paměti namapována část spustitelného souboru, ve které se segment Data nachází. Tato část je zarovnaná na velikost stránky virtuální paměti. Velikost segmentu Data ale nutně neodpovídá násobku velikosti stránky. Na konci poslední stránky tím vzniká volné místo, které je vynulováno a kam je umístěn začátek segmentu BSS.

V ideálním případě se do volného místa vejde celý BSS a není potřeba nic dalšího řešit. V našem případě tomu tak není – `z` je úmyslně voleno tak, aby samo zabíralo celou jednu stránku paměti. Systém kvůli tomu vytvořil oblast `08202000-08203000` a umístil do ní zbývající část pole `z`. Pokud je `z` dostatečně zmenšeno, tato oblast vůbec nevznikne.

- Oblast `08403000-08424000` odpovídá haldovému segmentu. To lze jednoduše poznat podle označení `[heap]`, ale také z výstupu programu. Do této oblasti spadá adresa v ukazateli `d`. Programem ohlášený program break se rovněž shoduje s koncovou adresou této oblasti.

- Oblasti `7fffff000000-7fffff3f1000` a `7fffff400000-7fffff62b000` obsahují sdílené knihovny, které program využívá. První z nich je standardní knihovna jazyka C (`libc-2.27.so`), druhá je knihovna dynamického linkeru (`ld-2.27.so`). Podle přístupových práv se dá odhadnout, kde se nacházejí segmenty Text nebo Data těchto knihoven.

- Oblast `7fffff7e0000-7fffff7e2000` program vytvořil anonymním mapováním. Začátek oblasti odpovídá reportované hodnotě ukazatele `m`. Souhlasí i velikost rovna dvojnásobku velikosti stránky.

- V oblasti `7fffff7ef000-7fffffef000` se nachází zásobník volání. To lze opět jednoduše identifikovat označením `[stack]`, ale také adresou, na které se nachází proměnná `l`.

- Poslední oblastí je `7fffffef000-7ffffff0000` s označením `[vdso]`. Zde se nachází malá dynamická knihovna, kterou Linuxový kernel mapuje do každého uživatelského procesu [26].

Kapitola 2

Paměťový manažer GNU C

V této kapitole bude popsána implementace paměťového manažeru knihovny GNU C (glibc). Primárním zdrojem informací je práce od autorů Block, Dewald [27], kde je zkoumáno fungování paměťového manažeru ve verzi 2.23 z roku 2016.

Nová verze knihovny vychází každého půl roku. V době psaní tohoto textu je k dispozici již verze 2.33 vydaná v únoru 2021. Z časového harmonogramu [28] a zpráv k vydaným verzím knihovny plyne, že v paměťovém manažeru nedošlo od verze 2.23 k žádným výraznějším změnám. Výjimkou je verze 2.26 (srpen 2017), kdy byla do paměťového manažeru přidána optimalizace pro vícevláknové programy v podobě lokální cache.

Dalším zdrojem je glibc wiki [29]. Zdroj přímo neuvádí verzi knihovny, kterou popisuje, nicméně se zde o lokální cache lze dočíst. Dá se proto předpokládat, že popis reflektuje minimálně verzi 2.26.

Některé informace byly převzaty z článku od sploitfun [30]. Článek byl publikován již v roce 2015, svým obsahem však koresponduje s předchozími zdroji.

Pro získání některých dílčích detailů bylo při zpracovávání tématu rovněž nahlíženo do zdrojového kódu[5] glibc v aktuální verzi 2.33 [31]. Kód je poměrně dlouhý, ale velmi dobře a podrobně komentovaný.

Text používá následující termíny pro pojmenování různých skupin funkcí paměťového manažeru:

Alokační funkce představují skupinu funkcí používaných k dynamické alokaci paměti. Jedná se o funkce typu `malloc`, `calloc` a další. Spadá sem i volání `realloc` s `ptr == NULL`. Hovoří-li text bez dalšího rozlišení o funkci `malloc`, je tím myšlena celá tato skupina funkcí.

Uvolňovací funkce jsou funkce používané k uvolnění dynamicky alokované paměti. Zahrnuje `free` nebo volání `realloc` s argumentem `size == 0`.

[5] Soubory `malloc.c` a `arena.c`

Hovoří-li text o funkci `free`, je tím podobně jako u alokačních funkcí myšlena celá tato skupina.

Paměťové funkce označují alokační i uvolňovací funkce dohromady.

2.1 Kusy

Na nejnižší úrovni paměťový manažer GNU C knihovny pracuje s takzvanými *kusy* (anglicky *chunks*). Kus představuje malou souvislou část VAS procesu a může se nacházet ve dvou stavech – alokovaný a uvolněný. Alokovaný kus je takový, ke kterému program získal přístup zavoláním alokační funkce, uvolněný kus je naopak ten, který program uvolnil funkcí `free`.

Alokovaný kus disponuje dostatečným množstvím paměti potřebné pro uložení uživatelských dat požadované velikosti. Žádá-li uživatel 60 bytů paměti, paměťový manažer mu může vrátit kus, do kterého se reálně vejde například 72 bytů. Nemůže ale vrátit kus, do kterého by se 60 bytů nevešlo.

Při přechodu z alokovaného do uvolněného stavu paměťový manažer rozhodne, zda uvolněnou paměť vrátí operačnímu systému, nebo si ji ponechá pro další použití. Ve druhém případě do kusu uloží své servisní informace, které později využije pro znovupřidělení kusu při některé z dalších alokací. Jinými slovy, zavoláním uvolňovací funkce nevrací uživatel alokovanou paměť operačnímu systému, ale paměťovému manažeru. Ten se pak rozhodne, jakým způsobem s pamětí dále naloží.

Kusy jsou v paměti uloženy bezprostředně za sebou, jeden paměťový kus je popsán strukturou `malloc_chunk`, která vyznačuje jeho začátek. Jak bude později vysvětleno, ne všechny prvky struktury jsou za všech okolností validní a používané.

Struktura `malloc_chunk`

Člen	**Popis**
`mchunk_prev_size`	Velikost předchozího kusu.
`mchunk_size`	Velikost aktuálního kusu.
`fd`	Ukazatel na další uvolněný kus.
`bk`	Ukazatel na předchozí uvolněný kus.
`fd_nextsize`	Ukazatel na další uvolněný kus jiné velikosti.
`bk_nextsize`	Ukazatel na předchozí uvolněný kus jiné velikosti.

Každý kus musí mít alespoň takovou velikost, aby se do něj vešly první čtyři prvky `malloc_chunk`. Pro 32bitové prostředí to znamená, že nejmenší velikost kusu je 16 B, pro 64bitové prostředí 32 B.

Standard C navíc garantuje zarovnání ukazatelů získaných pomocí `malloc`. Ve 32bitovém prostředí se jedná o zarovnání na 8B adresy, v 64bitovém prostředí na 16B adresy. Protože na sebe jednotlivé kusy v paměti navazují, pro splnění požadavku na zarovnání je nutné, aby byla velikost každého kusu násobkem 8 (resp. 16).

To má za následek, že tři nejméně významné bity `mchunk_size` jsou nulové. Paměťový manažer je proto využívá k uchování dalších informací. Jedná se o bity:

A (0x04) Slouží k identifikaci arény (pojem aréna viz sekce 2.4). Hodnota 1 indikuje, že kus spadá do hlavní arény, hodnota 0 znamená, že kus je součástí některé z vláknových arén.

M (0x02) Paměťový manažer se může rozhodnout, že alokaci provede pomocí systémového volání `mmap`. K tomu dochází u velkých alokací, ve výchozím nastavení je hranicí pro provedení alokace tímto způsobem hodnota 128 KiB. Hranice se může za běhu postupně zvyšovat. Kusy alokované přes `mmap` mají nastaven bit M na 1 a ostatní bity jsou ignorovány.

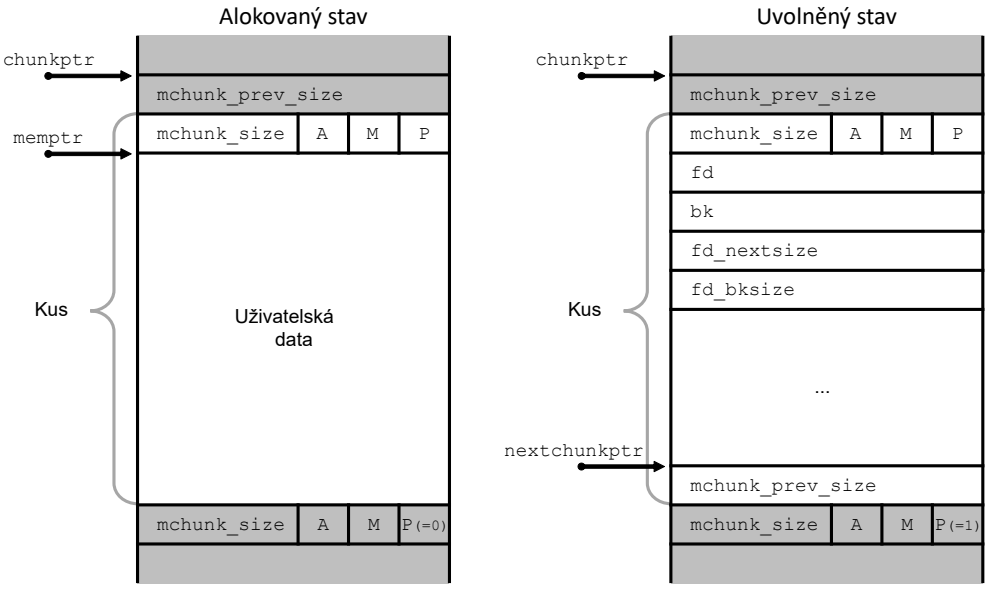
P (0x01) Pomocí tohoto bitu lze identifikovat stav předchozího kusu. Je-li předchozí kus v alokovaném stavu, bit je nastavenu na hodnotu 1.

Hodnota v `mchunk_size` je platná vždy, bez ohledu na stav kusu. U kusů v alokovaném stavu představuje jedinou přidanou režii paměťového manažeru. Zbytek kusu počínaje od `fd` je už použit pro uživatelská data. V 32bitovém prostředí je proto u kusu velikosti 16 B k dispozici 12 B pro uživatelská data, kus velikosti 24 B má 20 B pro uživatelská data atd. Počet bytů vyhrazených pro uživatelská data lze pro daný kus zjistit funkcí `malloc_usable_size` [32]. Ukazatele na uživatelská data a na začátek `mchunk_size` mezi sebou lze jednoduše převádět, adresy se liší pouze o `2 * sizeof(size_t)`.

Důležitým aspektem je, že `mchunk_prev_size` je potřebný až v momentě, kdy se předchozí kus nachází v uvolněném stavu. Paměťový manažer toho využívá tím způsobem, že uživatelská data alokovaného kusu přesahují až oblasti `mchunk_prev_size` kusu následujícího. Jinými slovy, poslední 4 (resp. 8) byty jednoho kusu, jsou ve skutečnosti `mchunk_prev_size` kusu následujícího.

Lépe je to patrné z obrázku 2.1. V jeho levé části je kus v alokovaném stavu. Ukazatel `chunkptr` ukazuje na místo, kde se nachází struktura `malloc_chunk`, ukazatel `memptr` představuje návratovou hodnotu funkce `malloc`. Kus je na obrázku vyznačen způsobem, který odpovídá jeho logickému rozložení, tedy jako oblast uživatelských dat anotovaná jeho velikostí. Bit P následujícího kusu je nastaven na 0.

Po uvolnění kusu (pravá část obrázku) se hodnota bitu P následujícího kusu změní na 1 a do prostoru uživatelských dat uvolněného kusu jsou doplněny ostatní servisní informace (`fd`, `bk` a další). Hodnota `mchunk_size` je



Obrázek 2.1: Paměťový kus podle, [29]

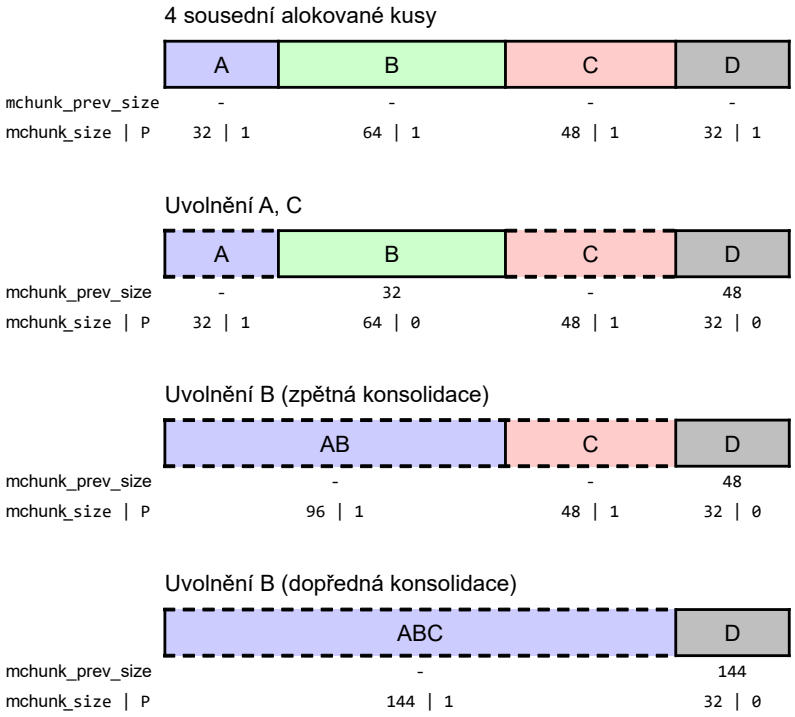
zkopírována na konec prostoru uživatelských dat. Toto místo už ale odpovídá `nextchunkptr->mchunk_prev_size`.

2.2 Konsolidace

Konsolidace je mechanismus, který má za cíl částečně řešit problémy s fragmentací paměti tím, že vyloučí existenci dvou po sobě jdoucích uvolněných kusů. Namísto toho spojí tyto dva kusy do jednoho většího kusu, který může být následně využit pro obsluhu alokačního požadavku a v případě potřeby opět rozdělen na dva. Jeden, který odpovídá požadované velikosti alokace, a druhý, který zůstane v uvolněném stavu.

Konsolidace probíhá ve dvou fázích. První fáze se označuje jako *zpětná* konsolidace a dochází při ní ke spojení kusu aktuálního a předchozího. Komplementem je *dopředná* konsolidace, kdy se aktuální kus naopak spojuje s kusem následujícím. V obou případech platí, že ke spojení může dojít jen ve chvíli, kdy je předchozí (resp. následující) kus v uvolněném stavu. Fáze jsou na sobě nezávislé. Je-li například předchozí kus uvolněný, ale následující nikoliv, dojde pouze ke konsolidaci zpětné.

Obrázek 2.2 koncept konsolidace názorně ilustruje. Na začátku existují čtyři sousední paměťové kusy v alokovaném stavu – symbolicky označeny jako A, B, C, D. Následně dojde k uvolnění kusů A, C. Pro kus B to znamená, že dojde ke změně hodnoty v bitu P a do `mchunk_prev_size` se překopíruje `mchunk_size` kusu A. Podobně pro kusy D a C. Položka `mchunk_prev_size` představuje tzv. *boundary tag*.



Obrázek 2.2: Konsolidace paměťových kusů

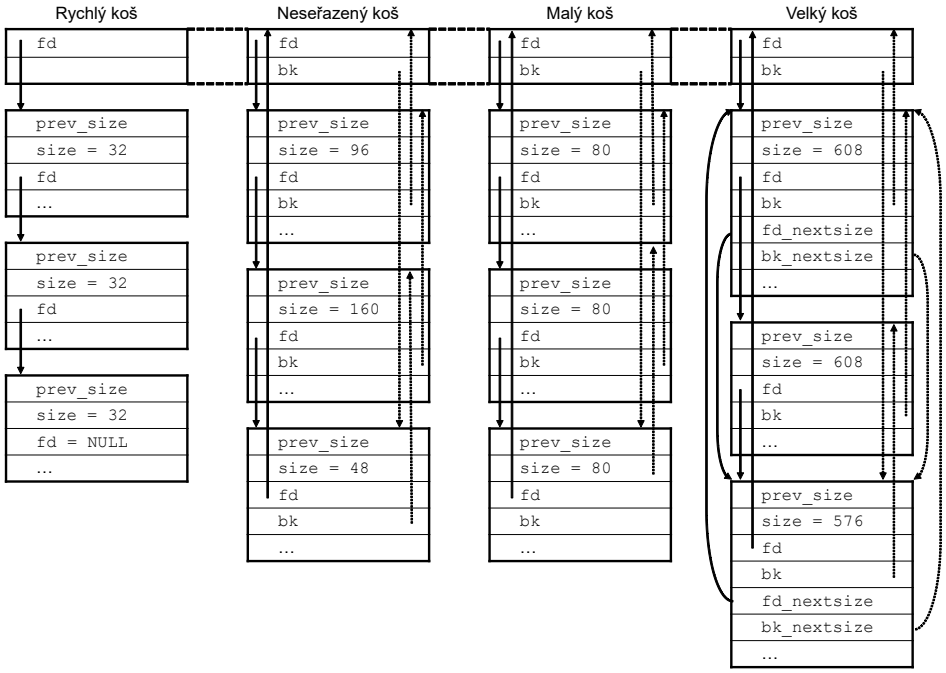
Ve chvíli, kdy je znám ukazatel na jeden paměťový kus, je možné díky `mchunk_size` dopočítat, kde se v paměti nachází kus následující. Ukazatel stačí jednoduše zvětšit o přečtený počet bytů. Díky boundary tag je možné provést to samé, akorát v opačném směru.

Následně dojde k uvolnění kusu B a započne zpětná konsolidace. Paměťový manažer přečte v kusu B bit P, čímž zjistí, že kus A je uvolněný. Tím pádem `mchunk_prev_size` obsahuje platnou hodnotu. Odečtením této hodnoty od ukazatele na kus B se získá ukazatel na kus A a dojde ke sloučení do jediného kusu AB. Kus A je zároveň vyjmut z jeho koše (pojem koš viz sekce 2.3).

Obdobně proběhne i dopředná konsolidace. Paměťový manažer opět zjistí, zda je následující kus, tedy C, uvolněn. Tentokrát k tomu ale potřebuje přečíst bit P, který se nachází až v kusu D. Poté dojde ke sloučení do jediného kusu ABC a kus C je vyřazen z koše.

2.3 Koše

Uvolněné kusy představují potenciální volnou paměť, kterou je možné v budoucnu použít. Paměťový manažer potřebuje být pro obsloužení alokačního požadavku schopen co nejrychleji vyhledat uvolněný kus vhodné velikosti a převést jej zpět do alokovaného stavu. Za tímto účelem ukládá uvolněné kusy



Obrázek 2.3: Druhy košů a jejich vnitřní struktura

do takzvaných *košů*. Koš je spojový seznam uvolněných kusů, které všechny patří do jedné arény. Existuje několik druhů košů. Ty se od sebe liší tím, jak velké kusy jsou do nich ukládány, a také způsobem, jakým paměťový manažer s kusy v těchto koších pracuje. Všechny druhy košů a jejich vlastnosti budou dále diskutovány. Obrázek 2.3 zachycuje vnitřní strukturu jednotlivých druhů košů.

Rychlé koše

Rychlé koše slouží k ukládání menších paměťových kusů. Operace nad rychlými koši jsou optimalizovány pro rychlost, a to i za cenu větší fragmentace. Kusy mohou být v případě potřeby přesunuty z rychlého koše do jiného druhu koše.

Rychlých košů je celkem 10, v každém z nich jsou kusy stejné velikosti. Ve výchozím nastavení paměťový manažer používá pouze 7 rychlých košů. Je ale možné nastavit, aby využíval 9 z nich. Poslední koš je nevyužitý a je přítomen pravděpodobně z důvodu zarovnání. Ve 32bitovém prostředí jsou do prvního koše (na indexu 0) vkládány kusy o velikosti 16 bytů, do druhého koše 24 bytů, do třetího koše 32 bytů a tak dále. V 64bitovém prostředí jsou v prvním koši kusy o velikosti 32 bytů, ve druhém 48 bytů atd, ve třetím 64 bytů atd.

Rychlé koše jsou implementovány formou jednosměrného spojového seznamu. Paměťový manažer udržuje v rámci arény pro každý koš ukazatel na první uvolněný kus. Konkrétně se jedná o pole `fastbinsY` ve struktuře `malloc_state`, kde jeden prvek pole odpovídá jednomu koši. To je buďto zmíněný ukazatel na uvolněný kus, nebo `NULL` v případě, že je koš prázdný. Zbytek spojového seznamu je realizován pomocí `fd` ve struktuře `malloc_chunk`.

Se spojovým seznamem se pracuje v LIFO pořadí. U kusů v rychlých koších neprobíhá konsolidace, při svém uvolnění proto nenastavují bit P následujícího kusu, ani jeho `mchunk_prev_size`. Existují nicméně dvě situace, za kterých ke konsolidaci dojde. První z nich je alokace takové velikosti, že pro ni nemůže být použit žádný z kusů v rychlých ani malých koších. Druhým případem je uvolnění kusu velikosti přesahující `FASTBIN_CONSOLIDATION_THREDHOLD`, což je ve výchozím nastavení 64 KiB.

V obou zmíněných situacích jsou proiterovány veškeré kusy všech rychlých košů, nad každým kusem proběhne konsolidace a následně dojde k přesunu výsledného kusu do neseřazeného koše.

Normální koše

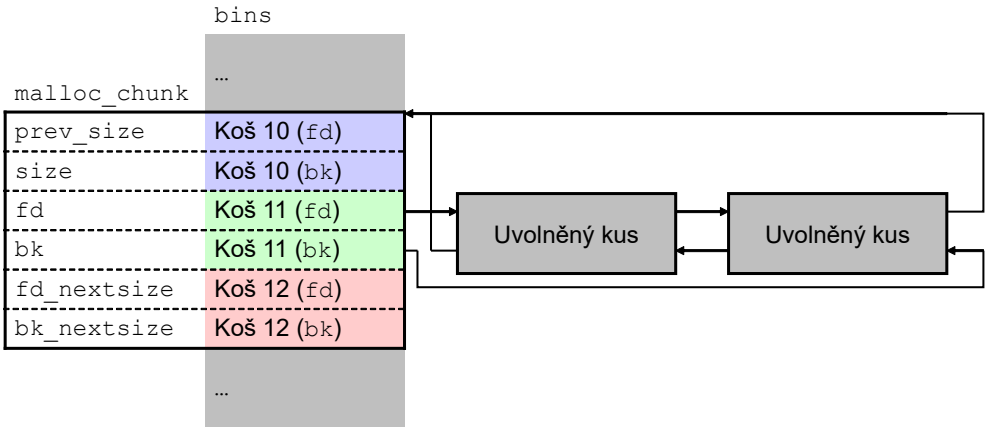
Vedle rychlých košů existují dále *normální koše*. Ty jsou pro změnu optimalizovány na nízkou fragmentaci paměti a kusy v nich uložené podléhají konsolidaci. Normální koše se dále dělí na velké, malé a jeden speciální tzv. neseřazený koš. Kusy v normálních koších jsou udržovány v obousměrně zřetězených kruhových spojových seznamech. Pro jejich reprezentaci se požívají ukazatele `fd` a `bk`.

Tyto dva ukazatele rovněž slouží jako přístupový bod do daného koše. Jsou ukládány do pole `bins` ve struktuře `malloc_state`. Díky znalosti rozložení struktury `malloc_chunk` je možné tuto dvojici v paměti zarovnat tak, že se jeví jako samostatný paměťový (pseudo)kus. Pseudokus díky tomu může být sám použit jako prvek kruhového spojového seznamu. Tato situace je pro koš č. 11 ilustrována obrázkem 2.4.

Neseřazený koš

Neseřazený koš je jediným z normálních košů, do kterého funkce `free` ukládá uvolněné kusy. Ve snaze ušetřit čas je kus při svém uvolnění a konsolidaci umístěn na počátek neseřazeného koše a to bez ohledu na jeho velikost.

Zařazení kusu do správného koše podle velikosti je odloženo až do chvíle dalšího volání `malloc`. Funkce `malloc` začne procházet kusy v neseřazeném koši a hledá takový, který by přesně odpovídal alokačnímu požadavku. Během této operace zařazuje nevyhovující kusy na správné místo v rámci velkých či malých košů. Podaří-li se v neseřazeném koši najít vhodně velký kus, pak tento nemusí být nikam zařazován a `malloc` jej může rovnou použít.



Obrázek 2.4: Reprezentace koše jako pseudokusu, podle [27]

Malé koše

Malé koše obsahují podobně jako rychlé koše pouze kusy stejné velikosti. Malých košů je celkem 62 a obsahují kusy v rozmezí velikosti 16 až 508 bytů (32bitové prostředí), respektive 32 až 1008 bytů (64bitové prostředí). Kusy jsou do malých košů rozděleny podobně, jako u rychlých košů, tedy v prvním koši jsou v 32bitovém prostředí kusy velikosti 16 bytů, ve druhém 24 bytů atd.

Velké koše

Velké koše jsou určeny pro kusy velikosti větší nebo rovny 512 bytům (1024 bytům v 64 bitovém prostředí). Velkých košů je 63, přičemž v jednom koši se mohou vyskytovat kusy různých velikostí. Kusy v rámci jednoho velkého koše jsou seřazeny podle velikosti v sestupném pořadí a jsou pro ně relevantní i položky `fd_nextsize` a `bk_nextsize`.

Na rozdíl od `fd`, což je ukazatel na další uvolněný kus v rámci spojového seznamu (který může mít stejnou velikost, jako kus aktuální), `fd_nextsize` ukazuje na další kus menší velikosti. Podobně pro `bk_nextsize`. Tímto způsobem je možné ve spojovém seznamu rychleji vyhledat správné místo, na které má být vložen nový uvolněný kus, protože tím odpadá potřeba iterovat přes kusy stejné velikosti. Ukazatele `fd_nextsize` a `bk_nextsize` jsou nastavovány jen pro první kus dané velikosti v rámci seznamu. Další kusy stejné velikosti mají tyto ukazatele nastaveny na `NULL`.

Jak již bylo zmíněno, každý z velkých košů obsahuje kusy v určitém rozmezí velikosti. Pro prvních 32 košů je toto rozmezí v 32bitovém prostředí vždy 64 bytů. První velký koš tedy obsahuje kusy velikosti 512 – 568 B, druhý koš kusy velikosti 576 – 632 B apod.

Následuje 16 košů s rozmezím 512 bytů, 8 košů s rozmezím 4096 bytů, 4 koše s rozmezím 32 768 bytů a 2 koše s rozmezím 262 144 bytů. Kusy, které

jsou natolik velké, že se nevejdou do žádného ze zmíněných košů, jsou umístěny do posledního koše.

2.4 Arény

Na nejvyšší úrovni paměťového manažeru stojí *arény*. Jedna aréna popisuje část VAS patřící pod správu paměťového manažeru a skládá se z jedné nebo více souvislých oblastí adresního prostoru. Každá z arén je popsána instancí struktury `malloc_state`, její souvislé paměťové oblasti pak jednotlivými instancemi struktury `heap_info`.

Důležité položky obou struktur budou detailněji diskutovány v následujících podsekcích. Pro získání základního přehledu je však vhodně již nyní představit seznam jejich deklarovaných členů.

Struktura `malloc_state`

Člen	**Popis**
`mutex`	Serializace přístupu do arény.
`flags`	Ukládá informaci, zda je paměťový prostor arény souvislý.
`have_fastchunks`	Označuje, zda jsou v rychlých koších nějaké volné kusy.
`fastbinsY`	Rychlé koše.
`top`	Představuje nealokovanou paměť arény. Ukazuje na tzv. vrchní kus.
`last_remainder`	Ukazuje na zbývající prostor po posledním rozděleném kusu.
`bins`	Pole pro normální koše.
`binmap`	Bitová mapa pro normální koše.
`next`	Ukazatel na další arénu.
`next_free`	Ukazatel na další arénu, která není využívána žádným vláknem.
`attached_threads`	Počet vláken využívajících tuto arénu.
`system_mem`	Velikost celé arény v bytech.
`max_system_mem`	Používá se při změně velikosti arény.

Struktura `heap_info`

Člen	Popis
`ar_ptr`	Ukazatel na arénu, jejíž je daná paměťová oblast součástí.
`prev`	Ukazatel na předchozí oblast.
`size`	Velikost oblasti v bytech.
`mprotect_size`	Počet bytů oblasti, které mají přístupová práva pro čtení a zápis. Používá se v kontextu rozšiřování a zkracování oblasti.
`pad`	Padding, zajišťuje správné zarovnání.

