

Vizualizace ladění aplikací

Visualization of application debugging

Tuto stránku nahradíte v tištěné verzi práce oficiálním zadáním Vaší diplomové či bakalářské práce.

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava*.

Zde vložte text dohodnutého omezení přístupu k Vaší práci, chránící například firemní know-how. Zde vložte text dohodnutého omezení přístupu k Vaší práci, chránící například firemní know-how. A zavazujete se, že

1. o práci nikomu neřeknete,
2. po obhajobě na ni zapomenete a
3. budete popírat její existenci.

A ještě jeden důležitý odstavec. A ještě jeden důležitý odstavec. A ještě jeden důležitý odstavec. A ještě jeden důležitý odstavec. A ještě jeden důležitý odstavec. Konec textu dohodnutého omezení přístupu k Vaší práci.

V Ostravě 16. dubna 2009

+++
.....

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

V Ostravě 16. dubna 2009

+++
.....

Rád bych na tomto místě poděkoval všem, kteří mi s prací pomohli, protože bez nich by tato práce nevznikla.

Abstrakt

Tato bakalářská práce se zabývá vizualizací ladění programů napsaných v jazyce C a C++. První část pojednává o obecných principech ladění programů. Druhá část popisuje běžně používané ladící nástroje a jejich grafické nádstavby. Třetí část se zabývá implementací grafického nástroje, který vizualizuje paměť a stav procesu během jeho ladění za využití existujících ladících nástrojů

Klíčová slova: ladění programů, vizualizace paměti

Abstract

This is English abstract. This is English abstract. This is English abstract. This is English abstract. This is English abstract. This is English abstract.

Keywords: typography, L^AT_EX, master thesis

Seznam použitých zkratek a symbolů

API	– Application Programmable Interface
GDB	– The GNU Project Debugger
GNU	– GNU's Not Unix!
GUI	– Graphical user interface

Obsah

1	Úvod	5
2	Principy ladění programů	6
2.1	Signály	6
2.2	Krokování	6
2.3	Obousměrné mapování zdrojového kódu na instrukce	7
2.4	Běžné konstrukce ladících nástrojů	7
3	Existující ladící nástroje	9
3.1	GDB	9
3.2	LLDB	10
3.3	Grafická rozhraní	11
4	Vizualizační nástroj Devi	12
4.1	Popis problému	12
4.2	Specifikace požadavků	12
4.3	Architektura	13
4.4	Vizualizační část	15
4.5	Ovládání laděného procesu	15
4.6	Vizualizace paměti procesu	15
5	Závěr	16
5.1	Další vývoj	16
	Reference	17
	Přílohy	18
A	Grafická rozhraní debuggerů	19

Seznam tabulek

Seznam obrázků

1	Architektura Devi	14
2	Textové rozhraní GDB (TUI)	20
3	Rozhraní programu DDD	21
4	Rozhraní programu CLion (zdroj: https://www.jetbrains.com/clion) . . .	22

Seznam výpisů zdrojového kódu

1 Úvod

Ladění je nezbytná součást vývoje programů, která dovoluje programátorům detailně sledovat a ovládat laděný proces, aby v něm mohli odhalit chyby a lépe pochopit jeho průběh. K tomuto slouží ladící programy, které vytváří asociaci mezi zdrojovým kódem a binárním spustitelným souborem a poskytují tak tvůrci kódu možnost ladit kód na vysoké úrovni abstrakce, tj. na úrovni samotného zdrojového kódu. Cílem této práce je vývoj grafického rozhraní, které bude nezávislé na použitém ladícím programu a bude umožňovat vizualizaci laděného programu pro lepší pochopení jeho vnitřního stavu.

První kapitola popisuje obecné principy ladění, které jsou společné pro všechny ladící programy. Druhá kapitola pojednává o existujících ladících programech a jejich grafických rozhraních. Třetí kapitola se věnuje návrhu a vývoji rozhraní pro přístup k ladícím programům a grafického rozhraní pro vizualizaci ladění programů.

Tato práce je zaměřená 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 v této práci používán termín debugger, jelikož se jedná o často používaný programátorský termín a v češtině pro něj neexistuje zavedená alternativa.

2 Principy ladění programů

Tato kapitola popisuje obecné principy fungování ladících nástrojů, způsob mapování binárních instrukcí programu zpět do jeho zdrojového kódu, krokování běžícího procesu a běžné konstrukce používané při ladění. Konkrétně je popis zaměřen na programy napsané v jazycích C a C++ v prostředí Linuxových systémů používající procesory z rodiny Intel x86. Popsané principy jsou ale obecné a lze je aplikovat na libovolný operační systém.

