Razhroščevalniki (Debuggerji)

Predstavitev razhroščevalnikov in njihovega delovanja v sistemu Linux za predmet Sistemska programska oprema na Fakulteti za računalništvo in informatiko. In HDH.

Avtor: Jakob Erzar

Kazalo

Pregled razhroščevalnikov

Opis

Razhroščevalnik je program, s katerim testiramo ali razhroščujemo drug ("ciljni") program. Njihov cilj je, da programerju pomagajo razumeti program in najti vzrok napake v programu. Omogočajo spremljanje poteka izvajanja ciljnega programa in programerju omogočajo, da ob katerikoli točki ustavi program ter pogleda stanje programa in preveri pravilnost njegovega delovanja. Vključuje lahko tudi t.i. *instruction set simulator* (ISS), torej simulator izvajanja ukazov na ciljni arhitekturi, kot npr. različni emulatorji. V tem primeru je izvajanje tipično počasnejše kot neposredno izvajanje, a lahko omogoča še več funkcionalnosti. Zaradi hitrosti izvajanja lahko razhroščevalniki ponujajo več načinov izvajanja - polno ali delno simulacijo.

Tipi razhroščevalnikov

Source-level (symbolic) debugger	**Machine-level debugger**
- višjenivojski programski jeziki	- strojna koda
- potrebno mapiranje	- bolj enostavno, 1:1

Z večjo uporabo višjenivojskih programskih jezikov je narasla potreba po debuggerjih, kjer je potrebno preslikovanje iz disassemblyja nazaj v izvorno kodo, kjer pa nastane težava - vsak ukaz v strojnem jeziku nima nujno enakega v izvornem jeziku. Zato je potrebno pri prevajanju dodati podatke o izvorni kodi, ki debuggerju pomagajo pri mapiranju kode nazaj.

Stand-alone debugger	**IDE (integrirano razvojno okolje)**
- program, ki je namenjen samo razhroščevanju	- vključuje tudi prevajalnik, povezovalnik, itd.
- več prostosti pri uporabi drugih orodij	- boljša povezava z prevajalnikom, bolj priročni

Razhroščevalniki so te dni pogosto vključeni v integrirano razvojno okolje, kar je veliko bolj priročno od uporabe posebnega programa za razhroščevanje.

Funkcionalnosti

- **prekinitvene točke (breakpoint)** - ustavljanje programa (ob določeni točki)
- **izvajanje programa korak za korakom** (single-stepping / step by step)
- **pregled trenutnih vrednosti** v registrih in spominu
- spreminjanje trenutnega stanja med tekom
- nadaljevanje izvajanja programa na drugi lokaciji
- prekinitvene točke pri dostopih do pomnilnika

Vzvratno razhroščevanje (reverse debugging)

Nekateri razhroščevalniki omogočajo vzvratno razhroščevanje, kar omogoča sprehod po izvajanju programa nazaj v času. Taki razhroščevalniki močno upočasnijo izvajanje programa, najboljši

upočasnijo le za 2x ali manj. Ponekod lahko pride tak tip razhroščevanja zelo dobro v poštev, vendar še vedno ni v pogosti rabi.

Debugger front-ends

Nekateri razhroščevalniki nudijo le CLI, saj s tem povečajo prenosljivost in zmanjšajo uporabo virov, a je za večino programerjev GUI veliko bolj enostaven in hiter za uporabo. Zato obstajajo tudi programi, ki so le GUI vmesniki za specifične CLI debuggerje (npr GDBGUI za GDB)

Primeri razhroščevalnikov

- GDB - GNU debugger
- WinDbg
- Microsoft Visual Studio Debugger
- Eclipse debugger API
- Valgrind
- LLDB
- Chrome Dev Tools
- Xdebug

GNU Debugger (GDB)

Prenosljiv razhroščevalnik, ki teče na večih Unix-like sistemih in deluje z večimi programskimi jeziki, kot Ada, C, C++, Objective-C, D, Pascal, Fortran, Go, Rust,... na velikem številu različnih arhitektur - X86 in X64, IA-64, ARM, Alpha, AVR, H8/300, Motorola 68000, MIPS, PA-RISC, PowerPC, SuperH, SPARC,...

Napisal ga je Richard Stallman v letu 1986 kot del njegovega GNU sistema, in je izdan pod GNU GPL licenco. Še vedno je v aktivnem razvoju (8.0 je izšla junija 2017), ki ga zdaj vodi GDB Steering Committee.