Hlavní a vláknové arény

V glibc existuje statická proměnná typu `malloc_state`, která uchovává informace o tzv. *hlavní aréně*. Oblast reprezentovaná hlavní arénou je rozprostřena přes celý haldový segment a paměťový manažer spravuje její velikost pomocí systémového volání `sbrk`. Haldový segment je souvislý, hlavní aréna proto nepoužívá instanci `heap_info`.

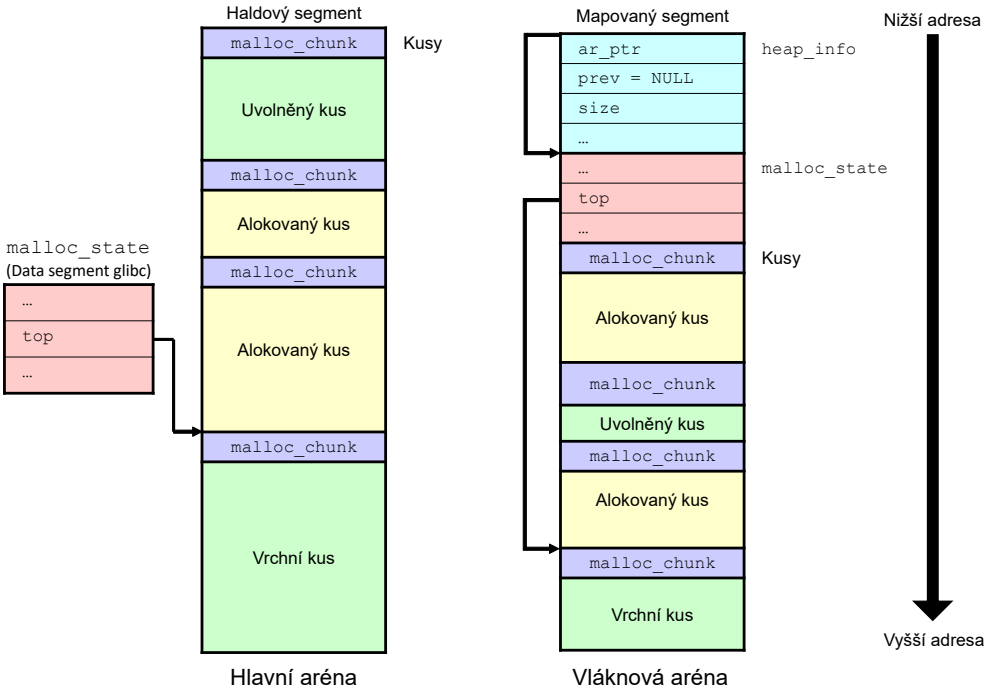
Haldový segment je v procesu pouze jeden, pro ostatní arény získává glibc paměť od operačního systému skrze systémové volání `mmap`. Arény vytvořené tímto způsobem se nazývají *vláknové arény*. Do jedné vláknové arény může spadat více namapovaných oblastí, každá na svém začátku obsahuje instanci `heap_info`. Instance `heap_info` jsou skrz `prev` vzájemně propojeny do spojového seznamu. První z oblastí má `prev` nastaveno na `NULL`.

Ve vláknové aréně existuje jediná instance struktury `malloc_state` umístěná bezprostředně za instancí `heap_info` v první namapované oblasti arény. Arény vzájemně tvoří kruhový spojový seznam, struktura `malloc_state` obsahuje ukazatel `next` odkazující na další arénu. Poslední z vláknových arén odkazuje zpět na arénu hlavní.

Obrázek 2.5 ukazuje rozvržení paměti pro hlavní arénu (levá část obrázku) a vláknovou arénu s jedinou namapovanou oblastí (pravá část obrázku). Na obrázku 2.6 je naopak znázorněna situace, kdy má vláknová aréna namapováno více segmentů.

Synchronizační mechanismy

Ve struktuře `malloc_state` existuje `mutex`, kterým je kontrolován přístup do jedné arény z více vláken. Ačkoliv určité operace, jako například práce s rychlými koši, mohou probíhat atomicky, většina ostatních operací vyžaduje uzamčení arény. To je jedním z důvodů pro existenci vláknových arén.



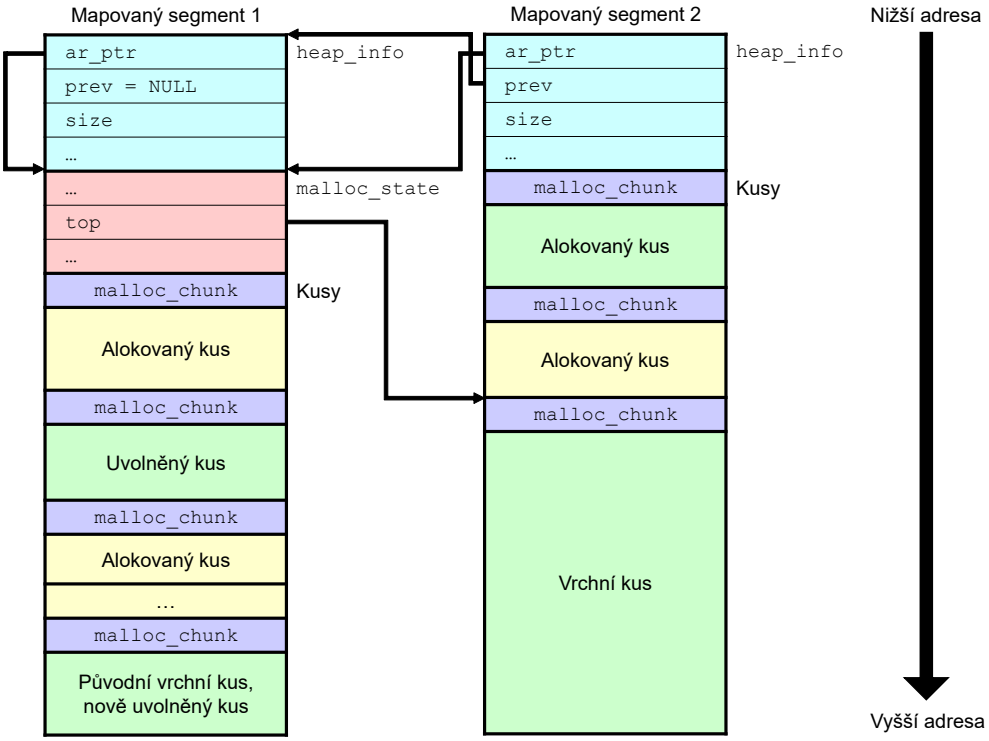
Obrázek 2.5: Hlavní a vláknová aréna, podle [30]

Pokud by veškeré paměťové operace probíhaly pouze nad hlavní arénou, představovalo by toto místo ve vícevláknových programech vzhledem k četnosti volání paměťových funkcí velmi úzké hrdlo celého programu. V praxi je poměrně častá situace, kdy si jedno vlákno alokuje paměť, využije ji a následně ji samo i uvolní. Má-li každé vlákno jednu dedikovanou arénu, mohou podobné operace probíhat bez toho, aniž by se vlákna navzájem blokovala.

Má to ale dva háčky. Prvním z nich je, že v programu může existovat velké množství vláken, ale v jeden moment je počet běžících vláken limitován počtem jader procesoru. Vyhrazovat jednu arénu pro každé vlákno se tak stává zbytečné a časově i paměťově nehospodárné. Knihovna proto zavádí limit na počet arén. Ve výchozím nastavení je u 32bitových prostředí maximální počet arén roven dvojnásobku počtu jader procesoru, pro 64bitová prostředí se jedná o osminásobek.

Druhým aspektem je, že uživateli samozřejmě nic nebrání v tom, aby alokace prováděl v jednom vlákně a uvolnění ve vlákně jiném. Uvolněné kusy paměti musí být navráceny do stejné arény, odkud byly alokovány. Synchronizační mechanismy proto musí být v paměťovém manažeru přítomny, arény je nutné zamykat, ale díky existenci více arén se lze alespoň částečně vyhnout vzájemnému blokování se všech vláken v programu.

Informace o počtu vláken, která danou arénu využívají, je uchovávána v proměnné `attached_threads`. Tato hodnota je inkrementována při vzniku



Obrázek 2.6: Vláknová aréna s více mapovanými oblastmi, podle [30]

vlákna a jeho přiřazení ke konkrétní aréně. Při zániku vlákna je naopak dekrementována. Vlákno může počas běhu programu používanou arénu měnit. To se děje ve chvílích, kdy je potřeba provést novou alokaci, ale je již vyčerpána veškerá dostupná paměť arény a nedaří se od operačního systému získat paměť další (např. selže volání `sbrk`). V takovém případě se vlákno pokusí přejít do jiné arény a alokovat paměť odtud.

Umístění kusů v paměti arény

Zbylá část prostoru za řídícími strukturami `malloc_state` a `heap_info` je vyhrazena pro jednotlivé kusy v alokovaném nebo uvolněném stavu. Poslední kus v aréně se nazývá vrchní kus a představuje nealokovanou paměť arény. Zpočátku zabírá celý zbytek arény. Na tento kus odkazuje proměnná `top` v příslušné instanci struktury `malloc_state`. Vrchní kus není nikdy zařazen do žádného z košů.

Nepodaří-li se při alokaci najít vhodný kus v některém z košů, je vrchní kus rozdělen na dva kusy. První z nich má velikost odpovídající požadavku alokace, je uveden do alokovaného stavu a navrácen uživateli. Druhý z kusů se stává novým vrchním kusem.

Není-li vrchní kus dostatečně veliký, paměťový manažer zvětší oblast arény.

U hlavní arény stačí posunout program break, u vláknové arény je potřeba namapovat oblast novou. V případě mapování je nová oblast anotována instancí `heap_info` a z původního vrchního kusu se stane kus uvolněný. Ten je pak umístěn do vhodného koše. Nově vzniklý vrchní kus se už bude nacházet v nově namapované oblasti. V aréně proto bude opět existovat právě jeden vrchní kus.

Při požadavku na alokaci malé velikosti vstupuje do hry `last_remainder`. Nepodaří-li se najít žádný uvolněný kus přesně odpovídající požadované velikosti, je možné alokační požadavek obsloužit rozdělením některého z větších kusů. Zbylá část kusu, tedy ta, která nebyla navrácena uživateli, se stává `last_remainder` a je vložena do neseřazeného koše. Následují-li další malé alokace, je možné `last_remainder` znovu a znovu dělit. Tím je podpořena datová lokalita – série po sobě jdoucích malých alokací bude do paměti umístěna na jedno souvislé místo.

Koše v aréně

Jak již bylo dříve zmíněno, v poli `fastbinsY` ve struktuře `malloc_state` se nacházejí rychlé koše. Pro rychlejší identifikaci, zda nějaké kusy v rychlých koších existují, je v téže struktuře současně uchovávána booleovská hodnota `have_fastchunks`.

Obdobné platí i pro normální koše v poli `bins` a bitovou mapu `binmap`. Zatímco `have_fastchunks` je společná pro všechny rychlé koše, `binmap` drží informaci o přítomnosti uvolněných kusů pro každý z normálních košů zvlášť.

2.5 Lokální cache

Od verze glibc 2.26 má každé vlákno pro uvolněné kusy svou vlastní *thread local cache* (zkráceně *tcache*). Ke kusům v tcache lze přistupovat bez toho, aniž by bylo nutné zamykat celou arénu. Ve vícevláknových aplikacích přináší tcache značný výkonnostní benefit.

Tcache je organizována podobně, jako rychlé koše – obsahuje pole `entries`, jeden prvek pole ukládá ukazatel na první prvek jednosměrného spojového seznamu kusů stejné velikosti. Se spojovým seznamem se opět pracuje v LIFO pořadí a jejich počet je omezen hodnotou `TCACHE_MAX_BINS`. Ve výchozím nastavení je tato hodnota 64. Do prvního tcache koše jsou ukládány nejmenší možné kusy, s rostoucím indexem roste i velikost ukládaných kusů. Pro 64bitové prostředí to znamená, že v `entries[0]` jsou umístěny kusy velikosti 32 B, v `entries[1]` kusy velikosti 48 B, v `entries[63]` kusy velikosti 1040 B.

Rozdíl oproti rychlým košům je v tom, že ukazatele v rámci spojového seznamu neukazují na další kus jako na strukturu `malloc_chunk`, ale ukazují přímo na oblast uživatelských dat. Počet kusů v každém tcache koši je udržován v poli `counts` a je limitován hodnotu `TCACHE_FILL_COUNT`. Ta je ve výchozím nastavení rovna 7.

Kusy v tcache mají při alokaci přednost před kusy v rychlých koších, jejich velikost však musí přesně odpovídat velikosti alokačního požadavku. Pokud pro požadovanou velikost neexistuje v tcache žádný odpovídající kus, paměťový manažer provede alokaci obvyklým způsobem z košů v aréně.

2.6 Algoritmy paměťových funkcí

Nyní již byly představeny všechny potřebné struktury a mechanismy, které paměťový manažer knihovny GNU C interně používá. Získané poznatky lze zkombinovat a uvést základní kroky jednotlivých paměťových funkcí.

Algoritmus `malloc`

1. Existuje-li v tcache kus přesně odpovídající požadované velikosti, je navrácen uživateli.

2. Pokud je požadavek dostatečně velký, je zavolána funkce `mmap`, paměť je získána přímo od operačního systému a navrácena volajícímu.

3. Obsahuje-li rychlý koš odpovídající velikosti nějaký uvolněný kus, je použit tento. Je-li k dispozici kusů více, je doplněna tcache.

4. Obsahuje-li malý koš odpovídající velikosti nějaký uvolněný kus, je použit tento. I v tomto případě je doplněna tcache.

5. Odpovídá-li požadovaná velikost velkému koši, je provedena konsolidace kusů v rychlých koších.

6. Nebyl-li doposud nalezen vhodný kus, začne se procházet neseřazený koš. Kusy v něm jsou konsolidovány a následně rozřazovány do odpovídajících malých či velkých košů. Najde-li se během tohoto procesu kus správné velikosti, procházení se ukončí a kus je navrácen.

7. Odpovídá-li požadovaná velikost alokace velkému koši, je prohledáván odpovídající velký koš a následující velké koše. Nalezne-li se během této operace volný kus, je navrácen.

8. Nachází-li se stále nějaké kusy v rychlých koších, jsou zkonsolidovány a algoritmus se vrací do kroku 4.

9. Nebyl-li stále nalezen vhodný uvolněný kus, dojde k rozdělení vrchního kusu na dva s případným rozšířením arény.

Algoritmus `free`

1. Je-li v odpovídajícím koši tcache volné místo, je kus umístěn sem.

2. Pokud je kus dostatečně malý, je umístěn do rychlého koše.

3. Jedná-li se o kus dříve alokovaný pomocí `mmap`, zavolá se `munmap` a kus je navrácen operačnímu systému.

4. Ve zbylých případech proběhne konsolidace. Uvolněný kus může být spojen i s vrchním kusem. V takovém případě se `top` přesouvá na začátek uvolněného kusu.

5. Jinak je kus umístěn do neseřazeného koše.

6. Pokud má kus větší velikost než `FASTBIN_CONSOLIDATION_THRESHOLD`, je navíc provedena konsolidace kusů v rychlých koších. Je-li nyní vrchní kus dostatečně velký, je část paměti navrácena operačnímu systému.

Algoritmus `realloc`

1. Je-li `ptr == NULL`, je volání ekvivalentní `malloc`.

2. Pokud je `size == 0`, funkce provede (re)alokaci nejmenšího možného kusu. To neplatí, je-li definováno `REALLOC_ZERO_BYTES_FREES`. V takovém případě se provede `free`.

3. Byl-li realokovaný kus dříve získán funkcí `mmap`, je postupně provedena posloupnost `malloc`, `memcpy`, `munmap`. To neplatí, pokud systém podporuje `mremap`. V takovém případě je realokace přenechána na operačním systému.

4. Jedná-li se o realokaci na menší velikost, je kus rozdělen na dva. První z nich začíná na původní adrese a je navrácen, druhý z nich je vložen zpět do arény jako uvolněný kus.

5. Pokud se jedná o realokaci s cílem rozšířit aktuální kus, je zkontrolován stav kusu následujícího. Pokud následuje uvolněný (nebo vrchní) kus a dohromady s aktuálním kusem mají dostatečnou velikost, jsou tyto dva kusy sloučeny. Po sloučení mohou být opět rozděleny stejným mechanismem, jako v předchozím bodu, a funkce navrátí původní ukazatel.

6. Jinak se použije posloupnost `malloc`, `memcpy`, `free`.

2.7 Sledování stavu paměťového manažeru

Fungování paměťového manažeru glibc lze částečně ovlivnit funkcí `mallopt` [33], případně nastavením proměnné prostředí `GLIBC_TUNABLES` [4, kap. 37]. To je výhodné ve chvílích, kdy je potřeba vyladit chování paměťového manažeru přesně pro účely konkrétního spouštěného programu.

Pro tuto práci jsou nicméně zajímavější prostředky, které glibc nabízí pro sledování stavu paměťového manažeru. Následuje seznam funkcí knihovny z hlavičkového souboru `<malloc.h>`, které může profiler využít pro získání dodatečných informací o dynamických alokacích paměti. Jejich popis vychází z manuálových stránek a zkoumání příslušných částí kódu knihovny.

`struct mallinfo2 mallinfo2();` [34]

Funkcí `mallinfo2` lze získat souhrnný náhled na stav arén. Návratovou hodnotou je instance stejnojmenné struktury.

Struktura `mallinfo2`

Člen	**Popis**
`arena`	Součet velikostí (`system_mem`) všech arén.
`ordblks`	Počet uvolněných kusů v normálních koších ve všech arénách. Zahrnuje i vrchní kus arény.
`smblks`	Počet uvolněných kusů v rychlých koších ve všech arénách.
`hblks`	Aktuální počet kusů alokovaných přes `mmap`.
`hblkhd`	Velikost aktuálně alokovaných kusů přes `mmap`.
`usmblks`	Nepoužívá se, je nastaven na 0.
`fsmblks`	Součet velikostí kusů v rychlých koších ve všech arénách.
`uordblks`	Součet velikostí kusů v alokovaném stavu ve všech arénách. Rozdíl `arena` - `fordblks`.
`fordblks`	Součet velikostí kusů v uvolněném stavu a vrchního kusu ve všech arénách.
`keepcost`	Velikosti vrchního kusu hlavní arény.

`void malloc_usable_size (void* ptr)` [32]
: Tato funkce byla zmíněna již dříve. Z ukazatele na uživatelská data alokovaného kusu dokáže přečíst jejich velikost. Její velkou výhodou je rychlost – na rozdíl od `mallinfo2` a dalších nevyžaduje zamykání arén.

`void malloc_stats()` [35]
: Funkce slouží k výpisu statistik na standardní chybový výstup. Pro každou arénu je vypsána její velikost a velikost kusů v alokovaném stavu. Jedná se o obdobu `arena` a `uordblks` z `mallinfo2`, jen pro každou arénu zvlášť. Na závěr funkce vypíše součet `arena` a `uordblks` všech arén a dále maximální počet bloků a bytů, které kdy byly v jeden okamžik alokovány přes `mmap`.

`int malloc_info (int options, FILE* stream)` [36]
: Funkce vytvoří textový řetězec ve formát XML, a zapíše jej do výstupního proudu `stream`. Argument `options` musí být 0. XML řetězec obsahuje informace o všech arénách.

Kapitola 3

Dostupné profilovací nástroje

Profilování představuje proces hledání míst v programu vhodných pro jeho optimalizaci. Program lze profilovat v mnoha směrech. Lze sledovat četnost volání jeho funkcí, čas strávený v konkrétních místech programu, interakce s cache procesoru nebo způsob, jakým program využívá paměť. Profilování je forma dynamické analýzy, probíhá tedy za běhu profilovaného programu. Nástroj, který profilování provádí, se nazývá *profiler*.

Profilerů existuje velké množství. Liší se od sebe svou oblastí zaměření, podporovanými platformami, principem fungování. Jednou z kategorií profilerů jsou profilery haldy. Ty se snaží odhalit možnou neefektivitu při práci s haldou. Dokáží například odhalit místa, kde program provádí příliš velké nebo příliš časté dynamické alokace. Tato kapitola představí nejrozšířenější zástupce z kategorie profilerů haldy dostupných pro Linuxové prostředí.

3.1 Valgrind

Valgrind [37] je populární sada nástrojů pro debugování a profilování programů. Dokáže detekovat problémy spjaté se správou paměti, odhalit časově závisle chyby ve vícevláknových programech, lze jej použít k odhalení funkcí, kde program tráví příliš mnoho času. Obsahuje rovněž nástroje pro sledování dynamických alokací.

Valgrind podporuje několik platforem, mezi nimi například x86/Linux či AMD64/Linux. Jde o volně šiřitelný software pod licencí GNU GPL v.2, poprvé vydaný v roce 2002. V době psaní této práce je k dispozici ve verzi 3.17.0 z 19. března 2021.

Princip fungování

Valgrind pracuje na principu dynamické binární instrumentace. Architektonicky se skládá ze dvou částí – jádra Valgrindu a nástrojového pluginu. Tyto dvě části dohromady tvoří jeden celek označovaný jako nástroj. Jádro po-

skytuje nízkoúrovňovou infrastrukturu pro podporu instrumentace programu. Obsahuje součásti jako JIT kompilátor, nízkoúrovňový paměťový manažer, plánovač vláken, stará se o zpracování signálů.

Jádro vykonává disassembling strojového kódu debugovaného programu a transformuje jej do VEX IR, což je interní mezikód RISCového typu. Tento proces probíhá just-in-time po malých blocích kódu z programu, které jsou aktuálně spouštěny. Vygenerovaný mezikód je následně předán nástrojovému pluginu, který provede instrumentaci – do mezikódu přidá své instrukce, které mají být při spuštění provedeny. Takto upravený mezikód je následně vrácen zpět jádru Valgrindu, které jej přeloží zpět do strojového kódu a spustí. Výsledný překlad je uložen do kódové cache a při dalším spuštění bloku může být znovupoužit. [38, 39]

Nástroje

Valgrind obsahuje několik nástrojů, které je možné použít. Každý nástroj obecně slouží k jinému účelu, funkcionalita některých z nich se však do určité míry překrývá. Mezi dostupné nástroje patří například: [40]

Memcheck Slouží k detekci problémů se správou paměti a je primárně určen pro debugování C/C++ programů. Memcheck zaznamenává všechna volání paměťových funkcí a kontroluje veškeré přístupy programu do paměti. Na základě toho je schopen odhalit, zda program:

- Nepřistupuje k paměti, ke které by přistupovat neměl. Jedná se například o oblasti, které nebyly dosud alokovány, nebo naopak již byly uvolněny.
- Nepoužívá neinicializované hodnoty.
- Řádně uvolňuje alokovanou paměť. Detekce úniků paměti, dvojího uvolnění.
- Nepoužívá překrývající se paměťové bloky u funkcí jako `memcpy` apod.

Cachegrind Tento nástroj se používá pro sledování využití CPU cache. Detailně simuluje I1, D1 a L2 cache procesoru a dokáže odhalit místa, kde dochází ke cache miss.

Callgrind Jedná se o rozšíření nástroje Cachegrind. Poskytuje stejné informace, jako Cachegrind, a navíc sleduje veškerá volání funkcí v programu a vytváří graf volání (anglicky *callgraph*). K vizualizaci výstupních dat je možné použít program KCachegrind.

Helgrind, DRD Tyto nástroje nacházejí uplatnění při debugování vícevláknových aplikací. Dokáží detekovat problémy jako jsou uváznutí nebo časově závislé chyby.

Massif Slouží k profilování haldy. Umožňuje nalézt místa v programu, kde dochází k dynamickým alokacím paměti. Tento nástroj bude podrobněji popsán níže.

DHAT Slouží rovněž k profilování haldy. Jeho primárním účelem je zachytit přístupy programu do dynamicky alokované paměti – tzn. sleduje operace čtení/zápis do alokovaného paměťového bloku. Dokáže odhalit situace, kdy program:

- Alokuje paměť, krátce ji použije, ale uvolní ji až na konci běhu.
- Provádí v rychlém sledu alokace a uvolnění, aniž by jeden blok zůstal alokovaný po delší dobu.
- Alokuje více paměti, než kolik následně reálně využije.
- Alokuje paměť, kterou vůbec nevyužije, nebo nadměrně používá jen některé její části.

Massif

Nástroj Massif [41] slouží k profilování haldy. V pravidelných časových intervalech pořizuje snímky haldy, které po skončení běhu programu dají dohromady detailní přehled o vývoji množství alokované paměti v čase. Volitelně dokáže sledovat i velikost programového zásobníku.

Data, která Massif za běhu programu sbírá, jsou ukládána do textového souboru. Není-li specifikováno jinak, soubor má název `massif.out.<pid>`. Obsah souboru je lidsky čitelný, pro lepší vizualizaci nasbíraných dat je však možné z příkazové řádky použít program `ms_print`. Alternativně lze použít GUI aplikaci `massif-visualizer` [42].

Snímky, které nástroj Massif generuje, se dělí do tří kategorií – *normální*, *detailní* a *špičkové*.

Normální snímky

Normální snímky uchovávají pouze základní informace o využití paměti. Příklad obsahu tohoto snímku přímo z výstupního souboru (tedy bez využití vizualizačních nástrojů) je následující:

```
#-----------
snapshot=2
#-----------
time=2032
mem_heap_B=2000
mem_heap_extra_B=32
mem_stacks_B=0
heap_tree=empty
```

Záznam `snapshot` říká, o kolikátý snímek se jedná, `time` udává čas pořízení snímku. Čas může být měřen v počtu provedených instrukcí, milisekundách nebo v počtu alokovaných bytů (to je případ uvedeného příkladu).

Následuje `mem_heap_B` a `mem_heap_extra_B`, oba záznamy udávají počet bytů alokovaných v době mezi předcházejícím a aktuálním snímkem. Konkrétně `mem_heap_B` je paměť skutečně alokovaná uživatelem, tedy například dvě volání `malloc(1000)`. Záznam `mem_heap_extra_B` pak reprezentuje přidanou režii paměťového manažeru. Ta zahrnuje anotaci alokovaných kusů jejich velikostí a případné extra zarovnávací byty.

Záznam `mem_stacks_B` udává velikost programového zásobníku. Je-li zásobníků více (vícevláknové programy), `mem_stacks_B` obsahuje součet velikostí všech zásobníků. Ve výchozím nastavení není tento údaj zaznamenáván, protože citelně zpomaluje běh profileru. V uvedeném příkladu je tato hodnota proto rovna nule.

Poslední záznam `heap_tree` reprezentuje kategorii snímku. Pro normální snímky je roven hodnotě `none` a indikuje, že na následujícím řádku souboru se už bude nacházet další snímek. U ostatních druhů snímků za tímto řádkem následují další data aktuálního snímku.

Detailní snímky

Dalším druhem snímků jsou snímky detailní. Ve výchozím nastavení je detailní každý desátý snímek. Jeho obsah ve výstupním souboru může vypadat například následovně:

```
#-----------
snapshot=24
#-----------
time=30344
mem_heap_B=10000
mem_heap_extra_B=24
mem_stacks_B=0
heap_tree=detailed
n3: 10000 (heap allocation functions) malloc/new/new[], ...
 n2: 8000 0x1086E1: g (in /home/toman/massif/a.out)
  n1: 4000 0x1086F7: f (in /home/toman/massif/a.out)
   n0: 4000 0x108740: main (in /home/toman/massif/a.out)
  n0: 4000 0x108745: main (in /home/toman/massif/a.out)
 n1: 2000 0x1086F2: f (in /home/toman/massif/a.out)
  n0: 2000 0x108740: main (in /home/toman/massif/a.out)
 n0: 0 in 1 place, below massif's threshold (1.00%)
```

Svým obsahem je detailní snímek zpočátku stejný, jako snímek normální. Obsahuje základní údaje o využití paměti programem. Odlišná je až hodnota `heap_tree`, která je v tomto případě nastavena na `detailed`. Za tímto řádkem následuje strom volání alokačních funkcí. Strom zachycuje aktuální stav haldy. Stavem haldy se rozumí informace o tom, kolik paměti je aktuálně alokováno z kterých míst programu. Strom je možné číst odshora dolů.

Na prvním řádku se nachází kořen stromu reprezentující volání libovolné alokační funkce jako `malloc` či `new`. Tímto uzlem procházejí všechny dynamické alokace paměti, v uvedeném příkladu se proto ve druhém sloupci nachází hodnota 10 000, což odpovídá `mem_heap_B`.

Každý z dalších uzlů stromu je pak odsazen mezerou podle hloubky, ve které se nachází, a reprezentuje konkrétní místo v jedné funkci, odkud došlo (tranzitivně) k dynamické alokaci paměti. Začátek záznamu je anotovaný písmenem `n` a počtem potomků uzlu. Kupříkladu `n2` říká, že daný uzel má dva potomky, `n0` značí list stromu.

