

VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Vizualizace ladění aplikací

Vizualization of applications debugging

Zadání bakalářské práce

Student:

Jakub Beránek

Studijní program:

B2647 Informační a komunikační technologie

Studijní obor:

2612R025 Informatika a výpočetní technika

Téma:

Vizualizace ladění aplikací
Vizualization of Applications Debugging

Jazyk vypracování:

čeština

Zásady pro vypracování:

Většina aplikací, které se používají v prostředí Linuxu pro ladění C/C++ aplikací, jsou konzolové aplikace. Hlavním cílem bakalářské práce bude vytvořit grafickou nadstavbu pro vybraný ladící nástroj, která by umožňovala krokovat aplikaci a vizualizovat stav paměti při ladění. Cíle práce lze shrnout v těchto bodech.

1. Seznamte se s ladícími nástroji pro C/C++ a vyberte vhodný nástroj.
2. Navrhněte infrastrukturu, která umožní program laděný pomocí toho vybraného nástroje vzdáleně ovládat a krokovat. Navržené řešení implementujte.
3. Vytvořte aplikaci, která bude řídit ladění programu a vizualizovat stav paměti při jeho vykonávání.

Seznam doporučené odborné literatury:

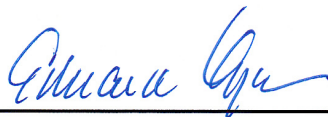
Podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Marek Běhálek, Ph.D.**

Datum zadání: 01.09.2015

Datum odevzdání: 29.04.2016



doc. Dr. Ing. Eduard Sojka
vedoucí katedry



prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 29. dubna 2016

.....

Chtěl bych poděkovat svému vedoucímu, Ing. Marku Běhálkovi, Ph.D., za pomoc při tvorbě této práce.

Abstrakt

Tato bakalářská práce se zabývá vizualizací paměti programů napsaných v jazycích C a C++, která má sloužit k usnadnění jejich ladění a snazšímu pochopení jejich průběhu. První část uvádí obecné principy ladění programů a metody jejich implementace. Druhá část popisuje a srovnává běžně používané ladící nástroje a jejich grafická rozhraní. Třetí část se zabývá návrhem a implementací knihovny pro komunikaci s debuggery a tvorbou grafického nástroje, který vizualizuje paměť laděného procesu.

Klíčová slova: ladění programů, vizualizace paměti

Abstract

This thesis deals with visualizing memory of programs written in C and C++. This visualization serves to facilitate debugging and understanding of program behaviour. The first part introduces general principles of program debugging and some approaches to their implementation. The second part describes and compares common debugging tools and their graphical interfaces. The third part discusses design and implementation of a library for communicating with debuggers and also describes a graphical tool that visualizes the memory of a debugged process.

Key Words: debugging, memory visualization

Obsah

Seznam použitých zkratek a symbolů	7
Seznam obrázků	8
1 Úvod	9
2 Principy ladění programů	11
2.1 Signály	11
2.2 Krokování	11
2.3 Metody zastavení procesu	12
2.4 Mapování zdrojového kódu na instrukce	13
2.5 Adresní prostor procesu	14
3 Existující ladící nástroje	16
3.1 GDB	16
3.2 LLDB	17
3.3 Grafická rozhraní	18
4 Architektura aplikace	22
4.1 Specifikace požadavků	22
4.2 Použité technologie	22
5 Knihovna pro komunikaci s debuggery	24
5.1 Python API pro GDB	25
5.2 Python API pro LLDB	25
5.3 Protokol GDB MI	26
6 Vizualizační aplikace	31
6.1 Vykreslovací komponenta	31
6.2 Používání aplikace	34
7 Testování dosažených výsledků	37
8 Závěr	39
Literatura	40
Přílohy	41
A Struktura projektu a instalační manuál	41

Seznam použitých zkratek a symbolů

ABI	– Application binary interface
API	– Application programmable interface
GCC	– GNU Compiler Collection
GDB	– GNU Project Debugger
GNU	– GNU's Not Unix!
GUI	– Graphical user interface
IPC	– Inter-process communication
JSON	– JavaScript Object Notation
TCP	– Transmission Control Protocol

Seznam obrázků

1	Zásobníkový rámeček platformy x86	15
2	Uživatelské rozhraní programu DDD	19
3	Uživatelské rozhraní programu CLion	20
4	Uživatelské rozhraní programu GDB (TUI)	21
5	Architektura <i>Devi</i>	23
6	Komunikace s laděným procesem	28
7	Třídní diagram části vykreslovací komponenty	32
8	Přehled vizualizačních prvků	33
9	Uživatelské rozhraní <i>Devi</i>	34

1 Úvod

Jazyky C a C++ jsou velmi komplexní a pracují s hardwarem počítače na nízké úrovni. Kvůli absenci automatické správy paměti a možnosti provádět nekontrolované operace s pamětovými ukazateli jsou velmi náchylné ke vzniku těžko odhalitelných chyb. Mnoho chyb, hlavně u programátorů začínajících s těmito jazyky, vzniká proto, že programátor nemá jasnou představu o tom, co se děje na pozadí programu za jeho běhu. Zdrojový kód poskytuje pouze silně statický popis programu a nedokáže plně zachytit dynamiku pamětových struktur běžícího programu. Vznik a zánik zásobníkových rámců při volání funkcí, dealokace proměnných, rozdíl mezi alokací na zásobníku a na haldě, to vše jsou věci, které jsou potřebné pro pochopení fungování programů napsaných v jazycích C a C++. Přitom tyto informace nejsou ve většině výukových kurzů a učebnic programování explicitně vysvětleny, anebo jsou odloženy až do pokročilých lekcí. Programátorovi, který tomuto chování plně nerozumí, pak průběh programu může přijít neintuitivní a neumí si vysvětlit, proč se program chová jinak, než by měl.

Pro nalezení chyb v programu a lepší porozumění jeho chování lze použít **debugger** (ladicí nástroj). Debuggery vytváří asociaci mezi zdrojovým kódem a spustitelným souborem a poskytují tak tvůrci kódu možnost ladit kód na vysoké úrovni abstrakce, tj. na úrovni samotného zdrojového kódu. Umožňují spustit program, na předem zvolených místech ho zastavit a poté jej spouštět po malých úsecích kódu (tzv. ho *krokovat*), což se používá pro nalezení úseků programu obsahujících chyby. Existující debugery pro jazyky C a C++ na Linuxových systémech mají velké množství funkcí, ale samy o sobě poskytují svému uživateli obvykle pouze ladění pomocí terminálu, což je pro velké programy nepřehledné a pomalé řešení. Proto pro ně vzniklo velké množství grafických rozhraní, ať už samostatných nebo integrovaných v komplexních vývojových prostředích, které práci s nimi usnadňují. Tato rozhraní typicky zobrazují stav paměti procesu v textové podobě, která je sice vhodná pro rychlé zjištění hodnot proměnných, ale nezobrazuje vztahy mezi jednotlivými objekty ani jejich umístění v paměti. Nedovoluje si tak udělat ucelenější, globální pohled na paměť procesu a pochopit, co přesně se v ní děje po provedení určitého příkazu. Tento širší pohled by mohla poskytnout grafická reprezentace, která by umožnila popsat umístění objektů v jednotlivých pamětových segmentech, zobrazit vztahy mezi nimi (např. pamětové ukazatele) a poskytnout snadno pochopitelný pohled na celkový stav paměti.

Grafická vizualizace paměti programů není novým tématem, zabývalo se jí už mnoho programů. Za zmínku stojí **Python Tutor**¹, který vizualizuje paměť programu ve formě objektů a odkazů mezi nimi. [8] Primárně je určen pro jazyk Python, ale obsahuje rozšíření i pro jazyky Java, JavaScript, TypeScript, Ruby, C a C++. Tento nástroj ovšem nezobrazuje stav programů za jejich běhu, pouze zpětně zobrazuje jejich stav, který je průběžně ukládán. Dovoluje tedy ladit program až po jeho skončení (tzv. post-mortem debugging). To je v určitých případech, například u masivně paralelních aplikací, jediným použitelným způsobem ladění. Nicméně užitečnější je ladit programy přímo za jejich běhu a mít tak možnost ovlivňovat jejich průběh. Tento

¹<http://pythontutor.com>

nástroj navíc za běhu programu ukládá po každém příkazu (resp. instrukci) celkový stav paměti a generuje tak velký objem dat. Kvůli toho je obtížné ho použít pro ladění větších a složitějších programů.

Sorva [11, s. 140] ve své práci o vizuální simulaci programů uvádí a kategorizuje několik desítek programů, které se zabývají zobrazováním stavu běžícího procesu v rozličných formách. Většina z nich slouží jako nástroje pro výuku programování, dle zjištění Sorvy totiž vizualizace skrytých procesů usnadňuje studentům pochopit, jak programy fungují. [11, s. 212]

Cílem této práce je vytvořit jednoduše použitelné grafické rozhraní, které bude umožňovat ladit programy a vizualizovat paměť procesu ve formě diagramu objektů. Mělo by být užitečné jak pro výuku programování a pochopení fungování programů napsaných v jazycích C a C++, tak pro ladění reálných aplikací. Diagram paměti by měl zobrazovat jednotlivé objekty a struktury, jejich hodnotu, umístění v paměti a vztahy s ostatními objekty. Tímto by měl poskytnout svým uživatelům snadnější představu o tom, co se v děje v paměti programu za jeho běhu. Kromě ladění může být tento diagram použit i pro vizualizaci průběhu určitých typů algoritmů (např. třídících). Mimo vizualizace paměti by měl program umožňovat používat běžné funkce debuggerů, jako je krokování, inspekce a úprava paměti nebo přepínání mezi zásobníkovými rámci a vlákny.

Teoretický popis ladění v této práci i samotný vizualizační program je zaměřený na programy pro operační systémy založené na Linuxovém jádře. Pojmem Linuxový systém se v této práci myslí libovolná distribuce Linuxu. Pro označení programů, které umožňují ladění jiných programů, je zde používán termín debugger, jelikož se jedná o často používaný programátorský termín a v češtině pro něj neexistuje zavedená alternativa.

Kapitola 2 popisuje obecné principy ladění, které jsou společné pro používání libovolného debuggeru. Vysvětluje také terminologii potřebnou pro pochopení dalšího textu. Po ní následuje kapitola 3 pojednávající o existujících debuggerech a jejich grafických rozhraních. Kapitola 4 popisuje architekturu aplikace a technologie, které jsou v ní použity. Kapitola 5 se věnuje návrhu a vývoji komunikačního rozhraní pro přístup k debuggerům, které bylo dále použito pro vytvoření vizualizační aplikace, popsané v kapitole 6.

2 Principy ladění programů

Tato kapitola popisuje obecné principy fungování debuggerů, způsob mapování binárních instrukcí programu zpět na jeho zdrojový kód, krokování běžícího procesu a běžné konstrukce používané při ladění. Konkrétně je popis zaměřen na programy napsané v jazycích C a C++ pro Linuxové systémy s procesory s instrukční sadou rodiny Intel x86 (popř. x86-64). Tato platforma je běžně používána pro vývoj C a C++ programů a je i cílovou platformou programu tvořeného v této bakalářské práci. Popsané principy jsou ale obecné a lze je aplikovat na libovolný operační systém.

2.1 Signály

Ladění procesu vyžaduje mít možnost číst jeho paměť a ovládat jeho běh. Čtení paměti je nezbytné k získání informací o proměnných. Ovládání procesu a hlavně jeho zastavení je nezbytné, protože procesy provádějí obrovské množství instrukcí za vteřinu a zkoumat takto rychle se měnící datový tok by bylo obtížné. Proces lze zastavit pomocí signálu zaslaného operačním systémem. Signály jsou zprávy, které lze v procesu odchytnout a zareagovat na ně. [12, s. 21] Slouží pro komunikaci mezi procesy a fungují jako softwarová obdoba hardwarových přerušení procesoru. Jakmile proces obdrží signál, který očekává, tak si uloží hodnoty svých registrů a přejde do procedury, která tento signál obslouží. Pokud proces obdrží signál, pro který si nepřipravil žádnou reakci, tak se provede implicitně nadefinovaná akce pro daný typ signálu. V Linuxových systémech je definováno několik desítek standardních signálů v závislosti na verzi a typu operačního systému. Na signály SIGKILL, sloužící k okamžitému ukončení procesu, a SIGSTOP, sloužící k zastavení procesu, nemá proces možnost zareagovat ani zjistit, že mu byly poslány.