2.1 Signály

Pro ladění programu je nutné mít možnost číst jeho paměť, aby šlo zkoumat hodnoty proměnných za jeho běhu, a také ho zastavit, jelikož programy za běhu provádějí obrovské množství instrukcí za vteřinu a zkoumat takto rychle se měnící datový tok by bylo obtížné. Aby šlo proces zastavit, musí mu jiný proces anebo sám operační systém zaslat signál. Signály jsou zprávy, které lze zaslat běžícímu procesu, ten si je může odchytnout a zareagovat na ně.[1, s. 21] Slouží pro meziprocení komunikaci 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**¹, 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 ladící nástroje, které proces sledovaný pomocí ptrace mohou po jeho zastavení znovu spustit, přistupovat k jeho paměti a ovlivňovat jeho průběh. Pokud je sledovaný proces potomkem procesu, který ho sleduje, bude při jeho spuštění vyvolán signál SIGTRAP, který dovolí rodiči odchytnout začátek provádění potomka. Jakmile je proces zastavený, může ho ladící nástroj tzv. 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). Ladící nástroje obvykle nabízejí krokování na vyšší úrovni než pouze po jedné instrukci, jelikož to by bylo zbytečně zdlouhavé (u vyšších programovacích jazyků se jeden řádek

¹<http://linux.die.net/man/2/ptrace>

zdrojového kódu může mapovat na desítky až stovky instrukcí). Obvykle jsou dostupné následující typy krokování:

Step over - krokování po řádku zdrojového kódu

Step in - stejná funkce jako step over, ale program se zastaví, pokud vstoupí dovnitř funkce

Step out - program bude pokračovat, dokud neskončí funkce, ve které se právě nachází

2.3 Obousměrné mapování zdrojového kódu na instrukce

Aby mohly ladící nástroje nabízet krokování na úrovni (řádků) zdrojového kódu, musí umět namapovat zdrojový kód na instrukce vygenerovaného spustitelného programu i instrukce zpět na zdrojový kód. 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 mapováním zdrojového kódu a vloží je do přeloženého programu. V překladačích jazyka C/C++ se tohoto dá standardně dosáhnout použitím řepínače **-g**. Existuje několik formátů ukládání těchto metadat, dnešním de facto standardem na Linuxových systémech je DWARF². Ten ukládá proměnné, datové typy, procedury a další údaje ze zdrojového kódu ve stromové struktuře. Pro ušetření místa obsahuje instrukce pro speciální konečný automat, který implementují ladící nástroje a pomocí něho poté získávají informace o původním zdrojovém kódu.

Samotné mapování zdrojového kódu není pro ladící nástroj užitečné, pokud je výsledný program zoptimalizovaný překladačem. Po optimalizaci totiž program nemusí obsahovat všechny původní proměnné, funkce a jeho průběh ani nemusí přesně odpovídat jeho zdrojovému kódu. Při použité málo agresivní optimalizace někdy lze programy úspěšně ladit, ale pro zajištění co nejpresnějšího ladění programů je obvykle nutné optimalizace úplně vypnout. Toho lze v překladačích obvykle dosáhnout použitím přepínače **-O0**.

2.4 Běžné konstrukce ladících nástrojů

Breakpoint Většina ladících nástrojů poskytuje svým uživatelům možnost zastavit běh laděného procesu pomocí tzv. breakpointu. 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 ladící nástroj nejdříve zjistí z daného 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, a lze jím tak nahradit libovolnou instrukci[2, s. 306]. Pokud by měla více než jeden byte, mohlo by ses stát, že by tato

²<http://dwarfstd.org>

instrukce přepsala více než jednu instrukci, což by mohlo způsobit nevalidní chování programu. Jakmile program během svého běhu provede tuto instrukci, vyvolá se signál SIGTRAP, který ladící nástroj odchytí a laděný proces se tímto zastaví. Pokud se uživatel rozhodne proces opět sputit, ladící nástroj zkopíruje původní instrukci programu (kterou si dříve uložil) na místo, kde vložil přerušení, nastaví na ni ukazatel příští instrukce a proces opět spustí. Některé procesory nabízí také hardwarový breakpoint, který sice může být rychlejší, ale obvykle kvůli tomu, že je implementován v hardwaru, tak poskytuje vytvoření pouze několika breakpointů zároveň.

Tracepoint V některých případech není možné laděný proces pozastavit k prozkoumání jeho paměti, jelikož jeho průběh může záviset na reálně uběhlém čase a zastavení tedy 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.

Watchpoint Pokud je potřeba zastavit program ne na konkrétním místě, ale při změně dané hodnoty v paměti, lze použít watchpoint. Ten se může hodit například pro kontrolu změn globálních proměnných. Pokud nenabízí procesor hardwarovou podporu pro watchpointy, ladící nástroj prochází laděný proces instrukci po instrukci, testuje hodnotu sledované paměti a pokud se tato hodnota změní, tak program zastaví. Tento proces může zpomalit laděný proces až o dva řády³.

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

³<https://sourceware.org/gdb/onlinedocs/gdb/Set-Watchpoints.html>

3 Existující ladící nástroje

Ladících nástrojů pro jazyky C a C++ existuje velké množství, v této kapitole jsou popsány dva z nejpoužívanějších nástrojů, GDB a LLDB, které byly dále použity při implementaci vizualizačního nástroje. Jsou zde taky popsány jejich vybrané grafické nástavby.