Cesta od kořene k listu stromu odpovídá jednomu backtrace. Z výstupu tak lze zjistit místa programu, odkud jsou alokační funkce volány. V uvedeném snímku lze například vyčíst backtrace:

```
10000 (heap allocation functions) malloc/new/new[], ...
8000 0x1086E1: g (in /home/toman/massif/a.out)
4000 0x1086F7: f (in /home/toman/massif/a.out)
4000 0x108740: main (in /home/toman/massif/a.out)
```

To odpovídá situaci, kdy funkce `main` zavolala funkci `f`, která zavolala funkci `g`, která v tomto kontextu alokovala 4000 B. Funkce `g` však byla zavolána i napřímo z `main`, kde rovněž alokovala 4000 B. Celkově je tedy `g` zodpovědná za alokaci 8000 bytů paměti, což je reflektováno uvedeným výstupem.

Špičkové snímky

Poslední kategorií snímků jsou snímky špičkové. Jak název napovídá, špičkový snímek označuje moment, kdy množství paměti alokované z haldy dosáhlo globálního maxima. Špičkový snímek je vždy nejvýše jeden a svou strukturou odpovídá snímku detailnímu – rozdíl je pouze v hodnotě `heap_tree=peak`.

Zachycená špička množství alokované paměti nemusí přesně odpovídat špičce reálné. Ve výchozím nastavení Massif zachytí takovou špičku, jejíž velikost se od špičky skutečné liší maximálně o 1 %.

Nastavitelné parametry

Chování profileru Massif lze v mnoha ohledech modifikovat pomocí parametrů. Mezi významné z nich patří například:

`--heap=<yes|no> default: yes`
: Specifikuje, zda se má provádět profilování haldy.

`--stacks=<yes|no> default: no`
: Specifikuje, zda se má provádět profilování zásobníku. Kombinací s předchozím parametrem je tedy možné například vypnout profilování haldy a sledovat pouze velikost zásobníku. Ve výchozím nastavení je profilování zásobníku vypnuté, protože signifikantně zpomaluje běh programu.

`--pages-as-heap=<yes|no> default: no`
: Massif ve výchozím nastavení sleduje alokace na úrovni paměťových funkcí. Nastavením tohoto parametru je možné nahradit výchozí chování tak, že profiler bude sledovat přímo alokace stránek paměti provedené systémovým voláním `mmap` apod.

`--depth=<number> default: 30`
: Maximální hloubka stromu alokací.

`--alloc-fn=<name>`
: Dovoluje specifikovat jména vlastních alokačních funkcí. To je užitečné například ve chvílích, kdy program využívá obalující funkce nad funkcí `malloc`. Uvedené obalující funkce pak nejsou zahrnuty do stromu volání, protože nepřinášejí žádnou užitečnou informaci.

`--peak-inaccuracy=<m.n> default: 1.0`
: Upravuje přesnost, s jakou Massif zachytí špičku alokované paměti. Čím nižší hodnota, tím delší doba profilování.

`--time-unit=<i|ms|B> default: i`
: Nastavení časové jednotky pro profilování. Čas je možné měřit v instrukcích (`i`), milivteřinách (`ms`) nebo v počtu alokovaných bytů (`B`).

`--detailed-freq=<n> default: 10`
: Frekvence, s jakou mají být pořizovány detailní snímky. Ve výchozím nastavení je každý desátý snímek detailní.

3.2 Heaptrack

Heaptrack [43] je profiler haldy určený pro operační systémy Linux a FreeBSD. První verze tohoto profileru vyšla v roce 2017 (v1.0.0) a v době psaní této práce je k dispozici ve verzi 1.3.0 ze září 2020. Heaptrack je volně šiřitelný nástroj pod licencí GNU LGPL v. 2.1. Jeho autorem je Millian Wolff, který na svých webových stránkách [44] popisuje princip fungování profileru.

Heaptrack zaznamenává všechny dynamické alokace a uvolnění paměti, které profilovaný program za běhu provede. Každá alokace je anotována svým backtrace, což následně umožňuje najít v programu její přesný původ. Heaptrack je možné použít například za účelem nalezení míst, kde program:

- Alokuje příliš mnoho paměti – ať už velikostí jednotlivých alokací, nebo četností volání alokačních funkcí.
- Alokuje paměť, kterou následně neuvolní (únik paměti).
- Vytváří dočasné alokace. To jsou alokace paměti, které jsou okamžitě následovány jejich dealokací.

Princip fungování

Jádro profileru Heaptrack tvoří sdílená knihovna `libheaptrack_preload.so` (dále jen Heaptrack knihovna). Heaptrack knihovna je při startu nahrána do profilovaného programu pomocí proměnné prostředí `LD_PRELOAD`. Knihovna nahrazuje funkce `malloc` a spol. svou vlastní implementací. Proto kdykoliv dojde v profilovaném programu k volání alokační funkce, je namísto výchozích funkcí paměťového manažeru ve skutečnosti zavolána jejich náhrada z Heaptrack knihovny. Náhradní implementace se v principu skládá ze dvou kroků:

1. Zavolání skutečné funkce paměťového manažeru a navrácení výsledku uživateli. Adresa příslušné funkce je získána pomocí funkcí dynamického linkeru.

2. Zpracování a uložení informace o volání. Heaptrack knihovna vytvoří backtrace a spolu se záznamem o proběhnuvší alokaci jej zapíše do dočasného výstupního souboru.

Výstupní soubor tak obsahuje záznamy o všech alokacích, které v profilovaném programu proběhly, a to v pořadí, ve kterém k nim došlo. Heaptrack zároveň do výstupního souboru v pravidelných časových intervalech (10 ms) zapisuje časové razítko a RSS. Pro tento účel využívá vlastní dedikované vlákno [43, `libheaptrack.cpp`]. Díky tomu je možné později zobrazit vývoj paměťové stopy profilovaného programu v čase.

Vedle profilovaného programu v systému zároveň běží proces s názvem `heaptrack_interpret` (dále jen interpreter). Jeho úkolem je další zpracování nasbíraných dat a vytvoření finálního výstupního souboru. Backtrace získaný Heaptrack knihovnou má podobu návratových adres ze zavolaných funkcí. Interpreter se stará o jeho symbolizaci, tj. transformaci do podoby jmen volaných funkcí.

Heaptrack knihovna a interpreter spolu komunikují přes zmíněný dočasný soubor. Tím je ve skutečnosti pojmenovaná roura (`mkfifo`), do které Heaptrack knihovna zapisuje data. Dočasný soubor je přesměrován na standardní vstup interpreteru, který příchozí data čte a zpracovává. Po zpracování interpreter na svůj standardní výstup vypisuje obsah finálního souboru. A konečně, standardní výstup interpreteru je přesměrován do `gzip`, který vytváří komprimovaný finální výstupní soubor.

Finální výstupní soubor už obsahuje všechny informace potřebné pro prezentaci výsledků profilování. Tzn. finální soubor může být přenesen na jiný stroj a tam uživatel může k náhledu jeho obsahu použít některý z vizualizačních nástrojů profileru Heaptrack. Pomineme-li kompresi, finální i dočasné soubory jsou čistě textové. Jejich přesný formát však není oficiálně zdokumentován.

Heaptrack podporuje *runtime attaching*. Je schopen se napojit na již běžící proces a začít jej profilovat. Tato funkcionalita je velmi užitečná pro profilování dlouze běžících procesů typu serverová aplikace. Lze si představit situaci, kdy se po měsíci běhu takové aplikace začnou projevovat problémy spjaté s dynamicky alokovanou pamětí, například je podezření na úniky paměti. Aplikaci ale nelze vypnout, spustit profiler a čekat další měsíc, až se problémy znovu objeví. Mnohem lepší je se k aplikaci připojit a okamžitě zjistit, co se děje. A právě to Heaptrack umožňuje.

Součástí profileru Heaptrack je proto knihovna `libheaptrack_inject.so`, která v již běžícím procesu dokáže zachytit volání alokačních funkcí a spustit namísto výchozích funkcí paměťového manažeru vlastní kód. Samotného připojení se k profilovanému programu je dosaženo pomocí debuggeru GDB. Další detaily podrobně vysvětluje sám autor Heaptracku ve své přednášce [22].

Ke spuštění profileru Heaptrack uživatelsky přívětivou cestou se používá shellový skript s názvem `heaptrack`. Ten zastřešuje všechny výše uvedené části projektu, stará se o správné spuštění programu s `LD_PRELOAD`, spuštění GDB a připojení se k běžícímu procesu atd.

Analýza nasbíraných dat

Heaptrack nabízí k analýze a vizualizaci nasbíraných dat dva nástroje. Prvním z nich je konzolový `heaptrack_print`, druhým je GUI aplikace s názvem `heaptrack_gui`. Práce s GUI aplikací je z uživatelského pohledu přívětivější a v porovnání s konzolovým nástrojem nabízí více možností, jakými lze na nasbíraná data nahlížet.

Pro představení základních informací, které je možné o profilovaném programu díky profileru Heaptrack zjistit, je nadále podrobněji popsán pouze konzolový nástroj `heaptrack_print`. Jeho argumentem je výstupní soubor profileru a dále je možné specifikovat například:

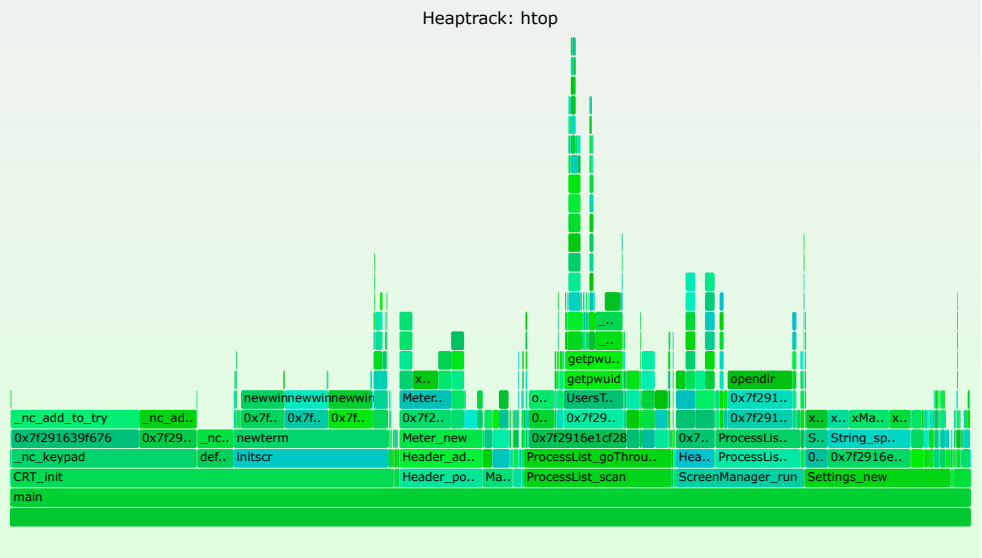
`--print-allocators=<1|0> default: 1`
: Vypíše místa v programu, odkud byly alokační funkce nejčastěji volány.

`--print-peaks=<1|0> default: 1`
: Vypíše místa v programu, odkud bylo v jeden moment alokováno nejvíce paměti.

`--print-temporary=<1|0> default: 1`
: Vypíše místa v programu, kde proběhlo nejvíce dočasných alokací.

`--print-leaks=<1|0> default: 0`
: Vypíše místa v programu, odkud byla alokována paměť, která následně nebyla uvolněna.

`--print-overall-allocated=<1|0> default: 0`
: Vypíše místa v programu, odkud bylo celkově alokováno nejvíce paměti (nebere v potaz uvolnění).



Obrázek 3.1: Ukázka Flamegraph

`--print-histogram <file>`
: Vytvoří alokační histogram. Alokační histogram obsahuje počty alokací, které požadovaly konkrétní počet bytů. Výstupem `heaptrack_print` je v tomto případě textový soubor, který na každém řádku obsahuje dvojici čísel. První číslo udává velikost alokace, druhé číslo celkový počet alokací této velikosti.

`--print-flamegraph <file>`
: Vytvoří textový soubor, který je dále možné externím nástrojem[6] převést do podoby obrázku ve formátu SVG. Flamegraph slouží pro přehledné znázornění stromu volání alokačních funkcí. Ukázka tohoto grafu na obrázku 3.1 byla vytvořena profilováním několika vteřin běhu programu `htop`.

`--print-massif <file>`
: Vytvoří textový soubor, který svým formátem odpovídá výstupu profileru Massif. Pro prezentaci nasbíraných dat tak lze použít i vizualizační nástroje právě pro Massif.

Na konci výstupu `heaptrack_print` se dále nacházejí souhrnné informace, jako je celková doba běhu profilovaného programu, celkový počet alokovaných bytů, počet volání alokačních funkcí atd.

[6] `https://github.com/brendangregg/FlameGraph`

Výhody v porovnání s Massif

- Při profilování profilerem Heaptrack je spouštěn původní kód programu, bez překladu do mezikódu a zpět. Přidaná režie se navíc projeví jen při volání alokačních funkcí. V místech, kde se alokace neprovádí, běží program bez zpomalení.
- Výstup profileru Heaptrack je podrobnější. Data nejsou za běhu nijak agregována, do výstupu je zaznamenána každá alokace a každé uvolnění paměti.
- Heaptrack podporuje runtime attaching.

Nevýhody v porovnání s Massif

- Bez úprav zdrojového kódu profilovaného programu nedokáže Heaptrack sledovat alokace prováděné přes systémová volání.
- Heaptrack neumožňuje profilování zásobníku.

3.3 Profiler v gperftools

V roce 2005 společnost Google zveřejnila svou interně využívanou implementaci paměťového manažeru s názvem TCMalloc jako součást projektu Google Performance Tools. Od té doby se projekt rozdělil na dva:

1. TCMalloc [11], který obsahuje pouze paměťový manažer. Jedná se současnou implementaci, kterou Google využívá ve svých aplikacích.
2. gperftools [10], který vedle paměťového manažeru dále obsahuje 3 profilovací nástroje – profiler haldy, heap checker a profiler CPU. První zmíněný bude podrobněji popsán níže.

Nutno podotknout, že implementace paměťových manažerů v těchto dvou projektech nejsou identické. První z projektů je primárně určen pro interní potřeby společnosti Google, nemusí proto dbát na obecnou přenositelnost kódu napříč různými platformami. Paměťový manažer v gperftools je naopak určen pro širší použití v externích projektech.

Heap profiler

Pomocí profileru haldy [10, `docs/heapprofile.html`] v gperftools je možné odhalit místa v programu, kde dochází k alokaci velkého množství paměti, a dokáže určit, kolik paměti je v daný moment z jakého místa programu alokováno. Případně je jím možné lokalizovat úniky paměti.

Profiler je součástí knihovny `libtcmalloc.so`, ve které se nachází i implementace paměťového manažeru. Aby se profiler dostal do paměti procesu a mohl pracovat, je nutné nutné knihovnu k programu buďto dynamicky přilinkovat, nebo při spuštění použít `LD_PRELOAD`. Obě varianty mají za následek, že dojde k náhradě výchozího paměťového manažeru za TCMalloc.

Samotné profilování lze následně zapnout nastavením proměnné prostředí `HEAPPROFILE` na hodnotu obsahující název výstupního souboru. Alternativou je volání funkcí `HeapProfilerStart` a `HeapProfilerStop` přímo z programu. Ukázka spuštění profileru:

```
$ HEAPPROFILE=/tmp/my_out.hprof \
LD_PRELOAD=/usr/local/lib/libtcmalloc.so ./my_program
```

Profiler periodicky vytváří výstupní soubory a ukládá je na disk. Při způsobu spuštění uvedeném výše budou postupně vznikat soubory:

```
/tmp/my_out.hprof.0001.heap
/tmp/my_out.hprof.0002.heap
/tmp/my_out.hprof.0003.heap
...
```

Každý ze souborů obsahuje jeden snímek stavu haldy. Soubory mají textový formát a obsahují seznam míst programu (backtrace), ze kterých kdy byla alokována paměť. Každé místo je anotováno dalšími informacemi:

- Počet doposud neuvolněných alokací.
- Velikost neuvolněné paměti.
- Celkový počet alokací.
- Celkový počet alokovaných bytů.

Obsažené backtraces nejsou symbolizované. Obsahují tedy pouze návratové adresy funkcí. Spolu s tím je v souboru uveden seznam namapovaných oblastí (výpis `/proc/self/maps`). Symbolizace se provádí až během analýzy a vizualizace nasbíraných dat. Za tímto účelem existuje v gperftool Perl skript `pprof`. Chování profileru je možné ovlivnit nastavením několika proměnných prostředí:

`HEAP_PROFILE_ALLOCATION_INTERVAL default: 1073741824 (1Gb)`
: Profiler vytvoří nový snímek pokaždé, když program v souhrnu alokuje specifikovaný počet bytů (od posledního snímku).

`HEAP_PROFILE_INUSE_INTERVAL default: 104857600 (100 Mb)`
: Profiler vytvoří nový snímek pokaždé, když se velikost paměti alokované programem zvýší o specifikovaný počet bytů.

`HEAP_PROFILE_TIME_INTERVAL default: 0`
: Profiler vytvoří nový snímek po uplynutí specifikovaného počtu vteřin od posledního snímku.

`HEAPPROFILESIGNAL default: disabled`
: Profiler vytvoří nový snímek ve chvíli, kdy proces obdrží specifikovaný signál.

`HEAP_PROFILE_MMAP default: false`
: Specifikuje, zda má profiler sledovat i alokace provedené přes systémová volání. Tj. profiler sleduje jak `malloc` a spol., tak i `mmap` a spol.

`HEAP_PROFILE_ONLY_MMAP default: false`
: Specifikuje, zda má profiler sledovat pouze alokace provedené přes systémová volání. Tj. profiler sleduje pouze `mmap` a spol.

`HEAP_PROFILE_MMAP_LOG default: false`
: Specifikuje, zda má profiler na standardní chybový výstup logovat volání `mmap` a `munmap`.

Při porovnání profileru v gperftools s dříve uvedenými nástroji má tento profiler svým výstupem blíže k nástroji Massif. Rovněž agreguje data a jeho výstup má podobu snímků stavu haldy. Každý snímek obsahuje téměř totožné informace, jako detailní snímek z nástroje Massif. Stejně tak umožňuje profilování na úrovni systémových volání. Co naopak neumožňuje, je sledování přesné velikosti programového zásobníku.

Co týče způsobu spouštění profilovaného programu, má gperftools pro změnu blíže k profileru Heaptrack. Profiler je rovněž součástí sdílené knihovny a při běhu je spouštěn skutečný strojový kód programu, bez překladu do IR a zpět. Nepodporuje však runtime attaching.

3.4 Další nástroje

memusage

Nástroj memusage [45, 46] je součástí glibc. K profilování využívá samostatnou knihovnu s názvem `libmemusage.so`, která je do procesu nahrána přes `LD_PRELOAD`. Dokáže zjistit počet volání alokačních funkcí, detekovat úniky paměti, zobrazit alokační histogram, vytvořit graf zobrazující spotřebu paměti v čase. Umí sledovat paměť nejenom na haldě, ale i na zásobníku. Lze jej použít i pro sledování alokací na úrovni systémových volání.

MALT

Dalším z profilerů haldy je Malloc Tracker [47], zkráceně MALT. Ten rovněž využívá techniku s `LD_PRELOAD`, jeho výstupem je soubor ve formátu

JSON nebo LUA. Případně dokáže vygenerovat soubor kompatibilní s výstupem Callgrind. MALT nabízí velmi detailní náhled na sesbíraná data. Lze například zjistit počty alokací z jednotlivých vláken programu, do alokačního histogramu je zahrnut i časový aspekt, je možné si prohlížet zdrojové kódy programu s vyznačenými řádky, které jsou zodpovědné za volání alokačních funkcí. MALT dále umožňuje sledovat velikost zásobníku a umí určit počet globálních proměnných v programu.

Memoro

Pro velmi detailní profilování haldy je možné použít nástroj Memoro [48]. Vedle zachycení volání alokačních funkcí dokáže Memoro u dynamicky alokované paměti monitorovat i operace čtení a zápis. Funguje na principu statické instrumentace, k čemuž využívá infrastruktutu LLVM/Clang. Profilovaný program musí být překompilován speciálně upravenou verzí překladače Clang. Na místa, kde program přistupuje do paměti, jsou vložena volání funkcí profileru. Tím se profiler o takovém přístupu dozví a může jej dále zpracovat. O principu fungování profileru Memoro podrobně pojednává článek [49].

Kapitola 4

Získání backtrace

Klíčovou funkcionalitou profileru haldy je lokalizace míst v programu, kde probíhají volání alokačních funkcí. K tomu potřebuje být profiler schopen vytvořit backtrace. V této kapitole budou diskutovány možnosti, jakými to lze provést. Velkým zdrojem inspirace pro tuto kapitolu je přednáška od autora profileru Heaptrack [22] a dále projekt Backward-cpp [50].

V úvodu kapitoly se společně pokusíme vytvořit backtrace „vlastnoručně“. Vytvoříme k tomu C++ program, který dokáže bez použití dalších knihoven sám sobě přečíst a vypsat návratové adresy ze zásobníku. Získaný výstup programu následně s pomocí jiných běžně dostupných nástrojů převedeme do podoby lidsky čitelných jmen zavolaných funkcí.

Ve zbytku kapitoly budou pro jednotlivé kroky celého procesu představeny dostupné knihovny a funkce, s jejichž pomocí lze proces získání backtrace provést robustní programovou cestou.

4.1 Vlastnoruční backtrace

Ze sekce 1.4 již víme, jakým způsobem je na CPU architektuře x86 organizován zásobník. Víme, že existuje ukazatel na začátek aktuálního zásobníkového rámce uchovávaný v registru `ebp`. Víme, že na jím odkazované adrese se nachází záloha ukazatele na začátek předchozího zásobníkového rámce. Víme, že na offsetu `ebp + 4` se nachází návratová adresa do volající funkce. To je dostatek informací k tomu, abychom si návratové hodnoty ze zásobníku vypsali sami.

Přesuňme se na x86_64. Registr `ebp` se změní na `rbp`. Návratová adresa bude ležet na offsetu `ebp + 8`. Ostatní principy zůstanou nezměněné. Můžeme tak přijít s programem zachyceným v kódu 4.1.

```
1  #include <iostream>
2  #define stack_read(rbp, offset) \
3      reinterpret_cast<uintptr_t*>((rbp))[(offset)]
4
5  uintptr_t anchor = 0;
6
7  void backtrace()
8  {
9      register uintptr_t rbp_ asm ("rbp");
10     uintptr_t rbp = rbp_;
11     do {
12         std::cout << std::hex
13                   << stack_read(rbp, 1) << '\n';
14         rbp = stack_read(rbp, 0);
15     } while (anchor > rbp);
16 }
17
18 void baz() { backtrace(); }
19 void bar() { baz(); }
20 void foo() { bar(); }
21
22 int main()
23 {
24     uintptr_t dummy;
25     anchor = reinterpret_cast<uintptr_t>(&dummy);
26     foo();
27     while(true);
28 }
```

Kód 4.1: Čtení návratových adres ze zásobníku

Úkolem programu je vypsat návratové adresy do funkcí `baz`, `bar`, `foo`, `main`. Funkce `backtrace` získá na řádcích 9–10 hodnotu registru `rbp`. Ten v tu chvíli odpovídá jejímu zásobníkovému rámci. Následně je v `do-while` cyklu přečtena a vypsána návratová adresa do volající funkce. Řádek 14 přečte zálohu ukazatele na předešlý zásobníkový rámec a cyklus pokračuje do další iterace.

Ve funkci `main` je vytvořena lokální proměnná `dummy`, jejíž adresa je uložena do globální proměnné `anchor`. Pomocí `anchor` je ukončen cyklus ve funkci `backtrace` – cyklus díky tomu nepokračuje za rámec funkce `main`. Program následně zavolá `foo->bar->baz->backtrace`. Poté uvázne v nekonečné smyčce.

Program použijeme čistě pro demonstrativní účely – využívá znalosti dané architektury procesoru a vlastnosti konkrétního překladače. Z pohledu jazyka C++ má nedefinované chování a nejedná se jakkoliv robustní nebo přenositelné řešení.

Program nyní zkompilujeme překladačem GCC (v. 8.4.0). Pro vytvoření debugovacích informací použijeme přepínač `-g` a kompilovat budeme bez optimalizací. Následně program spustíme.

Můžeme získat následující výstup:

```
$ g++ -g backtrace.cpp -o backtrace
$ ./backtrace
7f7ec8000989
7f7ec8000995
7f7ec80009a1
7f7ec80009cb
^Z
[1]+  Stopped                 ./backtrace
```

Dobrou zprávou je, že program nespadnul a souhlasí nám počet získaných návratových adres. Špatnou zprávou je, že výstup programu je kvůli ASLR pro každý jeho běh jiný.

Aby nám návratové adresy byly k něčemu užitečné, musíme za běhu programu zjistit ještě informace o jeho *modulech*. Modulem se rozumí samotný spustitelný soubor nebo sdílená knihovna. Konkrétně potřebujeme pro každý modul znát jeho *bázovou adresu* a cestu k modulu. Bázová adresa označuje počáteční adresu oblasti ve VAS procesu, kam byl modul dynamickým linkerem namapován. Proces máme pozastaven, jeho VAS si můžeme nechat vypsat:

```
$ cat /proc/`pidof backtrace`/maps
...
7f7ec8000000-7f7ec8001000 r-xp 00000000 00:00 204636    /home/toman/backtrace
7f7ec8201000-7f7ec8202000 r--p 00001000 00:00 204636    /home/toman/backtrace
7f7ec8202000-7f7ec8203000 rw-p 00002000 00:00 204636    /home/toman/backtrace
...
```

Hledanou bázovou adresou je `7f7ec8000000`. Tuto adresu odečteme od dříve získaných návratových adres. Tím dostaneme offsety `989`, `995`, `9a1`, `9cb`. Nyní již můžeme běžící proces ukončit.

Získané offsety následně předáme nástroji `addr2line` [51]. Přepínačem `-f` řekneme, že chce vypsat jméno funkce, do které návratová adresa spadá. Přepínačem `-C` zapneme *name demangling*, čímž dostaneme lidsky čitelná jména funkcí. Pomocí `-e ./backtrace` zadáme cestu k našemu programu. Pokud by získané návratové adresy patřily do sdílené knihovny, zadávali bychom cestu k této knihovně. Nástroj `addr2line` nám dá následující výstup:

```
$ addr2line -f -C -e ./backtrace 989 995 9a1 9cb
baz()
/home/toman/backtrace.cpp:18
bar()
/home/toman/backtrace.cpp:19
foo()
/home/toman/backtrace.cpp:20
main
/home/toman/backtrace.cpp:27 (discriminator 1)
```

Výstup vypadá slibně. Zobrazila se nám jména funkcí, které jsme zavolali, včetně informací o souboru se zdrojovým kódem, kde se nachází jejich definice. Pokud bychom program zkompilovali bez přepínače `-g`, nástroj by stále nalezl

jména funkcí, ale namísto cesty k souboru se zdrojovým kódem by se zobrazilo „`??:?`“.

Nástroj `addr2line` nám ukazuje i čísla řádků. Číslo řádku u volající funkce označuje místo, kde bude běh programu pokračovat po návratu z funkce volané. To lze vidět u získaného čísla řádku ve funkci `main`. Funkce `main` je v tomto kontextu volající, `foo` je volaná. K volání dochází na řádku 26. Po návratu z `foo` bude tedy program pokračovat řádkem 27.

To není příliš praktické. Potřebovali bychom zjistit místo, kde k volání došlo, ne kam se z něj program vrátí. Můžeme se podívat, jak situace vypadá v assembleru:

```
$ objdump -d ./backtrace
...
0000000000000998 <_Z3foov>:
998:   55                      push   %rbp
999:   48 89 e5                mov    %rsp,%rbp
...
00000000000009a4 <main>:
9a4:   55                      push   %rbp
9a5:   48 89 e5                mov    %rsp,%rbp
...
9c6:   e8 cd ff ff ff          callq  998 <_Z3foov>
9cb:   eb fe                   jmp    9cb <main+0x27>
...
```

Vidíme, že instrukce `callq`, kterou je zavolána `foo`, je umístěna na adresách `9c6`–`9ca`. My jsme ale nástroj `addr2line` žádali o zpracování `9cb`. Tam se už ale nachází `while` cyklus realizovaný instrukcí `jmp`. Chceme-li získat číslo řádku volání `foo`, musíme zadat některou z adres, která odpovídá instrukci `callq`. Od získané návratové adresy zde postačí odečíst 1 a `addr2line` už začne reportovat řádek volání, nikoliv řádek návratu. Z jeho výstupu zároveň zmizí `(discriminator 1)`.