Oddaljeno razhroščevanje

GDB omogoča tudi oddaljeno razhroščevanje - GDB instanca na enem sistemu lahko preko TCP / IP ali serijske naprave komunicira z programom, ki razume GDB protokol. Tak program lahko ustvarimo z povezovanjem programa z določenimi GDB datotekami ali pa uporabimo gdbserver.

Ker nima svojega GUI vmesnika, le CLI, obstaja zanj veliko GUI vmesnikov, ali pa programov, ki lahko z njim komunicira.

Primeri GDB front-endov

- KDbg (del KDE razvojnih orodij)
- Emacs
- Qt Creator
- Xcode
- CodeBlocks
- Eclipse C/C++ Development Tools
- CodeLite

- Visual Studio

Delovanje razhroščevalnikov

ptrace (Linux)

```
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

> The ptrace() system call provides a means by which one process (the ”tracer”) may observe and control the execution of another process (the ”tracee”), and examine and change the tracee’s memory and registers. It is primarily used to implement breakpoint debugging and system call tracing. - `man 2 ptrace`

Sistemski klic ptrace() omogoča procesu spremljanje in nadzoroavanje izvajanja drugega procesa.

Pripenjanje procesu (attaching to a process)

Procesu se lahko pripnemo na dva načina - z uporabo `PTRACE_TRACEME`, s katerim starševskem procesu dovolimo, da nadzira ta proces. Če se želimo priključiti že zagnanem procesu pa lahko namesto tega uporabimo ukaz `PTRACE_ATTACH`.

Iz man page o PTRACE_TRACEME:

> While being traced, the tracee will stop each time a signal is delivered, even if the signal is being ignored. (An exception is SIGKILL, which has its usual effect.) The tracer will be notified at its next call to waitpid(2) (or one of the related "wait" system calls); that call will return a status value containing information that indicates the cause of the stop in the tracee. While the tracee is stopped, the tracer can use various ptrace requests to inspect and modify the tracee. The tracer then causes the tracee to continue, optionally ignoring the delivered signal (or even delivering a different signal instead).

Varnost - Yama

Primer uporabe PTRACE_TRACEME

Zaradi dolžine primera sem ga premaknil v datoteko ptrace_traceme_example.c

Primeri ukazov `gdb` in `ptrace`

gdb	ptrace	Razlaga
`(gdb) start`	PTRACE_TRACEME	Omogoča staršu, da lahko spremlja proces
`(gdb) attach pid`	PTRACE_ATTACH	Pripetje procesu, ki se že izvaja

gdb	ptrace	Razlaga
`(gdb) stop`	`kill(child_pid, SIGSTOP)` (or PTRACE_INTERRUPT)	Ustavi ciljni proces
`(gdb) continue`	PTRACE_CONT	Nadaljuje izvajanje procesa
`(gdb) info registers`	PTRACE_GET(FP)REGS(ET) in PTRACE_SET(FP)REGS(ET)	Omogoča pregled in spreminjanje vrednosti registrov
`(gdb) x`	PTRACE_PEEKTEXT in PTRACE_POKETEXT	Dovoljuje pregled in pisanje po spominu.

Prekinitvene točke (breakpoints)

Nastavljanje prekinitvene točke

Ko razhroščevalniku naročimo, naj na izbrano vrstico v kodi doda ustavitveno točko, si **razhroščevalnik ta ukaz shrani v spomin**, nato pa **na to mesto zapiše**:

- poseben ukaz (na x86 je to `INT 3`)
- nedefinirano kodo (npr. pri ARM imajo v ta namen rezervirane posebne kode, ki so v dokumentaciji označene kot nedefinirane, da se lahko uporabljajo v ta namen)

V gdb lahko prekinitveno točko nastavimo z ukazom `(gdb) br *ADDRESS`.

Pasti (traps)

Razhroščevalniki si pomagajo s **pastmi**, ki se sprožijo, ko procesor ne more normalno delovati zaradi napake ali napačnih podatkov. Past se lahko obravnava podobno kot prekinitev - sprožijo se enaki mehanizmi, npr. push vseh registrov na stack, izvajanje prekinitvenega servisnega podprograma, itd., le da jo sproži programska oprema namesto strojne. Primeri sprožitve pasti:

- deljenje z ničlo
- nepodprt ukaz
- ustavitvena točka (breakpoint)
