VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Vizualizace ladění aplikací

Visualization of application debugging

2016 Jakub Beránek

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 práce, 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. 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 popisuje obecné principy ladění programů. Jsou zde popsány konstrukce, které se při ladění používají, způsob, jakým ladící nástroje provádějí ladění programů a také existující ladících nástrojy a jejich grafické nástavby. Druhá část popisuje možnosti komunikace s ladícími nástroji. Je v ní popsána implementace grafického nástroje, který vizualizuje pamět 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, LaTeX, 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

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Ladění je nezbytná součást vývoje programů, která dovoluje programátorům detailně sledovat a ovládat průběh běžícího procesu a také číst a zapisovat jeho paměť. K tomuto slouží ladící nástroje, 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.

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 UNIXových systémů na platformě Intel x86. Popsané principy jsou ale obecné a lze je aplikovat na libovolnou operační systém.

2.1 Signály

Pro ladění programu je nutné mít možnost číst jeho paměť, aby šly 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 odchytit a zareagovat na ně.[1, s. 21] Slouží pro meziprocesní 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 UNIXových 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. UNIXové 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 ladící nástroje, které proces sledovaný pomocí ptrace můžou 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 rodičovi odchytit 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)[2]. Ladící nástroje obvykle nabízí krokování na vyšší úrovni než pouze po jedné instrukci, jelikož to by bylo zbytečně zdlouhavé (u vyšších programovacích jazyků se jeden řádek 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 UNIXových systémech je DWARF[3]. 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 nejpřesnějšího 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[4, s. 306]. Pokud by měla více než jeden byte, mohlo by ses stát, že by tato 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 řády[5].

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 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 nádstavby.

3.1 GDB

GDB (The GNU Debugger) je standardním ladícím nástrojem pro UNIXové 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[6]. 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 UNIX (např. 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[7]. 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 LLDB je založen na sadě knihoven, které využívají infrastruktury LLVM a překladače Clang. LLVM je univerzální překladače, 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[8]. 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 a 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 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.

Název	IDE	Aktivní	GDB	LLDB
DDD	Ne	Ne	Ano	Ne
TUI	Ne	Ano	Ano	Ano

Tabulka 1: Srovnání grafických rozhraní ladících nástrojů

3.3 Grafická rozhraní

V této sekci jsou popsány vybrané grafické nádstavby pro GDB a LLDB.

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.

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.

KGdb

Clion Eclipse (UltraGDB) NetBeans Code::Blocks KGdb WinGDB

3.3.1 Srovnání grafických rozhraní

4 Implementace vizualizačního nástroje

Jazyky C a C++ jsou velmi komplexní a dovolují programátorům pracovat s hardwarem počítače na hodně nízké úrovni. Kvůli mnohým svým vlastnostem, jako je absence automatické správy paměti nebo možnost provádět nebezpečné operace s ukazateli do paměti, jsou ale také velmi náchylné ke vzniku četných chyb. Používání ladícího nástroje, který dokáže detailně zkoumat paměť běžícího procesu a krokovat ho, je tedy v těchto jazycích nutností. Existující ladící programy jako je GDB nebo LLDB fungují dobře, ale poskytují uživateli pouze ladění v příkazové řádce, což je pro velké programy velmi pomalé a nepřehledné řešení. Nad těmito nástroji proto vznikly různé grafické rozšíření, ať už samostatné, anebo integrované ve vývojových prostředích. Ty obvykle poskytují mnohem pohodlnější ladění programů, které je díky tomuto i snadněji zařazené do celého pracovního postupu vytváření kódu.

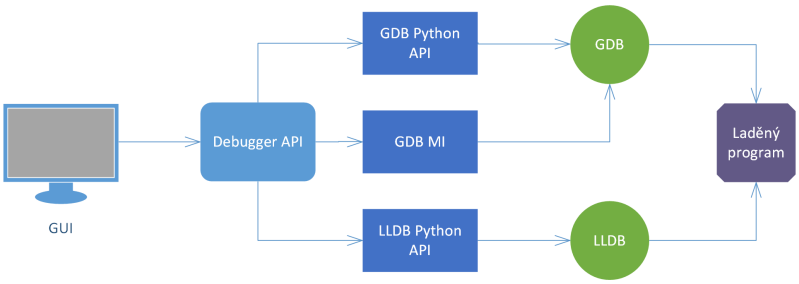
Cílem této práce bylo navrhnout a naimplementovat program, který bude ladění programu vizualizovat ve formě grafu, a otestovat, jestli tento postup pomáhá při ladění programů a vizualizaci jednoduchých algoritmů. Mimo jiné by tento program měl nabízet také funkce, které jsou obvyklé v jiných grafických rozhraních pro ladící nástroje, jako je krokování programu, zobrazování zdrojového kódu, přepínání zásobníkových rámců a ovládání vláken. V této kapitole je popsána architektura a implementace tohoto programu.