4.2 Programový backtrace

Při implementaci vlastního profileru bude nutné celý předchozí postup provést programovou cestou. Krátké shrnutí kroků, které je potřeba ke získání kompletního backtrace zajistit:

1. Zjištění návratových adres ze zavolaných funkcí v programu. Tento proces se často označuje jako *stack unwinding*.
2. Zjištění rozložení paměti procesu. Každý získaný IP je potřeba přiřadit k jeho modulu. To zahrnuje nalezení cesty k modulu a bázové adresy.
3. Provedení symbolizace. Se znalostí IP a bázové adresy se zjistí adresa symbolu v modulu. Pomocí ní pak jméno odpovídajícího symbolu. Jsou-li obsaženy debugovací informace, půjde získat i jména zdrojových souborů, příslušné řádky kódu a informace o inlined funkcích.

4. Posledním krokem je úprava jmen symbolů do lidsky čitelné podoby. Jména symbolů budou ve formátu, v jakém je vidí linker. Bude nutné provést name demangling. Tím se získají jména funkcí tak, jak je vidí překladač. To stále nemusí být vhodný formát. C++ funkce, ve kterých figurují šablony, budou mít dlouhé a nepřehledné názvy. Je vhodné nabídnout uživateli možnost šablony odfiltrovat.

S výjimkou zkrácení jmen C++ funkcí jsou všechy kroky příliš komplexní na to, aby mělo smysl se pokoušet je psát od nuly. V následujících sekcích proto budou představeny dostupné knihovny a funkce, které s konstrukcí backtrace pomohou.

4.3 Stack unwinding

Funkce `backtrace` v glibc

Knihovna GNU C nabízí k provedení stack unwinding funkci s příznačným názvem `backtrace` [52]. Deklarace funkce se nachází v hlavičkovém souboru `<execinfo.h>`. Jak ukazuje kód 4.2, funkce je velmi jednoduchá na použití.

```
#include <execinfo.h>
#include <iostream>

void print_backtrace()
{
    static constexpr int MAX_SIZE = 64;
    void* addresses[MAX_SIZE];
    int n = backtrace(addresses, MAX_SIZE);
    for (int i = 0; i < n; ++i) {
        std::cerr << addresses[i] << '\n';
    }
}
```

Kód 4.2: Stack unwinding, `backtrace` v glibc

Podle dostupné dokumentace funkce pracuje na podobném principu, jako dřívější vlastnoruční čtení adres ze zásobníku. Pro svou činnost vyžaduje, aby na zásobníku byly uloženy ukazatele na jednotlivé rámce. Překladač však může v rámci optimalizace `-fomit-frame-pointer` provést eliminaci těchto ukazatelů. Tím se naboural celý algoritmus čtení adres ze zásobníku.

Na rozdíl od vlastnoručního procházení dokáže funkce `backtrace` z glibc tuto situaci detekovat a další procházení zásobníku ukončí. Program nespadne, ale seznam získaných IP nemusí být úplný. Použití této funkce v profileru by obecně vyžadovalo, sestavení programu a všech jeho sdílených knihoven s přepínačem `-fno-omit-frame-pointer`.

Funkce pro obsluhu výjimek

Stack unwinding je přirozená funkcionalita jazyka C++. Dochází k němu kdykoliv, když program vyhodí výjimku. Kompilátor definuje sadu funkcí, které se o celý mechanismus obsluhy výjimek starají. V případě GCC jsou poskytovány knihovnou `libgcc.a` nebo `libgcc_s.so.1` [53, kap. 4.5]. Lze mezi nimi nalézt funkce `_Unwind_Backtrace` a `_Unwind_GetIPInfo`. Jejich deklarace se nachází v hlavičkovém souboru `<unwind.h>` a pro získání návratových adres je lze použít způsobem demonstrovaným v kódu 4.3.

```
#include <unwind.h>
#include <iostream>

void print_backtrace()
{
    _Unwind_Backtrace([](_Unwind_Context* ctx, void*)
        -> _Unwind_Reason_Code {
          int dummy;
          _Unwind_Ptr ip = _Unwind_GetIPInfo(ctx, &dummy);
          std::cerr << std::hex << ip << '\n';
          return _URC_NO_REASON;
    }, nullptr);
}
```

Kód 4.3: Stack unwinding, funkce pro obsluhu výjimek

Backtrace získaný pomocí těchto funkcí již není závislý na existenci ukazatelů na rámce zásobníku. Namísto toho získává informace o zásobníkových rámcích z DWARF `.eh_frame` tabulky [54]. To je mnohem spolehlivější metoda. Překladač tuto tabulku vytváří pro C i C++ programy, zůstane navíc přítomna i při zapnutých optimalizacích. Při překladu jazyka C by musel uživatel vytvoření této tabulky explicitně vypnout. To lze udělat přepínačem `-fno-asynchronous-unwind-tables`. U jazyka C++ je pak potřeba vypnout i mechanismy pro zpracování výjimek (`-fno-exceptions`).

Projekt libunwind

Knihovna libunwind [55, 56] definuje přenosné C API, kterým lze požadované funkcionality snadno dosáhnout. Knihovna interně provádí cachování, což dělá stack unwinding v porovnání s dříve zmíněnými metodami velmi rychlým.

Pro profiler je toto důležitá vlastnost, protože stack unwinding musí být proveden při každém volání alokační funkce. Jakákoliv neefektivita v této fázi se citelně podepíše na celkové době běhu profilovaného programu.

V hlavičkovém souboru `<libunwind.h>` je deklarována funkce s názvem `unw_backtrace`, kterou lze použít stejným způsobem, jako `backtrace` v glibc. Její použití ukazuje kód 4.4.

```
#define UNW_LOCAL_ONLY
#include <libunwind.h>
#include <iostream>

void print_backtrace()
{
    static constexpr int MAX_SIZE = 64;
    void* addresses[MAX_SIZE];
    int n = unw_backtrace(addresses, MAX_SIZE);
    for (int i = 0; i < n; ++i) {
        std::cerr << addresses[i] << '\n';
    }
}
```

Kód 4.4: Stack unwinding, `unw_backtrace` v libunwind

Knihovna podporuje i *remote unwinding*. To je funkcionalita umožňující získat backtrace z jiného procesu. Definicí makra `UNW_LOCAL_ONLY` před `#include <libunwind.h>` lze tuto funkcionalitu vypnout, což dělá lokální stack unwinding ještě rychlejším.

Použití `unw_backtrace` nevyžaduje ukazatele na zásobníkové rámce, bude proto fungovat i v optimalizovaných programech. Jednoduchost použití, rychlost a přesnost získaných výsledků dělá z libunwind velmi vhodnou volbu pro použití v profileru. Ostatně, používá ji i Heaptrack [22].

4.4 Přiřazení IP k modulu

Dalším krokem celého procesu je přiřazení získaných IP k jejich modulům. Tento krok musí stále probíhat v době běhu profilovaného programu. Není už ale nutné, aby probíhal při každém volání alokační funkce. Rozložení modulů v rámci VAS zůstává stejné po celou dobu běhu programu. Postačí proto, když profiler toto rozložení zjistí a zaznamená na začátku běhu. IP, které následně nasbírá, už půjde k modulům přiřadit i po ukončení programu.

To ovšem neplatí v případě, že program používá funkce dynamického linkeru z libdl, jako jsou `dlopen` a `dlclose` [57]. Pomocí nich může program za běhu otevírat a uzavírat další sdílené knihovny. Sdílená knihovna je při svém otevření namapována na nějaké místo ve VAS a tam přetrvá až do svého uzavření. Po tuto dobu může profiler zjistit cestu ke knihovně i její bázovou adresu. Jakmile je ale knihovna zavřena, tato informace se ztrácí. Pro profiler to znamená, že kromě paměťových funkcí potřebuje sledovat i volání `dlopen`, `dlclose` a patřičně aktualizovat seznam modulů.

Ve vlastnoručním řešení byl pro účely získání informací o VAS využit soubor `maps` z `proc` filesystému. Tento soubor je možné z programu obvyklým způsobem otevřít — `fopen("/proc/self/maps")` — a jeho obsah patřičně naparsovat.

Funkce `dl_iterate_phdr`

Elegantnější řešení nabízí funkce `dl_iterate_phdr` [58]. Ta je deklarována v hlavičkovém souboru `<link.h>`. Funkce projde všechny moduly v procesu a pro každý z nich jednou zavolá `callback`, který je jejím argumentem. Ukázka použití této funkce se nachází v kódu 4.5.

```
#include <link.h>
#include <iostream>

void print_modules()
{
    dl_iterate_phdr([](dl_phdr_info* info, size_t /* size */,
        void* /*user data */) -> int {
        const char* name = info->dlpi_name;
        void* base = reinterpret_cast<void*>(info->dlpi_addr);

        // name is empty for the executable itself
        if (!name || !name[0]) {
            name = "self";
        }

        std::cerr << "Module "       << name <<", "
                  << "Base address " << base << '\n';
        return 0;
    }, nullptr);
}
```

Kód 4.5: Získání seznamu modulů – `dl_iterate_phdr`

Tímto způsobem lze získat stejné informace, jako ze souboru `maps`. Pro přiřazení IP do modulu stačí vzít získaný seznam bázových adres, vzestupně jej seřadit a najít první adresu, která je větší než zkoumaný IP.

Funkce `dladdr`

Alternativu nabízí funkce `dladdr` [59]. Ta je deklarovaná v hlavičkovém souboru `<dlfcn.h>`. Jejím argumentem je přímo IP a dále ukazatel na instanci `Dl_info`. Funkce se přímo pokusí přiřadit IP do některého z aktuálně namapovaných modulů. Pokud uspěje, instance `Dl_info` bude obsahovat:

Struktura `Dl_info`

Člen	Popis
`dli_fname`	Cesta k modulu.
`dli_fbase`	Bázová adresa modulu.
`dli_sname`	Jméno symbolu, do nějž IP spadá. Tzn. pokud IP bude návratová adresa do funkce `foo`, pak `dli_sname` bude obsahovat řetězec `"foo"`.
`dli_saddr`	Adresa nalezeného symbolu – `&foo`.

Funkce indikuje úspěch nenulovou návratovou hodnotou. V takovém případě jsou validní `dli_fname` a `dli_fbase`. Jméno symbolu a jeho adresa jsou platné pouze tehdy, obsahují-li nenulový ukazatel.

Funkce `dladdr` má částečný přesah do fáze symbolizace. Nejedná se však o příliš spolehlivý způsob, jakým symbolizaci provést. Nedokáže totiž určit jména lokálních symbolů – je třeba kompilovat s `-export-dynamic`. Funkce zároveň nepracuje s debugovacími informacemi. S její pomocí tak profiler nezjistí soubory se zdrojovým kódem, čísla řádků atd.

4.5 Symbolizace

Knihovna glibc v hlavičkovém souboru `<execinfo.h>` nabízí funkci s názvem `backtrace_symbols` [52], které je možné předat pole získaných návratových adres. Funkce pro každou z adres najde odpovídající jméno symbolu. Funkce je jednoduchá na použití, nedává však dostatečně podrobné výsledky. Zároveň musí být volána v době běhu programu.

Ve vlastnoručním řešení byla symbolizace řešena nástrojem `addr2line`. Ten lze použít i po terminaci programu. Jediným předpokladem je, že jsou k dispozici v nezměněné podobě všechny moduly, které program za běhu používal.

Profiler pochopitelně může `addr2line` nebo podobný nástroj k symbolizaci využít. Lze si však vyrobit i svůj vlastní program, který symbolizaci zvládne provést. Obecně bude jeho vstupem trojice: cesta k modulu, bázová adresa, IP[7]. Výstupem bude (za ideálních okolností):

- Název funkce, do které IP spadá.
- Cesta k souboru se zdrojovým kódem, kde byla funkce definována.
- Číslo řádku a sloupce ve zdrojovém souboru, které odpovídá IP. Např. pokud IP spadá do funkce `foo` a tento IP je návratovou adresu funkce `bar`, je snahou určit řádek kódu, kde `foo` zavolala `bar`.

[7] Případně rovnou rozdíl IP a bázové adresy.

- Informace o inlinovaných funkcích. Pro posloupnost volání funkcí `foo -> bar -> baz -> qux` může kompilátor při optimalizacích provést inlining `bar` a `baz` do `foo`. Ve strojovém kódu tím pádem funkce `foo` volá rovnou `qux`. Cílem je odhalit, že k inliningu došlo, a pro každou z inlinovaných funkcí opět zjistit soubor, číslo řádku atd.

Všechny požadované informace lze z modulu vyčíst, úspěšnost však záleží na způsobu, jakým byl zkompilován. Obvykle nebývá problém s prvním bodem, tj. určení názvu funkce, kam IP spadá. Tuto informaci lze zjistit z tabulky symbolů. Ta bývá při výchozím způsobu kompilace přítomna. Získání ostatních informací už je však závislé na tom, zda modul obsahuje i debugovací informace. Aby je kompilátor vygeneroval, je potřeba tuto volbu explicitně zapnout přepínačem `-g`.

Spustitelné soubory a sdílené knihovny jsou uloženy ve formátu ELF, samotné debugovací informace pak ve formátu DWARF. Pro jejich čtení lze využít některou z dostupných knihoven:

libbfd Knihovnu Binary File Descriptor library (BFD) používá projekt GNU pro nízkoúrovňovou manipulaci s objektovými soubory. Je součástí kolekce GNU Binutils [60], která mimo jiné obsahuje GNU linker `ld`, nástroj pro zobrazení informací o objektových souborech `objdump`, nástroj pro čtení ELF souborů `readelf`, ale i již známý `addr2line`.

libdw Knihovna libdw je součástí projektu elfutils [61]. Elfutils je podobně jako GNU Binutils sada nástrojů pro čtení, vytváření a modifikaci ELF souborů. Dokáže najít a pracovat s DWARF debugovacími informacemi. Obsahuje nástroje jako `eu-objdump`, `eu-readelf`, `eu-addr2line`. Milian Wolff ve videu [22] ukazuje, jak knihovnu použít.

libelf, libdwarf Projekt elfutils dále obsahuje knihovnu libelf. Ta v kombinaci s libdwarf [62] dokáže námi požadovanou funkcionalitu rovněž nabídnout.

Nepříjemnou vlastností všech výše uvedených knihoven je náročnost jejich použití. Zmíněné knihovny slouží k obecnému čtení a manipulaci s ELF soubory a nenabízí jednoduchou cestu, jak je použít pouze pro účely symbolizace.

Projekt Backward-cpp [50] nabízí snadno použitelné C++ API pro provedení (nejen) symbolizace. Projekt dokáže pracovat se všemi výše uvedenými knihovnami. Makry preprocesoru si lze zvolit, která z knihoven má být použita. Bohužel ale nenabízí způsob, jakým symbolizaci provést až po skončení běhu programu. Při konstrukci profileru tak lze uvažovat několik možností:

1. Bude upřadnostněna jednoduchost použití a symbolizaci bude profiler provádět přes Backward-cpp za běhu, například v separátním vláknu. To pochopitelně může mít negativní dopad na dobu běhu.

2. K symbolizaci bude po ukončení běhu programu použit některý z existujících nástrojů jako `addr2line` (libbfd) nebo `eu-addr2line` (libdw). To s sebou nese nutnost znát přesný formát výstupu nástroje a umět jej naparsovat.

3. S pomocí Backward-cpp zmíněné knihovny experimentálně porovnat a zjistit, která z nich je při symbolizaci nejúspěšnější. Na jejím základě lze následně vyrobit vlastní nástroj šitý na míru potřebám profileru. Experimentální porovnání bude provedeno v následující sekci.

4.6 Porovnání symbolizačních knihoven

Pro provedení experimentu byl zvolen projekt docopt.cpp [63]. Docopt slouží pro snadné zpracování argumentů předaných programu na příkazové řádce. Jeho funkce `docopt::docopt` na základě *usage message* vygeneruje parser, kterým ověří, že předané argumenty jsou validní, a následně je ve formě mapy vrátí zpět volajícímu.

V experimentu byl Docopt sestaven jako sdílená knihovna, která byla dynamicky přilinkována k testovacímu programu `naval_fate`. Testovací program `naval_fate` slouží jako ukázkový příklad použití Docopt. Pro účely experimentu byl tento program modifikován náhradou `::operator new` za vlastní verzi. Ta ve svém výchozím nastavení prováděla dynamické alokace pomocí funkce `malloc`. Před zavolání funkce `docopt::docopt` však byla přepnuta do režimu, kdy s pomocí Backward-cpp při každém jejím zavolání vytvořila symbolizovaný backtrace. Po návratu z funkce `docopt::docopt` byla navrácena zpět do původního režimu. Tím bylo zajištěno zachycení pouze těch dynamických alokací, které měly původ v Docopt knihovně.

Docopt knihovna byla následně sestavena překladačem GCC v úrovních optimalizace `-O0`, `-O1`, `-O2`, `-O3` a `-Os`. Pro každou úroveň byla dále vytvořena i verze s debugovacími informacemi (přepínač `-g`). Všech 10 verzí knihovny Docopt bylo následně skrz modifikovanou verzi `naval_fate` testováno všemi dostupnými symbolizačními metodami, které Backward-cpp pro Linux nabízí. Program `naval_fate` byl spouštěn s argumenty `ship new FIT-CVUT`.

Výsledky experimentu jsou zobrazeny v tabulkách 4.1 a 4.2. Sloupce tabulky obsahují sledované veličiny, řádky použitou metodu symbolizace. Řádky jsou organizovány do skupin podle úrovně optimalizace. Ve všech případech došlo k 8 655 volání alokační funkce. Tolikrát byl tedy vytvořen i backtrace. Symbolizace byla provedena pro každý unikátní IP právě jednou. Pro danou úroveň optimalizace byl počet unikátních IP vždy stejný. Spolu s úrovní optimalizace je uveden v popisku skupiny řádků.

Sledované veličiny zahrnují čas, jak dlouho trvalo volání sledované funkce `docopt::docopt`, a dále výsledky symbolizace. První sloupec s názvem Funkce označuje počet IP, ke kterým se podařilo přiřadit jméno funkce z objektového

				Zdrojový kód				Inlinované funkce			
	lib.	Čas (s)	Funkce	Soubor	Funkce	Řádek	Sloupec	Soubor	Funkce	Řádek	Sloupec
-O0 IP: 579	glibc	9.17	504	0	0	0	0	0	0	0	0
	bfd	0.31	579	72	0	1	0	0	0	0	0
	dw	0.39	579	1	0	1	0	0	0	0	0
	dwarf	0.15	504	0	0	0	0	0	0	0	0
-O1 IP: 274	glibc	0.57	224	0	0	0	0	0	0	0	0
	bfd	0.22	274	47	0	1	0	0	0	0	0
	dw	0.26	274	1	0	1	0	0	0	0	0
	dwarf	0.11	224	0	0	0	0	0	0	0	0
-O2 IP: 252	glibc	0.55	199	0	0	0	0	0	0	0	0
	bfd	0.21	252	50	0	1	0	0	0	0	0
	dw	0.26	252	1	0	1	0	0	0	0	0
	dwarf	0.11	199	0	0	0	0	0	0	0	0
-O3 IP: 253	glibc	0.56	189	0	0	0	0	0	0	0	0
	bfd	0.22	253	61	0	1	0	0	0	0	0
	dw	0.27	253	1	0	1	0	0	0	0	0
	dwarf	0.11	189	0	0	0	0	0	0	0	0
-Os IP: 275	glibc	0.81	220	0	0	0	0	0	0	0	0
	bfd	0.23	275	52	0	1	0	0	0	0	0
	dw	0.27	275	1	0	1	0	0	0	0	0
	dwarf	0.11	220	0	0	0	0	0	0	0	0

Tabulka 4.1: Porovnání symbolizačních knihoven, verze bez debug. informací

souboru. V ideálním případě je hodnota rovna počtu IP. Z výsledků je patrné, že knihovny libbfd a libdw byly v tomto ohledu úspěšné ve všech případech.

Skupina sloupců vyznačená jako Zdrojový kód obsahuje nenulové hodnoty tehdy, podařilo-li se získat nějaké informace o zdrojových kódech programu. Sloupec Soubor udává počet IP, ke kterým se podařilo nalézt jméno souboru se zdrojovým kódem. Sloupec s názvem Funkce udává počet IP, ke kterým se podařilo přiřadit jejich název z informací o zdrojovém souboru. Je započítána pouze tehdy, pokud se liší od názvu funkce z objektového souboru,[8] a tedy přináší nějakou další informaci. Sloupce s názvem Řádek a Sloupec pak udávají počet IP, ke kterým se podařilo dohledat i přesný řádek resp. sloupec ve zdrojovém kódu.

Jediná knihovna libbfd dokázala v této kategorii odhalit část zdrojových souborů i bez debugovacích informací. Na druhou stranu ve verzi s debugovacími informacemi se s její pomocí nepodařilo určit žádný sloupec ve zdrojovém

[8] Viz Resolved trace v [50] a popis `object_function` a `source.function`

				Zdrojový kód				Inlinované funkce			
	lib	Čas (s)	Funkce	Soubor	Funkce	Řádek	Sloupec	Soubor	Funkce	Řádek	Sloupec
`-O0 -g` IP: 579	glibc	9.16	504	0	0	0	0	0	0	0	0
	bfd	2.50	579	570	0	569	0	0	0	0	0
	dw	3.52	579	569	568	569	568	0	0	0	0
	dwarf	21.43	504	567	553	567	567	0	0	0	0
`-O1 -g` IP: 274	glibc	0.56	224	0	0	0	0	0	0	0	0
	bfd	3.46	274	266	10	265	0	576	211	576	0
	dw	5.18	274	265	264	265	264	576	576	576	576
	dwarf	30.90	224	263	239	263	263	573	573	573	573
`-O2 -g` IP: 252	glibc	0.55	199	0	0	0	0	0	0	0	0
	bfd	3.52	252	244	10	243	0	596	202	596	0
	dw	4.75	252	243	242	243	242	596	596	596	596
	dwarf	27.98	199	241	222	241	241	593	593	593	593
`-O3 -g` IP: 253	glibc	0.56	189	0	0	0	0	0	0	0	0
	bfd	3.61	253	245	5	244	0	571	184	571	0
	dw	4.78	253	244	243	244	243	571	571	571	571
	dwarf	29.07	189	242	223	242	242	568	568	568	568
`-Os -g` IP: 275	glibc	0.77	220	0	0	0	0	0	0	0	0
	bfd	2.18	275	266	8	265	0	416	174	416	0
	dw	3.21	275	265	264	265	264	416	416	416	416
	dwarf	19.83	220	264	245	264	264	416	416	416	416

Tabulka 4.2: Porovnání symbolizačních knihoven, verze s debug. informacemi

kódu. Knihovny libdw a libdwarf dokázaly sloupce určit a v této kategorii si vedly srovnatelně. Funkce `backtrace_symbols` je mimo hru.

Poslední skupina sloupců zachycuje informace o inlinovaných funkcích. Pro každý IP je určen seznam funkcí, které jsou v daném místě inlinované. Seznam je následně proiterován a je zkoumáno, zda se podařilo odhalit zdrojový soubor (sloupec Soubor), jméno inlinované funkce (sloupec Funkce) a pozici ve zdrojovém souboru (sloupce Řádek a Sloupec). Tzn. je-li v místě IP inlinováno n funkcí, tento IP může zvětšit hodnoty v příslušných sloupcích o n.

Nejúspěšnějšími jsou zde knihovny libdw a libdwarf. Knihovna libbfd sice odhalila srovnatelný počet inlinovaných funkcí, určit jejich jméno se však povedlo jen zhruba ve třetině případů. Ani u inlinovaných funkcí neodhalila sloupce ve zdrojovém kódu.

Celkově z experimentu plyne, že pro získání dostatečně podrobného backtrace je potřeba mít program zkompilovaný s debugovacími informacemi. Bez nich se u optimalizovaných programů nepodaří odhalit inlining.

Nejvyšší úroveň detailu poskytla knihovna libdw. Vždy se jí podařilo určit

jméno funkce v objektovém souboru a odhalila nejvíce jmen u inlinovaných funkcí.

Knihovna libbfd byla o něco úspěšnější v případě absence debugovacích informací, nedokázala však určit sloupce ve zdrojovém kódu. Stejně tak měla problém se jmény některých inlinovaných funkcí.

Knihovna libdwarf nedokázala určit jména některých symbolů v objektovém souboru a zároveň měla problém s dobou výpočtu.

4.7 Vliv optimalizací na obsah backtrace

Předchozí experiment určil vhodnou symbolizační knihovnu. Nezodpověděl však otázku, jak symbolizovaný backtrace v optimalizovaném programu ve skutečnosti vypadá. V dalším experimentu byl proto vytvořen jednoduchý program, jehož úkolem je zavolat posloupnost funkcí a vypsat backtrace:

```
89 struct MyStruct { MyStruct() { print_backtrace(); } };
90 void baz() { auto ptr = std::make_unique<MyStruct>(); }
91 template <int N> void bar() {
92     if constexpr (N == 0) { baz();}
93     else { bar<N-1>(); }
94 }
95 void foo() { bar<3>(); }
96 int main() { foo(); }
```

Funkce `print_backtrace` zavolaná z konstruktoru `MyStruct` byla definována s `__attribute((noinline))`, aby v optimalizovaných verzích programu nedocházelo k jejímu inliningu.

Symbolizace byla opět provedena přes Backward-cpp se zvolenou knihovnou libdw. Program byl kompilován v optimalizačních úrovních `-O0` až `-O3`, ve verzi s debugovacími informacemi i bez nich. Počínaje úrovní `-O1` výše byl výstup vždy stejný. Výsledky experimentu následují níže.

`-O0`

```
0x7fbc260036dd: print_backtrace() in /home/toman/test
0x7fbc26005890: MyStruct::MyStruct() in /home/toman/test
0x7fbc26006c7c: std::_MakeUniq<MyStruct>::__single_object \
                std::make_unique<MyStruct>() in /home/toman/test
0x7fbc26003d78: baz() in /home/toman/test
0x7fbc26009835: void bar<0>() in /home/toman/test
0x7fbc26008d46: void bar<1>() in /home/toman/test
0x7fbc26007d70: void bar<2>() in /home/toman/test
0x7fbc26006d15: void bar<3>() in /home/toman/test
0x7fbc26003da4: foo() in /home/toman/test
0x7fbc26003db0: main in /home/toman/test
0x7fbc24fd1bf6: __libc_start_main in /lib/x86_64-linux-gnu/libc-2.27.so
  from __libc_start_main at ../csu/libc-start.c:310:?
0x7fbc26003489: _start in /home/toman/test
```

`-O0 -g`

```
0x7f9e29e036dd: print_backtrace() in /home/toman/test
  from print_backtrace at /home/toman/test.cpp:64:17
0x7f9e29e05890: MyStruct::MyStruct() in /home/toman/test
  from MyStruct at /home/toman/test.cpp:89:47
0x7f9e29e06c7c: std::_MakeUniq<MyStruct>::__single_object \
                std::make_unique<MyStruct>() in /home/toman/test
  from make_unique<MyStruct> at /usr/include/c++/8/bits/unique_ptr.h:835:30
0x7f9e29e03d78: baz() in /home/toman/test
  from baz at /home/toman/test.cpp:90:52
0x7f9e29e09835: void bar<0>() in /home/toman/test
  from bar<0> at /home/toman/test.cpp:92:32
0x7f9e29e08d46: void bar<1>() in /home/toman/test
  from bar<1> at /home/toman/test.cpp:93:20
0x7f9e29e07d70: void bar<2>() in /home/toman/test
  from bar<2> at /home/toman/test.cpp:93:20
0x7f9e29e06d15: void bar<3>() in /home/toman/test
  from bar<3> at /home/toman/test.cpp:93:20
0x7f9e29e03da4: foo() in /home/toman/test
  from foo at /home/toman/test.cpp:95:20
0x7f9e29e03db0: main in /home/toman/test
  from main at /home/toman/test.cpp:96:17
0x7f9e28dd1bf6: __libc_start_main in /lib/x86_64-linux-gnu/libc-2.27.so
  from __libc_start_main at ../csu/libc-start.c:310:?
0x7f9e29e03489: _start in /home/toman/test
```

`-O1`

```
0x7f4f3ea025bf: print_backtrace() in /home/toman/test
0x7f4f3ea04cab: baz() in /home/toman/test
0x7f4f3ea04cf0: main in /home/toman/test
0x7f4f3d9d1bf6: __libc_start_main in /lib/x86_64-linux-gnu/libc-2.27.so
  from __libc_start_main at ../csu/libc-start.c:310:?
0x7f4f3ea023c9: _start in /home/toman/test
```

`-O1 -g`

```
0x7fdcf10025bf: print_backtrace() in /home/toman/test
  from print_backtrace at /home/toman/test.cpp:64:17
0x7fdcf1004cab: baz() in /home/toman/test
  from baz at /home/toman/test.cpp:89:47
  (inlined) MyStruct at /home/toman/test.cpp:89:19
  (inlined) make_unique<MyStruct> at /home/toman/test.cpp:90:52
0x7fdcf1004cf0: main in /home/toman/test
  from main at /home/toman/test.cpp:92:32
  (inlined) bar<0> at /home/toman/test.cpp:93:20
  (inlined) bar<1> at /home/toman/test.cpp:93:20
  (inlined) bar<2> at /home/toman/test.cpp:93:20
  (inlined) bar<3> at /home/toman/test.cpp:95:20
  (inlined) foo at /home/toman/test.cpp:96:17
0x7fdceffd1bf6: __libc_start_main in /lib/x86_64-linux-gnu/libc-2.27.so
  from __libc_start_main at ../csu/libc-start.c:310:?
0x7fdcf10023c9: _start in /home/toman/test
```

Z experimentu je patrné, jak zásadní vliv mají debugovací informace na výslednou podobu backtrace při zapnutých optimalizacích. Mezi úrovněmi `-O0 -g` a `-O1 -g` došlo ke ztrátám některých informací o funkci (chybí typ návratové hodnoty), podařilo se ale odhalit jména všech funkcí participujících v řetězci volání.