- pisanje v zaščiten spomin

Večina CPE ima posebne ukaze za proženje pasti, ki so namenjeni razhroščevalniku. Na x86 arhitekturi je to ukaz `INT 3`. Sam ukaz `INT X` sproži programsko prekinitev, kjer `X` predstavlja prekinitiev, ki naj se sproži (0-255). Npr., ukaz `INT 0x21` (33 v desetiškem sistemu) bo PC nastavil na 34. vektor v prekinitveni tabeli. Ene izmed bolj znanih prekinitev na x86 so:

- INT 0x21 - MS-DOS API call, s katerim je glede na vrednosti v registrih mogoče izvajati IO operacije.
- INT 0x80 - Unix načeloma ne uporablja programskih prekinitev, razen 0x80, ki ga uporablja za sistemske klice. Tudi v tem primeru v registre vpišemo parametre, nato pa izvedemo ta ukaz.
- INT 3 - Namenjen razhroščevalnikom za nastavljanje prekinitvene točke. Ta ukaz je poseben, ker se zapiše **le z enim bajtom** - Njegov opcode je `0xCC`, čeprav se načeloma INT X zapiše z

dvema bajtoma, torej `0xCD 0x03`. Ker so nekateri ukazi na x86 lahko dolgi samo en bajt, s tem ob nastavitvi prekinitvene točke ne povozimo še drugih ukazov.

Izvedba prekinitvene točke

Po tem, ko smo prekinitveno točko nastavili in na tisto mesto postavili past, lahko CPE poženemo in počakamo, da CPE pride do te točke. Ko bo CPE do tja prišel, se bo past sprožila, pognal se bo ustrezen podprogram v operacijskem sistemu in proces razhroščevalnika bo dobil signal, da se je ciljni program ustavil. Razhroščevalnik nato:

1. Zamenja past (`INT 3`) z prvotnim ukazom na tistem mestu
2. PC zmanjša za 1, ker se je po izvedbi pasti premaknil za eno predaleč
3. Poda nadzor uporabniku, in ta lahko vidi vrednosti spremenljivk, klicni sklad, itd.
4. Če uporabnik ne odstrani prekinitvene točke na tem mestu, razhroščevalnik na to mesto past spet doda. *(Ker mora najprej izvesti še ta ukaz, jo najprej doda na naslednji ukaz, se s tem pri naslednjem ukazu ustavi, in jo zdaj nastavi na pravi ukaz, naslednjega pa spet nadomesti s provtno kodo in izvede).*

Primer nastavljanja in izvajanja prekinitvene točke s ptrace

V datoteki ptrace_setting_breakpoint_example.md

Pogojne prekinitvene točke (conditional breakpoints)

To so prekinitvene točke, ki se izvedejo le ob danem pogoju, npr. ko je indeks v zanki enak določeni vrednosti. Najbolj enostavno jih je implementirati tako, da razhroščevalnik uporablja navadne prekinitvene točke, nato pa preveri, če se pogoji ujemajo danim podatkom, in le v tem primeru poda nadzor uporabniku. Če uporabljamo oddaljen razhroščevalnik (npr. preko gdbserver) lahko prenos iz ciljne naprave na uporabnikovo napravo predstavlja veliko overheada, zato lahko pogoje preko gdb stuba preverimo kar na ciljni napravi, ali pa uporabimo interpreter, naložen kot deljen objekt.

Programske prekinitvene točke (software breakpoints)

Ta tip prekinitvenih točk, ki sem ga opisal, imenujemo programske prekinitvene točke (software breakpoints). Pri programskih prekinitvenih točkah praktično ni omejitve v številu točk (razen spomina, seveda), potrebujemo pa spremembe programske kode programa, kar je lahko nevarno in počasno.

Razhroščevalska strojna oprema (debug hardware)

Procesor lahko vsebuje dodatno logiko, ki je namenjena zgolj razhroščevanju. To lahko vključuje npr.

- razhroščevanje z ustavljanjem procesorja (halting mode debugging)
- enostavno izvajanje korak za korakom (single stepping)
- strojna podpora za prekinitvene točke in watchpointe

Taka strojna oprema je draga glede na prostor, ki je na voljo na siliciju - ker *boljše zmožnosti razhroščevanja* ni najlažje tržiti, se načrtovalcem procesorjem ne splača preveč prostora nameniti

takim komponentam, saj bi bil ta prostor lahko namenjen še večjim številom enot za hitrejše procesiranje. Zato je taka strojna oprema ponavadi omejena in lahko nastavimo npr. 4 prekinitven točke.