2.2 Krokování

Operační systémy obvykle poskytují nástroj, pomocí kterého lze buď spustit proces, anebo se připojit k již běžícímu procesu, a následně ho ovládat a přistupovat k jeho paměti. Linuxové systémy pro tento účel poskytují systémové volání **ptrace**², které umožňuje zachytávat signály zaslané sledovanému procesu. Proces sledovaný pomocí funkce ptrace je zastaven při přijetí jakéhokoliv signálu (kromě signálu SIGKILL, který se pokusí proces okamžitě ukončit). Tohoto mechanismu využívají debuggery, které proces sledovaný pomocí ptrace můžou po jeho zastavení znovu spustit, přistupovat k jeho paměti a ovlivňovat jeho průběh. Pokud je proces potomkem procesu, který ho sleduje, bude při jeho spuštění vyvolán signál SIGTRAP, který dovolí jeho rodiči na tuto událost zareagovat. Jakmile je proces zastavený, může ho debugger krokovat, tedy spouštět instrukci po instrukci. K tomu lze použít funkci ptrace s příznakem PTRACE_SINGLESTEP, která provede přesně jednu instrukci v laděném procesu (proces se také zastaví, pokud se dostane na vstupní nebo výstupní bod systémového volání).

²<http://linux.die.net/man/2/ptrace>

Debuggery obvykle nabízí krokování na vyšší úrovni než pouze po jedné instrukci, protože to by bylo zbytečně zdlouhavé (u vyšších programovacích jazyků se jeden řádek zdrojového kódu může mapovat na desítky až stovky instrukcí). Při krokování se program obnoví a po provedení určitého počtu instrukcí anebo při vyvolání určité události se opět zastaví. Programátor tak může běh programu posouvat po malých částech, sledovat, jak se mění jeho paměť a případně identifikovat místo, kde je v programu chyba. Debuggery obvykle nabízí následující typy krokování.

- **Krok po řádku** - program se obnoví, provede instrukce odpovídající jednomu řádku zdrojového kódu a poté se opět zastaví
- **Krok dovnitř funkce** - funguje stejně jako krok po řádku, ale program se zastaví i po každém zavolání funkce
- **Krok ven z funkce** - program se obnoví a bude běžet, dokud se nevrátí z funkce, ve které se právě nachází

2.3 Metody zastavení procesu

Pokud by programátor musel krokovat celý program po instrukcích anebo řádcích zdrojového kódu, tak by ladění trvalo velmi dlouho. Proto se obvykle nejprve zvolí zajímavá místa v programu, ve kterých se má program zastavit, poté se program spustí a nechá se běžet, dokud na takto označené místo nenarazí. Takto lze přeskočit dlouho trvající úseky programu bez nutnosti manuálního krokování. Většina debuggerů poskytuje následující možnosti označení míst pro zastavení procesu.

2.3.1 Breakpoint

Jedná se o označení řádku v zdrojovém kódu programu, na kterém se program za běhu zastaví a umožní tak uživateli prozkoumat paměť procesu a krokovat ho. Nejčastěji je implementován tak, že debugger zjistí z vybraného řádku adresu instrukce ve vygenerovaném spustitelném souboru, kterou tento řádek představuje, uloží si ji a nahradí ji instrukcí přerušení s kódem 3. Toto přerušení je určeno speciálně pro ladění procesů, jelikož generuje instrukci o velikosti jednoho bytu, kterou lze nahradit libovolnou jinou instrukcí. [9, s. 306] Pokud by tato instrukce měla více než jeden byte, mohlo by se stát, že by přepsala následující instrukci, což by mohlo způsobit nevalidní chování programu.

Jakmile program za běhu provede toto přerušení, tak se vyvolá signál SIGTRAP, který debugger odchytí a laděný proces se tímto zastaví. Pokud se uživatel rozhodne proces opět spustit, debugger zkopíruje původní instrukci programu (kterou si dříve uložil) na místo, kde vložil přerušení, nastaví na ni ukazatel příští instrukce (registr *instruction pointer*) a proces opět spustí. [1] Některé procesory nabízí také hardwarový breakpoint, který sice může být rychlejší, ale obvykle kvůli hardwarové implementaci umožňuje vytvořit pouze několik breakpointů zároveň.

2.3.2 Tracepoint

V některých případech není možné laděný proces pozastavit k prozkoumání jeho paměti, například pokud jeho průběh závisí na reálně uběhlém čase a jeho zastavení může způsobit, že program neproběhne korektně. Pro tyto situace lze použít tracepoint, u kterého se uvede lokace v programu a paměť, která má být sledována. Pokaždé, když se laděný proces dostane na tuto lokaci, tak je uložena sledovaná paměť a po skončení běhu procesu si lze zpětně prohlédnout, jak se tato paměť v průběhu programu měnila.

2.3.3 Watchpoint

Pokud je potřeba zastavit program ne na konkrétním místě, ale při změně určité hodnoty v paměti, lze použít watchpoint. Ten se může hodit například pro kontrolu změn globálních proměnných. Pokud nenabízí procesor hardwarovou podporu pro watchpointy, debugger musí procházet laděný proces instrukci po instrukci, testovat hodnotu sledované paměti a pokud se tato hodnota změní, tak program zastavit. To může zpomalit laděný proces až o dva řády. [4]

2.3.4 Catchpoint

Tuto konstrukci lze použít pro zachycení událostí jako je načtení sdílené knihovny, vyvolání hardwarové či softwarové výjimky, provedení systémového volání anebo obdržení signálu. Většina těchto událostí je ze své podstaty asynchronní, nelze u nich tedy dopředu určit, kdy budou vyvolány a použití klasického breakpointu pro ně není možné.

2.4 Mapování zdrojového kódu na instrukce

Aby mohly ladící nástroje nabízet krokování na úrovni (řádků) zdrojového kódu, musí mít asociaci mezi zdrojovým kódem a instrukcemi vygenerovaného spustitelného programu. Jelikož programy psané v jazycích C a C++ jsou kompilované a po jejich překladu nejsou ve výsledném binárním souboru téměř žádné informace o jejich zdrojovém kódu, musí být přeloženy ve speciálním režimu, který při překladu vygeneruje metadata s údaji o zdrojovém kódu a vloží je do přeloženého programu. V překladačích jazyka C/C++ se toho dá standardně dosáhnout použitím přepínače **-g**. Existuje několik formátů ukládání těchto metadat, dnešním de facto standardem na Linuxových systémech je DWARF³. Ten ukládá proměnné, datové typy, procedury a další údaje ze zdrojového kódu ve stromové struktuře. Debuggery poté využívají těchto metadat pro vytvoření asociace mezi instrukcemi a zdrojovým kódem programu.

Tato metadata nicméně nejsou pro debugger použitelná, pokud je laděný program vygenerován s optimalizacemi. Poté totiž program nemusí obsahovat všechny původní proměnné, funkce a jeho průběh ani nemusí přesně odpovídat původnímu zdrojovému kódu. Při použití málo agresivní optimalizace lze někdy programy úspěšně ladit, ale pro zajištění co nejpřesnějšího ladění

³<http://dwarfstd.org>

programů je obvykle nutné optimalizace zcela vypnout. Toho lze v překladačích obvykle dosáhnout použitím přepínače **-O0**. Sada překladačů GCC nabízí od verze 4.8 přepínač **-Og**, který se snaží dosáhnout kompromisu a optimalizovat program co nejvíce bez použití optimalizací, které kolidují s laděním.⁴

2.5 Adresní prostor procesu

Adresní prostor procesů je rozdělen do několika segmentů. [12, s. 53] Jedním z nich je zásobník (*stack*), což je úsek paměti, který obsahuje lokální proměnné a parametry funkcí a údaje o jejich jednotlivých voláních. Pokaždé, když proces zavolá funkci, tak se na zásobník uloží tzv. *zásobníkový rámec*. Informace o současném rámci jsou uloženy v procesorových registrech *esp* a *ebp* na 32-bitových systémech a *rsp* a *rbp* na 64-bitových systémech. Registr *ebp* ukazuje na bázi současného rámce a registr *esp* na jeho vrchol, který má ale ve skutečnosti nižší adresu v paměti, jelikož zásobník obvykle roste v paměti směrem dolů. Při zavolání funkce se na zásobník vloží předané parametry, návratová adresa, na kterou se má skočit po dokončení funkce, hodnota báze předchozího rámce a lokální proměnné dané funkce (v tomto pořadí). Jelikož registr *ebp* ukazuje na neměnné místo v zásobníkovém rámci, tak se kód funkce na lokální proměnné a parametry obvykle odkazuje relativně k jeho hodnotě.

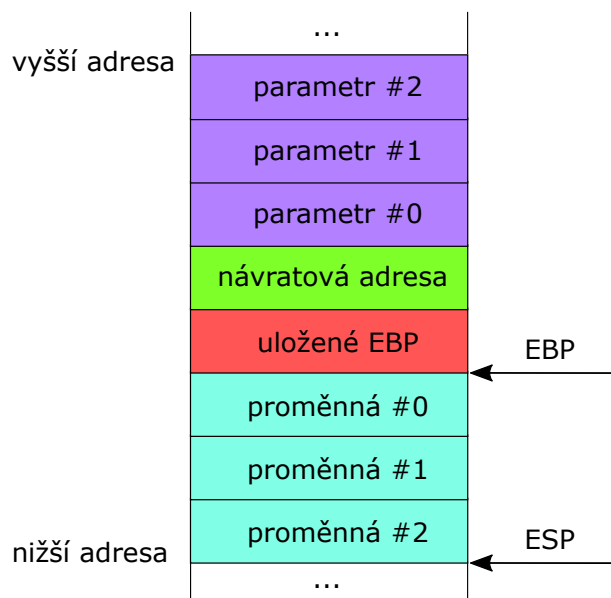
Na architektuře x86-64 je navíc ještě definována červená zóna [10, s. 16], což je oblast o velikosti 128 bytů, která je umístěná pod ukazatelem vrcholu zásobníku (registr *rsp*). Lze jí použít k libovolným účelům, překladače tento prostor obvykle využívají u listových funkcí (tj. funkcí, které nevolají žádné další funkce). U nich se totiž nemusí nijak měnit hodnota *rsp*, a proto zde můžou libovolně používat paměť bez zbytečné změny hodnoty tohoto registru. Je zaručeno, že tato paměť nebude modifikována žádnými obslužnými procedurami signálů ani přerušení.

Parametry se na zásobník předávají od nejpravějšího parametru (z hlediska volání funkce ve zdrojovém kódu), parametry nejvíce vlevo tedy skončí nad návratovou adresou a funkce bude vždy znát jejich pozici a bude k nim moci přistoupit.⁵ Z těchto prvních parametrů poté funkce může odvodit celkový počet předaných parametrů.⁶ Pokud by se parametry předávaly zleva, tak počáteční parametry by skončily na dopředu neznámém místě na zásobníku, a volaná funkce by nevěděla, kolik parametrů jí bylo předáno. Ukázku zásobníkového rámce platformy x86 si lze prohlédnout na obrázku 1.

⁴<https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>

⁵Detaily ukládání parametrů a získávání návratové hodnoty ze zásobníkového rámce jsou silně závislé na konkrétní architektuře a použitém způsobu volání funkcí (*call convention*). Zde je popsána standardní konvence jazyka C – *cdecl* – pro 32-bitovou architekturu.

⁶Například funkci `printf` se nejprve předá formátovací řetězec, a podle toho, kolik je v něm zástupných symbolů, se poté načte potřebný počet parametrů ze zásobníku.



Obrázek 1: Zásobníkový rámec platformy x86

Z hlediska ladění je zásobník zajímavý, jelikož jeho stav reprezentuje funkci, ve které je proces zastaven, a zároveň také všechna volání funkcí, které dané situaci předcházely. Lze tak analyzovat, jakou cestou se proces dostal do určitého stavu, což je často jedna z prvních věcí, která programátora při ladění zajímá. Pokud překladač vygeneroval kód, který při volání každé funkce korektně provádí její prolog (uložení registru *ebp*) i epilog (obnovení registru *ebp*), je pro debugger jednoduché provést zpětný průchod zásobníkem a zjistit informace o všech zásobníkových rámcích. Kromě použití při ladění lze tyto údaje použít například i pro implementaci výjimek jazyka C++.

Toto chování nicméně spotřebuje jeden registr procesoru pro ukládání informací o rámcích a vyžaduje provedení celého zmíněného postupu při každém zavolání funkce, a to může proces zpomalovat. Při návrhu binárního programovacího rozhraní (ABI) architektury x86-64 bylo rozhodnuto, že toto chování nebude vynucováno [10, s. 16]. Překladače této skutečnosti obvykle využívají, rámce explicitně nezaznamenávají a adresy lokálních proměnných a parametrů počítají relativně od vrcholu zásobníku, který je uložen v registru *rsp*. Poté je ale někdy nemožné zobrazit zásobník volaných funkcí v debuggeru, a proto je vhodné se ujistit, že se rámce zaznamenávají⁷. Debuggery můžou informace o volaných funkcích získat i z ladících metadat, pokud je s nimi daný program přeložen, nebo analýzou strojového kódu programu, ale tato řešení nemusí být vždy spolehlivá.

⁷U překladače GCC to lze zajistit pomocí přepínače *-fno-omit-frame-pointer*. Tato volba je implicitně zapnutá při úrovni optimalizace *-O0*.