Bez debugovacích informací je výstup v úrovni `-O0` stále ještě smysluplný, od `-O1` výše se už projevuje inlining a velká část informací o toku programu tím mizí.

4.8 Úprava jmen symbolů

Posledním krokem v procesu získání backtrace je převod jmen získaných symbolizací do hezké, lidsky čitelné podoby. Jména symbolů získaných z objektových souborů budou totiž v případě jazyka C++ vypadat zhruba takto:

```
_ZNKSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE4sizeEv
```

Nad získanými jmény je potřeba provést name demangling. Backward-cpp se o tento krok postará automaticky, v případě tvorby vlastního symbolizačního nástroje jej však bude potřeba řešit. Stejně jako pro symbolizaci, i pro name demangling existují programy, které jej dokážou provést. Příkladem budiž `c++filt` z Binutils.

V tomto případě však není složité ani programové řešení. V hlavičkovém souboru `<cxxabi.h>` existuje funkce s názvem `abi::__cxa_demangle` [64, kap. 28], která tuto funkcionalitu implementuje. Její použití ukazuje kód 4.6 převzatý z [22].

```
#include <cxxabi.h>

std::string demangle(const std::string& name)
{
    // mangled C++ names starts with "_Z"
    if (name.size() < 3 || name[0] != '_' || name[1] != 'Z') {
        return name;
    }

    char* demangled = abi::__cxa_demangle(name.c_str(), nullptr,
                                          nullptr, nullptr);
    if (!demangled) {
        return name;
    }

    std::string result = demangled;
    free(demangled);
    return result;
}
```

Kód 4.6: Name demangling pomocí `abi::__cxa_demangle`

Funkce `demangle` převede jméno symbolu do následující podoby:

```
std::__cxx11::basic_string<char, std::char_traits<char>,
                           std::allocator<char> >::size() const
```

Z tohoto názvu už je dobře patrné, že se jedná o členskou funkci `size` třídy `std::string`. Nepříjemné ovšem je, že se při kompilaci ztratila informace o použitém `typedef`. Těžce vydobytá jména funkcí, ve kterých jakkoliv figurují šablony, budou ve výsledku nepřehledná a půjde z nich těžko vyčíst vlastní jméno funkce. To se týká členských funkcí šablonových tříd, šablonových funkcí, parametrů funkcí typu šablonová třída apod.

Konkrétní instance šablon nejsou často pro uživatele tolik podstatné a je možné je z výstupu volitelně vynechat. Na závěr kapitoly je proto ukázán kód 4.7, který se postará o jejich odfiltrování. Kód je opět inspirován profilerem Heaptrack [43, `accumulatedtracedata.cpp`].

```
std::string shorten_templates(const std::string& demangled_name)
{
    std::string result;
    int depth = 0;
    for (char c : demangled_name) {
        if (!depth) { // not inside any template
            result.push_back(c);
        }
        // handle C++ operators
        bool is_cpp_operator = ends_with(result, "operator<")  ||
                               ends_with(result, "operator>")  ||
                               ends_with(result, "operator<<") ||
                               ends_with(result, "operator>>");

        if ((c != '<' && c != '>') || is_cpp_operator) {
            continue;
        }

        if (c == '<') { // template starts here
            if (!depth) { // shorten it
                result += "...";
            }

            ++depth;
        } else { // template ends here
            --depth;
            if (!depth) {
                result += c;
            }
        }
    }
    return result;
}
```

Kód 4.7: Odfiltrování šablon z názvů funkcí

Po provedení `shorten_templates` bude název funkce vypadat následovně:

```
std::__cxx11::basic_string<...>::size() const
```

Kapitola 5

Konstrukce vlastního profileru

V této kapitole si společně projdeme úvodními kroky při konstrukci profileru haldy. Ukážeme si, jakým způsobem lze odposlouchávat paměťové funkce a vyřešíme nečekané problémy, které se při tom objeví. Zároveň prodiskutujeme téma inicializace a ukončení profileru – pokusíme se uvnitř profilovaného programu vytvořit instanci vlastní třídy `Profiler`. Není to tak snadné, jak se na první pohled může zdát.

Získané poznatky budou v závěru kapitoly uplatněny pro návrh profileru haldy s nízkou latencí.

5.1 Sledování alokačních funkcí

Samotná idea profileru je vlastně velmi prostá. Kdykoliv v profilovaném programu dojde k dynamické alokaci paměti, profiler tuto informaci zachytí, zpracuje a uloží. Po skončení běhu profilovaného programu profiler vytvoří z nasbíraných dat svůj výstup. Z výstupu by měl být uživatel schopen zjistit, jakým způsobem program s haldou pracoval. Na základě toho pak může svůj program dále optimalizovat.

Na první problém narazíme ještě předtím, než vůbec začneme vlastní profiler programovat. Potřebujeme vyřešit, jakým způsobem injektovat do již zkompilovaného programu vlastní kód. Dále potřebujeme zajistit, aby se náš kód zavolal, kdykoliv v profilovaném programu dojde k volání paměťové funkce.

Memory Allocation Hooks v GNU C

Částečné řešení druhého z problémů nalezneme v manuálu knihovny glibc [4, s. 3.2.3.9]. Manuál zmiňuje, že pro úpravu chování alokačních funkcí lze použít proměnné `__malloc_hook`, `__realloc_hook` a `__free_hook` z hlavičkového souboru `<malloc.h.>`. Do zmíněných proměnných je možné uložit ukazatele na vlastní funkce, které se zavolají při každém volání paměťových funkcí. Manuál uvádí příklad, jak s tímto mechanismem pracovat.

Pro náš profiler je ale toto řešení nepoužitelné. Nijak se tím neřeší otázka vložení vlastního kódu do profilovaného programu, podle Linuxového manuálu [65] navíc není tento mechanismus bezpečný ve vícevláknových programech a byl označen jako *deprecated*.

Technika `LD_PRELOAD`

Manuál GNU C knihovny [4, kap. 3.2.5] zmiňuje možnost náhrady výchozích implementací paměťových funkcí pomocí `LD_PRELOAD` [66]. Tuto techniku jsme ostatně již mohli vidět v kapitole 3. Používá ji Heaptrack nebo profiler v gperftools.

`LD_PRELOAD` je proměnná prostředí, do které lze uložit seznam sdílených knihoven. Předtím, než začne zavaděč nahrávat sdílené knihovny linkované samotným programem, nahraje do procesu knihovny z tohoto seznamu.

To je pro nás důležitý moment. Definuje-li některá z knihoven vlastní verzi funkce `malloc`, bude tato použita namísto výchozí implementace ze standardní knihovny.

5.2 Náhrada `malloc` technikou `LD_PRELOAD`

Pojďme vyzkoušet, zda nám bude technika s `LD_PRELOAD` fungovat. Vytvoříme jednoduchý program `main.c`, který zavolá funkci `malloc`. Pokud volání uspěje, program vrátí hodnotu 0, v opačném případě vrátí hodnotu 1. Dále vytvoříme soubor `hook.c`, ve kterém funkci `malloc` definujeme tak, že vždy navrátí hodnotu `NULL`:

```
// main.c
#include <stdlib.h>

int main() { return malloc(1) == NULL; }

//hook.c
#include <cstddef>

void* malloc(size_t size) { return NULL; }
```

Nyní oba zdrojové kódy zkompilujeme. Soubor `main.c` jako obyčejný program, `hook.c` jako sdílenou knihovnu s pozičně nezávislým kódem. Následně program spustíme. Nejprve obvyklým způsobem, podruhé s `LD_PRELOAD` nastavenou na naši knihovnu:

```
$ gcc main.c -o main
$ gcc -shared -fPIC hook.c -o libhook.so
$ ./main ; echo $?
0
$ LD_PRELOAD=./libhook.so ./main ; echo $?
1
```

Vidíme, že mechanismus se chová dle očekávání. Při běžném spuštění se zavolala výchozí definice funkce `malloc`, ve druhém běhu se již spustila naše definice. Teoreticky mohlo při druhém běhu dojít k situaci, kdy se přeci jen zavolala výchozí definice, které se jen nepodařilo alokovat požadovaný jeden byte. To lze nicméně snadno vyvrátit tím, že namísto navrácení `NULL` zavoláme například funkci `abort`.

Je potřeba myslet na to, že `hook.c` jsme kompilovali překladačem jazyka C. Pokusíme-li se sestavit stejný kód pomocí překladače C++, přestane mechanismus fungovat:

```
$ mv hook.c hook.cpp
$ mv libhook.so libhook_c.so
$ g++ -shared -fPIC hook.cpp -o libhook_cpp.so
$ LD_PRELOAD=./libhook_cpp.so ./main ; echo $?
0
```

Důvod, proč se tak děje, již známe z kapitoly 4. Naše funkce `malloc` při přeložení C++ překladačem podléhala name mangling. Z pohledu linkeru tak dostala úplně jiné jméno. O tom se můžeme ostatně přesvědčit:

```
$ nm libhook_c.so | grep malloc
00000000000005e5 T malloc
$ nm libhook_cpp.so | grep malloc
0000000000000555 T _Z6mallocm
```

Linker hledal funkci `malloc`, v naší knihovně ale našel jen `_Z6mallocm`. Pokračoval proto v hledání dál a `malloc` objevil až v glibc. Vzniklý problém lze ale snadno vyřešit. V C++ kódu stačí funkci deklarovat se specifikátorem `extern "C"`.

5.3 Volání skutečné implementace `malloc`

Prozatím umíme do profilovaného programu dostat vlastní kód, který se spustí zavoláním `malloc`. Program od takového volání ale očekává navrácení alokované paměti. Nemáme-li v úmyslu vytvářet vlastní paměťový manažer, musíme nějakým způsobem zavolat funkce toho stávajícího. K tomu nám poslouží funkce dynamického linkeru `dlsym`: [67]

`void* dlsym(void* handle, const char* symbol);`

> Funkce `dlsym` nalezne ve sdílené knihovně identifikované pomocí `handle` symbol s názvem `symbol` a navrátí jeho adresu. Pro `handle` lze použít speciální hodnotu `RTLD_NEXT`, kdy funkce nalezne další výskyt požadovaného symbolu v pořadí za aktuální knihovnou.

Tím je část problému vyřešena. Funkcí `dlsym` budeme schopni získat adresu originální funkce `malloc` a tu pak zavolat pro obsloužení alokačního požadavku. Na vhodné místo stačí umístit kód profileru, který volání zpracuje a uloží.

Je však třeba myslet na to, že kód profileru bude nejspíše sám potřebovat alokovat nějakou paměť. Za tím účelem bude volat `malloc`. Musíme proto rozlišit kontext, v jakém je funkce `malloc` volána – pro alokace původem z profileru nesmíme rekurzivně volat funkce profileru.

Pojďme nyní získané poznatky aplikovat. Funkci `malloc` rozšíříme tak, aby při každém jejím zavolání vypsala požadovanou velikost alokace a získaný ukazatel na alokovanou paměť. Možnou implementaci ukazuje kód 5.1.

```
//hook.cpp
#include <cstdio>
#include <cstdlib>
#include <dlfcn.h>

using malloc_t = void* (*)(size_t);

namespace {
thread_local bool hooked = false;
}

template <class Signature>
Signature hook(const char* symbol) noexcept
{
    void* fn = dlsym(RTLD_NEXT, symbol);
    if (!fn) {
        std::abort();
    }

    return reinterpret_cast<Signature>(fn);
}


extern "C" void* malloc (size_t size)
{
    thread_local auto orig_malloc = hook<malloc_t>("malloc");
    void* ret = (*orig_malloc)(size);

    if (hooked) {
        return ret;
    }

    hooked = true;
    printf("malloc(%zu) -> %p\n", size, ret);
    hooked = false;

    return ret;
}
```

Kód 5.1: Sledování volání funkce `malloc`

5.4 Konstruktory v ostatních knihovnách

Kód můžeme opět zkompilovat a přes `LD_PRELOAD` spustit náš testovací program. Při kompilaci je potřeba přilinkovat knihovnu libdl. Program následně porovnáme s výstupem konkurenčního profileru Heaptrack.

```
$ g++ -shared -fPIC hook.cpp -o libhook.so -ldl
$ LD_PRELOAD=./libhook.so ./main
malloc(1) -> 0x7fffb8124260
$ heaptrack
...
heaptrack stats:
        allocations:            2
...
```

Jak je možné, že Heaptrack zaevidoval dvě alokace, ale my pouze jednu? Výstup profileru Heaptrack můžeme přes jeho vizualizační nástroje dále prozkoumat. Zjistíme, že Heaptrack zaznamenal dynamickou alokaci velikosti přes 72 KB, ke které došlo v `libstdc++`. K té ovšem nemohlo dojít z našeho C programu. Přes `ldd` si ostatně můžeme zjistit, jaké knihovny program `main.c` linkuje. Knihovna `libstdc++` mezi nimi není.

Heaptrack funguje na stejném principu, jako náš profiler. Používá vlastní implementace paměťových funkcí a udržuje si přehled o tom, jestli je funkce volána z profilovaného programu, nebo Heaptracku samotného. Stejně jako v našem případě je jakákoliv funkce volaná „pod `malloc`“ považována za funkci z programu, kdežto funkce „nad `malloc`“ za funkci profileru.

Tento přístup ale přestane fungovat ve chvíli, kdy v profileru bude existovat například globální objekt třídního typu, který při své konstrukci provede dynamickou alokaci paměti. Během spouštění programu dojde k volání konstruktorů všech takových objektů – jak těch z programu samotného, tak těch z dynamicky linkovaných knihoven, včetně knihovny profileru. To se stane ještě předtím, než je vůbec zavolána funkce `main`.

Pro profiler je ale takové volání ve stavu „pod `malloc`“. Z logiky věci pak plyne, že při zavolání `malloc` z těla konstruktoru bude alokace profilerem zaznamenána. Problém se ale netýká jen objektů. GCC například dovoluje označit funkce atributem `__attribute__((constructor))`. Pro kód uvnitř těla funkcí bude platit to samé – zavolá-li se `malloc`, profiler to zaznamená.

V rámci kódu samotného profileru se tomuto lze vcelku efektivně bránit. Stačí v těle konstruktoru nastavit hodnotu proměnné `hooked` na `true`. Tím se mechanismus sledování alokací vypne a funkce `malloc` pouze zavolá skutečnou alokační funkci z paměťového manažeru. V podobných situacích je ale potřeba na toto chování myslet a být důsledný při prevenci zaznamenávání nechtěných alokací.

To, co platilo pro kód profileru, bohužel neplatí pro knihovny linkované k profileru. I v nich se mohou podobné konstrukty nacházet a předejít nechtěnému zaznamenání takové alokace je v podstatě nemožné. A to je pravdě-

podobně i případ druhé alokace, kterou zaznamenal profiler Heaptrack. Pokud bychom v našem profileru chtěli pro výpis použít například `std::cout` namísto `printf`, museli bychom inkludovat hlavičkový soubor `<iostream>`. V hlavičkovém souboru se nachází definice statické instance třídy ze standardní knihovny s názvem `std::ios_base::Init`, která je zodpovědná za inicializaci C++ proudů. A právě během inicializace dojde k dynamické alokaci paměti.

To můžeme ostatně rovnou ověřit. Do souboru `hook.cpp` přidáme na první řádek `#include <iostream>`. Zkompilujeme a profiler spustíme obvyklým způsobem. Na výstupu se objeví:

```
malloc(72704) -> 0x7fffd7089260
malloc(1) -> 0x7fffd709be80
```

Nyní je náš výstup sice nesprávný, ale zato konzistentní s výstupem Heaptrack. Jak moc velký problém to pro uživatele profileru představuje? V tomto případě asi není situace příliš dramatická. Při profilování C++ programů lze předpokládat, že by k této alokaci došlo tak jako tak.

Ve zbylých případech je to nepříjemné, protože uživatel dostane nepřesná data. Na druhou stranu uživatel bude pravděpodobně hledat odpověď na jiné otázky – proč jeho program provádí tolik dynamických alokací? Jaká místa svého kódu by mohl optimalizovat? Jednu nesprávně zaznamenanou alokaci, navíc ze standardní knihovny, snadno přehlédne.

Obecně ale použití externích knihoven představuje pro profiler překážku, se kterou se při psaní běžných aplikací nesetkáváme. Vždy je potřeba vyzkoušet, zda námi zvolená knihovna nezanese do výstupu profileru falešně pozitivní výsledky.

5.5 Sledování ostatních alokačních funkcí

Prozatím jsme napojeni na funkci `malloc`. Pojďme pokročit dále a začít sledovat i ostatní alokační funkce. Z těch standardních pro jazyk C se jedná o `calloc`, `realloc`, `free` a `aligned_alloc`. Je vhodné sledovat i rozšiřující funkce, které může profilovaný program používat – `posix_memalign`, `valloc`, `cfree` apod. Ty však nemusí být na cílové platformě vždy přítomné, je potřeba počítat s možností, že jejich hledání přes `dlsym` může selhat. Zda konkrétní funkce v systému existují lze zjistit při sestavování profileru a pomocí maker preprocesoru případně vypnout kompilaci náhrad za neexistující funkce.

Na implementaci není nic složitého a lze postupovat v podstatě identicky, jako u funkce `malloc`. Je vhodné držet se principu, že u funkcí, které paměť alokují, chceme nejprve provést skutečnou alokaci a teprve potom volání analyzovat. U funkcí, které paměť uvolňují, je třeba postupovat opačně. Tímto se vyhneme možným problémům ve vícevláknových aplikacích. Dokud paměť reálně neuvolníme, nemůže ji paměťový manažer přiřadit jinému vláknu.

Problematický je z tohoto pohledu `realloc`. U něj dopředu nevíme, zda dojde k rozšíření aktuálního kusu paměti, nebo bude výsledek principiálně

odpovídat zavolání `free` a následnému `malloc`. Pokud pouze vypisujeme výsledky přes `printf`, tak v tomto není příliš rozdíl. Budeme-li ale později chtít zkoumat samotné ukazatele, musíme analýzu rozdělit na dvě části – jedna část před voláním skutečného `realloc`, druhá část po něm.

Máme-li všechny funkce připraveny a knihovnu `libhook.so` zkompilovánu, můžeme vytvořit nový testovací program:

```
//test.c
#include <stdlib.h>

int main()
{
    void* ptr = malloc(10);
    free(ptr);
    ptr = calloc(10, 10);
    ptr = realloc(ptr, 20);
    free(ptr);
    return 0;
}
```

```
$ gcc test.c -o test
$ LD_PRELOAD=$./libhook.so ./test
malloc(10) -> 0x7fffc5b6b260
free(0x7fffc5b6b260)
calloc(10, 10) -> 0x7fffc5b6b280
realloc(0x7fffc5b6b280, 20) -> 0x7fffc5b6b280
free(0x7fffc5b6b280)
```

Profiler funguje, na výstupu se zobrazí očekávaný výsledek. Co když zkusíme nějaký jiný program? Vezměme třeba `ls`:

```
$LD_PRELOAD=./libhook.so ls
Segmentation fault (core dumped)
```

Kde se stala chyba? Odpověď na tuto otázku získáme přes GDB:

```
$ gdb ls
(gdb) set environment LD_PRELOAD libhook.so
(gdb) start
...
Program received signal SIGSEGV, Segmentation fault.
__dlsym (...) at dlsym.c:68
(gdb) backtrace
... void* (*hook<...>(...))(...) () from libhook.so
... calloc () from ./libhook.so
... _dlerror_run (...) at dlerror.c:140
... __dlsym (...) at dlsym.c:70
... void* (*hook<...>(...))(...) () from libhook.so
... calloc () from ./libhook.so
... _dlerror_run (...) at dlerror.c:140
... __dlsym (...) at dlsym.c:70
... void* (*hook<...>(...))(...) () from libhook.so
...
```

Problém je zjevný. Funkce `dlsym` může interně zavolat `calloc`. Ten v tu chvíli ještě nezná ukazatel na skutečný `calloc`, tak zavolá `dlsym`. Tím program skončí v nekonečné rekurzi.

S takovým chováním naše implementace nepočítala. V proměnné `hooked` bude potřeba rozlišovat tři možné stavy. Toho lze docílit například zavedením výčtového typu se třemi hodnotami:

`Program`
: Volání paměťové funkce pochází z programu. V takovém případě chceme zavolat skutečnou verzi paměťové funkce a analyzovat výsledek.

`Profiler`
: Volání paměťové funkce pochází z profileru. V takovém případě pouze zavoláme skutečnou verzi paměťové funkce.

`Hook`
: Volání paměťové funkce pochází z `dlsym`. V takovém případě musíme nějak zajistit alokaci resp. uvolnění paměti.

Stav `Hook` je poměrně problematický. Na volání musíme reagovat navrácením platného ukazatele na dostatečně velké volné místo v paměti. K tomu ale nemůžeme využít funkce paměťového manažeru – v tu dobu ještě neznáme jejich adresu. Jak z toho ven? Máme dvě možnosti.

1. Funkcionalitu paměťového manažeru delegujeme na operační systém. Nacházíme-li se ve stavu `Hook`, pak pro každé volání alokační funkce vytvoříme funkcí `mmap` nové anonymní mapování, při volání dealokační funkce jej pomocí `munmap` zrušíme.

   Výhody tohoto řešení:

   - Jedná se o poměrně flexibilní řešení, alokace i dealokace mohou probíhat v libovolném pořadí.
   - Získaná paměť bude zarovnána na velikost stránky a všechny její byty budou nastaveny na 0.

   Nevýhody tohoto řešení:

   - Pokud nevíme, jak přesně vypadá implementace `dlsym`, pak tento přístup nemusí fungovat. Mohlo by se stát, že `dlsym` v jednom volání paměť alokuje a ve druhém volání uvolní. Druhé volání by ale mohlo pocházet z profilovaného programu. V tu chvíli se bude profiler nacházet ve stavu `Program` a ukazatel na uvolňovanou paměť předá skutečnému paměťovému manažeru. To nejspíš vyústí v pád celého programu.

- Získaná paměť bude mít velikost násobku stránky, pro malé alokace se zbytečně plýtvá místem.

2. Vytvoříme si dostatečně velkou oblast paměti, ze které budeme postupně „ukrajovat“ potřebné místo. Oblast můžeme vytvořit dvěma způsoby. Buďto při prvním zavolání alokační funkce ve stavu `Hook` opět vytvoříme anonymní mapování, nebo si definujeme vhodně velký statický buffer inicializovaný na 0. V obou případech si budeme udržovat ukazatel na začátek doposud nevyužitého místa v bufferu a s každou proběhnuvší alokací ve stavu `Hook` budeme ukazatel náležitě zvyšovat.

   Výhody tohoto řešení:

   - Odpadá problém s interní fragmentací při malých alokacích.
   - Dealokace nejsou závislé na stavu – stačí kontrolovat, zda se uvolňované místo nachází v bufferu či nikoliv.
   - Řešení se statickým bufferem je jednoduché.

   Nevýhody tohoto řešení:

   - Dojde-li v bufferu místo, nemáme jej jak rozšířit. Pokud oblast vytvoříme pomocí anonymního mapování, můžeme zkusit `mremap`. V principu tím ale jen odsuneme stejný problém na později.
   - Chceme-li splnit požadavky na zarovnání, které alokační funkce kladou, musíme se o něj postarat sami. Při alokaci nestačí posunout ukazatel na volné místo v bufferu o požadovaný počet bytů, ale je potřeba jej zaokrouhlit nahoru.
   - Budeme postaveni před otázku volby vhodné velikosti bufferu – ta musí být dostatečná, aby stačila na celý běh programu, ale zase ne příliš velká, abychom zbytečně neplýtvali pamětí.

Ať už zvolíme první nebo druhou variantu, vždy skončíme s řešením, které má své nedostatky. Můžeme se je pokoušet odstranit, ale tím v konečném důsledku začneme vytvářet vlastní paměťový manažer. To není naším cílem. My potřebujeme jednoduché řešení vyhovující jedné konkrétní situaci.

Budeme-li problém dále zkoumat, zjistíme, že v rámci `dlsym` dochází k jedinému zavolání `calloc(1, 32)` a na navrácený ukazatel je následně zavolána funkce `free`.

U druhého z nabízených řešení by proto teoreticky stačilo mít buffer pro 32 bytů a ten neustále recyklovat. Nemáme nicméně jistotu, že `dlsym` neskončí v nějaké jiné výpočetní větvi, kde bude alokovat více paměti, klidně i pomocí jiné alokační funkce.

Jako kompromisní řešení se proto jeví statický buffer o velikosti několika kilobytů. Toto řešení ostatně používá i Heaptrack [43, `heaptrack_preload.cpp`].

Dále je potřeba dostat nějak lépe pod kontrolu způsob volání `dlsym`. Doposud jsme ukazatele na původní alokační funkce ukládali do `thread_local` proměnných. To mělo svůj důvod. Samotná funkce `dlsym` je thread-safe, pokud si ji každé vlákno zavolá samo, není potřeba řešit synchronizaci vláken při ukládání navrácené adresy.

Funkce `dlsym` nicméně předpokládá, že je thread-safe i volání `calloc`. Tím nám padá výhoda použití `thread_local`. Buďto musíme zajistit, aby bylo bezpečné volat `calloc` ve stavu `Hook` z více vláken, nebo musíme `dlsym` volat v jeden okamžik jen z jednoho vlákna.

Vzhledem k tomu, že `dlsym` vrátí pro jednou hledanou funkci vždy stejnou adresu, můžeme všechna potřebná volání provést právě jednou při startu procesu v rámci inicializace a výsledky uložit do statických proměnných.

Proběhne-li vše úspěšně, nemusíme již `dlsym` dále řešit. Pokud některé volání selže, můžeme profiler rovnou ukončit a uživatel nebude muset čekat, až se chyba projeví po nějaké době běhu.

5.6 Inicializace a ukončení profileru

V tuto chvíli už umíme spolehlivě odposlouchávat paměťové funkce a máme vyřešeny problémy s tím spojené. Nyní nastal čas se posunout o krok dále a zaměřit se na samotný profiler.

Aktuálně provádíme pouze jednoduchý výstup pomocí `printf`. Naší snahou dále bude vytvořit instanci třídy `Profiler`, která bude uchovávat data nezbytná pro profilování a bude zodpovědná za výstup profileru. Třída bude mít členské funkce jako `handleMalloc`, `handleFree` apod. Tyto členské funkce budou později volány namísto současného `printf`.

Musíme ale vyřešit otázku, jak takovou instanci vlastně vytvořit. Pokud má instance existovat napříč různými voláními paměťových funkcí, nemůže se jednat o lokální proměnnou. Vcelku přirozeně se nabízí globální proměnná na úrovni souboru.

Z dřívějška víme, že budeme muset rozlišovat stav profileru, aby nedošlo k chybnému zaznamenání případné alokace uvnitř konstruktoru. Na úrovni jednoho souboru to ale není problém zařídit. Ani tak ale nebude tento přístup fungovat. To ilustruje následující ukázka:

```
// mylib.h
struct S
{
    S() { ptr = malloc(1234); }
    ~S() { free(ptr); }
    void* ptr;
};

extern S obj;


// mylib.cpp
#include "mylib.h"
S obj {};


// main.cpp
#include "mylib.h"

int main()
{
    free(malloc(4321));
    return obj.ptr == nullptr;
}


// hook.cpp
namespace {
struct Profiler
{
    Profiler() { puts("Constructor"); }
    ~Profiler() { puts("Destructor"); }
};

Profiler profiler;
}
```

Testovací program v `main.cpp` využívá sdílenou knihovnu `libmylib.so` sestavenou ze souborů `mylib.h` a `mylib.cpp`. Knihovna `libmylib.so` deklaruje třídu `S` a definuje její globálně přístupnou instanci `obj`.