3.1 GDB

GDB (The GNU Debugger) je standardním ladícím nástrojem pro Linuxové systémy, často je v těchto systémech 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[3]. Podporuje velké množství rodin 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++, pokud je dodržen jejich standard. 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 ladící nástroj, jako je načtení ladících metadat v mnoha formátech (podporuje DWARF i několik dalších formátů), vytváření breakpointů, tracepointů a watchpointů, krokování programu a čtení i zápis paměti programu. Mimo to ale nabízí i pokročilé funkce, které ovšem nemusí být podporované všemi procesory a platformami, s kterými GDB jinak umí pracovat.

Vzdálené ladění GDB dokáže být spuštěn na jednom zařízení a ladit program spuštěný 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 (kernelů) operačního systému, čehož je využito například v programu KGDB, který se používá k ladění jader operačních systémů Linux a FreeBSD pomocí sériového připojení.

Ladění vícevláknových aplikací Pokud GDB ladí program, který využívá více než jedno vlákno, může pracovat v několika rozlišných módech[4]. 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, tak, aby ostatní 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 zbytek vláken stále běží.

Provádění výrazů Pomocí GDB lze nejenom číst a zapisovat paměť laděného procesu na úrovni bytů v adresním prostoru procesu, ale v podporovaných jazycích, hlavně v

C a C++, lze také provádět libovolné jazykové výrazy, volat funkce programu a systémová volání a pracovat s hodnotami na úrovni proměnných laděného programu.

Spolupráce s Valgrindem Valgrind je nástroj pro profilování a kontrolu paměťové korektnosti programů, který se využívá k hledání paměťových chyb, jako je například memory leak. Vytváří virtuální stroj, ve kterém spouští zkoumaný program a kvůli této vlastnosti jej nelze ladit klasickými přístupy. GDB poskytuje možnost připojit se k programu spuštěnému ve Valgrindu a vzdáleně ho takto ladit.

Analýza logu z ukončeného procesu Procesy, které se ukončí s chybou, např. po vyvolání výjimky, můžou vygenerovat výpis paměti (core dump), který lze poté načíst v GDB a zanalyzovat ho. Lze tak například zobrazit stav zásobníku volání funkcí v momentu, kdy program zhavaroval, a zjistit tak, který kód programu způsobuje chybu.

Zpětné provádění instrukcí Při ladění nastává často situace, kdy proces zajde moc daleko a přeskočí instrukci, kterou chce uživatel zkoumat. GDB umí spouštět určité instrukce zpětně, a může tedy krokovat program nejenom dopředu, ale i dozadu. Všechny změny a vedlejší efekty, které proběhly v paměti, jsou tak smazány a navráceny do původního stavu (pokud to daná platforma a stav programu dovoluje).

GDB nemá vlastní grafické rozhraní, je ovládán z příkazové řádky. Kromě toho ale podporuje také spouštění skriptů v Pythonu pomocí API, které bylo použito pro implementaci vizualizačního nástroje a je popsáno dále v textu.

3.2 LLDB

Ladící nástroj LLDB⁴ 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 optimalizovat a vygenerovat z ní dále spustitelný soubor pro libovolnou kompatibilní platformu. Obsahuje také kompletní implementaci standardní knihovny jazyka C++, která plně podporuje jeho nejnovější standard, C++11. Clang je nádstavbou LLVM, která analyzuje a překládá programy v jazyce C a C++. Celá LLVM architektura je postavena na modulárních komponentech, které spolupracují a dají se lehce využít ve formě knihovny. Nabízí tak modernější alternativu k programům GCC a GDB. Ty jsou sice stabilnější a prověřenější, ale jelikož existují už desítky let a musí udržovat zpětnou kompatibilitu, tak je těžší je využít jako modul do jiného programu. Z tohoto důvodu rovněž pomaleji přecházejí k novým standardům.

Umí ladit programy napsané v jazycích C, C++, Objective-C a Swift na platformách OS X, Linux, Free BSD a Window. Podporuje tedy méně jazyků i platforem, než GDB, ale narozdíl od něho je podporován, a stal se také standardním ladícím nástrojem, i operačními systémy OS X a iOS. Nabízí většinu standardních funkcí ladících nástrojů, jako

⁴<http://llvm.org>

je krokování kódu, vytváření breakpointů a 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. Některé z nich jsou samostatné programy podporující pouze ladění, další jsou jednou z mnoha součástí integrovaných vývojových prostředí. Rozhraní těchto nástrojů se obvykle skládá z textového editoru, který obsahuje zdrojový kód, ovládacích prvků, které kontrolují průběh laděného procesu. Dále také často nabízí manipulaci a zobrazování registrů, lokálních proměnných a parametrů funkcí. Ukázky uživatelského rozhraní jednotlivých programů si lze prohlédnout v příloze A. Následuje popis jednoho zástupce ze skupiny samostatných (DDD), integrovaných (Clion) a textových (TUI) uživatelského rozhraní pro ladění programů.