3 Existující ladící nástroje

Debuggery jsou komplexní programy, jejichž vývoj vyžaduje detailní znalost programovacího jazyka i platformy, pro kterou je daný debugger určen. Jazyky C a C++ samy o sobě neposkytují téměř žádné prostředky usnadňující ladění, na rozdíl od jazyků s běhovým prostředím, jako je například C# nebo Java. Proto je vývoj debuggerů pro tyto jazyky obtížnější a není jich mnoho. V této kapitole jsou popsány dva z těch nejpoužívanějších, GDB a LLDB, které byly také použity při implementaci vizualizačního nástroje. Poté následuje popis vybraných grafických rozhraní těchto debuggerů.

3.1 GDB

GDB (The GNU Debugger) je standardním ladícím nástrojem pro Linuxové systémy, často je v Linuxových distribucích už předinstalovaný. Jeho hlavním zaměřením je ladění programů napsaných v jazycích C a C++, ale podporuje mimo jiné i Adu, Objective-C, Pascal, Fortran, Javu a Go. [6] Podporuje také velké množství instrukčních sad a procesorů, například ARM, AVR, Itanium, MIPS, PowerPC, SPARC a samozřejmě x86 i x86-64. Lze jej spustit i na platformě Windows pomocí prostředí emulujících Linux, jako je Cygwin nebo MinGW. Byl vydán v roce 1986 a k roku 2016 stále vycházejí nové verze.

Umí spolupracovat s programy přeloženými libovolným překladačem jazyků C a C++. Navíc ještě obsahuje speciální podporu pro překladač GCC (GNU Compiler Collection), který pro něj umí vygenerovat dodatečná ladící metadata. GDB obsahuje základní funkce nezbytné pro každý debugger, jako je načítání ladících metadat v mnoha formátech (podporuje DWARF i několik dalších formátů), vytváření breakpointů, tracepointů a watchpointů, krokování a čtení i zápis paměti programu. Neobsahuje vlastní grafické rozhraní, je ovládán z příkazové řádky. Kromě toho ale podporuje také spouštění skriptů v Pythonu pomocí API, které je popsáno v kapitole 5.1. Dále nabízí i pokročilé funkce, které ovšem nemusí být podporované všemi procesory a platformami, s kterými GDB jinak umí pracovat.

GDB dokáže ladit programy spuštěné na jiném zařízení pomocí síťové komunikace (obvykle pomocí protokolu TCP). Toto může být užitečné, pokud není dostupný fyzický přístup k systému, který je potřeba odladit. Vzdálené ladění se dá využít také k ladění jádra (kernelu) operačního systému, čehož je využito například v programu KGDB, který umí ladit kernely operačních systémů Linux, FreeBSD a NetBSD pomocí sériového připojení.

Pokud GDB ladí program, který využívá více než jedno vlákno, může pracovat v několika rozličných módech. [5] V All-stop módu se při zastavení jednoho vlákna zastaví také všechna ostatní vlákna, aby šlo mezi vlákny přepínat a číst jejich paměť bez toho, aby se paměť mezitím nějak změnila. Pokud je nutné zastavit pouze jedno vlákno, aby ostatní vlákna mezitím pokračovala v běhu, lze použít tzv. Non-stop mód, který vždy zastaví pouze vlákno, které narazí na breakpoint, a zbytek vláken nechá běžet. S tímto módem je vhodné použít asynchronní ovládání

GDB, pomocí kterého lze zasílat ladící příkazy programu i za jeho běhu a ovládat tak pouze zastavené vlákno, i když ostatní vlákna běží.

Pomocí GDB lze číst a zapisovat paměť laděného procesu na úrovni bytů v adresním prostoru procesu. V podporovaných jazycích, hlavně v C a C++, lze také provádět libovolné výrazy daného jazyka, volat funkce a systémová volání a pracovat s hodnotami na úrovni proměnných laděného programu.

Procesy, které se ukončí neočekávaně, např. po vyvolání výjimky, mohou před svým ukončením vygenerovat výpis své paměti (core dump), který lze poté načíst a analyzovat v GDB. Lze tak například zjistit, jak vypadal zásobník volání funkcí v momentu, kdy program zhavaroval, z čehož se dá odvodit, která část kódu programu způsobila jeho selhání.

Při ladění často nastává situace, kdy proces přeskočí instrukci nebo řádek zdrojového kódu, který chce programátor zkoumat. Poté je nutné program spustit od začátku, což může být časově náročné. GDB umí spouštět určité instrukce zpětně, a může tedy krokovat program nejenom směrem dopředu, ale i dozadu, což umožňuje vrátit se v procesu zpátky bez nutnosti jeho opětovného spuštění. Všechny změny a vedlejší efekty, které proběhly v paměti, jsou tak smazány a navraceny do původního stavu (pokud to daná platforma a stav programu dovoluje).

3.1.1 Spolupráce s Valgrindem

Valgrind je nástroj pro profilování a kontrolu paměťové korektnosti programů, který se využívá mimo jiné k hledání paměťových chyb jako je přístup k neinicializované paměti nebo neuvolnění naalokované paměti.⁸ Spouští zkoumaný program ve virtuálním stroji po překompilování kódu programu, do kterého přidává sledování alokací a přístupů do paměti. Kvůli těmto úpravám nelze program ve Valgrindu ladit klasickým přístupem. GDB poskytuje možnost připojit se k programu spuštěnému ve Valgrindu a vzdáleně ho ladit.

3.2 LLDB

Ladící nástroj LLDB⁹ je založen na sadě knihoven, které využívají infrastruktury LLVM a překladače Clang. LLVM je univerzální překladač, který dokáže překládat velké množství jazyků do své vnitřní, jazykově nezávislé reprezentace, kterou umí dále optimalizovat a vygenerovat z ní spustitelný soubor pro libovolnou kompatibilní platformu. Obsahuje také kompletní implementaci standardní knihovny jazyka C++ (*libc++*), která plně podporuje standard C++14. Clang¹⁰ je nádstavbou pro LLVM, která analyzuje a překládá programy v jazyce C a C++. Celá LLVM architektura je postavena na modulárních komponentách, které spolupracují a dají se lehce využít ve formě knihovny. Nabízí tak modernější alternativu k programům GCC a GDB. Ty jsou sice stabilnější a mají zatím více funkcí, ale jelikož existují už desítky let, musí udržo-

⁸<http://valgrind.org>

⁹<http://lldb.llvm.org>

¹⁰<http://clang.llvm.org>

vat zpětnou kompatibilitu a nebyly navrhovány modulárně, takže je těžší je využít jako modul v jiném programu.

LLDB umí ladit programy napsané v jazycích C, C++, Objective-C a Swift na platformách OS X, Linux, FreeBSD a Windows. Podporuje tedy méně jazyků i platforem, než GDB, ale na rozdíl od něj je podporován operačními systémy OS X a iOS, na kterých je standardním debuggerem namísto GDB. Nabízí většinu standardních funkcí debuggerů jako je krokování kódu, vytváření breakpointů nebo čtení a zápis paměti procesu. Jelikož je stále ve vývoji, tak zatím neobsahuje některé pokročilejší funkce, které nabízí GDB, například zpětné provádění instrukcí.

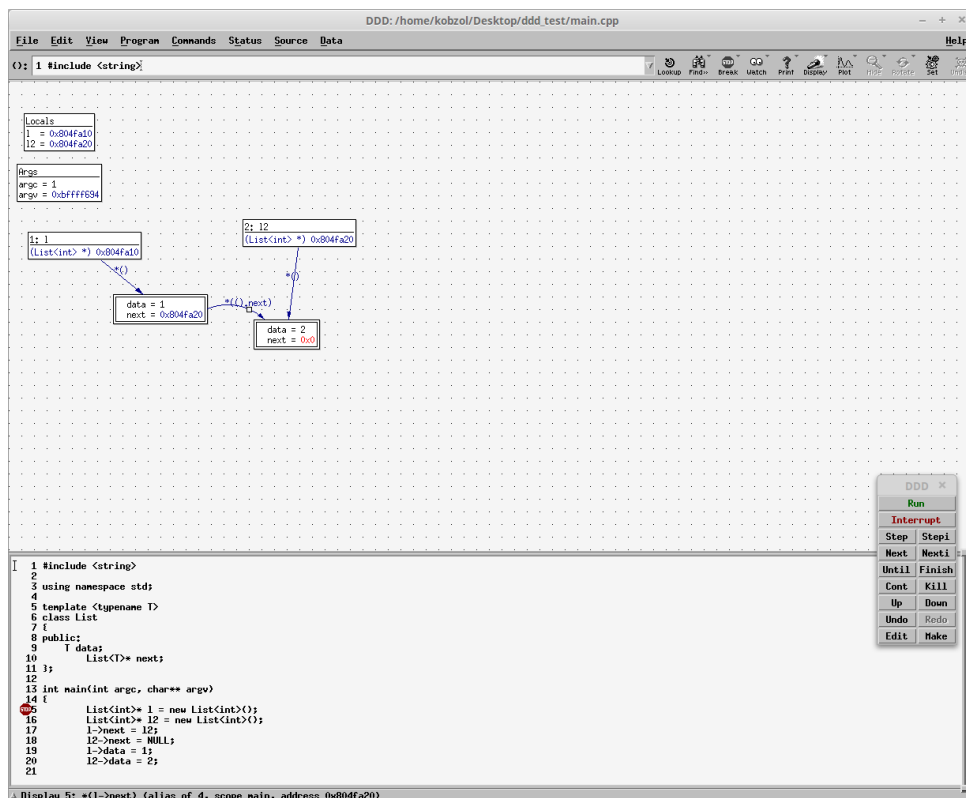
3.3 Grafická rozhraní

Grafických rozhraní pro debuggery GDB a LLDB existuje několik desítek. [2] Některé z nich jsou samostatné programy podporující pouze ladění, další jsou jednou z mnoha součástí integrovaných vývojových prostředí. Tyto nástroje se obvykle skládají z textového editoru, který obsahuje zdrojový kód, a ovládacích prvků, které kontrolují průběh laděného procesu. Dále také často nabízí manipulaci a zobrazení registrů, lokálních proměnných a parametrů funkcí. Rozhraní těchto nástrojů jsou obvykle velmi podobná, proto zde uvádím pouze zástupce z jednotlivých kategorií (samostatné, integrované a textové rozhraní).

3.3.1 DDD

DDD¹¹, neboli Data Display Debugger, je samostatné grafické rozhraní podporující velké množství debuggerů, mimo jiné GDB, pydb, DBX nebo Ladebug. Mimo klasického zobrazování zdrojového kódu programu nabízí i pokročilé vizualizační funkce. Umí kreslit grafy z hodnot paměti procesu anebo zobrazovat vztahy mezi objekty v paměti ve formě grafu. Jeho poslední verze vyšla v roce 2009, není už tedy v současnosti aktivně udržován. Ukázkou vizualizace paměti v DDD si lze prohlédnout na obrázku 2.

¹¹<https://www.gnu.org/software/ddd>

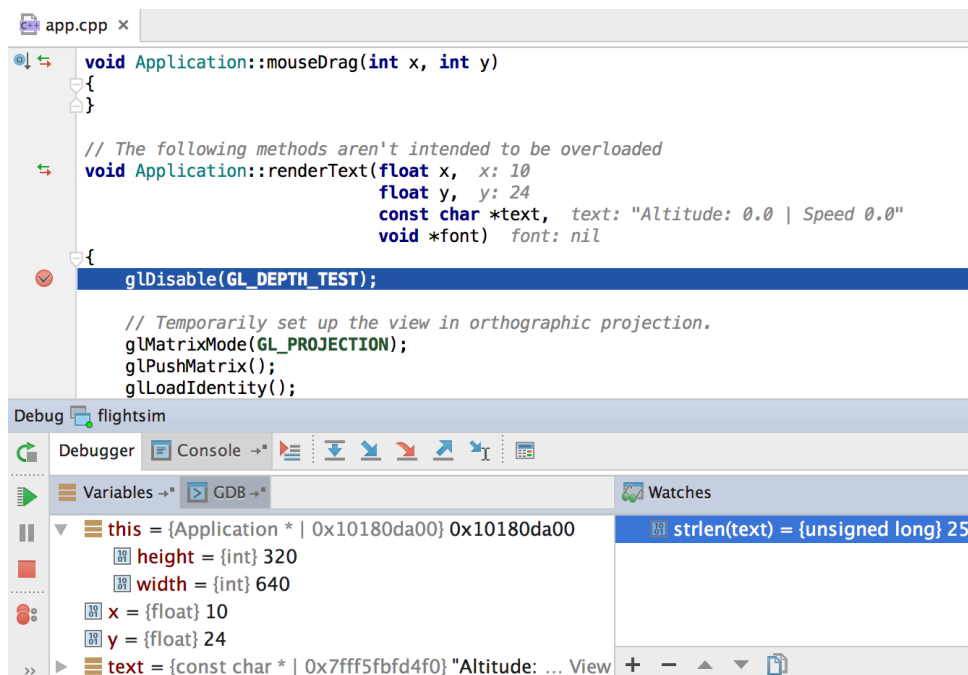


Obrázek 2: Uživatelské rozhraní programu DDD

3.3.2 Clion

Clion¹² je integrované vývojové prostředí založené na vývojové platformě IntelliJ. Nabízí mimo jiné statickou analýzu kódu psaného v jazycích C a C++, což pomáhá v odhalování velkého množství chyb již během psaní programu. Tato analýza zároveň usnadňuje ladění kódu poskytováním automatického doplňování výrazů a proměnných, které lze v laděném procesu sledovat. Během ladění Clion zobrazuje vedle názvů proměnných ve zdrojovém kódu jejich současnou hodnotu, což urychluje sledování stavu paměti procesu. Rozhraní Clionu, kde lze vidět hodnoty proměnných vypsané přímo ve zdrojovém kódu, je zobrazeno na obrázku 3.