V profilovací knihovně přibyla třída `Profiler` a její instance `profiler`. V nahrazených funkcích knihovna stále využívá výstup v podobě `printf`. Vše zkompilujeme a spustíme. Tentokrát kromě `LD_PRELOAD` nastavíme ještě `LD_DEBUG=reloc`:

```
$ g++ mylib.cpp -fPIC -shared -o libmylib.so
$ g++ main.cpp -o main -L. -lmylib
$ g++ -shared -fPIC hook.cpp -o libhook.so -ldl
$ LD_DEBUG=reloc LD_PRELOAD=./libhook.so ./main
```

Můžeme získat následující výstup:

```
773:     calling init: /usr/lib/x86_64-linux-gnu/libstdc++.so.6
malloc(72704) -> 0x7fffcb2cf260
...
773:     calling init: libmylib.so
malloc(1234) -> 0x7fffcb2e0e70
773:     calling init: /home/toman/libhook.so
malloc(4096) -> 0x7fffcb2e1350
Constructor
773:     initialize program: ./main
773:     transferring control: ./main
malloc(4321) -> 0x7fffcb2e2360
free(0x7fffcb2e2360)
773:     calling fini: ./main [0]
773:     calling fini: /home/toman/libhook.so [0]
Destructor
773:     calling fini: libmylib.so [0]
free(0x7fffcb2e0e70)
...
```

Pomocí `LD_DEBUG` lze podrobněji sledovat činnost zavaděče programu. Zde konkrétně vidíme, v jakém pořadí probíhala inicializace jednotlivých modulů. Výstup profileru nám pak ukazuje, jaké paměťové funkce byly při inicializaci volány. Podstatným momentem je pro nás konstrukce a destrukce instance `profiler`. Vidíme, že k dynamické alokaci paměti z `libmylib.so` došlo ještě před konstrukcí `profiler` a uvolnění bylo provedeno až po destrukci `profiler`. Pokud bychom bezmyšlenkovitě nahradili `printf` výstup za volání `profiler.handleMalloc(...)`, začneme se brzy potýkat s nedefinovaným chováním: [14, kap. 15.7]

„*For an object with a non-trivial constructor, referring to any non-static member or base class of the object before the constructor begins execution results in undefined behavior. For an object with a non-trivial destructor, referring to any non-static member or base class of the object after the destructor finishes execution results in undefined behavior.*“

Je určitě myslitelné, že profiler bude potřeba nějak inicializovat, tzn. bude mít netriviální konstruktor. Musíme zajistit, aby k použití objektu došlo až po jeho řádné konstrukci. Nabízí se následující možnosti:

1. Knihovnu `libhook.so` zkompilujeme s přepínačem `-z initfirst`. V takovém případě zavaděč inicializuje naši knihovnu jako první. To bohužel funguje jen do chvíle, dokud nějaká jiná z profilovaným programem používaných sdílených knihoven nepoužívá `initfirst` také. To je případ například libpthread. Zavaděč v takové situaci provede prioritní inicializaci pouze jedné z knihoven a na inicializaci ostatních nebude mít přepínač žádný vliv. Náš problém to tedy neřeší.

2. Veškeré profilování budeme provádět pouze počas života (*lifetime*) objektu `profiler`. V jednovláknových aplikacích nebude problém toto období vymezit, vystačíme si s obyčejnou globální bool proměnnou. Při více vláknech se však může stát, že *detached* vlákna budou volat členské funkce profileru, zatímco hlavní vlákno bude provádět jeho destrukci. Zároveň tímto způsobem nezaznamenáme všechna volání paměťových funkcí. Vše, co se uděje před konstrukcí a po destrukci, se nepromítne do výsledků profilování.

3. Instanci `profiler` vytvoříme při prvním zachyceném volání paměťové funkce a nebudeme ji nikdy destruovat. Je pravděpodobné, že k prvnímu zavolání paměťové funkce dojde ještě před vytvořením ostatních vláken programu. Robustní řešení však musí počítat i s možností, že tomu tak není, a zajistit bezpečné vytvoření instance.

Jako jediné použitelné řešení se tak jeví poslední z navrhovaných možností. Pojďme se nyní podívat, jak jej realizovat. Pro vytvoření instance potřebujeme v zásadě dvě věci – alokovat paměť a v ní následně instanci zkonstruovat. Potřebnou paměť můžeme alokovat staticky, tzn. vytvoříme dostatečně velký statický buffer. Pomocí výrazu `new` (*placement new*) v bufferu následně zkonstruujeme instanci profileru. Jak taková implementace může vypadat ukazuje kód 5.2.

```
Profiler& profiler()
{
    alignas(alignof(Profiler)) static uint8_t
        storage[sizeof(Profiler)];
    static Profiler* instance = new (storage) Profiler;
    return *instance;
}
```

Kód 5.2: Vytvoření instance profileru

S touto úpravou můžeme nyní konečně nahradit `printf` výpis profileru za `profiler().handleMalloc(...)`.

Jistotu funkčnosti řešení nám dává přímo standard jazyka C++. Využíváme zde dvě staticky alokované proměnné – `storage` a `instance`. Standard jazyka zaručuje, že paměť pro ně potřebná bude k dispozici po celou dobu běhu programu: [14, kap. 6.7.1.1]

„*All variables which do not have dynamic storage duration, do not have thread storage duration, and are not local have static storage duration. The storage for these entities shall last for the duration of the program.*“

U proměnné `instance` navíc provádíme dynamickou inicializaci[9], o níž standard praví: [14, kap. 9.7.4]

„*Dynamic initialization of a block-scope variable with static storage duration or thread storage duration is performed the first time control passes through its declaration; such a variable is considered initialized upon the completion of its initialization. ... If control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for completion of the initialization.* “

Máme tedy zaručenu i thread-safe konstrukci. Není striktně nutné provádět statickou alokaci `storage` a používat *placement new*. V rámci dynamické inicializace je možné provést „klasický“ výraz `new`. Paměť pro profiler bude v takovém případě alokována z haldy. Takové řešení nalezneme v profileru Heaptrack [43, `libheaptrack.cpp`]. Dynamickou inicializaci je rovněž možné nahradit voláním `std::call_once`.

Vzhledem k tomu, že máme v plánu instanci profileru nikdy nedestruovat, nemůžeme nikdy provést ani uvolnění paměti, ve které se nachází. Pokud paměť alokujeme dynamicky, dojde k úniku paměti. Ovšem ani statická alokace únik paměti nevyřeší. Bude-li profiler uchovávat nějaká dynamicky alokovaná data (stačí např. jeden `std::vector`), která mají být dostupná po celou dobu běhu programu, pak k úniku paměti stejně dojde.

Tím se pomalu dostáváme k problému s ukončením profileru. Předpokládejme profiler, který jednoduše počítá jednotlivé alokace a na konci profilování chce celkový počet alokací vypsat. Dvě standardní možnosti, které zajistí spuštění kódu při ukončování programu jsou destruktory statických objektů a funkce `std::atexit`. Překladač může dále podporovat anotaci funkce atributem `__attribute((destructor))`.

Ani jedna z těchto možností nám však nezajistí, že po jejím zavolání už program neprovede žádnou dynamickou alokaci nebo uvolnění. To lze experimentálně ověřit pomocí `LD_DEBUG`. Zjistíme, že ve všech případech bude kód zavolán v kroku `calling fini` naší knihovny. Cokoliv dalšího už nemáme pod kontrolou – nejsme schopni rozhodnout, zda se jedná o poslední volání paměťové funkce a měl by tudíž být proveden výstup profileru.

Co s tím? Než si na tuto otázku odpovíme, připomeňme si krátce, jak fungují jiné profilery. Obecně je lze rozdělit do dvou kategorií:

Precizní profiler zaznamenává do svého výstupu všechna volání paměťových funkcí. Dodržuje při tom pořadí volání, a to i napříč různými vlákny programu. Po skončení běhu programu lze z výstupu zrekonstruovat stav haldy pro libovolný časový okamžik. Stavem haldy je myšlena informace o počtu aktuálně alokovaných kusů a bytů paměti z konkrétních míst programu. Zástupcem této kategorie je Heaptrack.

[9] Neplést s dynamickou alokací.

Snímkovací profiler si stav haldy monitoruje interně a výstup provádí vždy po uplynutí zvoleného časového[10] kvanta. Zástupcem této kategorie je profiler v gperftools.

Při hledání řešení našeho problému s výstupem se můžeme výše uvedeným inspirovat. Počas normálního běhu programu můžeme vytvářet výstup v pravidelných časových intervalech. To nám navíc zajistí, že pokud bude program ukončen nějakou neobvyklou cestou (signál), nepřijdeme o všechna doposud nasbíraná data.

Destruktorem nebo pomocí funkce zaregistrované přes `std::atexit` detekujeme, že se běh programu blíží svému konci. V ten moment opět vytvoříme výstup. Pokud chceme mít kompletní data o celém běhu programu, je možné profiler přepnout do precizního režimu a nadále už provádět výstup při každém zavolání paměťové funkce.

Tím máme připravenu potřebnou infrastrukturu, nyní zbývá vytvořit samotné srdce profileru. Existuje celá řada možností, jak k tomuto úkolu přistoupit. V závěrečné sekci této kapitoly bude představen návrh profileru se zaměřením na minimální latenci profilování ve vícevláknových aplikacích.

5.7 Profilování s nízkou latencí

Zásadní nedostatek existujících profilerů tkví v tom, že při své činnosti synchronizují vlákna profilovaného programu. Lze to sledovat například v profileru Heaptrack. Všechna vlákna sdílí jeden výstupní soubor. Zápis do tohoto souboru představuje kritickou sekci profileru – v jeden moment může do souboru zapisovat pouze jedno vlákno. Ostatní vlákna čekají. Provádí-li program velké množství dynamických alokací z různých vláken, toto místo se stane úzkým hrdlem celého profileru.

Autoři Langr, Kočička ve svém článku [68] představují koncept profilování haldy s nízkou latencí. Jedná se o strategii, kdy každé vlákno profilovaného programu ukládá informace o proběhnuvších dynamických alokacích do své lokální cache. Všechny lokální cache jsou uloženy v globálním poli, jejich obsah proto zůstává k dispozici i po ukončení vlákna. Při terminaci programu (detekováno destruktorem) je celé pole hlavním vláknem proiterováno a je vypsán obsah každé lokální cache.

Vlastní návrh

Tato strategie je velmi dobře kombinovatelná s dříve diskutovaným prováděním výstupu v pravidelných časových intervalech. Profiler totiž může při svém startu vytvořit *sběračské* vlákno, které bude zodpovědné za sběr dat

[10] Čas může být měřen i v jiných jednotkách než vteřinách. Například v počtu alokovaných bytů atp.

z lokálních cache a tvorbu výstupu. Lokální cache bude označována jako *profil vlákna*.

Sběračské vlákno bude po většinu času uspané, pravidelně se však bude probouzet. Při probuzení proiteruje přes profily všech vláken a sesbírá v nich uložená data. Vláknové profily spojí do jednoho *sloučeného profilu* a ten bude zapsán do výstupního souboru. Období mezi dvěma výstupy profileru se nazývá *runda*.

Profiler konstruovaný tímto způsobem nepotřebuje po většinu běhu synchronizaci programových vláken při každé paměťové operaci. Kritickou sekcí je v tomto případě přístup k profilu vlákna. Do něj programové vlákno zapisuje a sběračské vlákno z něho čte. Synchronizace proto může probíhat pouze na úrovni profilu vlákna. Sběračské vlákno tím v jeden okamžik zablokuje nejvýše jedno vlákno programové.

Profily vláken budou organizovány v obousměrně zřetězeném spojovém seznamu. Při první dynamické alokaci v daném vlákně dojde k vytvoření profilu vlákna a jeho zařazení na začátek spojového seznamu. Při ukončení vlákna bude jeho profil ze spojového seznamu vyňat. Tyto akce budou spolu se sběrem dat vyžadovat výlučný přístup ke spojovému seznamu. Dá se předpokládat, že v porovnání s četností alokací nebudou programová vlákna vznikat a zanikat příliš často.[11] Možné zablokování programového vlákna při jeho vzniku či zániku by proto nemělo přinést výrazný výkonnostní dopad.

Ukončení vlákna bude detekováno destruktorem `thread_local` objektu. Stejně jako v případě ukončování profileru, i zde bude nutné uvažovat situaci, že vlákno po provedení destruktoru ještě provede dynamickou alokaci nebo uvolnění paměti. Pro tento účel bude na konci spojového seznamu profilů existovat jeden speciální, *sdílený* profil.

Po destrukci profilu vlákna bude toto vlákno všechny své další profilovací informace zapisovat do sdíleného profilu. Ke sdílenému profilu může být v jeden moment přiřazeno více vláken, která se budou navzájem blokovat. I zde se ale dá předpokládat, že paměťových operací neproběhne při ukončování vlákna příliš velké množství a nedojde k výraznému zpomalení profilování.

Pro další urychlení synchronizace mezi programovým a sběračským vláknem budou pro každé vlákno existovat ve skutečnosti dva jeho profily. Do prvního z nich bude programové vlákno zapisovat, druhý bude prázdný. Při sběru dat dojde k prohození ukazatelů na tyto dva profily. Sběračské vlákno tak dostane profil naplněný daty, ta zpracuje a profil vyprázdní. Programové vlákno tím dostane čistý profil, do kterého bude po celou příští rundu ukládat data.

Nesmí dojít k situaci, kdy programové vlákno bude ukládat data do profilu a sběračské vlákno prohodí ukazatele a začne profil zpracovávat. Před přístupem k profilu vlákna proto programové vlákno získá zámek, vytvoří a uloží data a zámek uvolní. Sběračské vlákno může prohození provést až ve chvíli,

[11] Pokud by tak bylo, pak má program jiný problém, než neefektivní práci s haldou.

kdy samo získá tento zámek. Dá se předpokládat, že prohození ukazatelů bude rychlá operace a sběračské vlákno proto nebude držet zámek příliš dlouhý čas.

Profiler detekuje konec běhu programu některou z dříve prezentovaných technik. V tu chvíli dojde k nastavení příznaku indikujícího konec profilování a sběračské vlákno bude ukončeno. Hlavní vlákno následně projde a zpracuje všechny ještě existující vláknové profily.

Od tohoto momentu, kdykoliv nějaké vlákno ukončí zápis do svého profilu a detekuje příznak konce profilování, provede zároveň zpracování a výstup dat, které právě uložilo. Zde už bude docházet k vzájemnému blokování se programových vláken. Opět se dá ale předpokládat, že v období od zavolání destruktoru profileru do reálného ukončení programu nedojde k enormně velkému počtu volání paměťových funkcí.

Sledované veličiny

Navržený profiler nedokáže sledovat celkový stav haldy. Pokud by měl udržovat seznam aktuálně alokovaných paměťových kusů, musel by být tento seznam globálně přístupný všem vláknům. Nelze jej udržovat pouze lokálně, protože alokace kusu může proběhnout v jednom vlákně a jeho uvolnění ve vlákně jiném. Manipulace se seznamem by vyžadovala výlučný přístup. Při velkém množství alokovaných kusů by seznam navíc zabíral velké množství paměti a práce s ním by mohla být pomalá.

Do profilu vlákna proto budou ukládány pouze ty informace, které vlákno může zjistit v kontextu volání jedné paměťové funkce. Půjde sledovat:

Počet alokací a uvolnění V profilu vlákna budou existovat dva čítače obsahující počet volání alokačních resp. uvolňovacích funkcí, ke kterým došlo během jedné rundy.

Zaznamenána budou pouze ta volání, během kterých skutečně došlo k alokaci nebo uvolnění. Program může validně provést `free(NULL)`, takové volání ale nemá reálně žádný efekt. Z argumentů volání a návratové hodnoty skutečné paměťové funkce profiler určí, kdy čítač inkrementovat.

Velikost alokované paměti Z argumentů alokačních funkcí profiler snadno zjistí množství paměti, které bylo po dobu jedné rundy alokováno.

U uvolňovacích funkcí to však bude problém – v argumentu je pouze ukazatel na alokovanou oblast, nikoliv však její velikost. Bez seznamu aktuálně alokovaných paměťových kusů nelze přesně zjistit, kolik bytů paměti ukazatel reprezentuje.

Lze však použít funkci `malloc_usable_size`. Z kapitoly 2 je známo, že každý paměťový kus je interně anotován svou velikostí. Tato velikost nutně neodpovídá velikosti, kterou uživatel požadoval – reálně bude paměťový kus o pár bytů větší. To ale není problém. Pro zachycení vývoje

trendu velikosti haldy v čase bude získaná informace dostatečná. Volání této funkce je navíc velmi rychlé.

Celkově je tak pro jednu rundu možné sledovat dvě hodnoty:

1. Kolik paměti program požadoval alokovat (argument alok. funkce).
2. Rozdíl velikosti uvolněné a alokované paměti. Alokační funkce tuto hodnotu zvýší o `malloc_usable_size(ptr)` bytů. Funkce uvolňovací ji o stejnou hodnotu zase sníží.

Alokační histogram Kombinací sledování velikostí alokací a jejich počtu lze získat alokační histogram – pro každou provedenou velikost alokace stačí v profilu udržovat jeden čítač počtu alokací této velikosti. Implementačně to lze provést například pomocí mapy, kde klíčem je velikost a hodnotou počet alokací.

Backtrace U každé dynamické alokace je dále možné určit místo v programu, ze kterého k ní došlo, tzn. provede se stack unwinding. Pro každý unikátní backtrace pak lze uchovávat další informace, jako počet volání, velikost, případně rovnou alokační histogram.

Další informace dodá do výstupu profileru samo sběračské vlákno při zpracovávání jedné rundy. Snadno dokáže zjistit čas, který uplynul od inicializace profileru. Dá se předpokládat, že k inicializaci dojde velmi krátce po startu procesu. Sběračské vlákno může dále zajistit informace o VSZ nebo RSS.

Kapitola 6

Heaprof

V praktické části diplomové práce byl na základě předchozího návrhu vytvořen profiler haldy s názvem Heaprof. Jeho zdrojový kód je k dispozici na přiloženém CD v adresáři `zdrojove_kody/heaprof`. Je rovněž dostupný na fakultním GitLabu v repozitáři `https://gitlab.fit.cvut.cz/tomanj26/heaprof`.

Implementace je realizována v programovacím jazyce C++ ve standardu C++17. Vzhledem k charakteru projektu používá některá z nestandardních GNU rozšíření jazyků C/C++. Externí knihovny, které Heaprof využívá, budou uvedeny v sekci 6.2

6.1 Struktura projektu

Knihovna profileru `libheaprof_profile.so`

Jádrem celého projektu je sdílená knihovna `libheaprof_profile.so`. Jejím nahráním přes `LD_PRELOAD` se do profilovaného programu dostane kód profileru a dojde k náhradě výchozí implementace paměťových funkcí za jejich upravené verze.

Kdykoliv program zavolá některou z paměťových funkcí, je toto volání zachyceno a předáno do profileru k dalšímu zpracování. Specifikací proměnné prostředí `HEAPROF_MODE=<n>` lze zvolit režim, ve kterém má profiler pracovat. Jsou podporovány 3 různé režimy:

Základní režim, `n=1`
: V základním režimu bude profiler fungovat jako jednoduchý čítač volání paměťových funkcí. Určí, kolikrát došlo při běhu k dynamické alokaci a uvolnění paměti. Nerozlišuje přitom, o jakou paměťovou funkci se jednalo. Dále určí, kolik si program nárokoval dynamické paměti, a bude v čase sledovat velikost aktuálně alokovaných paměťových kusů.

Rozšířený režim, `n=2`
: V rozšířeném režimu je funkcionalita profileru rozšířena o tvorbu his-

togramu alokací. Po skončení profilování tak lze zjistit, kolikrát došlo k dynamické alokaci určité velikosti.

Plný režim, `n=3`
V plném režimu profiler navíc zaznamenává backtrace. Pro každé unikátní místo v programu je udržován histogram alokací. Z výstupu je tak navíc možné určit, které programové místo je zodpovědné za kolik alokací jaké velikosti.

Proměnná prostředí `HEAPROF_ROUND_INTERVAL` udává počet milisekund, po které trvá jedna runda. Jméno výstupního souboru je specifikováno proměnnou `HEAPROF_OUTPUT_FILENAME`. Výstupní soubor používá vlastní binární formát.

Spouštěcí skript `heaprof`

Pro snadné spuštění profileru lze využít Bash skript `heaprof`. Skript se postará o nastavení všech potřebných proměnných, včetně `LD_PRELOAD`, a spuštení profilování. Po skončení profilování vypíše celkovou dobu běhu programu. Jeho použití je následující:

```
heaprof [options] -- <executable> [<argument>...]
```

Mezi `[options]` patří:

`-m, --mode <value> default: 3`
Nastavení režimu profileru.

`-i, --interval <msec> default: 1000`
Specifikace doby trvání jedné rundy.

`-o, --output <filename> default: heaprof.<program>.<pid>.out`
Určení názvu výstupního souboru.

`-v, --version`

`-h, --help`

Vizualizační nástroj `heaprof_cmd`

Z příkazové řádky lze pro náhled na profilerem nasbíraná data využít program `heaprof_cmd`. Program načte výstupní soubor profileru a v závislosti na zvoleném příkazu vytvoří výstup v lidsky čitelné podobě. Jeho použití je následující:

```
heaprof_cmd [<command>] [options] <file>
```

Argument `<file>` udává cestu k výstupnímu souboru. Příkaz `<command>` může být jeden z níže uvedených:

`overview`
: Vypíše základní přehled o výsledcích profilování. To zahrnuje například jméno profilovaného programu, počet volání alokačních funkcí, celkovou velikost alokované paměti nebo počet neuvolněných paměťových bloků. Není-li specifikován `<command>`, je výchozím příkazem právě `overview`. Tento příkaz je dostupný pro všechny režimy profilování.

`timeline`
: Zobrazí informace o paměťových operacích v čase. Výstupem je tabulka, kde jeden řádek zachycuje jednu rundu. Sloupce tabulky zahrnují čas, kdy daná runda skončila, počet alokací a uvolnění, které během rundy proběhly, přibližnou velikost aktuálně alokované paměti, RSS a kumulativní velikost alokací. Tento příkaz je dostupný pro všechny režimy profilování.

`histogram`
: Zobrazí alokační histogram. Histogram má opět podobu tabulky. První sloupec udává velikost alokace, druhý sloupec udává počet alokací této velikosti. Tento příkaz je není dostupný v základním režimu profilování.

`filter --size=<n>`
: Nalezne všechny alokace velikosti `<n>` a vypíše jejich backtrace. V kombinaci s příkazem `histogram` tak lze odhalit místa, kde program provádí velké množství alokací stejné velikosti. Na těchto místech může být uplatněn například *memory pooling* nebo *small size optimization*.

`tree`
: Zobrazí strom volání alokačních funkcí. Ten má podobnou strukturu, jako strom volání v detailním snímku nástroje Massif.

`hotspots [--top=<n>]`
: Nalezne a vypíše seznam míst v programu, kde došlo k největšímu počtu alokací. Dále vypíše seznam míst, odkud bylo celkově alokováno nejvíce paměti. Argumentem `--top=<n>` lze specifikovat, kolik různých počtů a velikostí má být zobrazeno.

`flame [--flame-use-size]`
: Vytvoří strom volání alokačních funkcí v textovém formátu, který lze následně převést do podoby SVG obrázku. Princip převodu je stejný, jako u `heaptrack_print -print-flamegraph`.

  Flamegraph ve výchozím nastavení obsahuje počty volání alokačních funkcí. Přepínačem `--flame-use-size` lze Flamegraph vytvořit velikost alokované paměti.

Výstup `heaprof_cmd` lze dále modifkovat pomocí přepínačů `[options]`. To se týká částí, které pracují s backtrace. Lze specifikovat:

`-j, --just-function-name`
: V backtrace nebudou zahrnuty cesty k modulům, informace o zdrojových souborech a inlinovaných funkcích.

`-t, --shorten-templates`
: Z názvů funkcí budou odfiltrovány šablony.

`-r, --reverse`
: Otočí strom volání alokačních funkcí. Ve výchozím nastavení reprezentuje kořen stromu „prostor nad `malloc`“. S tímto přepínačem bude kořen stromu v „prostoru pod `main`“. Přepínač je relevantní pro příkazy `tree` a `flame`.

`-v, --version`

`-h, --help`

Hook knihovna `libheaprof_hook.so`

Vedle profileru v projektu dále existuje samostatná Hook knihovna s názvem `libheaprof_hook.so`. Hook knihovnu je možné dynamicky přilinkovat k vlastnímu programu, čímž dojde k náhradě výchozích paměťových funkcí. S Hook knihovnou je možné interagovat přes C Heaprof API.

Heaprof API se nachází v hlavičkovém souboru `heaprof_api.h` a jsou zde deklarovány dvě skupiny funkcí. První skupinou jsou funkce opatřené `__attribute((weak))`. Tento atribut je v Heaprof API definován jako makro `WEAK_SYMBOL`. Program může definovat jejich libovolnou podmnožinu. Definované funkce pak budou Hook knihovnou volány.

Druhá skupina funkcí slouží pro komunikaci v opačném směru, tj. z programu do Hook knihovny. Tyto funkce jsou knihovnou přímo implementovány, nemají proto `WEAK_SYMBOL`. Heaprof API obsahuje:

`WEAK_SYMBOL void heaprof_malloc(void* ptr, size_t size)`
: Tuto funkci zavolá Hook knihovna pokaždé, když dojde k úspěšné dynamické alokaci paměti přes libovolnou z alokačních funkcí. V argumentu `ptr` je ukazatel na alokovanou oblast, `size` je velikost, která byla po alokační funkci požadována. K volání `heaprof_malloc` dojde po volání skutečného `malloc`.

`WEAK_SYMBOL void heaprof_free(void* ptr);`
: Tuto funkci zavolá Hook knihovna pokaždé, když dojde k uvolnění paměti. V argumentu `ptr` je ukazatel, který bude předán funkci `free`. K volání `heaprof_free` dojde před voláním skutečného `free`.

`WEAK_SYMBOL void heaprof_initialize();`
: Tuto funkci zavolá Hook knihovna právě jednou, a to před prvním voláním `heaprof_malloc` nebo `heaprof_free`. Případná ostatní vlákna jsou blokována do doby, dokud nedojde k návratu z `heaprof_initialize`.

`WEAK_SYMBOL void heaprof_finish();`
: Tuto funkci zavolá Hook knihovna ve chvíli, kdy se blíží konec běhu programu. Volání nezaručuje, že nemohou současně probíhat další volání `heaprof_malloc` a `heaprof_free`, ani že po návratu k jejich dalšímu volání nedojde. V době volání `heaprof_finish` už budou zdestruovány statické objekty v programu.

`void heaprof_dont_track_this_thread();`
: Tuto funkci implementuje Hook knihovna a program ji může použít pro vypnutí sledování volajícího vlákna. Použití této funkce způsobí, že Hook knihovna nebude pro volající vlákno nadále volat `heaprof_malloc` ani `heaprof_free`. Funkce může být volána z více vláken současně.

`void heaprof_track_this_thread();`
: Opak předchozí funkce, dojde k obnově sledování tohoto vlákna.

6.2 Sestavení projektu

Pro úspěšnou kompilaci projektu jsou vyžadovány následující nástroje:

- Překladač C++ podporující standard C++17.

  Vyzkoušeny jsou překladače GCC verze 8.4.0 a Clang verze 6.0.0.
- CMake verze 3.10 a vyšší.
- Build systém (Make, Ninja).

V CMake je možné zvolit, které části projektu mají být sestaveny:

`HEAPROF_BUILD_PROFILER <ON|OFF> default ON`
: Sestavení knihovny profileru `libheaprof_profile.so`.

`HEAPROF_BUILD_HOOKS <ON|OFF> default OFF`
: Sestavení Hook knihovny `libheaprof_hook.so`.

`HEAPROF_BUILD_VIEWER_CMD <ON|OFF> default ON`
: Sestavení vizualizačního nástroje `heaprof_cmd`.

Externí knihovny

Knihovna profileru využívá pro stack unwinding a symbolizaci backtrace projekt Backward-cpp. V CMake lze nastavit, jaké knihovny mají být pro tyto činnosti použity:

`STACK_WALKING_UNWIND`
: Pro stack unwinding budou použity funkce pro obsluhu výjimek.

`STACK_WALKING_BACKTRACE`
: Pro stack unwinding bude použita funkce `backtrace` z glibc.

`STACK_WALKING_LIBUNWIND default`
: Pro stack unwinding bude použita knihovna libunwind.

`STACK_DETAILS_DW`
: Pro symbolizaci bude použita knihovna libdw.

`STACK_DETAILS_BFD`
: Pro symbolizaci bude použita knihovna libbfd.

`STACK_DETAILS_DWARF`
: Pro symbolizaci budou použity knihovny libdwarf a libelf.

`STACK_DETAILS_BACKTRACE_SYMBOL`
: Pro symbolizaci bude použita `backtrace_symbols` z glibc.