TUI Text User Interface je grafickým rozhraním vestavěným přímo v GDB, které zobrazuje stav průběhu v několik terminálových oknech pro větší přehlednost programu. Je postaveno na knihovně *curses*, která umožňuje vytvářet textové uživatelské rozhraní s pokročilými funkcemi přímo v terminálu. TUI lze spustit předáním parametru **-tui** při spouštění GDB anebo stisknutím kláves CTRL+X či spuštěním příkazu *tui enable* za jeho běhu. LLDB obsahuje podobné rozhraní také, ale není zatím oficiální součástí nástroje, jedná se pouze o nezávazně vyvíjený doplněk.

DDD DDD, neboli Data Display Debugger, je grafické prostředí podporující velké množství ladících nástrojů, 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.

Clion Clion 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 velmi urychluje pochopení stavu výpočtu.

4 Vizualizační nástroj Devi

Tato kapitola obsahuje popis vizualizace ladění a návrh a implementaci vizualizačního nástroje.

4.1 Popis problému

Jazyky C a C++ jsou velmi komplexní a dovolují programátorům pracovat s hardwarem počítače na značně nízké úrovni. Kvůli mnohým vlastnostem, jako je absence automatické správy paměti nebo možnost provádět potencionálně nebezpečné operace s paměťovými ukazateli, jsou velmi náchylné ke vzniku těžko odhalitelných chyb. Používání debuggeru, který dokáže detailně zkoumat paměť běžícího procesu a krokovat ho, je tedy v těchto jazycích nutností, bez které by byl vývoj C a C++ aplikací výrazně pomalejší. Existující debuggery, jako je GDB nebo LLDB, mají velké množství užitečných funkcí, ale samy o sobě poskytují svému uživateli pouze ladění pomocí terminálu. To je pro velké programy velmi pomalé a nepřehledné řešení. Nad těmito nástroji proto vznikla různá grafická rozšíření, ať už samostatná nebo integrovaná v komplexních vývojových prostředích. Tato uživatelská rozhraní umožňují mnohem pohodlnější a jednodušší ladění programů. Díky tomu může být ladění přirozenou součástí vývoje programů, a ne pouze obtížnou činností nutnou při hledání skrytých chyb. I když grafické uživatelské rozhraní ladění značně usnadňuje, tak reprezentace paměti se v něm většinou omezuje na pouhý textový výpis názvů proměnných a jejich hodnot. To postačuje pro hledání chyb v programu, ale nedovoluje to plně zobrazit vztahy mezi objekty a jejich umístění v paměti. Zobrazení grafické reprezentace objektů v paměti může být také užitečné pro pochopení a sledování průběhu určitých (např. třídících) algoritmů.

4.2 Specifikace požadavků

Cílem této práce bylo navrhnout a naimplementovat program, který bude ladění programu vizualizovat ve formě grafu objektů v paměti, a otestovat, jestli tato grafická reprezentace usnadňuje ladění programů a pochopení průběhu jednoduchých algoritmů. Program by také měl poskytovat přístup k běžným funkcím debuggerů. Následuje seznam základních funkcí, které by měl svým uživatelům nabízet.

- Asynchronní komunikace s debuggerem
- Načítání binárních i zdrojových souborů
- Zobrazování zdrojového i strojového kódu programu
- Vytváření a odebrání breakpointů
- Manipulace s procesem (spuštění, 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
- Přepínání vláken
- Zobrazování registrů a paměti na úrovni bytů
- Manipulace s hodnotami proměnných
- Vizualizace objektů v paměti ve formě grafu

S touto sadou funkcí by měl být tento program plně použitelný pro ladění reálně používaných aplikací.

4.3 Architektura

Pro implementaci programu jsem zvolil programovací jazyk **Python**⁵ ve verzi 2.7. Novější verze 3.x nebyla použita z důvodu zachování kompatibility s již existujícími API pro debuggery, které v době tvorby této práce Python 3 ještě zcela nepodporovaly. Aplikaci jsem psal tak, aby bylo v případě potřeby snadné ji přepsat do Pythonu 3. Pro tuto konverzi lze použít i automatizované nástroje, například **2to3**⁶. Python jsem vybral, protože je vhodný k rychlému vývoji aplikací, existuje pro něj několik rozhraní pracujících s debuggery a umí snadno používat kód napsaný v jazycích C a C++. Navíc je multiplatformní, což by usnadnilo případný port aplikace na jiný operační systém.

Pro vývoj grafického rozhraní programu jsem vybral knihovnu **GTK+ 3**⁷. Jedná se o volně dostupný⁸, multiplatformní grafický software umožňující tvorbu uživatelského rozhraní, který je používán mimo jiné i některými manažery plochy pro Linux (např. v GNOME⁹). Kromě možnosti tvorby vlastních GUI prvků obsahuje několik desítek běžně používaných prvků, které jsou předpřipravené k okamžitému použití a usnadňují tak rychlý vývoj aplikací.