Strojne prekinitvene točke

Komparator, ki gleda programski števec in ga primerja z programsko vneseno vrednostjo. Ponavadi genererira napako pri prenosu (fetch exception), in pošlje SIGSEGV ali SIGTRAP. Do njih lahko dostopamo le s posebnimi ukazi, ni tako enostavno kot pri programskih.

Spominske prekinitvene točke (memory breakpoints, watchpoints)

To so točke, ki jih lahko nastavimo na določenem mestu v spominu, in se bo program ustavil ob dostopu do danega naslova. Izvedene so s komparatorjem, ki gleda na vodilo, na katerem je zapisan naslov v spominu. Lahko se sproži ob branju ali pisanju na spomin (ali oboje). Ponavadi generira data abort exception, in pošlje SIGSEGV ali SIGTRAP.

JTAG Debugger

Je eden izmed načinov, s katerim lahko razhroščejumo na vgrajenih sistemih. Za to poznamo več rešitev, ponavadi pa komunicirajo preko čim manjšega števila žic.

Razhroščevanje z ustavljanjem procesorja (halting mode debugging)

Tak tip razhroščevanja ustavi normalno izvajanje programa in procesorsko uro, še vedno pa tečejo zunanje naprave (RTC, DMA, ...). Na voljo imamo tudi kanal, preko katerega lahko v cevovod vstavimo ukaze in jih izvedemo, dostopamo pa lahko tudi do nekaterih posebnih sistemskih registrov. Do spomina lahko dostopamo z load in store ukazi, ki bodo šli v čez MMU in predpomnilnik, tako da bo dostop do spremenljivk vrnil pravilne vrednosti. Razhroščevalnik mora zato pravilno nadzorovati cevovod, pomnilniške banke, predpomnilnik... Vezje nam lahko omogoča tudi neposreden dostop do pomnilniškega vodila, vendar bo v tem primeru dostop do spremenljivk vrnil napačne vrednosti.

Informacije za razhroščevanje (debug information)

Če se želimo v poljubnem programskem jeziku ustaviti le na vrsticah, ki jih vidimo v izvorni kodi, ne pa za vsak ukaz v strojni kodi, mora razhroščevalnik vedeti, kako se vrstice v izvorni kodi mapirajo z izvorno kodo. Zato mu mora to prevajalnik nekako sporočiti. Eden izmed standardov za to je DWARF.

DWARF

Ime DWARF je povezano z datoteko ELF (Executable and Linkable Format), ki se uporablja za izvajanje in povezovanje. V DWARF lahko najdemo opise simbolov (funkcij, spremenljivk...). Primer je podan v datoteki dwarf_debug_info.md

Kljub dodatnim informacijam pa ima razhroščevalnik lahko še vedno težave, če je izhod prevajalnika preveč optimiziran - npr, če zaradi optimizacije kakšna funkcija manjka, prevajalnik

odstrani kakšno spremenljivko, c++ templates, pri katerih je več kot ena vrstica izvorne kode na en ukaz...

Klicni sklad (call stack)

Z uporabo frame pointerjev lahko ugotovimo, znotraj katerih funkcij poteka trenutni klic, in sestavimo ustrezen izpis. Pri ugotavljanju tega pa lahko pride do težav, če prevajalnik s skladom ne dela po standardnem načinu klicanja - calling conventionu ali pa zaradi optimizacije kakšno funkcijo kar odstrani (ker npr. vrne konstanto).

Povzetek

Razhroščevanje pride programerju zelo prav, vendar potrebuje tudi ustrezno podporo iz strani operacijskega sistema in strojne opreme. Razhroščevalnik lahko deluje veliko bolje, če mu prevajalnik poda ustrezne podatke. Težave lahko nastanejo tudi pri optimizacijah, ki precej otežijo delo.

Reference

- Jonathan B. Rosenberg. How Debuggers Work: Algorithms, Data Structures, and Architecture. John Wiley & Sons. ISBN 0-471-14966-7.
- Debuggers - Wikipedia
- How do debuggers (really) work - Video
- How debuggers work - Part 1
- How debuggers work - Part 2 - Breakpoints
- How debuggers work - Part 3 - Debugging information
- GDB The GNU Project Debugger
- GDB - Wikipedia
- man 2 ptrace
- INT (x86 instruction) - Wikipedia
- Writing a linux debugger