¹²<https://www.jetbrains.com/clion>



Obrázek 3: Uživatelské rozhraní programu CLion

Zdroj: <https://www.jetbrains.com/clion>

3.3.3 GDB TUI

Text User Interface je textové grafické rozhraní vestavěné přímo v GDB¹³, které zobrazuje stav laděného procesu v oknech terminálu. Je postaveno na knihovně curses, která umožňuje vytvářet textové uživatelské rozhraní s pokročilými funkcemi přímo v terminálu. TUI lze spustit předáním parametru **-tui** při spuštění GDB anebo stisknutím kláves CTRL+X či zadáním příkazu *tui enable* za jeho běhu. Ukázka TUI je na obrázku 4. LLDB obsahuje podobné rozhraní, to ale zatím není oficiální součástí nástroje, jedná se pouze o nezávazně vyvíjený doplněk.

¹³<https://sourceware.org/gdb/onlinedocs/gdb/TUI.html>

```
server.cpp
78 {
79     DiningTable* dining_table;
80     int port;
81
82     handle_args(argc, argv, &dining_table, &port);
83
84     HelpPrinter::print(LOG_INFO, "Hostina byla zapocata na portu %d\n", port);
85
86     while (true)
87     {
88         char buf[256];
89         int len = read(STDIN_FILENO, buf, sizeof(buf));
90
91         if (len < 1) break;
92
93         buf[len - 1] = '\0'; // vymazani \n
94
95         if (!strcmp(buf, "end"))
96         {
97             dining_table->stop_feast();
98             HelpPrinter::print(LOG_INFO, "Hostina byla ukoncena\n");
99         }
100         else if (!strcmp(buf, "start"))
101         {
102             dining_table->start_feast(port);
103             HelpPrinter::print(LOG_INFO, "Hostina byla obnovena\n");
104         }
105         else if (!strcmp(buf, "quit"))
106         {
107             dining_table->stop_feast();
108             HelpPrinter::print(LOG_INFO, "Vypinam program\n");
109             break;
110         }
111     }
112
113     delete dining_table;
114
115     return 0;
116 }

Multi-thre Thread 0xb7c84780 ( In: main                                L84    PC: 0x804a5ff
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=3, argv=0xbffff164) at server.cpp:78
(gdb) next
(gdb) next
[New Thread 0xb7c82b40 (LWP 2965)]
(gdb) [
```

Obrázek 4: Uživatelské rozhraní programu GDB (TUI)

4 Architektura aplikace

Cílem této práce je navrhnout a naimplementovat program, který bude ladění programu vizualizovat ve formě diagramu objektů v paměti, a otestovat, jestli tato grafická reprezentace usnadňuje ladění programů a pochopení průběhu jednoduchých algoritmů. Tato kapitola pojednává o architektuře tohoto nástroje a popisuje technologie, které byly pro jeho tvorbu použity.

4.1 Specifikace požadavků

Program by měl fungovat jako grafické rozhraní pro debugger. Jeho hlavní funkcí by měla být vizualizace stavu laděného procesu zobrazováním jeho adresního prostoru. Zobrazovat by měl hlavně zásobník, který představuje současný stav a pozici procesu a z hlediska ladění se jedná o první místo, kde lze začít hledat chyby. Aby se dal program použít pro ladění reálně používaných aplikací, měl by poskytovat přístup ke všem běžným funkcím debuggerů. Měl by také být nezávislý na použitém debuggeru. Následuje seznam základních funkcí, které by měl program svým uživatelům nabízet.

- Asynchronní komunikace s debuggerem
- Načítání binárních souborů a zdrojového i strojového kódu
- Vytváření a správa breakpointů
- Manipulace s procesem (spouštění, pozastavení, zastavení)
- Krokování (krok po řádku, krok dovnitř funkce, krok ven z funkce)
- Komunikace s procesem v reálném čase (přes standardní vstup a výstup)
- Přepínání zásobníkových rámců a vláken
- Zobrazování registrů a paměti na úrovni bytů
- Manipulace s hodnotami proměnných
- Vizualizace paměti ve formě diagramu objektů

4.2 Použité technologie

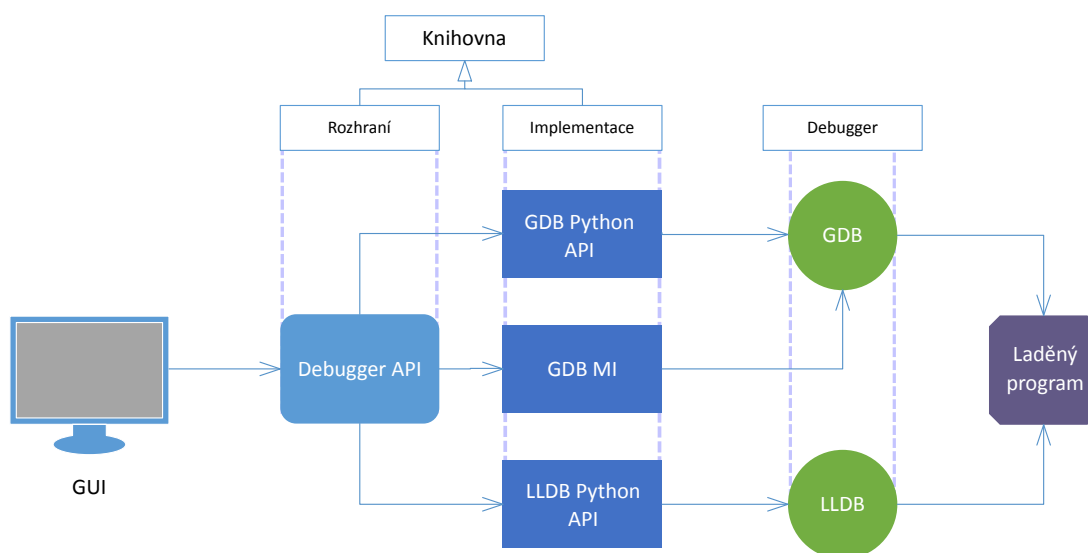
Pro implementaci programu jsem zvolil programovací jazyk **Python**¹⁴ ve verzi 2.7. Novější verze 3.x jsem nepoužil z důvodu zachování kompatibility s knihovnami, které jsou v programu použity a Python 3 plně nepodporují (mj. Python API pro LLDB a libclang). Python jsem vybral, protože je vhodný k rychlému vývoji aplikací, existují pro něj rozhraní pracující s debuggery GDB

¹⁴<http://www.python.org>

i LLDB a lze ho snadno vázat s kódem napsaným v jazycích C a C++. Navíc je multiplatformní, což by usnadnilo případný přenos aplikace na jiný operační systém.

Pro vývoj grafického rozhraní programu jsem vybral knihovnu **GTK 3**, která je volně dostupná pod licenci LGPL 2.1.¹⁵ GTK je multiplatformní grafický software umožňující tvorbu uživatelského rozhraní, který je používán mimo jiné i některými manažery plochy pro Linux (např. v GNOME¹⁶). Kromě možnosti tvorby vlastních GUI prvků obsahuje několik desítek běžně používaných prvků, které jsou připravené k okamžitému použití a usnadňují tak rychlý vývoj aplikací.

Vytvořil jsem dvě samostatné komponenty, knihovnu, která poskytuje rozhraní pro komunikaci s libovolným debuggerem a aplikaci s grafickým rozhraním, která slouží k vizualizaci paměti a k interakci s laděným procesem. Uživatelské rozhraní komunikuje s knihovnou, která zprostředkovává komunikaci s debuggerem. Pro účely této práce jsem otestoval tři implementace této komunikační vrstvy, které jsou popsány dále v textu. Debugger zajišťuje komunikaci a manipulaci s laděným procesem. Celá aplikace je tak rozdělena do několika vrstev, které jsou na sobě nezávislé. Architekturu aplikace si lze prohlédnout na obrázku 5.



Obrázek 5: Architektura *Devi*

¹⁵<http://www.gtk.org>

¹⁶<https://www.gnome.org/technologies>

5 Knihovna pro komunikaci s debuggery

Před tvorbou grafického rozhraní debuggeru bylo nejdříve potřeba vytvořit komunikační rozhraní, které by umožňovalo programově ovládat debugger. Pro tuto potřebu jsem vytvořil komunikační knihovnu, která definuje programovací rozhraní (API) pro práci s debuggerem. Díky tomuto rozhraní je uživatel knihovny odstíněn od konkrétní implementace debuggeru a způsobu komunikace s ním. Není tedy problém vytvořit implementace pro různé debuggery a používat je pomocí tohoto jednotného rozhraní. Tuto knihovnu jsem vyvinul nezávisle na vizualizační části aplikace, dá se tedy vyjmout a použít i v jiném projektu. Třídy v rozhraní knihovny slouží jako abstrakce pro práci s debuggerem. Pro implementaci komunikace s konkrétním debuggerem je nutné z těchto tříd podědit a naimplementovat všechny abstraktní funkce. Většina tříd v rozhraní neposkytuje téměř žádné společné chování pro své potomky, protože komunikace s různými debuggery může být implementována značně odlišným způsobem a společné chování pro všechny implementace by tak postrádalo smysl.

Vstupním bodem a také hlavní třídou rozhraní je **Debugger**, který představuje samotný ladící nástroj. Umožňuje načítat, spouštět a krokovat programy a také uchovává informaci o jejich stavu při ladění. Obsahuje několik komponent, které se starají o jednotlivé funkce potřebné pro ladění. **VariableManager** se stará o čtení registrů a paměti laděného procesu. Poskytuje funkce pro získávání hodnot a datových typů konkrétních proměnných a umožňuje je také měnit. **ThreadManager** umožňuje přepínání vláken a zásobníkových rámců procesu. O získávání současné pozice vlákna, práci se zdrojovými soubory programu a převod zdrojového kódu na strojový kód se stará komponenta **FileManager**. Breakpointy lze přidávat, mazat a vyhledávat pomocí třídy **BreakpointManager**. Čtení výstupu a zápis na vstup laděného procesu umožňuje **IOManager**. Správu informací o alokacích na haldě zajišťuje třída **HeapManager**.

Pro práci s debuggerem je nutné vytvořit i abstrakci samotného laděného procesu. Tak lze pro klienta knihovny zajistit zcela transparentní rozhraní nezávislé na použitém debuggeru a do určité míry i na jazyku laděného procesu. Vytvořil jsem několik tříd, které reprezentují běžící proces a jeho jednotlivé části. Tyto třídy reprezentují stav laděného procesu, který nezávisí na použitém debuggeru. Narozdíl od rozhraní pro debuggery tedy obsahují implementaci a netvoří tak pouze abstraktní rozhraní.

- **InferiorThread** představuje vlákno procesu. Aktivní je vždy pouze jedno vlákno, jehož údaje se používají při zobrazení současné pozice programu, seznamu zásobníkových rámců aj. Seznam všech vláken procesu spolu s informací o aktivním vlákně obsahuje třída **ThreadInfo**.
- **Variable** reprezentuje proměnnou laděného procesu. Obsahuje informace nezbytné pro identifikaci proměnné v programu, jako je adresa v paměti a jméno, a také její hodnotu v řetězcovém formátu. Pokud se jedná o strukturu, tak ta žádnou vlastní hodnotu nemá a je vždy potřeba přistoupit k jejímu konkrétnímu atributu. Dále je u proměnné uložena

cesta, pomocí které byla proměnná v programu nalezena, což umožňuje chovat se k proměnné jako k výrazu programovacího jazyka a používat ji v jiných výrazech. Kdyby totiž například proměnná `age` patřící do struktury `person` obsahovala pouze své jméno, nešlo by k ní přistoupit. Je tedy nutné si pro ni pamatovat, že přístup k ní lze provést pomocí výrazu `person.age`. Datový typ proměnné je reprezentován samostatnou třídou **Type**, která obsahuje název, kategorii a velikost daného typu.

