FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Razhroščevalnik GDB

Aljaž Šuštar

Ljubljana, 2021

1 Razhroščevalnik

1.1 Namen

Razhroščevalnik oziroma orodje za razhroščevanje je računalniški program za testiranje in razhroščevanje drugih (ciljnih) programov. Glavni namen razhroščevalnika je poganjanje ciljnega programa v nadzorovanem okolju, kar programerju omogoča nadzorovanje operacij v izvajanju ter spremljanju sprememb v računalniških virih (najpogosteje spremljanje pomnilnika, ki ga uporablja ciljni program). [1]

1.2 Glavne lastnosti razhroščevalnikov

Glavne funkcionalnosti, ki jih ponujajo razhroščevalniki, so nastavljanje točk zaustavitve (angl. breakpoint), izvajanje programa po korakih, spremljanje vrednosti spremenljivk med izvajanjem, pa tudi t.i. 'posnemi ter ponovno predvajaj' razhroščevanje (angl. record and replay debugging).

1.3 Uporabniški vmesnik

Glede na uporabniški vmesnik delimo razhroščevalnike predvsem na dva tipa: razhroščevalniki z grafičnim vmesnikom (angl. GUI - Graphical User Interface) ter razhroščevalniki, ki omogočajo interakcijo prek ukazne vrstice (angl. CLI - Command Line Interface). Razhroščevalniki z grafičnim vmesnikom so navadno vgrajeni v razvojno okolje (npr. PyCharm, IntelliJ ipd.), podpirajo pa lahko enega ali več razhroščevalnikov. Ne glede na uporabniški vmesnik so osnovne funkcionalnosti pri razhroščevalnikih enake.

2 Sistemski klic ptrace

2.1 Osnovno delovanje

Sistemski klic `ptrace` ponuja načine na katere lahko en proces (sledilec) opazuje in nadzoruje izvajanje drugega procesa (sledečemu) ter pregleduje njegov pomnilnik ter registre. Primarno se uporablja za razhroščevanje preko točk zaustavitve ter sledenje sistemskim klicem.[2]

Funkcija `ptrace` deluje tako, da iz glavnega programa preko jedra pošljemo signal programu, ki mu sledimo. Vsi signali, z izjemo enega, povzročijo, da se program, ki mu sledimo, ustavi, ne glede na to, kako ima implementiran rokovalnik za ta signal. Izjema je signal `SIGKILL`, ki naredi to, kar je njegov namen, torej nemudoma ustavi program.

2.2 Funkcija ptrace()

Podpis funkcije `ptrace` je sledeč:

`long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);`[2]

Parameter `enum __ptrace_request request` predstavlja številko zahteve, za katero želimo, da jo `ptrace` izvede. Možne vrednosti parametra so naštete v datoteki `ptrace.h`, na UNIX sistemih pa tudi z ukazom `man ptrace`.
Parametru `pid_t pid` podamo id procesa, ki mu želimo slediti.
Na naslov, na keterega kaže kazalec `void* addr`, `ptrace` zapisuje pridobljene naslove, lahko pa klicu naslov tudi podamo, odvisno od zahteve, ki je želimo izvesti.
V parameter `void *data` `ptrace` zapisuje pridobljene podatke (naprimer podatke o registrih), uporablja pa se tudi za podajanje podatkov sistemskemu klicu.

2.3 Primeri pogostih klicev ptrace

Nekaj primerov uporabe sistemskega klica `ptrace`:

`ptrace(PTRACE_SETOPTIONS, pid, NULL, options);`
Z zahtevo `PTRACE_SETOPTIONS` klicu sporočimo, da bi radi nastavili nastavitve za nadaljnje delo s `ptrace`. Nastavitve najdemo v datoteki `ptrace-shared.h`, oziroma z ukazom `man ptrace` na UNIX sistemih.

`ptrace(PTRACE_TRACEME);`
S tem klicem omogočimo sledenje procesu. Funkcijo pokličemo v procesu, ki mu želimo slediti. Ostale parametre `ptrace` ignorira. [2]

`ptrace(PTRACE_ATTACH, pid);`
Omogoči sledenje programu z id procesa `pid`. Razlika med tem klicem in prejšnjim je, da ta klic izvedemo iz sledilca (torej v primeru, da uporabljamo `fork`, iz starša). Sledečemu programu `ptrace` pošlje signal `SIGSTOP` a nimamo garancije, da se bo le-ta ustavil pred koncem klica `ptrace`. Zato navadno po klicu `ptrace` počakamo na ustavitev sledečega procesa (naprimer z `waitpid`). [2]

`ptrace(PTRACE_SYSCALL, pid);`
Zaustavi program, ki mu sledimo, ob vhodu oziroma izhodu iz sistemskega klica. Ciljni program se ob prejemu signala `SIGTRAP` zaustavi, kar razhroščevalniku omogoča vpogled v sistemski klic ter njegove parametre in izhodni status.[2]