Vizualizační nástroj jsem pojmenoval a dále jej v textu budu označovat jako *Devi*. Nástroj je tvořen dvěmi samostatnými komponentami, aplikací s grafickým rozhraním, která slouží k vizualizaci a k interakci s laděným procesem, a knihovny, která poskytuje rozhraní pro komunikaci s libovolným debuggerem. Obrázek 1 zachycuje pohled na architekturu aplikace z vysoké úrovně. Uživatelské rozhraní komunikuje s knihovnou, která zprostředkovává komunikaci s debuggerem. Na obrázku jsou uvedeny 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.

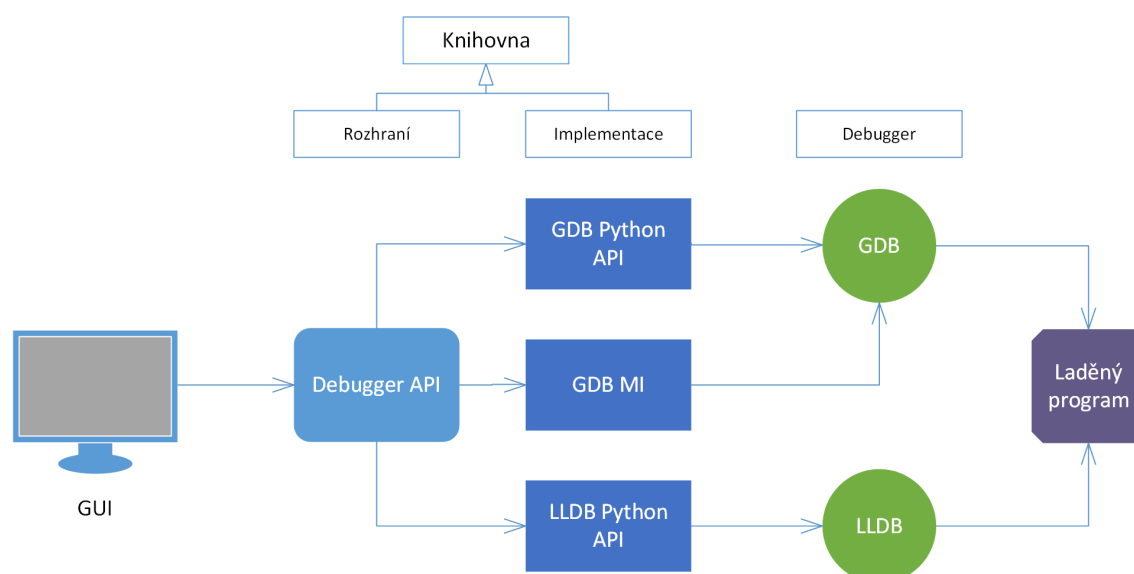
⁵<http://www.python.org>

⁶<https://docs.python.org/2/library/2to3.html>

⁷<http://www.gtk.org>

⁸Pod licencí LGPL 2.1

⁹<https://www.gnome.org/technologies>



Obrázek 1: Architektura Devi

4.3.1 Knihovna pro komunikaci s debuggery

Tato knihovna tvoří rozhraní pro komunikaci s libovolným ladícím nástrojem. Definuje abstraktní typy popisující laděný proces a není tak závislá na použitém ladícím nástroji. Není nijak závislá ani na grafickém rozhraní nástroje, lze ji tedy použít pro programovatelné ovládání debuggerů i v jiném projektu.

Knihovna obsahuje třídy představující části laděného procesu, které jsou dostatečně obecné na to, aby se daly aplikovat na libovolný proces i debugger. Následuje popis těchto tříd, které lze nalézt v modulu *debugger.debugee*.

Type - představuje datový typ proměnné, obsahuje jeho název, kategorii a velikost

Variable - představuje proměnnou, obsahuje její název, datový typ, adresu v paměti, hodnotu a potomky

Frame - představuje rámec zásobníku, obsahuje jeho úroveň v zásobníku a název a lokaci funkce, z které je vyvolán

InferiorThread - představuje vlákno procesu, obsahuje jeho identifikátor, název, stav a současný zásobníkový rámec

ThreadInfo - uchovává seznam všech vláken v procesu a také současně zvolené vlákno

Breakpoint - představuje breakpoint, obsahuje jeho identifikátor a zdrojový soubor a řádek, na kterém je umístěn

Register - představuje registr procesoru, obsahuje jeho název a hodnotu

Dále jsou v knihovně třídy, které tvoří obecné API pro přístup k debuggeru. Slouží jako rodičovské třídy pro jednotlivé implementace komunikace s debuggerem. Jejich smyslem není implementovat společné chování, jelikož jednotlivé debuggery a implementace přístupu k nim se od sebe můžou značně lišit, a tak poskytují pouze rozhraní. Následuje seznam tříd tohoto rozhraní.