- **Frame** je abstrakcí zásobníkového rámce, obsahuje informace o názvu funkce, z které je vyvolán a také uchovává své lokální proměnné a parametry.
- **Breakpoint** představuje breakpoint, který je identifikován zdrojovým souborem a řádkem, na kterém je umístěn.
- **Register** reprezentuje registr procesoru, obsahuje jeho název a hodnotu.
- **HeapBlock** uchovává informace o adrese a velikosti bloku paměti naalokovaného na haldě.

Pro účely této práce jsem pomocí rozhraní této knihovny vytvořil a otestoval několik implementací pro komunikaci s debuggery GDB a LLDB.

5.1 Python API pro GDB

GDB umožňuje použít Python pro ovládní a přístup k vnitřním funkcím GDB. [3] Většina nejpoužívanějších funkcí GDB je tak dostupná ve formě programovacího rozhraní (API). Funkce GDB, které nejsou v tomto API obsaženy, lze vyvolat pomocí přímého provádění textových příkazů. Nevýhoda tohoto přístupu je, že toto API neumí samo spustit instanci GDB a musí tak být načteno v již běžícím GDB procesu. Nelze jej tedy použít přímo jako modul v jiné aplikaci bez použití meziprocesní komunikace. Tento problém jsem vyřešil síťovou komunikací pomocí protokolu TCP (Transmission Control Protocol) se skriptem běžícím v procesu GDB. Po otestování tohoto způsobu komunikace s GDB jsem nicméně zjistil, že toto API není příliš stabilní a často havaruje. Kvůli tomu jsem se rozhodl tuto implementaci dále nerozšiřovat.

5.2 Python API pro LLDB

Stejně jako GDB i LLDB poskytuje API pro Python. Narozdíl od GDB rozhraní jej lze použít jako knihovnu, která sama vytváří instanci LLDB a je tedy jednoduché ji použít přímo v externím kódu. Pomocí tohoto API jsem vytvořil komunikační vrstvu, která implementuje většinu potřebných funkcí pro tvorbu vizualizačního nástroje. Abstrakce obsažené v tomto rozhraní se velmi podobají těm v rozhraní pro GDB a jejich sjednocení jsem použil jako inspiraci pro API mé komunikační knihovny. Nicméně stejně jako celé LLDB je toto rozhraní zatím ve vývoji a obsahuje drobné chyby, například nepřesné zobrazování stavu vláken a zásobníkových rámců. Tyto chyby jsem řešil s vývojáři LLDB¹⁷, a některé z nich již byly v nových verzích opraveny.

¹⁷<https://goo.gl/3g942j>

Nejnovější verze LLDB nicméně nejsou dostupné v běžných repozitářích balíčků a jejich překlad zahrnuje i překlad projektů LLVM a Clang, což mi přišlo jako zbytečně velká závislost. Kvůli těmto nedostatkům jsem do této implementace nepřidával podporu ladění vícevláknových aplikací ani některé další pokročilejší funkce.

5.3 Protokol GDB MI

MI (Machine Interface) je protokol pro ovládání GDB pomocí textových příkazů, které jsou na rozdíl od klasických příkazů používaných při manuálním ovládání debuggeru přizpůsobené pro snadné strojové zpracování. Obdobně jako Python API pro GDB poskytuje MI rozhraní pouze pro nejpoužívanější funkce. Zbytek funkcí lze používat pomocí běžných textových příkazů. Použití MI je běžným a doporučeným [2] způsobem pro programovou komunikaci s GDB a tvorbu grafických rozhraní. Existují volně dostupné knihovny pro Python i C/C++, které umí komunikovat pomocí MI protokolu. Nicméně abych pochopil fungování protokolu MI, měl co největší kontrolu nad komunikací s GDB a mohl používat i funkce, které tyto knihovny nenabízejí, tak jsem si vytvořil vlastní komunikační modul, který je popsán dále v textu.

5.3.1 Komunikace s debuggerem

Aby mohlo GDB komunikovat pomocí protokolu MI, tak se nejprve musí přepnout do MI módu, který lze zapnout parametrem **--interpreter=mi2** předaným při spuštění debuggeru. V tomto módu lze na jeho standardní vstup zasílat textové příkazy, které se okamžitě vyhodnotí a vrátí výsledek. Formát zpráv je dán specifikací¹⁸ a je přizpůsoben strojovému zpracování. Zprávy z debuggeru vždy obsahují hlavičku, která určuje, o jaký typ zprávy jde, a tělo, které obsahuje případné další informace o vyvolané události nebo provedeném příkazu. MI má sice definovaný základní formát zpráv, nicméně věci závislé na konkrétním laděném programu, jako jsou hodnoty proměnných, můžou být odesílány v různých formátech (popř. jsou místo nich dosazeny chybové hlášky), a toto chování není nikde zcela zdokumentováno. Může se tedy stát, že GDB vrátí data, která nepůjdou zpracovat a budou knihovnou ignorována.

Struktura MI zpráv je velmi podobná syntaxi datového formátu JSON, který je jedním ze standardních formátů pro výměnu dat¹⁹. Přijaté MI zprávy nejprve knihovna předzpracuje a poté je načte pomocí JSON parseru. Může se stát, že některé zprávy obsahující výše popsané nezdokumentované hodnoty nebudou správně zpracovány, nicméně stejný problém by existoval, i kdybych si napsal vlastní parser specifický pro formát MI zpráv, a proto jsem usoudil, že je toto řešení dostatečné. Komunikační knihovna poskytuje funkci pro odeslání příkazu, která jej odešle do GDB, počká, než jí přijde odpověď, která musí po každém příkazu následovat, a poté vrátí zpracovanou odpověď.

¹⁸https://sourceware.org/gdb/onlinedocs/gdb/GDB_002fMI-Command-Syntax.html

¹⁹<http://www.json.org>

Kromě této synchronní komunikace (zaslání příkazu - přijetí odpovědi) posílá GDB také zprávy o asynchronních událostech, jako je např. vytvoření vlákna nebo zastavení procesu. Grafické rozhraní debuggeru musí na tyto zprávy reagovat ihned poté, co jsou debuggerem zaslány. Kvůli toho se v komunikačním modulu vytváří samostatné vlákno, které v pravidelném časovém intervalu kontroluje výstup z debuggeru a pokud zjistí, že došlo k asynchronní události, tak vyvolá vlastní událost, ke které se může přihlásit libovolný klient. Při tomto periodickém čtení může dojít k souběhu, pokud by během čtení přišel od klienta knihovny požadavek na odeslání příkazu, který se také snaží číst výstup z GDB při čekání na svou odpověď. Z tohoto důvodu je čtení výstupu debuggeru zamknuto mutexem [12, s. 81], který potenciální souběh (chybu způsobenou nepředvídatelnou současnou změnou dat) vylučuje.

Některé funkce knihovny jsou naimplementované jinak než zasláním textového příkazu do GDB, například přerušení programu, které slouží k zastavení programu v libovolnou chvíli. Přerušeno je vyvoláno tak, že se procesu zašle signál SIGINT, který vyvolá zastavení procesu v GDB a to už je následně zpracováno knihovnou standardně, jako by se jednalo například o zastavení na breakpointu. Stejně tak násilné ukončení programu jej nejprve přeruší, a teprve poté pošle GDB příkaz k ukončení laděného procesu.

5.3.2 Komunikace s laděným procesem

GDB implicitně přesměrovává veškerý výstup z laděného procesu na svůj vlastní výstup. Toto chování není pro komunikační knihovnu vhodné, protože se výstup a vstup laděného procesu mísí s vstupem a výstupem debuggeru. Pro vyřešení tohoto problému lze v GDB pomocí příkazu *-inferior-tty-set* nastavit terminál, s kterým bude laděný proces komunikovat. Při použití tohoto příkazu se mi nepovedlo komunikovat s procesem v reálném čase kvůli vnitřnímu bufferování v terminálu. Rozhodl jsem se tedy použít jiné řešení, které při komunikaci s laděným procesem naprosto obchází GDB a je tak nezávislé na použitém debuggeru.

Před zapnutím laděného procesu knihovna v systémové složce pro dočasná data vytvoří čtyři speciální soubory, jeden pro vstup, dva pro výstup (klasický a chybový) a jeden pro alokace (popsáno dále v textu). Komunikace se soubory je znázorněna na obrázku 6. Tyto soubory jsou vytvořeny pomocí systémového volání `mkfifo`²⁰, které vytváří pojmenovanou rouru.

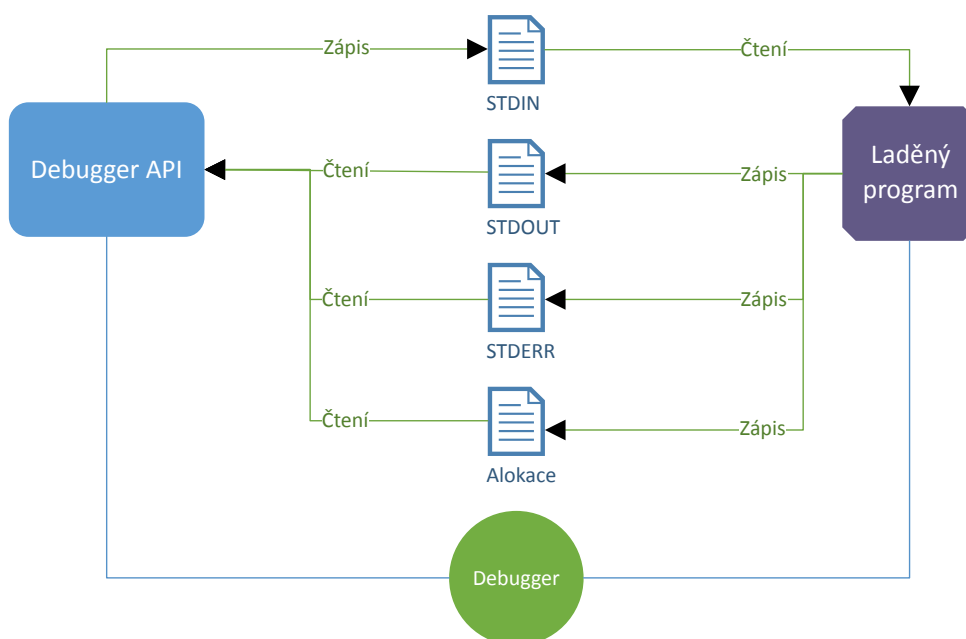
Pojmenované roury jsou prostředky meziprocesní komunikace (IPC), které mohou existovat i po ukončení procesu, který je vytvořil, a jsou globálně dostupné v celém systému, ne pouze v rámci procesu a jeho potomků, jako je tomu u klasických rour. Pojmenované roury jsou reprezentované (speciálním) souborem na disku, který lze otevřít pro čtení nebo zápis a komunikovat tak v reálném čase mezi procesy. Tato komunikace nicméně neprobíhá přes souborový systém, ale efektivně přímo v operační paměti.²¹ Na rozdíl od klasických souborů musí být tyto roury otevřené na obou svých koncích zároveň (tj. pro čtení i zápis), než je lze použít pro předání dat. Pokud se tedy nějaký proces pokusí otevřít rouru pro čtení, bude zablokován, dokud se nějaký

²⁰<http://linux.die.net/man/3/mkfifo>

²¹<http://linux.die.net/man/7/fifo>

jiný proces nepokusí otevřít danou rouru pro zápis a naopak. Při použití neblokujícího režimu lze rouru otevřít pro čtení, i pokud ji ještě nikdo neotevřel pro zápis. Pokus o otevření roury pro zápis v neblokujícím režimu ovšem selže, pokud předtím nebyla na svém druhém konci otevřena pro čtení.

Na straně knihovny se tyto soubory otevrou a poté pomocí nich lze zapisovat data na standardní vstup laděného procesu nebo číst data z jeho standardního (a chybového) výstupu. Při zapnutí laděného procesu v GDB je poté jeho standardní výstup i vstup přesměrován do těchto souborů. Po ukončení laděného procesu se tyto pojmenované roury smažou, aby se v systému nehromadily.



Obrázek 6: Komunikace s laděným procesem

5.3.3 Detekce dynamických alokací

Sledování dynamické alokace paměti na haldě laděného procesu je užitečná k zobrazení stavu adresního prostoru a také ke kontrole práce s pamětí. Systémová knihovna jazyka C obsahuje implementaci alokátoru, který se stará o alokaci a dealokaci paměťových bloků na haldě. Pro tuto práci s dynamickou pamětí se v jazyce C používají hlavně funkce **malloc**, **calloc**, **realloc** a **free**. V jazyce C++ se používají operátory **new** a **delete**, které ovšem také obvykle vnitřně volají již zmíněné C funkce, takže není nutné je explicitně zaznamenávat. Aby šlo sledovat alokace, je nutné zachytit volání všech funkcí pracujících s dynamickou pamětí a zaznamenat si jejich parametry a návratové hodnoty. Zachycení volání těchto funkcí lze zajistit různými způsoby. Já jsem si určil jako podmínku to, že půjde pracovat přímo se spustitelným souborem a nebude vyžadována úprava zdrojového kódu programu. Nutnost úpravy zdrojového kódu by totiž byla nepraktická u větších programů, navíc by ani nešla provést pro externí knihovny, které jsou

dostupné pouze jako binární distribuce. S ohledem na tyto požadavky jsem navrhl a otestoval následující způsoby sledování alokací.