`STACK_DETAILS_AUTO_DETECT default`
: V pořadí libdw, libbfd, libdwarf, glibc nalezne CMake první dostupnou knihovnu a ta bude pro symbolizaci použita.

Pro stack unwinding je důrazně doporučováno mít nainstalovanou knihovnu libunwind. Ve výchozím nastavení je tak i vyžadováno. V případě použití jiných stack unwinding metod bude profilování velmi pomalé. Vyzkoušena je verze libunwind 1.4.0. Pro symbolizaci je doporučována knihovna libdw. Vizualizační nástroj `heaprof_cmd` pro své setavení dále vyžaduje nainstalovanou knihovnu docopt.cpp. Tu využívá pro zpracování argumentů předaných na příkazové řádce. Heaprof dále vyžaduje přítomnost obvyklých knihoven jako je glibc, libdl nebo libpthread.

Kompilace

Jsou-li výše vyjmenované knihovny dostupné lze Heaprof získat a zkompilovat následujícím způsobem:

```
git clone https://gitlab.fit.cvut.cz/tomanj26/heaprof
cd heaprof
mkdir build
cd build
cmake ..
make
```

Po dokončení `make` bude adresář `build` obsahovat podadresáře `bin` a `lib`. V adresáři `bin` se bude nacházet spouštěcí skript `heaprof` a vizualizační nástroj `heaprof_cmd`. Adresář `lib` pak bude obsahovat knihovnu profileru `libheaprof_profile.so` a Hook knihovnu `libheaprof_hook.so`.

Projekt v současné době nepodporuje instalaci. Jako provizorní řešení je možné zahrnout adresář `bin` do proměnné prostředí `PATH`.

6.3 Implementace

V kořenu repozitáře se nachází adresář `src`, který obsahuje veškeré zdrojové kódy projektu Heaprof. Nachází se zde `heaprof_api.h`, spouštěcí skript profileru a konfigurační soubory pro CMake. Zdrojové kódy jsou dále podle oblasti svého určení organizovány do následujících podadresářů:

`src/format/`
: Třídy pro podporu tvorby a čtení výstupního souboru profileru.

`src/profiler/`
: Samotný profiler a Hook knihovna.

`src/utils/`
: Pomocné třídy a funkce, které profiler používá.

`src/viewer/`
: Vizualizace nasbíraných dat.

Nejzajímavější částí projektu je bezesporu obsah `src/profiler/`. Konceptuálně implementace reflektuje postupy a návrhy popsané kapitolou 5. Níže budou představeny jednotlivé zdrojové soubory z toho adresáře a budou popsány implementační detaily, které v nich lze nalézt.

Náhrada paměťových funkcí

Zdrojový kód: `profiler/hook.cpp`

Knihovna profileru a Hook knihovna spolu sdílejí stejný zdrojový kód určený k zachycení volání paměťových funkcí. V základu jsou sledovány funkce `malloc`, `calloc`, `realloc` a `free`. Při provádění CMake je detekováno, zda se na cílové platformě nacházejí další rozšiřující funkce. Jmenovitě jsou to `posix_memalign`, `aligned_alloc`, `valloc`, `memalign`, `pvalloc` a `cfree`. Pomocí maker preprocesoru je pak rozhodnuto, zda tyto funkce budou zakompilovány do knihovny profileru resp. Hook knihovny. Stejně tak jsou dále sledovány funkce `dlopen` a `dlclose`.

Při každém zavolání paměťové funkce je kontrolováno, zda již došlo k inicializaci. Inicializace probíhá pomocí funkce `std::call_once` a má dvě fáze. V první fázi dojde k volání `dlsym` pro zjištění skutečných adres všech nahrazených funkcí. Získané ukazatele jsou uloženy do statických proměnných a jsou tudíž dále přístupny všem vláknům. Ve druhé fázi dojde k zavolání API funkce `heaprof_initialize` a registraci volání `heaprof_finish` přes `std::atexit`.

Pro určení původu volání paměťové funkce je zaveden výčtový typ s názvem `CallOrigin` s možnými hodnotami `Program`, `Profiler`, `Hook`. Pro každé vlákno existuje lokální objekt tohoto typu.

Volání paměťových funkcí ve stavu `Program` jsou delegována do profileru skrz Heaprof API pro další zpracování. Ve stavu `Profiler` je pouze zavolána

původní funkce paměťového manažeru. Stav `Hook` pak řeší problém s rekurzivním voláním `calloc` z `dlsym`. Za tímto účelem existuje třída `FakeAllocator` obsahující statický buffer velikosti 8 KiB.

V tomto zdrojovém souboru jsou dále definovány již zmiňované API funkce `heaprof_track_this_thread` a `heaprof_dont_track_this_thread`. Jejich zavolání má efekt přehození `thread_local bool` příznaku, který je následně při volání paměťových funkcí kontrolován.

Implementace Heaprof API

Zdrojový kód: `profiler/api_impl.cpp`

Knihovna profileru implementuje `WEAK_SYMBOL` Heaprof API funkce. Při zavolání funkce `heaprof_initialize` je přečten obsah definovaných `HEAPROF_*` proměnných prostředí a na základě získaných hodnot dojde pomocí *placement-new* k vytvoření instance třídy `Profiler`. Na tuto instanci jsou dále delegována všechna ostatní API volání. Instance není nikdy destruována.

V tomto souboru jsou zároveň ošetřeny případné výjimky vyhozené při volání členských funkcí instance `Profiler`. Výjimky indikují závažnou chybu, kvůli které profiler nemůže nadále pokračovat ve své činnosti. V takovém případě by se odchycením a ignorováním výjimky mohl zachránit alespoň běh samotného programu. Uživatele však při profilování zajímá výstup profileru, nikoliv výsledek profilovaného programu. Proto dojde-li k této situaci, je rovnou zavolána funkce `std::abort`.

Backtrace

Zdrojový kód: `profiler/backtrace.h`, `profiler/backtrace.cpp`

Třída `Backtrace` nabízí členskou funkci `Backtrace::loadHere`. Zavoláním této funkce dojde v místě volání ke stack unwinding. Získané návratové adresy budou uchovány v interním poli instance `Backtrace` a pro přístup k nim třída nabízí API podobné STL kontejnerům.

K provedení stack unwinding třída původně využívala projekt Backward-cpp. Ukázalo se však, že Backward-cpp nedokáže dostatečně efektivně pracovat s knihovnou libunwind a profilování bylo v takovém případě velmi pomalé.

Z toho důvodu je při kompilaci Heaprof se `STACK_WALKING_LIBUNWIND` pro stack unwinding napřímo využita funkce `unw_backtrace` z libunwind. Pro ostatní metody stack unwinding je ale projekt Backward-cpp stále využíván.

Symbolizace backtrace probíhá za běhu programu v separátním vlákně. Za tímto účelem existuje funkce `resolve`. Ta v argumentu přijímá jeden IP a interně použije Backward-cpp pro provedení veškeré potřebné práce. Výsledkem volání je instance `ResolvedSymbol` obsahující informace o modulu, souboru se zdrojovým kódem a seznamem inlinovaných funkcí.

Profiler

Zdrojový kód: `profiler/profiler.h`, `profiler/profiler.cpp`

Třída `Profiler` deklaruje čistě virtuální členské funkce `Profiler::malloc`, `Profiler::free`, `Profiler::dlclose` a `Profiler::finish`. Ty jsou volány z implementace Heaprof API a slouží k informování profileru o nastalé události.

Pro podporu různých režimů byla využita technika známá jako *curiously recurring template pattern* (CRTP). Z třídy `Profiler` je zděděna šablonová třída `ProfilerBase<Profile, Derived>`, která redefinuje výše uvedené virtuální funkce a obsahuje kód společný pro všechny režimy. Šablonový parametr `Profile` je typ představující profil vlákna. Šablonový parametr `Derived` je typ třídy, která z `ProfilerBase` dědí (CRTP).

Provedením `static_cast<Derived*>(this)` získá `ProfilerBase` ukazatel typu zděděné třídy a může zavolat její členské funkce, které implementují části specifické pro konkrétní režim. Tím je dosaženo statického polymorfismu.

V konstruktoru `ProfilerBase` dojde k otevření výstupního souboru a zapsání základních profilovacích informací jako jsou PID, cesta ke spustitelnému souboru programu nebo nastavení profileru.

Dále je v konstruktoru založeno a spuštěno sběračské vlákno. To je realizováno třídou `utils::Timer`. V prvotní verzi implementace Heaprof bylo sběračské vlákno zodpovědné za vytvoření sloučeného profilu, zpracování dat (symbolizace backtrace) a zápisu celé rundy do výstupního souboru. Ukázalo se však, že dokud nejsou naplněny interní cache, symbolizace sběračské vlákno příliš zpomaluje a profiler z počátku běhu neprovádí výstup v požadovaném intervalu. Z tohoto důvodu bylo přidáno ještě jedno vlákno – *zpracovatelské*.

Zpracovatelské vlákno je spuštěno rovněž z konstruktoru `ProfilerBase` a komunikuje se sběračským vláknem na principu producent-konzument. Sdílená fronta je realizována třídou `utils::TaskQueue`. Sběračské vlákno se tak stará pouze o vytvoření sloučeného profilu, vytvoření časového razítka, zjištění aktuálního RSS a vložení výsledku do fronty.

Zpracovatelské vlákno sloučený profil následně zprocesuje a provede zápis celé rundy do souboru. Tato funkcionalita je pro každý režim specifická a je proto implementována v třídách zděděných z `ProfilerBase`. Jedná se o třídy `SimpleProfiler`, `ExtendedProfiler` a `FullProfiler`.

Třída `FullProfiler` obsahuje zmíněné interní cache. Aby při každé rundě nedocházelo k zápisu stejných backtraces, je každý backtrace do výstupního souboru zapsán pouze jednou a jednotlivé záznamy o alokacích se na něj odkazují pomocí indexu. Profiler proto uchovává mapování backtrace na index. Jeden IP je často součástí více backtraces. Aby nedocházelo k opakovaným symbolizacím stejného IP, profiler dále uchovává i mapu těch již symbolizovaných. Každý symbol je do souboru opět zapsán pouze jednou a záznam o backtrace tudíž obsahuje seznam indexů symbolů. Podobně je uchovávána i mapa všech textových řetězců, které během symbolizace vznikly.

Členská funkce `ProfilerBase::finish` ukončí sběračské vlákno a vytvoří sloučený profil. Ten vloží do fronty a následně počká, dokud zpracovatelské vlákno frontu zcela nevyprázdní. Od této chvíle přebírají programová vlákna úkol vlákna sběračského. Jakákoliv dynamická alokace nebo uvolnění paměti profilovaným programem způsobí, že zodpovědné programové vlákno samo vytvoří sdílený profil, vloží jej do fronty a opět počká, dokud zpracovatelské vlákno frontu nevyprázdní.

Profil vlákna je opět realizovaný s využitím techniky CRTP. Šablonová třída `ProfileBase<Derived>` implementuje čítač volání uvolňovacích funkcí a pomocí `malloc_usable_size` počítá rozdíl velikosti alokované a uvolněné paměti. Tato funkcionalita je společná pro všechny typy profilů vlákna. Pro každý režim je dále zděděna třída, která řeší zpracování volání alokačních funkcí. Pro základní režim existuje třída `SimpleProfile` obsahující čítač volání alokačních funkcí a počítající celkovou velikost alokované paměti. Třída `ExtendedProfile` implementuje pomocí `std::unordered_map` alokační histogram. A konečně, třída `FullProfile` obsahuje `std::unordered_map`, kde klíčem je `Backtrace` a hodnotou další `std::unordered_map` reprezentující alokační histogram.

Spojový seznam profilů vláken

Zdrojový kód: `profiler/profile_storage.h`

Šablonová třída `ProfileStorage<Profile>` zastřešuje spojový seznam profilů vláken a řeší záležitosti spojené s výlučným přístupem. Interně využívá `std::list`. Každý z jeho prvků obsahuje dvě instance typu `Profile`, mutex a příznak indikující, který z profilů aktuálně používá programové vlákno (*aktivní* profil).

Programové vlákno s instancí této třídy interaguje přes členskou funkci `ProfileStorage<Profile>::onLocal`, jejímž argumentem je `callback`. Při prvním zavolání této funkce z daného vlákna dojde k vytvoření příslušného prvku na začátku spojového seznamu. V lokální[12] `thread_local` proměnné je udržován iterátor na tento prvek. Dále je vytvořena lokální `thread_local` instance třídy `utils::ScopeGuard`. Ta při své destrukci přenastaví iterátor na sdílený prvek umístěný na konci spojového seznamu. Do `callback` je předáván aktivní profil, k němuž je zajištěn výlučný přístup.

Členská funkce `ProfileStorage<Profile>::forEach` je využívána sběračským vláknem. Jejím argumentem je opět `callback`. Tato funkce dovoluje bezpečně iterovat přes celý spojový seznam. U každého prvku dojde ke změně příznaku aktivního profilu a původně aktivní profil je předán do `callback` k dalšímu zpracování a vyčištění.

[12] Ve smyslu lokální pro funkci.

Volání platformě specifických funkcí

Zdrojový kód: `profiler/system.h`, `profiler/system.cpp`

V místech, kde profiler potřebuje používat nestandardní funkce jazyka C++, volá členské funkce třídy `System`. Jejím smyslem je umístit všechny platformě specifické funkcionality do jednoho zdrojového souboru, což by v budoucnu mělo usnadnit přechod i na jiné platformy. Instanci této třídy je možné vytvořit lokálně v místě, kde je její užití potřeba. Příklady funkcionalit, které třída nabízí:

- PID.
- Příkaz, kterým byl program spuštěn – `/proc/self/cmdline`.
- Cesta k profilovanému programu – ze symlinku `/proc/self/exe`.
- Aktuální RSS – `/proc/self/statm`.
- Práce s proměnnými prostředí.

6.4 Experimentální vyhodnocení

V praktické části práce bylo dále provedeno experimentální vyhodnocení efektivity navrženého a implementovaného řešení. Primární oblastní zájmu byl dopad profilování na celkovou dobu běhu programu.

Testovací prostředí

Heaprof byl v plném režimu porovnáván s profilery Massif (Valgrind) verze 3.15.0, Heaptrack verze 1.1.80 a profilerem haldy v gperftools verze 2.9.1. Na testovacím systému se nacházela GNU C knihovna ve verzi 2.31. Pro překlad Heaprof a jednotlivých benchmarků byl využit překladač GCC ve verzi 8.4.0. Použitým operačním systémem byla Linuxová distribuce Ubuntu 20.04.1.

Experiment probíhal na stroji s procesorem AMD Ryzen 7^{TM} 2700U s frekvencí 2.2 GHz, 4 fyzickými jádry a podporou paralelního běhu 8 vláken. Stroj disponoval 2×4 GiB DDR4 pamětí RAM na frekvenci 2 400 MHz.

Použité benchmarky

Za účelem experimentu byla vytvořena sada 7 benchmarků. Pět z nich bylo převzato z jiných zdrojů, dva jsou vlastní. Ve všech případech se jedná o vícevláknové aplikace, které provádějí velké množství dynamických alokací. Každý benchmark byl se zkoumaným profilerem spuštěn pro $P = 1, 2, 4, 8$ vláken. Následuje popis benchmarků včetně použitých parametrů:

threadtest [69] Každé vlákno provede 1000 iterací. V každé iteraci vlákno alokuje a následně uvolní $3 \times 10^4/P$ objektů. Kód benchmarku byl převzat z [70].

linux-scalability Každé vlákno provede 10^7 alokací a paměť následně uvolní. Všechny alokace mají velikost 32 B. Kód benchmarku byl převzat z [70].

shbench [69] Vlákna v 2×10^6 / P iteracích náhodně alokují a uvolňují objekty náhodné velikosti v rozmezí 1 až 1000 B. Kód benchmarku byl převzat z [71].

binary-trees Vlákna rekurzivně konstruují binární stromy. Každý uzel binárního stromu je dynamicky alokován. Maximální hloubka stromu je omezena na 15. Kód benchmarku byl převzat z [10].

hash-table Každé vlákno v 7 milionech iterací vkládá do hash tabulky na náhodné pozice náhodně velká pole. Kód byl převzat z [68].

parse-json Každé vlákno pomocí knihovny [72] načítá velký (170 MB) JSON soubor. Při načítání dochází k mnoha alokacím z různých míst knihovny.

queue Každé vlákno udržuje frontu alokovaných objektů. V každé iteraci se vlákno náhodně rozhodne, zda alokuje nový objekt a vloží jej na začátek fronty, nebo uvolní a odstraní poslední prvek fronty. Počet alokovaných objektů jedním vláknem je omezen na 30 milionů.

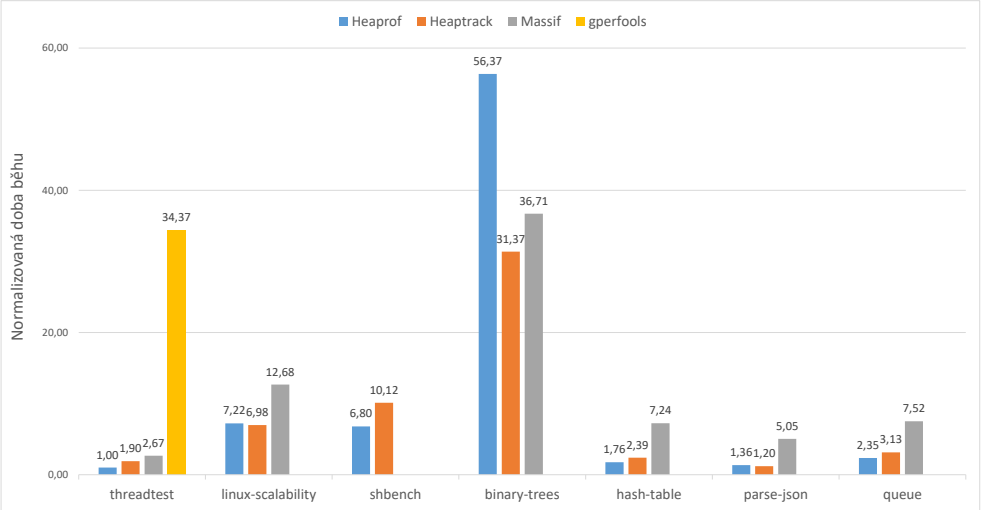
Pro každý benchmark B byl pro všechna zkoumaná P změřen referenční čas $T_B(P)$ s přesností na setiny vteřiny. Naměřená $T_B(P)$ se v závislosti na benchmarku a počtu vláken pohybovala v rozmezí od desetin vteřiny (binary-trees) až po jednotky minut (shbench).

Porovnání s ostatními profilery – čas

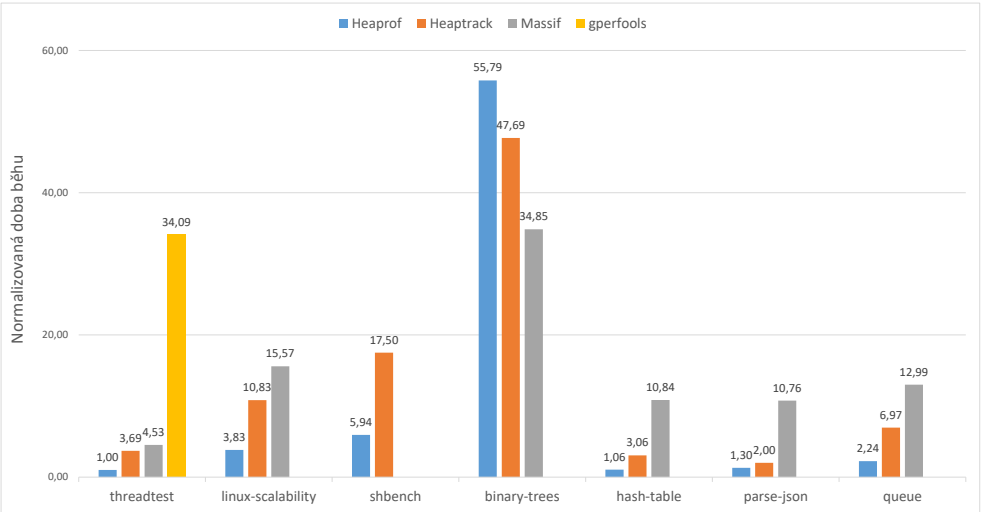
Následně byl pro každý profiler H měřen čas $T_B^H(P)$, který zahrnoval celkovou dobu běhu počínaje spuštěním profileru až do jeho úplného ukončení. Na základě těchto hodnot byla vypočítána normalizovaná doba běhu jako $R_B^H(P) = T_B^H(P)/T_B(P)$. Ta označuje koeficient, o který se profilováním prodloužila doba běhu programu. Naměřené výsledky jsou vyobrazeny v grafech na obrázcích 6.1, 6.4, 6.3, 6.4

Ne všechna měření byla úspěšná. Profiler v gperftools vykazoval takové zpomalení programu, že byl zkoumán pouze pro benchmarky binary-trees a threadtest. V threadtest zpomaloval program na více než 30násobek obvyklé doby běhu, v binary-trees se při $P = 8$ jednalo o 1395násobek. Aby nedošlo ke zkreslení hodnot, jsou v grafech pro gperftools zahrnuty pouze výsledky z threadtest.

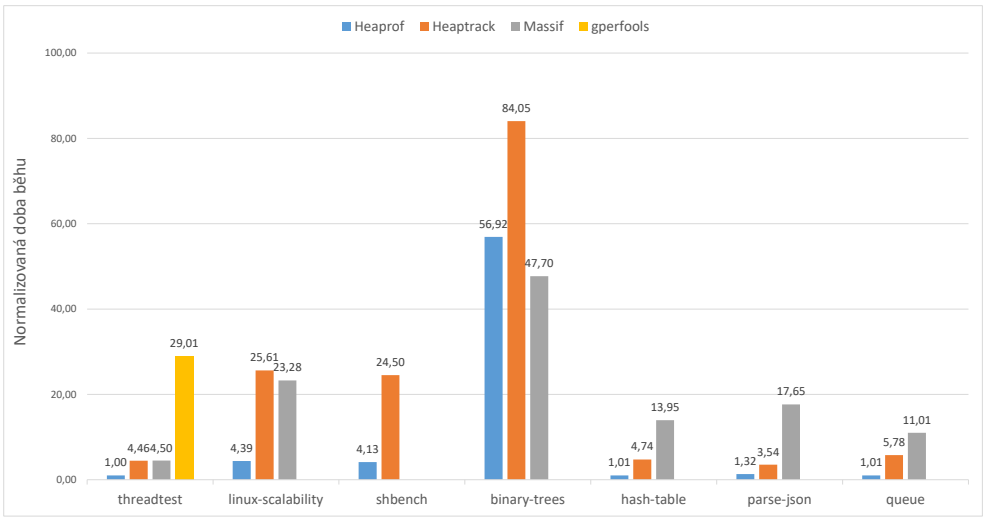
U některých měření dále chybí výsledky nástroje Massif. Zde ve všech případech došlo k vyčerpání veškeré dostupné paměti a celkovému zamrznutí systému. To se pro všechna P týkalo benchmarku shbench a pro $P = 8$ dále benchmarků hash-table a parse-json.



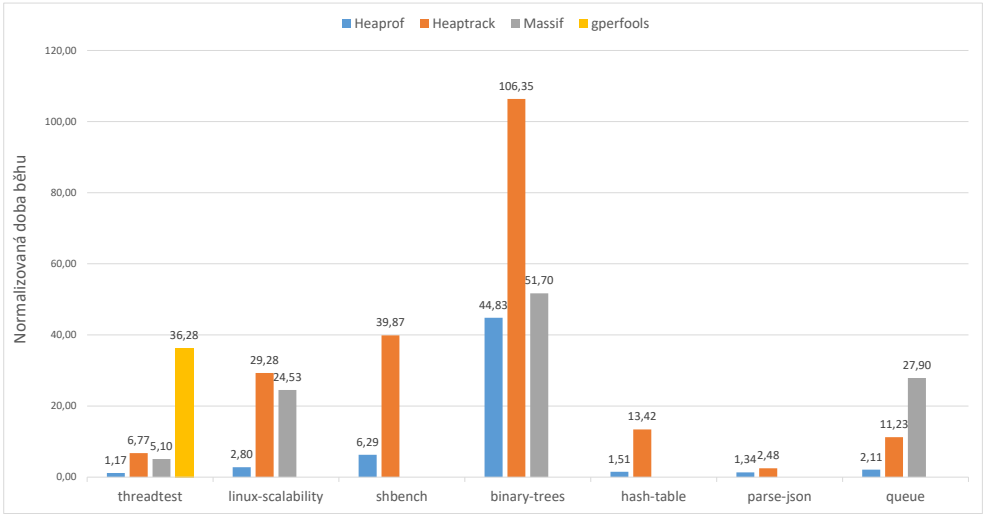
Obrázek 6.1: Srovnání profilerů, 1 vlákno



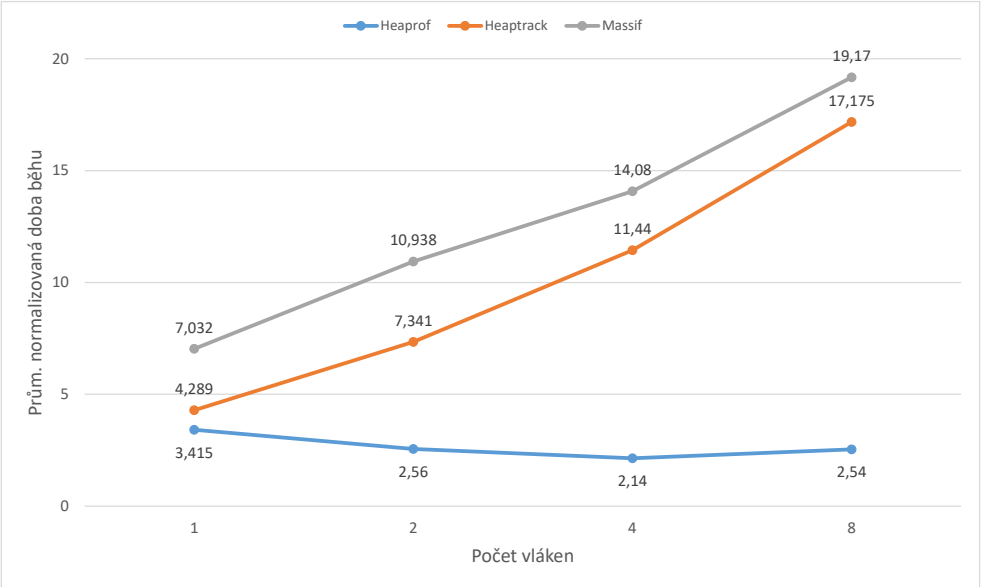
Obrázek 6.2: Srovnání profilerů, 2 vlákna



Obrázek 6.3: Srovnání profilerů, 4 vlákna



Obrázek 6.4: Srovnání profilerů, 8 vláken



Obrázek 6.5: Závislost počtu vláken na průměrné době profilování

Heaptrack nezvládl zcela dokončit běh benchmarku parse-json pro $P = 8$ a v závěru profilování spadl. Tento problém se vyskytoval i při opakovaných pokusech. Porovnáním výstupu s výsledky Heaprof bylo zjištěno, že profileru Heaptrack chybělo zaznamenat posledních cca. 100 alokací z více než 141 milionů. Do výsledků byl proto Heaptrack v tomto benchmarku zahrnut.

Největší problém činil pro všechny profilery benchmark binary-trees. Důvodem jsou alokace v rekurzivních funkcích. Ty totiž mají za následek, že každý backtrace, který je během nich vytvořen, je unikátní co do pořadí obsažených IP. Jelikož počet listů v binárním stromu roste exponenciálně s hloubkou stromu, začne exponenciálně růst i velikost interní cache Heaprof. Z naměřených hodnot lze usuzovat, že obdobnou interní cache mají i ostatní profilery. U ostatních benchmarků je z naměřených hodnot patrné, že Heaprof podával ve vícevláknových aplikacích bezkonkurenčně nejlepší výsledky. Při $P = 8$ prodloužil Heaprof dobu běhu benchmarku hash-table o 51 %. V případě profileru Heaptrack se už jednalo o 1242 %. Pro lepší ilustraci – standardní doba běhu tohoto benchmarku byla 15,4 vteřiny, Heaprof dokončil svou činnost po 23,3 vteřinách, Heaptrack po 3 minutách a 26 vteřinách běhu. To představuje 8,8násobné zrychlení celého profilování. U ostatních benchmarků je situace velmi podobná. Nejmenší rozdíl byl zaznamenán u parse-json, kde Heaprof přidal 34 % doby běhu a Heaptrack 148 %.