Debugger - představuje abstrakci ladícího nástroje, který umožňuje načtení binárního souboru programu, jeho spuštění, zastavení a krokování. Dále také uchovává stav laděného procesu a obsahuje všechny ostatní komponenty rozhraní vyjmenované níže.

VariableManager - stará se o čtení registrů a paměti procesu a o získávání a změnu jeho proměnných

ThreadManager - stará se o získávání a volbu vláken a zásobníkových rámců

FileManager - stará se o získávání současné pozice a hlavního souboru laděného procesu. Poskytuje také převod zdrojového kódu na strojový kód.

BreakpointManager - stará se o přidávání, mazání a načítání breakpointů

IOManager - stará se o komunikaci s laděným procesem pomocí čtení z jeho standardního a chybového výstupu a zápisu do jeho standardního vstupu

Pro účely této práce jsem v této knihovně vytvořil několik implementací pro komunikaci s GDB a LLDB, které budou dále popsány.

4.3.1.1 Python API pro GDB GDB obsahuje rozhraní ¹⁰, které nabízí možnost načtení skriptů v Pythonu 2, které můžou ovládat GDB a přistupovat k jeho API. To nabízí většinu nejpoužívanějších funkcí GDB ve formě tříd a metod. Funkce GDB, které v API nejsou obsaženy, lze vyvolat pomocí přímého provádění textových příkazů. Nevýhoda tohoto API je, že neumí samo spustit instanci GDB a musí tak být načteno v již běžícím procesu GDB. Nelze jej tedy použít přímo v kódu aplikace, protože samotné API běží v jiném procesu. Tento problém byl vyřešen komunikací se skriptem běžícím v procesu GDB pomocí TCP/IP. Jelikož je ale toto řešení komplikovanější než ostatní způsoby práce s GDB a při testování nebylo stabilní, tak jeho implementace nebyla dokončena a byl zvolen jiný postup, který je popsán níže.

¹⁰<https://sourceware.org/gdb/onlinedocs/gdb/Python-API.html>

4.3.1.2 Protokol GDB/MI MI je komunikační rozhraní pro ovládání GDB pomocí textových příkazů. Nejedná se o klasické příkazy používané při manuálním ovládání GDB, ale o speciální příkazy přizpůsobené pro jednoduché strojové zpracování. Obdobně jako Python API pro GDB, obsahuje rozhraní pro nejpoužívanější funkce GDB a zbytek funkcí lze používat pomocí běžných textových příkazů. Použití MI je běžným a doporučeným [5] způsobem pro programovou komunikaci s GDB a psaní grafických rozhraní. Existují volně dostupné knihovny pro Python i C/C++, které umí komunikovat s GDB pomocí MI protokolu, nicméně za účelem pochopení fungování tohoto protokolu a maximální volnosti v implementaci byl vytvořen vlastní komunikační modul.

4.3.1.3 Python API pro LLDB Stejně jako GDB, i LLDB poskytuje API v Pythonu. Lze jej načíst do procesu LLDB a automatizovat průběh ladění, navíc jej ale lze také použít jako knihovnu, která sama vytváří instance LLDB a je tedy jednoduché ji použít přímo v externím kódu. Stejně jako samotné LLDB je ale zatím ve vývoji a obsahuje drobné chyby, například nepřesné zobrazování stavu vláken.

4.4 Vizualizační část

4.5 Ovládání laděného procesu

4.6 Vizualizace paměti procesu

5 Závěr

5.1 Další vývoj

IDE - kompilace, projekty, doplňování kódu backendy pro více ladících nástrojů rozšíření vizualizace