Zastavení procesu při alokaci Pokud se na všechna volání alokačních funkcí umístí breakpoint, tak se proces zastaví při všech alokacích i dealokacích dynamické paměti. Poté si z něj lze přečíst údaje o alokaci, zaznamenat si je a obnovit běh procesu. Toto řešení se ukázalo jako funkční, ale zastavování procesu při každé alokaci laděný proces zpomalovalo. Dále vytváření skrytých breakpointů a jejich odlišení od breakpointů vytvořených uživatelem komplikovalo implementaci komunikační knihovny. Navíc by se tento způsob zachytávání musel implementovat pro každý debugger zvlášť. Kvůli těmto důvodům jsem se rozhodl toto řešení nepoužít.

Obalení alokačních funkcí vlastními funkcemi Alokační funkce v jazyce C jsou definovány v systémové knihovně, která poskytuje implementaci standardních funkcí a systémových volání. Často používanou implementací této knihovny na Linuxových systémech je glibc²². Při spuštění procesu na Linuxu se vyhledají dynamické knihovny, které daný proces vyžaduje, a poté se namapují do jeho adresního prostoru. Použitím proměnné prostředí **LD_PRELOAD** lze určit seznam sdílených knihoven, které budou přednostně nahrány ke spouštěnému procesu²³. Jelikož se tyto knihovny nahrávají jako první, symboly obsažené v nich mají přednost před později nahrávanými symboly. Pokud se tedy pro proces nahrají dvě sdílené knihovny, obě obsahující vlastní implementaci funkce se stejnou deklarací, tak při zavolání této funkce se použije implementace z knihovny, která byla nahrána dříve.

Těto skutečnosti jsem využil pro vytvoření sdílené knihovny, která obsahuje deklarace dříve zmíněných alokačních funkcí. Pro alokaci a namapování paměti lze využít i další funkce, např. **mmap** nebo **memalign**, ty se ale běžně v programech nevyskytují, a tak nejsou v této knihovně sledovány. Při spuštění procesu je nahrána před systémovou knihovnou jazyka C a všechny dynamické alokace z laděného procesu poté volají funkce z mé knihovny. V ní dochází k otevření pojmenované roury, která slouží k posílání dat o alokacích mé komunikační knihovně. Při alokaci či dealokaci se nejprve odešlou informace o jejím typu a případné další parametry, jako je velikost alokovaných dat nebo adresa dealokované paměti. Poté je zavolána původní funkce ze systémové knihovny, která provede samotnou (de)alokaci.

Adresa původní funkce musí být zjištěna za běhu programu pomocí funkce **dlsym**.²⁴ Prosté použití jejího názvu by totiž vyústilo v nekonečné rekurzivní volání, jelikož by se opět zavolala funkce z přednačtené knihovny. Funkce **dlsym** nicméně sama může alokovat malé množství dynamické paměti, její naivní použití by tedy vyústilo v nekonečnou rekurzi a přetečení zásobníku. Aby k tomuto nedošlo, tak se ve sdílené knihovně vytváří statické pole s 1024 vynulovanými byty, které je použito pro přidělení paměti vyžádané před nebo během načítání adresy alokačních funkcí. Pokud později dojde k dealokaci této paměti pomocí funkce **free**, tak je tato

²²<https://www.gnu.org/software/libc/>

²³<http://linux.die.net/man/8/ld.so>

²⁴<http://linux.die.net/man/3/dlsym>

dealokace ignorována a není předána dále do systémové knihovny, protože ta nemá o této statické paměti žádné informace. V komunikační knihovně běží vlákno, které se stará o čtení informací o alokacích. Uchovává si jejich seznam a umožňuje jiným komponentám knihovny reagovat na alokace pomocí událostí.

Nevýhodou tohoto řešení je, že nebude fungovat, pokud bude laděný program používat staticky přilinkovanou systémovou knihovnu²⁵. Tato situace ale není obvyklá a v případě potřeby může tvůrce laděného programu pro účely ladění systémovou knihovnu linkovat dynamicky. Otevření souboru pro komunikaci a získávání adresy původních funkcí dále musí být synchronizováno, aby nedošlo k souběhu u vícevláknových aplikací. Toto řešení však nevyžaduje zastavení procesu při každé alokaci a poskytuje zachytávání alokačních funkcí nezávislé na použitém debuggeru, proto jsem se jej rozhodl použít.

²⁵Linker *ld* připojuje knihovny k spustitelnému souboru staticky při použití přepínače **-static**

6 Vizualizační aplikace

Pro samotné ladění aplikací a využití komunikační knihovny jsem vytvořil aplikaci *Devi*. Ta poskytuje grafické rozhraní pro veškeré naimplementované funkce knihovny a dále k nim přidává vizualizaci paměti procesu. Z popsaných implementací komunikace s debuggerem jsem zvolil komunikaci s GDB pomocí protokolu MI, protože tento způsob komunikace byl nejstabilnější. Nicméně uživatelské rozhraní používá výhradně mnou navržené API popsané v kapitole 5, takže implementaci komunikační vrstvy lze v případě potřeby jednoduše vyměnit. Jelikož se funkcionalita a návratové hodnoty příkazů GDB v různých verzích značně liší, tak komunikační vrstva předpokládá spolupráci s konkrétní verzí GDB. Při kompilaci projektu se tato verze GDB stáhne a nainstaluje do složky projektu²⁶. Přibalení konkrétní verze debuggeru používají i jiná uživatelská rozhraní nebo vývojové prostředí, například CLion.

Devi při svém spuštění vytvoří instanci komunikačního rozhraní pro ovládání debuggeru. Komunikační mezivrstva je asynchronní, nelze například dopředu předvídat, v který okamžik se proces zastaví, kdy se vytvoří nové vlákno atd. Pro obsluhu těchto událostí jsem použil návrhový vzor Observer [7, s. 326], který umožňuje jednoduše reagovat na libovolné události v aplikaci. Při vyvolání události v debuggeru (změna stavu procesu, alokace, změna zásobníkového rámce) se aktualizuje uživatelské rozhraní. To je díky tomu udržováno aktuální a konzistentně zobrazuje stav debuggeru.

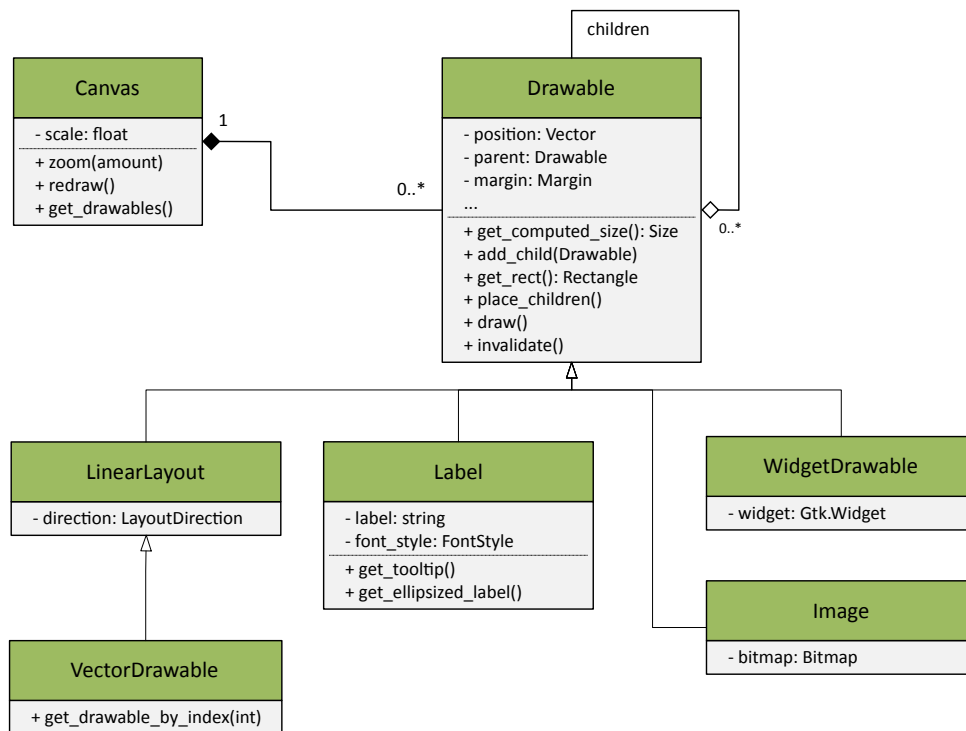
Jelikož události komunikační vrstvy jsou obvykle vyvolané jiným než hlavním vláknem, tak při jejich obsluze nelze přímo modifikovat uživatelské rozhraní. V GTK, stejně jako u většiny ostatních GUI frameworků, lze totiž pracovat s prvky grafického rozhraní pouze z hlavního, tzv. GUI vlákna. Pokud je tedy pro obsluhu události nutná změna uživatelského rozhraní, tak se do fronty hlavního vlákna vloží funkce, která bude hlavním vláknem zavolána, až zpracuje dříve vzniklé události. Protože je velmi lehké zapomenout na zákaz úpravy GUI z jiných vláken (což může nedeterministicky způsobit pád aplikace nebo jiné problémy), tak jsem vytvořil dekorátor, který tato volání blokuje. Pokud dekoruje funkci, tak při jejím zavolání nejprve pomocí inspekce zásobníku zkontroluje, jestli je současné vlákno hlavní, a pokud není, tak vyvolá výjimku. Při použití tohoto dekorátoru u všech funkcí manipulujících s uživatelským rozhraním lze spolehlivě odchytnout veškeré modifikace GUI z jiných vláken.

6.1 Vykreslovací komponenta

Pro vykreslování paměti procesu jsem vytvořil komponentu, která se stará o vykreslování a uspořádání jednoduchých dvojrozměrných objektů (čtyřúhelníky, Bézierovy křivky, obrázky, text atd.). Pro nízkoúrovňové kreslicí operace používá Cairo²⁷, univerzální open-source 2D grafickou knihovnu, která je kompatibilní s nástrojem GTK. Třídní diagram této komponenty si lze prohlédnout na obrázku 7.

²⁶Podrobnější informace jsou uvedeny v příloze A

²⁷<http://cairographics.org>



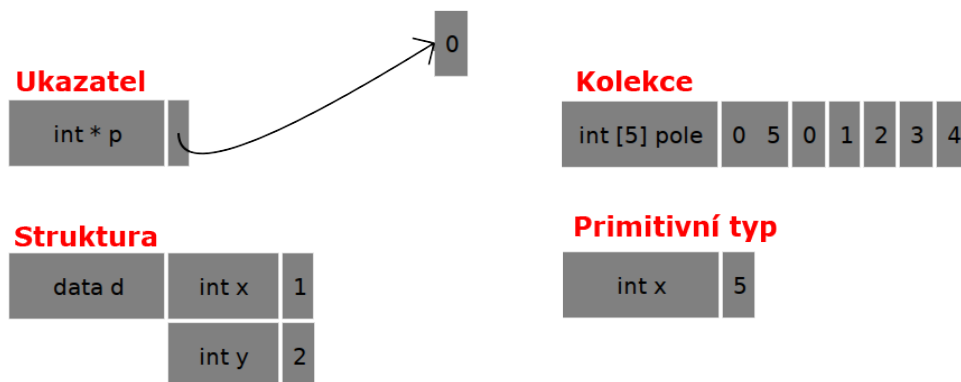
Obrázek 7: Třídní diagram části vykreslovací komponenty

Třída **Canvas** slouží jako plátno; udržuje hierarchii vykreslovaných objektů, deleguje na ně události myši a klávesnice a aplikuje globální transformace (translace a změnu měřítka). Sama o sobě je grafickou komponentou GTK, takže ji lze transparentně použít jako součást GUI. Obsahuje několik kreslicích vrstev, do kterých lze uložit kreslicí příkazy, které se provedou v určeném pořadí, jakmile jsou ostatní kreslicí operace dokončeny. To umožňuje vykreslovat objekty do různých vrstev nezávisle na tom, kdy byl příkaz k jejich vykreslení vyvolán.