Pro implementaci programu byl vybrán programovací jazyk Python[1] ve verzi 2.7. Novější verze Python 3 nebyla použita z důvodu zachování kompatibility s API pro LLDB, které Python 3 nepodporuje. Python byl vybrán, protože je vhodný k rychlému vývoji aplikací, existuje pro něj několik rozhraní pracujících s ladícími nástroji a umí snadno používat existující kód v jazycích C a C++. Navíc je multiplatformní, což by usnadnilo případný port aplikace na jiný operační systém. Pro vývoj grafického rozhraní programu byla použita knihovna GTK+ 3[2], což je volně dostupný[3], multiplatformní grafický software umožňující tvorbu uživatelského rozhraní. Kromě možnosti tvorby vlastních GUI prvků obsahuje několik desítek často používaných prvků, které jsou předpřipravené k okamžitému použití a usnadňují tak rychlý vývoj aplikací.

4.1 Architektura nástroje

Nástroj tvoří dvě samostatné komponenty, aplikace s klasickým grafickým rozhraním, která slouží k interakci s uživatelem nástroje, a knihovny, která poskytuje rozhraní pro komunikaci s ladícími nástroji. Následuje popis obou komponent.

[1] http://www.python.org
[2] http://www.gtk.org
[3] Pod licencí LGPL 2.1



Obrázek 1: Architektura nástroje

4.1.1 API pro komunikaci s ladícími nástroji

Tato komponenta je knihovna napsaná v Pythonu, která tvoří rozhraní pro komunikaci s libovolným ladícím nástrojem. Toto rozhraní definuje abstraktní typy popisující laděný proces a není tak závislé na použitém ladícím nástroji. Není nijak závislé ani na grafické komponentě nástroje, lze jej tedy vyjmout a použít pro programové ovládání ladících nástrojů i v jiném projektu. Pro účely této práce bylo vytvořeno několik implementací pro komunikaci s GDB a LLDB, které budou dále popsány.

4.1.1.1 Python API pro GDB

GDB obsahuje rozhraní [9], 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.

4.1.1.2 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 [10] 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.

4.1.1.3 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.

4.1.1.4 Debugger v Pythonu

(Dopsat debugger?)

4.2 Vizualizační část

4.3 Ovládání laděného procesu

4.4 Vizualizace paměti procesu

5 Závěr

Jakub Beránek

Reference

Knižní zdroje

[1] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice Hall, 1987. ISBN: 0131429388.

Online zdroje

[2] *Funkce ptrace*. URL: `http://linux.die.net/man/2/ptrace`.

[3] *DWARF Debugging Standard*. URL: `http://dwarfstd.org/`.

[4] *Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference*. 1999. URL: `http://download.intel.com/design/intarch/manuals/24319101.pdf`.

[5] *Watchpointy v GDB*. URL: `https://sourceware.org/gdb/onlinedocs/gdb/Set-Watchpoints.html`.

[6] *Seznam jazyků podporovaných programem GDB*. URL: `https://sourceware.org/gdb/current/onlinedocs/gdb/Supported-Languages.html`.

[7] *Módy GDB pro práci s vícevláknovými aplikacemi*. URL: `https://sourceware.org/gdb/onlinedocs/gdb/Thread-Stops.html`.

[8] *LLVM*. URL: `http://llvm.org/`.

[9] *Python API pro GDB*. URL: `https://sourceware.org/gdb/onlinedocs/gdb/Python-API.html`.

[10] *Grafická rozhraní GDB používající MI*. URL: `https://sourceware.org/gdb/wiki/GDB%20Front%20Ends`.