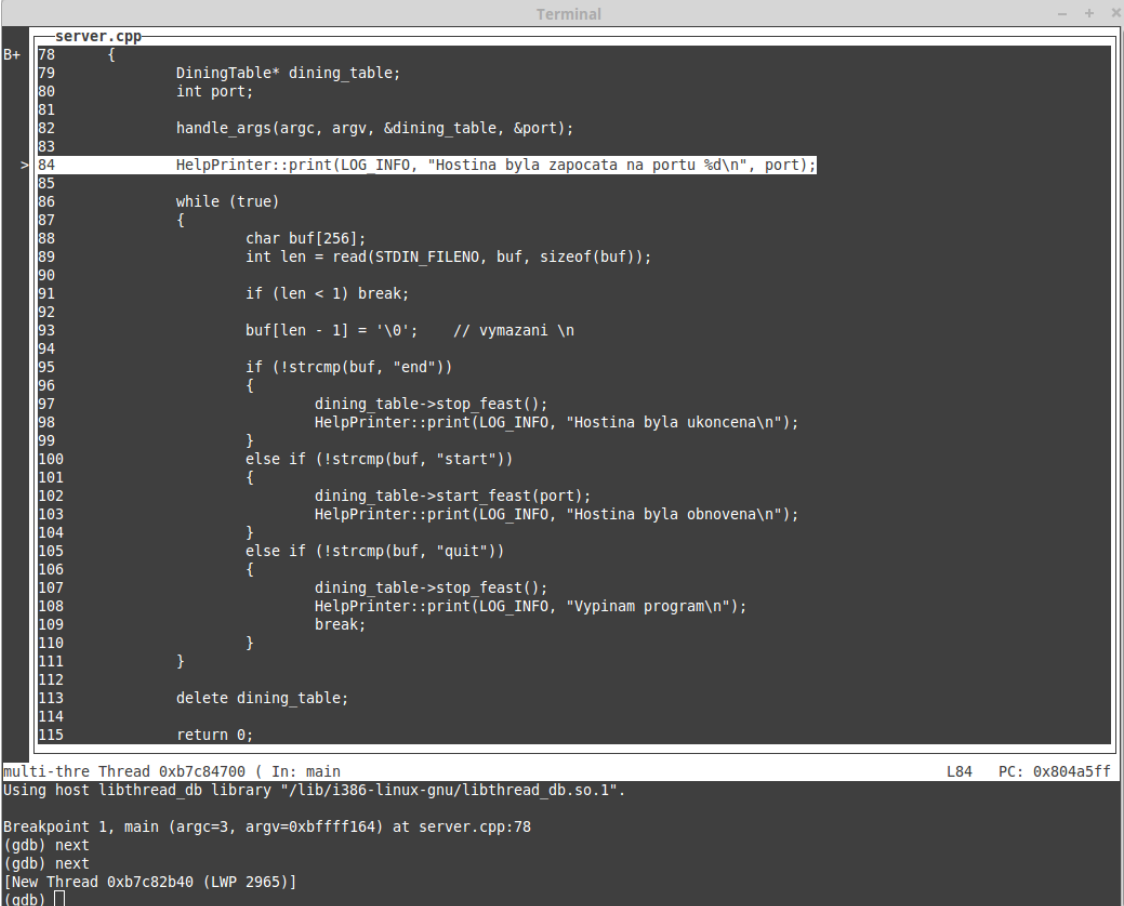
Jakub Beránek

Reference

Odkazy

1. 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.
2. *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>.
3. *Seznam jazyků podporovaných programem GDB*. Dostupný z WWW: <https://sourceware.org/gdb/current/onlinedocs/gdb/Supported-Languages.html>.
4. *Módy GDB pro práci s vícevláknovými aplikacemi*. Dostupný z WWW: <https://sourceware.org/gdb/onlinedocs/gdb/Thread-Stops.html>.
5. *Grafická rozhraní GDB používající MI*. Dostupný z WWW: <https://sourceware.org/gdb/wiki/GDB%20Front%20Ends>.