Vykreslovatelné objekty reprezentuje třída **Drawable**, která je uspořádána do stromové struktury dle návrhového vzoru *Composite*. [7, s. 183] Každá její instance může tvořit buď list stromu, který se umí sám vykreslit, anebo obsahuje potomky, na které deleguje uživatelský vstup a určuje jejich umístění na plátně. Ke změně umístování jednotlivých prvků je k dispozici určení pozice, vnější a vnitřní mezery a určení minimálních, maximálních a požadovaných rozměrů. Podle těchto omezení se vypočte rozměr a ohraničující obdélník objektu a objekty poté mohou být svými rodiči vyskládány na plátno. Tato sada vlastností umožňuje vykreslit jednoduché struktury objektů, které stačí pro základní zobrazení stavu paměti běžícího procesu. Dále je ještě k dispozici několik vlastností pro změnu stylu (například barva pozadí nebo rámeček). Abych do této komponenty nemusel reimplementovat komplexní uživatelské prvky, jako je například pole pro zadávání textu, tak v ní lze vytvořit prvek, který obaluje libovolný GTK grafický prvek, který je poté absolutně pozicován na plátno.

Pomocí této komponenty jsem vytvořil několik tříd specializovaných na vykreslování datových typů jazyka C a C++. Aplikace při zastavení laděného procesu načte z debuggeru zásob-

níkové rámce spolu s jejich proměnnými a dále také všechny objekty, na které ukazují ukazatele ze zásobníku. Všechny proměnné jsou rekurzivním průchodem namapovány na odpovídající vykreslovací komponentu, která je předána plátnu k vykreslení. Následuje seznam kreslících prvků specializovaných na jednotlivé datové typy programů. Jejich přehled je na obrázku 8.



Obrázek 8: Přehled vizualizačních prvků

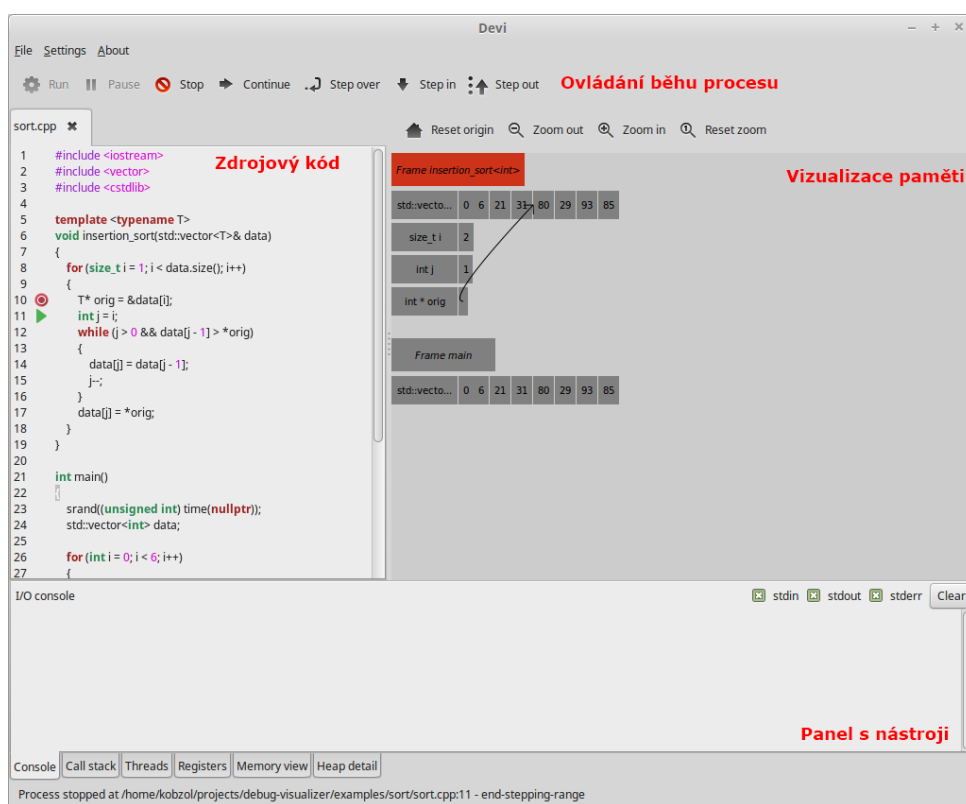
1. Jednoduché datové typy, jako jsou čísla, znaky, řetězce, výčetové typy (enumerace) a typy, které nelze plně vizualizovat (funkce), jsou zobrazeny klasickým textovým popisem. Při kliknutí na ně se zobrazí textový vstup, pomocí kterého lze změnit jejich hodnotu.
2. Ukazatele a reference jsou vizualizovány šipkou (její dráha je určena kubickou Bézierovou křivkou), která ukazuje na objekt na adrese dané ukazatelem či referencí. Kliknutím na ukazatel a přetáhnutím kurzoru na jinou proměnnou lze změnit hodnotu ukazatele na její adresu. *Devi* umí zkontrolovat, jestli se typ cílové proměnné shoduje s typem ukazatele, a tedy jestli je tato akce validní. Nicméně změna hodnoty ukazatele není v programu nijak omezena, protože pro potřeby ladění je někdy užitečné s ukazateli provést i operace, které vyžadují přetypování anebo které nejsou zcela validní.
3. Složené datové typy (struktury, třídy a svazy²⁸) jsou reprezentovány hierarchicky. U každého atributu struktury je zobrazen jeho název a hodnota, jejíž podoba závisí na jeho datovém typu. Jelikož mapování proměnných na grafické objekty probíhá rekurzivně, lze takto zobrazit libovolně zanořené hierarchické struktury.
4. Pole a dynamické vektory (typ `std::vector` ze standardní šablonové knihovny C++) jsou reprezentovány výčtem svých prvků. Jelikož takovéto kolekce mohou obsahovat obrovský počet prvků, tak je vždy načtena a zobrazena pouze jejich část. Pomocí dvou číselných hodnot lze u každé kolekce zvolit, kolik prvků se z ní má zobrazit a na které pozici v kolekci má zobrazený výčet začínat. Aby se krátké kolekce nemusely manuálně rozkliknout, tak se implicitně pro každou kolekci zobrazuje prvních deset prvků.

²⁸Datový typ `union`

Devi obsahuje podporu vizualizace kontejnerů typu `std::vector`. Ten je často používanou reprezentací pole s dynamickou velikostí a proto se v *Devi* zobrazuje stejně jako klasické pole. Ostatní kontejnery a třídy ze standardní knihovny nemají speciální podporu a jsou tak zobrazeny jako běžné třídy.

6.2 Používání aplikace

Devi po svém spuštění zobrazí hlavní okno aplikace, které si lze prohlédnout na obrázku 9. Většina oken a panelů v *Devi* popisuje nebo modifikuje stav laděného procesu a z toho důvodu s nimi lze pracovat pouze když je proces zastavený. Dnešní počítače zvládnou provádět miliardy operací za vteřinu a tak častá aktualizace informací o stavu procesu by postrádala smysl. Následuje popis funkcí jednotlivých částí rozhraní.



Obrázek 9: Uživatelské rozhraní *Devi*

Hlavní menu V menu lze načíst spustitelný soubor aplikace, což je nutný první krok, pokud tuto aplikaci chceme v *Devi* ladit. Spustitelný soubor musí být přeložený s ladícími symboly, jinak ladění nebude spolehlivě fungovat. Dále zde lze upravit spouštění laděného procesu. Nastavit jdou předané parametry prostředí a příkazové řádky a pracovní adresář procesu.

Panel ovládání procesu Tento panel obsahuje nejpoužívanější příkazy sloužící k ovládání běhu procesu, jako je jeho zapnutí, přerušení, násilné ukončení a několik typů krokování. Jednotlivé příkazy jsou automaticky blokovány a povolovány podle stavu laděného procesu, nelze tedy např. zapnout proces, když už běží, anebo ho krokovat, když není zastavený.

Zdrojový kód *Devi* obsahuje okno, které zobrazuje obsah zdrojového kódu souboru v textovém editoru se zvýrazněním syntaxe. Zobrazeno může být několik souborů naráz, lze mezi nimi přepínat pomocí záložek. Při načtení spustitelného souboru se automaticky načte a zobrazí soubor, který obsahuje vstupní bod daného programu (funkci *main*). Pokud se laděný proces zastaví na breakpointu v nějakém zdrojovém souboru, tak je tento soubor automaticky otevřen a zobrazen v editoru. Ten se posune tak, aby byl daný breakpoint vidět a uživatel ho nemusel hledat. Pokud chce uživatel manuálně přidat breakpoint v jiném, než hlavním zdrojovém souboru, může využít funkce načtení libovolného zdrojového souboru z hlavního menu.

Samotný textový editor nepovoluje změny zdrojového kódu, jelikož k tomu debugger není určen a neobsahuje ani žádnou vestavěnou podporu pro překlad kódu do spustitelné podoby. Kód je tak zobrazen pouze pro čtení, aby náhodou nedošlo k jeho nechtěné změně. Ve sloupci vedle zdrojového kódu lze přidávat a odebírat breakpointy. Zároveň se v něm při běhu procesu ukazuje, na kterém řádku v daném souboru je proces zastavený. Pokud je laděný proces zastavený, tak při přesunutí kurzoru myši nad proměnnou ve zdrojovém kódu se zobrazí její současná hodnota v procesu. Tato funkcionality je zajištěna pomocí knihovny Clang.

Konzole Pomocí konzole lze komunikovat s laděným procesem. Zobrazuje se v ní obsah jeho standardního (*stdout*) i chybového (*stderr*) výstupu. Lze do ní také psát text, který se po odřádkování odešle na standardní vstup (*stdin*) laděného procesu. Výstup, chybový výstup i vstup lze filtrovat pomocí zaškrťovacích tlačítek.

Zásobník funkcí V tomto okně je zobrazen stav zásobníku volaných funkcí pro aktivní vlákno. Lze se mezi nimi přepínat, což způsobí aktualizaci ukazatele současného řádku v editoru zdrojového kódu. Mezi zásobníkovými rámci lze přepínat i pomocí vizualizačního plátna, které je popsáno dále.

Seznam vláken Zde je zobrazen seznam všech vláken laděného procesu spolu s jejich základními informacemi, jako je název nebo id. Pokud je proces zastavený, lze se mezi přepínat. Jelikož změna vlákna způsobí i změnu zásobníku funkcí, tak při ní dochází ke kompletnímu překreslení vizualizačního plátna.

Seznam registrů Pokud je proces zastavený, tak je zde zobrazen seznam názvů a hodnot všech registrů procesoru. To může sloužit k nízkoúrovňovému ladění procesu, např. při práci s jazykem symbolických adres.

Zobrazení paměti V tomto okně lze zadat hexadecimální adresu paměti, ze které se načte 160 bytů, které jsou poté zobrazeny ve formě dvou tabulek. V první tabulce je přímá hodnota bytů ve formě čísla s rozsahem 0 až 255, ve druhé tabulce je pak jejich ASCII reprezentace, vhodná pro zkoumání textových polí v paměti. Kromě adresy v paměti lze zadat také název proměnné ze současně aktivního zásobníkového rámce. Adresa této proměnné se poté odvodí a zobrazí se hodnota její paměti.

Detail haldy Toto okno zobrazuje graf velikosti paměti alokované na haldě v závislosti na čase. Pokud laděný proces běží, tak se hodnota grafu aktualizuje co vteřinu a zobrazuje tak stav haldy v reálném čase. Dále se zde zobrazuje současný počet naalokovaných bloků na haldě, její velikost v bytech a celkový počet alokací a dealokací zaznamenaných od spuštění procesu.

Vizualizační plátno Na tomto plátně se při zastavení procesu vykreslí všechny zásobníkové rámce aktivního vlákna. V každém rámci je vykreslen seznam lokálních proměnných a parametrů dané funkce. U každé proměnné je zobrazen její název a současná hodnota, při podržení myši nad ní se dále zobrazí dodatečné informace, jako je její adresa v paměti. Aktivní zásobníkový rámec lze změnit kliknutím na hlavičku některého z vykreslených rámců, současný aktivní rámec je zvýrazněn červenou barvou. Po zastavení procesu se spustí načítání zásobníkových rámců a jejich proměnných. Toto načítání může trvat pro velký počet dat až několik vteřin, proto je prováděno na vedlejším vlákně, aby aplikace zůstala responzivní.

7 Testování dosažených výsledků

Funkčnost a použitelnost *Devi* jsem otestoval laděním C++ programů, přičemž jsem vždy jeho použití srovnával s laděním stejného programu v nástroji CLion (jeho popis lze najít v sekci 3.3.2). Clion používá GDB obdobným způsobem jako *Devi* a nabízí tak téměř identickou sadu funkcí pro ladění. Uživatelské rozhraní CLionu je samozřejmě mnohem pokročilejší než rozhraní mé aplikace, nicméně při testování jsem nenarazil na situaci, ve které bych potřeboval využít funkci, kterou můj program neobsahoval. Ovládání a ladění tak z mého pohledu bylo v obou aplikacích srovnatelné, s několika rozdíly, které jsou popsány níže.

Načítání a vizualizace proměnných po zastavení procesu může v *Devi* trvat až několik vteřin, přičemž tento čas je z velké části stráven vykonáváním požadavků uvnitř GDB, a nelze jej tedy v mém programu ovlivnit. V mém programu se dále vytváří grafická reprezentace paměti (oproti textového popisu v CLionu) a načítání stavu paměti tak může trvat déle.