Zajímavé je srovnání hodnot $R_B^{\text{Heaprof}}(P)$ pro jeden zvolený benchmark napříč všemi P. Tato hodnota totiž na rozdíl od ostatních profilerů nemá lineární trend růstu v závislosti na P. To znamená, že profiler nebránil aplikaci ve škálování. Přidal sice jistou časovou režii, ta se však pro různá P do celkové

Benchmark	Alokace	Heaprof	Heaptrack	Massif
threadtest	30 M	10 KB	10,6 MB	19 KB
linux-scalability	80 M	5 KB	775 KB	17 KB
shbench	2,1 G	174 KB	2,27 GB	N/A
binary-trees	101 M	209 MB	417 MB	56 KB
hash-table	112 M	30 KB	128 MB	N/A
parse-json	141 M	123 KB	142 MB	N/A
queue	243 M	16 KB	162 MB	109 KB

Tabulka 6.1: Velikost výstupních souborů profilerů

doby běhu promítla jen jako multiplikativní konstanta. Graf na obrázku 6.5 zachycuje pro jednotlivé profilery závislost počtu vláken na průměrné normalizované době profilování. Z průměru je vynechán benchmark binary-trees.

Z výsledků pro $P = 1$ je patrné, že Heaprof má stále své výkonnostní rezervy. Kromě diskutovaného binary-trees jej Heaptrack předstihl ještě v benchmarcích linux-scalability a parse-json. Nabízí se tedy možnost další optimalizace sekvenčních částí Heaprof.

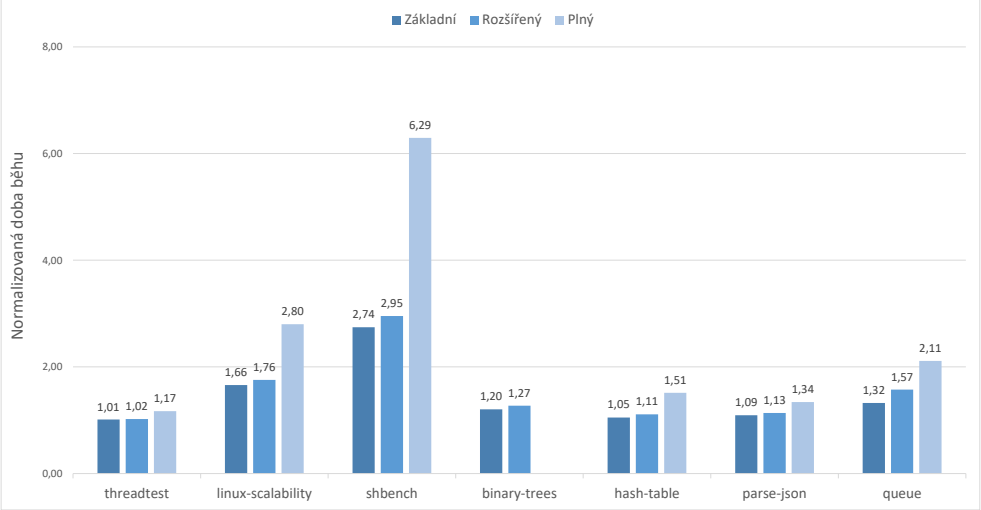
Porovnání s ostatními profilery – velikost výstupu

Jedním z důležitých parametrů profileru je i velikost výstupního souboru. Tabulka 6.1 zachycuje velikosti souborů, které jednotlivé profilery během experimentu vygenerovaly. V prvním sloupci je uveden název benchmarku, ve druhém sloupci přibližný počet alokací, ke kterým za běhu benchmarku došlo. Následují velikosti výstupů profilerů Heaprof, Heaptrack a Massif. Tabulka zachycuje profilování s $P = 8$. Bylo ověřeno, že Heaprof zachytil stejný počet alokací, jako Heaptrack. Jedinou výjimku tvoří již zmiňovaný problém profileru Heaptrack s dokončením běhu parse-json. Tam Heaprof naměřil o 137 alokací více.

Výstupní soubory Heaprof ani Massif nejsou komprimované a jejich velikost je srovnatelná. Naproti tomu Heaptrack generuje o několik řádů větší, komprimované soubory. O jejich dekompresi a čtení se automaticky starají vizualizační nástroje profileru Heaptrack. Z pohledu uživatele není potřeba soubory dekomprimovat. V tabulce je proto uvedena jejich komprimovaná velikost.

Porovnání režimů Heaprof

Na závěr experimentálního vyhodnocení byly porovnány jednotlivé režimy, které Heaprof podporuje. Měření proběhlo na celé sadě benchmarků s fixním $P = 8$. Výsledky zobrazuje graf na obrázku 6.6. Pro větší přehlednost je z grafu odstraněn výsledek plného režimu v benchmarku binary-trees.



Obrázek 6.6: Srovnání režimů Heaprof

Naměřené výsledky odpovídají očekávání. Nejmenší časovou režii k profilování přidává základní režim. Tvorba histogramu alokací si vyžádá více času, oproti základnímu režimu se ale nejedná o výrazný rozdíl. Tvorba backtrace už profilování zpomaluje citelněji. V případě benchmarku binary-trees došlo k poklesu doby běhu profilovaného programu z 45násobku normální doby běhu na 1,2 resp. 1,27násobek.

Shrnutí

Celkově experimentální vyhodnocení ukázalo, že běžně rozšířené nástroje nedokáží při profilování paměťově náročných vícevláknových aplikací dostatečně škálovat. Při odhlédnutí od problémů s alokacemi v rekurzivních funkcích zaznamenal Heaprof svůj nejhorší výsledek v benchmarku linux-scalability, kde při sekvenčním provádění způsobil nárůst doby běhu na 7,22násobek. S rostoucím počtem vláken se však tato hodnota dále nezvětšovala a při $P > 2$ už Heaprof vykazoval ve všech benchmarcích násobně menší zpomalení profilovaného programu. Poměrně realistický benchmark parse-json Heaprof nikdy nezpomalil o více, než 36 %.

Výsledky ukazují, že navržená metoda profilování s nízkou latencí je v praxi realizovatelná a může významným způsobem napomoci při profilování vícevláknových programů.

6.5 Budoucí rozšíření a vylepšení

Jakkoliv dokázal Heaprof konkurovat ostatním nástrojům svou rychlostí, nedokáže už konkurovat svou robustností. V praxi byly například zaznamenány

problémy s otevíráním sdílených knihoven. Používá-li profilovaný program funkci `dlopen`, v určitých případech selže volání originální funkce. Prozatím se však nepodařilo zjistit důvod tohoto chování.

Zásadním vylepšením, kterým bude muset Heaprof do budoucna projít, je odsun symbolizace do pozdější fáze spuštěné až po ukončení celého profilování. Tím zanikne nutnost používat zpracovatelské vlákno a některé z interních cache. Sníží se tak časová i paměťová náročnost profilování. Z porovnání dostupných symbolizačních knihoven vyplynulo, že nejvhodnější knihovna pro tento účel bude libdw. Celkově bude potřeba změnit formát výstupního souboru, ukládat informace o rozložení paměti procesu a vytvořit nový program, který dokáže vzniklý výstupní soubor načíst, doplnit symbolizační informace a opět uložit.

Interní cache pro backtrace bude muset zůstat zachována, nabízí se ale některé její optimalizace. V současnosti není omezena její velikost a v případě, že program alokuje paměť z rekurzivních funkcích, dojde brzy k vyčerpání dostupné paměti. Dalo by se uvažovat nad zavedením limitu pro uložené backtraces, při jehož dosažení by došlo k vyprázdnění celého obsahu cache. Případně by mohla fungovat na principu *least recently used* (LRU). Dala by se rovněž konstruovat jako strom a indexy ukládat do jeho uzlů.

Ačkoliv Heaprof produkuje řádově menší výstupní soubory než například Heaptrack, dá se předpokládat, že při profilování několika desítek minut nebo jednotek hodin běhu programu už začne být výstupní soubor příliš velký. Nabízí se proto možnost jeho komprimace.

Heaprof aktuálně dokáže sledovat jen rodičovský proces. Provede-li program `fork`, potomek už nebude profilován. Dá se uvažovat nad zavedením podpory profilování i v takovýchto případech. Bylo by rovněž možné podporovat runtime-attaching.

Výstupní soubor profileru je aktuálně možné prohlížet pouze z příkazové řádky. V repozitáři už ale lze nalézt i základ GUI prohlížeče. Ten bude v budoucnu znamenat významné vylepšení, protože umožní snadnější interakci s nasbíranými daty. Jeho vývoj byl prozatím pozastaven a bude pokračovat až ve chvíli, kdy bude symbolizace přesunuta do pozdější fáze a dojde k ustálení formátu výstupního souboru. Dalším vylepšením bude možnost instalace celého projektu.

Heaprof by dále mohl obsahovat API umožňující ovládání profileru přímo z profilovaného programu. Dalo by se tak například spustit profilování pouze určité části kódu. Hudbou vzdálené budoucnosti je pak podpora dalších platforem. Pro systémy, na kterých existuje mechanismus `LD_PRELOAD`, by nemuselo být složité zprovoznit alespoň základní a rozšířený režim profilování.

Závěr

V diplomové práci byla analyzována problematika operací souvisejících s dynamickou alokací paměti doprovázená popisem paměťového manažeru v GNU C knihovně. Byly prozkoumány existující nástroje pro profilování haldy a byl diskutován princip jejich fungování.

Důležitou funkcionalitou profileru je získání backtrace v běžícím programu. Práce představila prostředky dostupné pro operační systém Linux, s jejichž pomocí lze tento úkon provést. Profiler haldy musí být také schopen zachytit volání alokačních a uvolňovacích funkcí. Často využívaná `LD_PRELOAD` technika s sebou přináší svá úskalí, kterým musí profiler čelit. Práce tyto problematické aspekty rozebrala a představila jejich možná řešení.

Přínosem práce byl dále návrh profileru haldy s nízkou latencí postavený na principu maximálního využití lokálních cache. Ve vícevláknových aplikacích není díky navržené metodě nutné při každé dynamické alokaci zajišťovat výlučný přístup k prostředku sdílenému všemi vlákny profilovaného programu.

V praktické části práce byl vytvořen profiler Heaprof využívající navrženou metodu. Experimentální vyhodnocení implementovaného profileru ukázalo, že Heaprof sice přidal k běhu programu měřitelnou časovou režii, ta však na rozdíl od jiných dostupných profilerů nerostla s tím, jak rostl počet vláken v profilovaném programu. U osmivláknových benchmarků došlo v průměru k prodloužení doby běhu profilovaného programu na 2,5násobek, zatímco u konkurenčích profilerů se jednalo o 17,2násobek resp. 19,2násobek.

Tím byla zodpovězena otázka položená v úvodu práce: „Nebylo by možné vytvořit profiler haldy s podstatně nižší latencí?“ Odpovědí je ano.

Bibliografie

1. LOVE, Robert. *Linux System Programming*. 1. vyd. Sebastopol, CA: O'Reilly Media, Inc., 2007. ISBN 978-0-596-00958-8.
2. BOVET, Daniel P. a Marco Cesati. *Understanding the Linux Kernel*. 3. vyd. Sebastopol, CA: O'Reilly Media, 2006. ISBN 978-0-596-00565-8.
3. STEVENS, W. Richard a Stephen A. Rago. *Advanced programming in the UNIX® Environment*. 3. vyd. Upper Saddle River, New Jersey: Addison-Wesley Professional, 2013. The Addison-Wesley professional computing series. ISBN 978-0-321-63773-4.
4. LOOSEMORE Sandra, Richard M. Stallman et al. *The GNU C Library Reference Manual* [online]. 1993-2020. Ver. 2.32 [cit. 2021-03-14]. Dostupné z: `https://www.gnu.org/software/libc/manual/pdf/libc.pdf`.
5. *Linux manual page: alloca(3)* [online]. 2020-12-21 [cit. 2021-03-14]. Dostupné z: `https://man7.org/linux/man-pages/man3/alloca.3.html`.
6. KRZYZANOWSKI, Paul. Stack frames: A really quick explanation of stack frames and frame pointers. In: *PK.ORG* [online]. February 16, 2018 [cit. 2021-03-14]. Dostupné z: `https://www.cs.rutgers.edu/~pxk/419/notes/frames.html`.
7. *Linux manual page: brk(2)* [online]. 2020-12-21 [cit. 2021-03-15]. Dostupné z: `https://man7.org/linux/man-pages/man2/brk.2.html`.
8. IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R)). *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)*. 2008, s. 1–3874. Dostupné z DOI: `10.1109/IEEESTD.2008.4694976`.
9. *Linux manual page: mremap(2)* [online]. 2020-12-21 [cit. 2021-04-18]. Dostupné z: `https://man7.org/linux/man-pages/man2/mremap.2.html`.

10. GPERFTOOLS. gperftools. In: *Github* [online]. 3 Mar 2021 [cit. 2021-04-11]. Dostupné z: `https://github.com/gperftools/gperftools`. Licence BSD-3-Clause License, Release gperftools-2.9.1.

11. GOOGLE. TCMalloc. In: *Github* [online]. 3 Apr 2021 [cit. 2021-04-11]. Dostupné z: `https://github.com/google/tcmalloc`. Licence Apache-2.0 License, Commit cda8d49b4b7956433ba5eb5c0296882061075fe0.

12. JEMALLOC MEMORY ALLOCATOR. jemalloc. In: *Github* [online]. 5 Aug 2019 [cit. 2021-04-18]. Dostupné z: `https://github.com/jemalloc/jemalloc`. Release 5.2.1.

13. BEGUN. lockfree-malloc. In: *Github* [online]. 18 Jul 2018 [cit. 2021-04-18]. Dostupné z: `https://github.com/Begun/lockfree-malloc`. Licence LGPL-3.0 License, Commit 915f51b282c5a31b18fb9b96ef19da4-10b00a421.

14. ISO/IEC. *Working Draft, Standard for Programming Language C++* [online]. International Organization for Standardization, 2017-03-21 [cit. 2021-04-21]. Dostupné z: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf`. N4659 Commitee draft.

15. C++ Weekly - Ep 256 - C++11's Garbage Collector. In: *Youtube* [online]. 25. 1. 2021 [cit. 2021-03-14]. Dostupné z: `https://www.youtube.com/watch?v=jZ2pXlcDGFc`. Kanál uživatele C++ Weekly With Jason Turner.

16. ISO/IEC. *ISO/IEC 9899:201x — Programming languages — C* [online]. International Organization for Standardization, April 12, 2011 [cit. 2021-03-14]. Dostupné z: `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`. N1570 Commitee draft.

17. *Linux manual page: malloc(3)* [online]. 2020-12-21 [cit. 2021-03-14]. Dostupné z: `https://man7.org/linux/man-pages/man3/malloc.3.html`.

18. WHITNEY Tyler, Colin Robertson et al. *Microsoft Documentation: realloc* [online]. Microsoft, 09/11/2020 [cit. 2021-03-14]. Dostupné z: `https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/realloc`.

19. *Linux manual page: posix_memalign(3)* [online]. 2020-12-21 [cit. 2021-03-14]. Dostupné z: `https://man7.org/linux/man-pages/man3/valloc.3.html`.

20. WHITNEY Tyler, Kent Sharkey et al. *Microsoft Technical Documentation: _expand* [online]. Microsoft, 04/02/2020 [cit. 2021-03-14]. Dostupné z: `https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/expand`.

21. STANDARD C++ FOUNDATION. Memory Management. *isocpp.org* [online]. © Copyright 2021 [cit. 2021-04-16]. Dostupné z: `https://isocpp.org/wiki/faq/freestore-mgmt`.

22. CppCon 2019: Milian Wolff "How to Write a Heap Memory Profiler". In: *Youtube* [online]. 2. 10. 2019 [cit. 2021-04-08]. Dostupné z: https://www.youtube.com/watch?v=YB0QoWI-g8E. Kanál uživatele CppCon.

23. *Linux manual page: proc(5)* [online]. 2020-12-21 [cit. 2021-03-14]. Dostupné z: https://man7.org/linux/man-pages/man5/proc.5.html.

24. *Linux manual page: pmap(1)* [online]. 2020-06-04 [cit. 2021-03-14]. Dostupné z: https://man7.org/linux/man-pages/man1/pmap.1.html.

25. FREE SOFTWARE FOUNDATION, INC. Debugging with GDB: the GNU Source-Level Debugger. *sourceware.org* [online]. Tenth Edition. Copyright © 1988-2021 [cit. 2021-04-16]. Dostupné z: https://sourceware.org/gdb/current/onlinedocs/gdb/.

26. *Linux manual page: vdso(7)* [online]. 2020-12-21 [cit. 2021-03-14]. Dostupné z: https://man7.org/linux/man-pages/man7/vdso.7.html.

27. BLOCK, Frank a Andreas Dewald. *Linux Memory Forensics: Dissecting the User Space Process Heap*. Friedrich-Alexander-Universität Erlangen-Nürnberg, Dept. of Computer Science, April 2017. Tech. zpr., CS-2017-02. ISSN 2191-5008.

28. The GNU C Library Release Timeline. In: *glibc wiki* [online]. 2021-02-02 [cit. 2021-03-18]. Dostupné z: https://sourceware.org/glibc/wiki/Glibc%20Timeline. Naposledy editoval SiddheshPoyarekar dne 2021-02-02 12:17:10.

29. Overview of Malloc. In: *glibc wiki* [online]. 2021-02-02 [cit. 2021-03-18]. Dostupné z: https://sourceware.org/glibc/wiki/MallocInternals. naposledy editoval DJ Delorie dne 2019-05-10 23:10:47.

30. SPLOITFUN. Understanding glibc malloc. In: *sploitF-U-N* [online]. February 10, 2015 [cit. 2021-03-18]. Dostupné z: https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/.

31. GNU PROJECT. The GNU C Library. In: *sourceware.org* [online]. Dec 3 2020. Dostupné také z: https://sourceware.org/git/glibc.git. GIT repozitář.

32. *Linux manual page: malloc_usable_size(3)* [online]. 2020-12-21 [cit. 2021-03-18]. Dostupné z: https://man7.org/linux/man-pages/man3/malloc_usable_size.3.html.

33. *Linux manual page: mallopt(3) - Linux manual page* [online]. 2021-03-22 [cit. 2021-04-02]. Dostupné z: https://man7.org/linux/man-pages/man3/mallopt.3.html.

34. *Linux manual page: mallinfo(3)* [online]. 2021-03-22 [cit. 2021-04-02]. Dostupné z: https://man7.org/linux/man-pages/man3/mallinfo.3.html.

35. *Linux manual page: malloc_stats(3)* [online]. 2021-03-22 [cit. 2021-04-02]. Dostupné z: https://man7.org/linux/man-pages/man3/malloc_stats.3.html.

36. *Linux manual page: malloc_info (3)* [online]. 2021-03-22 [cit. 2021-04-02]. Dostupné z: https://man7.org/linux/man-pages/man3/malloc_info.3.html.

37. VALGRIND™ DEVELOPERS. Valgrind. *valgrind.org* [online]. Copyright © 2000-2021 [cit. 2021-04-04]. Dostupné z: https://www.valgrind.org/.

38. SEWARD, Julian a Nicholas Nethercote et al. *Valgrind Documentation: 2. Writing a New Valgrind Tool* [online]. Valgrind, 19 Mar 2021 [cit. 2021-04-04]. Dostupné z: https://www.valgrind.org/docs/manual/manual-writing-tools.html. Release 3.17.0.

39. NETHERCOTE, Nicholas a Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: Association for Computing Machinery. *SIGPLAN Not.* New York, NY, USA, 2007, sv. 42, s. 89–100. Č. 6. ISBN 9781595936332. ISSN 0362-1340. Dostupné z DOI: 10.1145/1273442.1250746.

40. SEWARD, Julian a Nicholas Nethercote et al. *Valgrind Documentation: Valgrind's Tool Suite* [online]. 19 Mar 2021 [cit. 2021-04-04]. Dostupné z: https://www.valgrind.org/info/tools.html. Release 3.17.0.

41. SEWARD, Julian a Nicholas Nethercote et al. *Valgrind Documentation: 9. Massif: a heap profiler* [online]. 19 Mar 2021 [cit. 2021-04-05]. Dostupné z: https://valgrind.org/docs/manual/ms-manual.html. Release 3.17.0.

42. KDE GITHUB MIRROR. massif-visualizer. In: *Github* [online]. 7 Sep 2017 [cit. 2021-04-27]. Dostupné z: https://github.com/KDE/massif-visualizer. Licence GPL-2.0 License, Release v0.7.0.

43. KDE GITHUB MIRROR. heaptrack - a heap memory profiler for Linux. In: *Github* [online]. 2 Sep 2020 [cit. 2021-04-14]. Dostupné z: https://github.com/KDE/heaptrack. Licence LGPL-2.1 License, Release v1.2.0 - improved stability.

44. WOLFF, Milian. Heaptrack - A Heap Memory Profiler for Linux. *milianw.de* [online]. December 02, 2014 [cit. 2021-04-08]. Dostupné z: https://milianw.de/blog/heaptrack-a-heap-memory-profiler-for-linux.html.

45. *Linux manual page: memusage(1)* [online]. 2021-03-22 [cit. 2021-04-13]. Dostupné z: https://man7.org/linux/man-pages/man1/memusage.1.html.

46. GAPPMEIER, Gerhard. Profiling Memory using GNU glibc tools. *Gergap's Weblog* [online]. January 20, 2017 [cit. 2021-04-13]. Dostupné z: https://gergap.wordpress.com/tag/memusage/.

47. MEMORY TRACKING TOOLS. MALT : Malloc Tracker. In: *Github* [online]. 17 Oct 2020 [cit. 2021-04-13]. Dostupné z: https://github.com/memtt/malt. Licence CeCILL-C, Release Version 1.2.1 (October 17, 2020).

48. EPFL VLSC. Memoro: A Detailed Heap Profiler. In: *Github* [online]. 14 Jan 2021 [cit. 2021-04-14]. Dostupné z: https://github.com/epfl-vlsc/memoro. Licence MIT License, Commit 5528924ea3a494813111a7c06255fb5d01d1da6a.

49. BYMA, Stuart a James R. Larus. Detailed Heap Profiling. In: Association for Computing Machinery. *SIGPLAN Not.* New York, NY, USA, 2018, sv. 53. Č. 5. ISSN 0362-1340. Dostupné z DOI: 10.1145/3299706.3210564.

50. BOURLET, François-Xavier. Backward-cpp. In: *Github* [online]. 5 Oct 2020 [cit. 2021-04-30]. Dostupné z: https://github.com/bombela/backward-cpp. Licence MIT License, Commit f7ad5142e1dee10065ca2bd23fbd585400d56fc9.

51. *Linux manual page: addr2line(1)* [online]. 2021-02-06 [cit. 2021-04-22]. Dostupné z: https://man7.org/linux/man-pages/man1/addr2line.1.html.

52. *Linux manual page: backtrace(3)* [online]. 2021-03-22 [cit. 2021-04-23]. Dostupné z: https://man7.org/linux/man-pages/man3/backtrace.3.html.

53. FREE SOFTWARE FOUNDATION, INC. GNU Compiler Collection (GCC) Internals. *gcc.gnu.org* [online]. Copyright © 1988-2021 [cit. 2021-04-16]. Dostupné z: https://gcc.gnu.org/onlinedocs/gccint/.

54. BASTIAN Théophile, Stephen Kell a Francesco Zappa Nardelli. Reliable and Fast DWARF-Based Stack Unwinding. *Proc. ACM Program. Lang.* 2019, roč. 3, OOPSLA, č. 146 (October 2019), s. 24. Dostupné z DOI: 10.1145/3360572.

55. WATSON, Dave et al. The libunwind project. *nongnu.org* [online]. [N.d.] [cit. 2021-04-04]. Dostupné z: https://www.nongnu.org/libunwind/.

56. LIBUNWIND. libunwind. In: *Github* [online]. 31 Mar 2020 [cit. 2021-04-14]. Dostupné z: https://github.com/libunwind/libunwind. Licence MIT License, Release libunwind 1.4.0.

57. *Linux manual page: dlopen(3)* [online]. 2021-03-22 [cit. 2021-04-24]. Dostupné z: https://man7.org/linux/man-pages/man3/dlopen.3.html.

58. *Linux manual page: dl_iterate_phdr(3)* [online]. 2021-03-22 [cit. 2021-04-24]. Dostupné z: https://man7.org/linux/man-pages/man3/dl_iterate_phdr.3.html.

59. *Linux manual page: dladdr(3)* [online]. 2021-03-22 [cit. 2021-04-24]. Dostupné z: https://man7.org/linux/man-pages/man3/dladdr.3.html.

60. FREE SOFTWARE FOUNDATION, INC. GNU Binutils. *gnu.org* [online]. 2021/04/13 [cit. 2021-04-25]. Dostupné z: http://www.gnu.org/software/binutils/.

61. DREPPER, Ulrich et al. ELFUTILS. *sourceware.org* [online]. [N.d.] [cit. 2021-04-25]. Dostupné z: https://sourceware.org/elfutils/.

62. ANDERSON, David. David A's DWARF Page. *prevanders.net* [online]. [N.d.] [cit. 2021-04-25]. Dostupné z: https://www.prevanders.net/dwarf.html.

63. DOCOPT. docopt.cpp: A C++11 Port. In: *Github* [online]. 17 Jun 2020 [cit. 2021-04-25]. Dostupné z: https://github.com/docopt/docopt.cpp. Licence Boost Software License – Version 1.0, Commit 6f5de76970be94a6f1e4556d1716593100e285d2.

64. CARLINI, Paolo et at. The GNU C++ Library Manual. *gcc.gnu.org* [online]. Copyright (C) 2008-2021 [cit. 2021-04-26]. Dostupné z: https://gcc.gnu.org/onlinedocs/libstdc++/manual/index.html.

65. *Linux manual page: malloc_hook(3)* [online]. 2021-03-22 [cit. 2021-03-23]. Dostupné z: https://man7.org/linux/man-pages/man3/malloc_hook.3.html.

66. *Linux manual page: ld.so(8)* [online]. 2021-03-22 [cit. 2021-03-23]. Dostupné z: https://man7.org/linux/man-pages/man8/ld.so.8.html.

67. *Linux manual page: dlsym(3)* [online]. 2021-03-22 [cit. 2021-03-23]. Dostupné z: https://man7.org/linux/man-pages/man3/dlsym.3.html.

68. LANGR, Daniel a Martin Kočička. Reducing the Impact of Intensive Dynamic Memory Allocations in Parallel Multi-Threaded Programs. In: IEEE Computer Society. *IEEE Transactions on Parallel & Distributed Systems*. Los Alamitos, CA, USA, 2020, sv. 31, s. 1152–1164. Č. 05. ISSN 1558-2183. Dostupné z DOI: 10.1109/TPDS.2019.2960514.

69. BERGER, Emery D. a Kathryn S McKinley et al. Hoard: a scalable memory allocator for multithreaded applications. In: Association for Computing Machinery. *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*. Cambridge, Massachusetts, United States: ACM, 2000, s. 117–128. ISBN 1-58113-317-0. Dostupné z DOI: http://doi.acm.org/10.1145/378993.379232.

70. BERGER, Emery. The Hoard Memory Allocator: A Fast Scalable ... In: *Github* [online]. 2 Jan 2019 [cit. 2021-04-30]. Dostupné z: `https://github.com/emeryberger/Hoard`. Licence Apache License Version 2.0, Release Version 3.13.

71. MICROQUILL INC. *smartheap/shbench* [online]. [N.d.] [cit. 2021-04-30]. Dostupné z: `http://www.microquill.com/smartheap/shbench/`.

72. LOHMANNR, Niels. JSON for Modern C++. In: *Github* [online]. 29 Apr 2021 [cit. 2021-04-30]. Dostupné z: `https://github.com/nlohmann/json`. Licence MIT License, Commit 8a29a6ecf4d228b5d7807a2df5ac352806da8ae8.

Příloha **A**

Seznam použitých zkratek

API Application Programming Interface

ASLR Address Space Layout Randomization

CPU Central Processing Unit

CRTP Curiously Recurring Template Pattern

DWARF Debugging with Arbitrary Record Formats

ELF Executable and Linkable Format

GNU GNU's Not Unix!

GUI Graphical User Interface

JIT Just-In-Time

JSON JavaScript Object Notation

LIFO Last In First Out

IP Instruction Pointer

IR Intermediate Representation

PID Process Identifier

RAM Random-Access Memory

RSS Resident Set Size

SVG Scalable Vector Graphics

XML Extensible Markup Language

VAS Virtual Address Space

VSZ Virtual Memory Size

Příloha **B**

Obsah přiloženého CD

```
cti_me.txt ..... Stručný popis obsahu CD
Jan_Tomanek_DP.pdf ..... Text práce ve formátu PDF
zdrojove_kody
    heaprof ..... GIT repozitář projektu Heaprof
        benchmark ..... Zdrojové kódy benchmarků
        src ..... Zdrojové kódy implementace
    thesis ..... Zdrojová forma práce ve formátu LaTeX
```