A Grafická rozhraní debuggerů

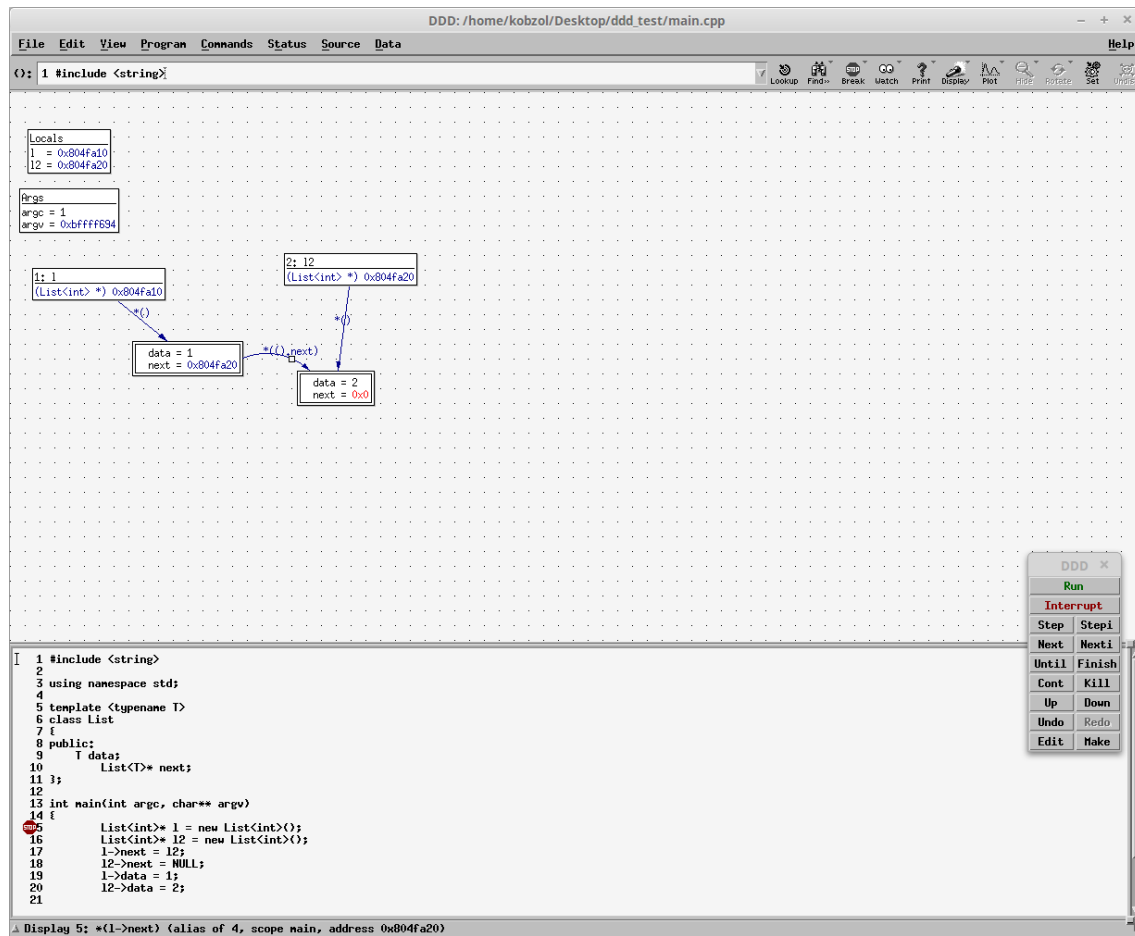


```
server.cpp
78 {
79     DiningTable* dining_table;
80     int port;
81
82     handle_args(argc, argv, &dining_table, &port);
83
84     HelpPrinter::print(LOG_INFO, "Hostina byla zapocata na portu %d\n", port);
85
86     while (true)
87     {
88         char buf[256];
89         int len = read(STDIN_FILENO, 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;
}

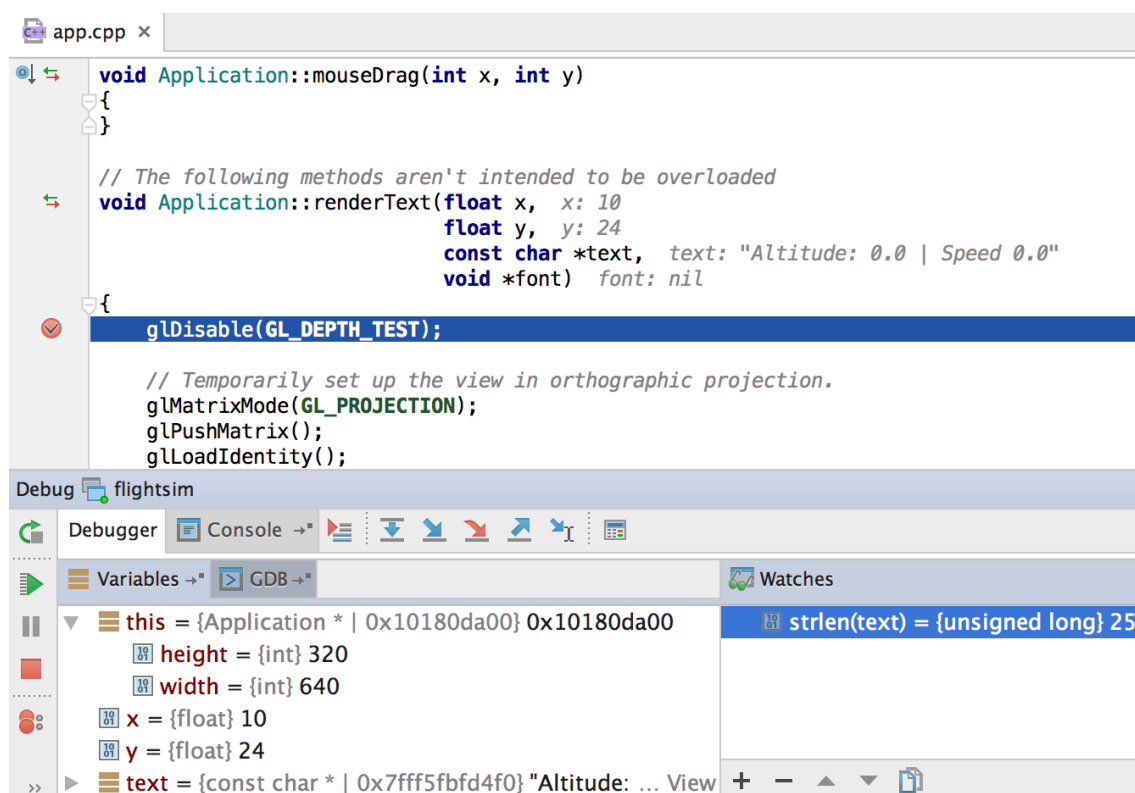
multi-thre Thread 0xb7c84700 ( In: main                                L84    PC: 0x804a5ff
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

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

Obrázek 2: Textové rozhraní GDB (TUI)



Obrázek 3: Rozhraní programu DDD



Obrázek 4: Rozhraní programu CLion (zdroj: <https://www.jetbrains.com/clion>)