Abych zajistil co největší kompatibilitu *Devi* s debuggerem, tak jsem jej odladil nad konkrétní verzi GDB. Nicméně údaje o laděném procesu kromě debuggeru závisí také na kódu laděného programu, verzi standardních a sdílených knihoven, DWARF formátu a dalších proměnných faktorech. Může se tedy stát, že pro stejný program přeložený jiným překladačem anebo jinou verzí stejného překladače bude GDB vracet odlišné ladící informace. V tomto případě zobrazené informace o stavu procesu nemusí být přesné ani kompletní. Nutno říci, že s tímto problémem se potýkají i ostatní rozhraní pro debuggery, která nejsou vždy schopná poskytnout přesné informace o laděném procesu. Vychází to z toho, že programy napsané v jazyku C po překladu do binární podoby neobsahují dostatek informací potřebných pro ladění. U jazyku C++ je situace ještě obtížnější, protože obsahuje šablony (*templates*), které do ladících informací přinášejí další komplexitu. Názvy datových typů se šablonovými parametry se můžou pro různé verze překladačů lišit, což způsobuje v *Devi* problémy při identifikaci typů, pro které existuje speciální vizualizátor (jako je například `std::vector`).

Vizualizace paměti v *Devi* při ladění umožňuje rychle prozkoumat stav programu a jednoduše ho upravit. Dovoluje také při ladění intuitivně sledovat průběh algoritmů a pochopit jejich fungování, případně odhalit v nich chyby. Nicméně při ladění větších programů se ukázalo, že vizuální reprezentace paměti je problémová u programů obsahujících velký počet dat. Při velkém množství zásobníkových rámců a proměnných je grafická reprezentace paměti nepřehledná a pro programátora může být obtížné se v ní jednoduše orientovat. Z těchto experimentů jsem došel k závěru, že ideální je vizualizaci zkombinovat s klasickým textovým popisem paměti ve formě stromu proměnných, který je lépe použitelný při velkém objemu dat laděného programu.

Pro otestování *Devi* jsem připravil několik jednoduchých programů v jazyce C++. Tyto programy obsahují jazykové konstrukce, na kterých lze snadno demonstrovat funkcionalitu *Devi* při ladění. Příklady jsou umístěné ve složce `examples`.

- **Sort** - tento program obsahuje implementaci jednoduchého třídění vkládáním (insertion sort) nad celými čísly uloženými v kolekci typu `std::vector`. Při krokování tohoto pro-

gramu v *Devi* lze sledovat dynamické změny obsahu vektoru i vizualizaci průběhu třídícího algoritmu.

- **Pointers** - tento program obsahuje celočíselné proměnné, ukazatel a C++ referenci. Ukazatel a reference (dále odkazy) se na dané proměnné odkazují a jejich hodnota se v cyklu vypisuje. Při ladění lze měnit hodnoty odkazů a sledovat změny ve výpisu aplikace.
- **Threads** - na tomto programu je demonstrována schopnost *Devi* ladit vícevláknové programy. Je v něm implementovaný klasický problém meziprocesní komunikace známý jako Problém producenta a spotřebitele. [12, s. 76] Vlákno, které se chová jako producent, generuje data pro druhé vlákno, které data spotřebovává. Zde je pro jednoduchost fronta sloužící pro komunikaci mezi vlákny reprezentována jediným znakem. Producent do tohoto znaku postupně ukládá znaky řetězce, který chce spotřebiteli předat, a ten zase znaky postupně odebírá a vypisuje je na standardní výstup. Obě vlákna se střídají, vždy se tedy jeden znak uloží do fronty, poté jej spotřebitel z fronty zkopíruje a vypíše. Synchronizace mezi vlákny je zajištěna pomocí mutexu a podmíněné proměnné²⁹, které dohromady tvoří monitor. [12, s. 81] Tyto synchronizační konstrukce jsou implementované pomocí knihovny *pthread*³⁰. Při ladění tohoto programu se lze přepínat mezi jednotlivými vlákny a jejich zásobníkovými rámci a detailně tak sledovat průběh komunikace.

²⁹http://linux.die.net/man/3/pthread_cond_wait

³⁰<http://man7.org/linux/man-pages/man7/pthreads.7.html>

8 Závěr

V rámci této práce jsem popsal principy ladění programů a srovnal nejpoužívanější debuggery pro Linuxové systémy a jejich grafická rozhraní. Dále jsem navrhl rozhraní komunikační knihovny pro ovládání debuggerů a otestoval několik implementací tohoto rozhraní. Pomocí této knihovny jsem vytvořil aplikaci, která poskytuje grafické rozhraní pro ovládání debuggeru a vizualizuje stav paměti laděného procesu. Po jejím otestování jsem došel k závěru, že může programátorům značně usnadnit pochopení fungování jejich programu a poskytnout o laděném programu více informací, než nabízí klasické metody zobrazení paměti.

Devi by mohl být v budoucnu rozšířen o další užitečné funkce. Po přidání alespoň základní správy souborů a možnosti překladu zdrojového kódu by mohl sloužit jako jednoduché vývojové prostředí pro výuku programování v jazycích C a C++. Díky tomu, že používá komunikační knihovnu s abstraktním rozhraním, je jednoduché přidat k němu podporu pro další debuggery. Určitě by bylo zajímavé rozšířit tuto knihovnu o komunikaci s jinými debuggery, než je GDB, a srovnat jejich funkcionalitu a stabilitu. Komunikační knihovnu by také šlo rozšířit o více funkcí, například o ladění po síti nebo připojování se k běžícím procesům. Zajímavé by bylo také přidat podporu pro Valgrind, což by umožnilo při ladění detekovat chyby v přístupu do paměti. Vizualizaci objektů by šlo rovněž dále prohloubit, například vytvářením odlišných pohledů nad stejnými daty, přidáním diagramů zobrazujících rozsáhlá data, podporou více paměťových kontejnerů standardní knihovny jazyka C++ nebo zobrazováním globálních proměnných. Jako doplněk k grafické reprezentaci by mohlo sloužit textové zobrazení proměnných ve stromové struktuře. Pro velké programy může načítání vizualizace paměti trvat dlouhou dobu, pro omezení tohoto problému by se dalo použít načítání dat na požádání (tzv. lazy loading) anebo opětovné vykreslování již vizualizovaných objektů.

Jakub Beránek

Literatura

1. BENDERSKY, Eli. *How debuggers work: Part 2 - Breakpoints* [online]. 2011 [cit. 2016-04-12]. Dostupný z WWW: <http://eli.thegreenplace.net/2011/01/27/how-debuggers-work-part-2-breakpoints>.
2. FREE SOFTWARE FOUNDATION, INC. *GDB front ends* [online]. [cit. 2016-04-12]. Dostupný z WWW: <https://sourceware.org/gdb/wiki/GDB%20Front%20Ends>.
3. FREE SOFTWARE FOUNDATION, INC. *Python API* [online]. [cit. 2016-04-12]. Dostupný z WWW: <https://sourceware.org/gdb/onlinedocs/gdb/Python-API.html>.
4. FREE SOFTWARE FOUNDATION, INC. *Setting Watchpoints* [online]. [cit. 2016-04-12]. Dostupný z WWW: <https://sourceware.org/gdb/onlinedocs/gdb/Set-Watchpoints.html>.
5. FREE SOFTWARE FOUNDATION, INC. *Stopping and Starting Multi-thread Programs* [online]. [cit. 2016-04-12]. Dostupný z WWW: <https://sourceware.org/gdb/onlinedocs/gdb/Thread-Stops.html>.
6. FREE SOFTWARE FOUNDATION, INC. *Supported Languages* [online]. [cit. 2016-04-12]. Dostupný z WWW: <https://sourceware.org/gdb/current/onlinedocs/gdb/Supported-Languages.html>.
7. GAMMA, Erich, HELM, Richard, JOHNSON, Ralph a VLISSIDES, John. *Design patterns: Elements of reusable object-oriented software*. Reading, Mass: Addison-Wesley, 1995. 395 s. ISBN 0-201-63361-2.
8. GUO, Philip J. Online Python Tutor: Embeddable Web-based Program Visualization for CS Education. In: *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*. Denver, Colorado, USA: ACM, 2013. S. 579–584. SIGCSE '13. ISBN 978-1-4503-1868-6. Dostupný také z WWW: <http://doi.acm.org/10.1145/2445196.2445368>.
9. INTEL CORPORATION. *Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference* [online]. 1999 [cit. 2016-01-24]. Dostupný z WWW: <http://download.intel.com/design/intarch/manuals/24319101.pdf>.
10. MATZ, Michael, HUBIČKA, Jan, JAEGER, Andreas a MITCHELL, Mark. *System V Application Binary Interface: AMD64 Architecture Processor Supplement* [online]. 2013 [cit. 2016-02-22]. Dostupný z WWW: <http://www.x86-64.org/documentation/abi.pdf>.
11. SORVA, Juha. *Visual program simulation in introductory programming education*. Espoo: Aalto Univ. School of Science, 2012. ISBN 9789526046266. Dostupný také z WWW: <http://doi.acm.org/10.1145/2445196.2445368>.
12. TANENBAUM, Andrew S. a WOODHULL, Albert S. *Operating Systems: Design and Implementation*. 3rd ed. Upper Saddle River, N.J.: Prentice Hall, 2006. 1080 s. ISBN 0-13-142938-8.

A Struktura projektu a instalační manuál

Soubory na přiloženém CD jsou uspořádány do následující adresářové struktury.

```
/.....instalační a spouštěcí skript a soubory s licenseми
├── debugger ..... zdrojové kódy komunikační knihovny
│   ├── analysis ..... analýza kódu
│   ├── gdbc ..... komunikace s GDB pomocí GDB Python API
│   ├── lldb ..... komunikace s LLDB pomocí LLDB Python API
│   ├── mi ..... komunikace s GDB pomocí rozhraní MI
│   ├── net ..... síťová komunikace mezi debuggerem a klientem
│   └── pycgdb ..... experimentální implementace debuggeru v Pythonu
├── examples ..... ukázky jednoduchých programů pro vyzkoušení Devi
├── gui ..... zdrojové kódy uživatelského rozhraní
│   └── drawing ..... vykreslovací framework a vizualizace paměti
├── res ..... vizuální zdroje aplikace
│   ├── css ..... CSS styly pro úpravu vzhledu aplikace
│   ├── gui ..... Glade soubory s nákresey GUI prvků
│   └── img ..... obrázky
├── tests ..... systémové testy
│   └── src ..... zdrojové kódy C++ aplikací pro testy
├── thesis ..... PDF soubor s textem této práce
└── util ..... pomocné skripty
```

Pro kompletní instalaci programu je nutné mít knihovnu GTK 3 (verze 3.10.8 nebo vyšší), překladač C++ a Pythonovské balíčky `enum34`, `matplotlib` a `clang`. Dále jsou požadovány (Debian) balíčky `texinfo` a `python-dev`. Pro spuštění testů je potřeba balíček `pytest` (ve verzi pro Python 2) a pro vygenerování dokumentace balíček `epydoc`. O sestavení programu se stará program *waf*. Pro jeho použití stačí spustit v interpretru příkazové řádky ve složce projektu `./waf <příkaz>`. Je možné pro něj použít následující příkazy.

- **configure** nakonfiguruje projekt, tento příkaz musí být zavolán před příkazem **build**
- **build** přeloží sdílenou knihovnu pro sledování dynamických alokací a stáhne a přeloží GDB 7.11
- **docs** vygeneruje dokumentaci programu do složky *docs*
- **download** stáhne všechny potřebné programy a balíčky pro běh aplikace pomocí balíčkového manažeru *Aptitude*

- **cleanall** vymaže dokumentaci, zkompilované soubory sdílené knihovny a GDB a soubory s byte kódem Pythonu

Pro rychlou instalaci lze použít přiložený skript `install.sh`, který provede `waf` příkazy `download`, `configure` a `build` (v tomto pořadí). Aplikaci lze otestovat pomocí příkazu `py.test` ve složce `tests`. Kompilace C++ programů pro testy vyžaduje na 64-bitových systémech balíček `g++-multilib`. *Devi* lze spustit pomocí skriptu `start.sh` v kořenovém adresáři projektu anebo spuštěním Python skriptu `initialize.py` ve složce `gui`. Lze mu předat jeden parametr příkazové řádky, a to cestu ke spustitelnému souboru, který se po spuštění aplikace načte. Binární soubory ukázek programů, které jsou umístěny ve složce `examples`, lze po sestavení projektu najít ve složce `build/examples`. Tyto ukázky lze opětovně přeložit pomocí příkazu `./waf build`.

Projekt je verzován pomocí verzovacího systému *git* a je dostupný také online na adrese <https://github.com/Kobzol/debug-visualizer>.