`ptrace(PTRACE_CONT, pid, NULL, data);`
Ponovno zažene ustavljeni proces. Parameter `data` lahko uporabimo, da ciljnemu programu dostavimo številko signala, če pa ga nastavimo na 0, se ciljnemu programu ne pošlje nič. S tem parametrom torej poskrbimo, da se signal (ne) dostavi ciljnemu programu. [2]

3 Razhroščevalnik GDB

3.1 Splošno

Razhroščevalnik GDB je razvila organizacija GNU Project. Podpira razhroščevanje različnih jezikov, kot naprimer C/C++, Fortran, Assembly itd. Z razhroščevalnikom upravljamo preko ukazne vrstice, zanj pa so na voljo tudi različni uporabniški vmesniki (nekaj jih je naštetih na povezavi). Omogoča razhroščevanje na isti napravi, v simulatorju ter na oddaljeni napravi [3]. Slednje je predvsem uporabno za razhroščevanje na vgrajenih sistemih, kjer se opazovani program izvaja na ciljni napravi, GDB pa teče na napravi, na kateri razvijamo programsko opremo. GDB podpira UNIX, Windows ter Mac OS X operacijske sisteme [3].

3.2 Notranje delovanje GDB

V naslednjih podpoglavjih sledi opis delovanja GDB. Dotaknil se bom uporabe sistemskega klica `ptrace` v GDB in nekaj uprorabnih ukazov za delo z lupino GDB. Potrebno pa se je zavedati, da v svoji zasnovi GDB za večino svojih operacij uporablja sistemski klic `ptrace`, ki je bil opisan v poglavju 2.

3.2.1 Uporaba sistemskega klica ptrace v GDB

Ko v ukazni lupini GDB poženemo ciljni program za razhroščevanje, GDB izvede sistemski klic `ptrace(PTRACE_ATTACH, pid)`. GDB nato kliče `ptrace` glede na ukaz uporabnika. Naprimer, če uporabnik poda ukaz `s` (izvedi naslednji ukaz ter ustavi izvajanje), potem GDB pokliče sistemski klic s parametrom `PTRACE_SINGLESTEP`, torej `ptrace(PTRACE_SINGLESTEP, pid)`. Če uporabnik poda ukaz `catch syscall`, pa GDB izvede klic `ptrace(PTRACE_SYSCALL, pid)`. [4]

3.2.2 Postavljanje zaustavitvenih točk (breakpoint)

Predpogoj za nastavitev točke zaustavitve (angl. breakpoint) je, da prevajalnik vključi dovolj informacij v izvršljivo datoteko, da razhroščevalnik lahko poveže izvorno kodo s strojnim ukazom. Če uporabljamo prevajalnik GCC, uporabimo stikalo `-g`. Z ukazom `break <ime funkcije/št. vrstice/naslov>` v ukazni lupini GDB razhrošče valniku povemo, kam naj postavi točko zaustavitve. Razhroščevalnik nato na to mesto vstavi poseben ukaz (npr. na arhitekturi x86 prekinitev št. 3 - `BREAKPOINT`). Seveda mora ob tem originalni ukaz ohraniti ter ga po prekinitvi ponolno obnoviti ter izvesti.

3.2.3 Pridobivanje uporabnih podatkov o programu

Ko se program zaustavi, bi radi videli določene podatke, ki so na voljo. To so naprimer vrednosti v registrih, vrednosti spremenljivk in podobno. GDB ima za te namene nekaj ukazov:

- `info registers <ime registra>` pridobi podatke o registrih oziroma registru, če podamo ime registra, kot naprimer `info registers eax`,
- `print <format> <ime spremenljivke>` izpiše trenutno vrednost podane spremenljivke v podanem formatu. Naprimer, ukaz `print /a argv` bi izpisal naslov tabele `argv`, ki jo v programskem jeziku C dobimo kot parameter funkcije `main`.

4 Primer razhroščevalnika

Za lažje razumevanje delovanja razhroščevalnika lahko sprogramiramo enostaven program. S pomočjo klica `ptrace` sledimo izvajanju programa ter spremembe v registrih. Za programiranje sem uporabil programski jezik C, izvorna koda pa je na voljo na povezavi: GitHub repozitorij debugger-example.

4.1 Struktura projekta

V direktoriju `src` se nahaja vsa izvorna koda programa. Glavni del se nahaja v datoteki `src/debugger.c` v kateri je tudi funkcija `main`. V direktoriju `test` se nahaja testni program `palindrome.c`, ki ga bom uporabil za testiranje razhroščevalnika. Program na 2 načina preveri, ali je podani niz palindrom - z rekurzijo ter iteracijo s skladom. Zadnji je še direktorij `ptrace`, v katerem so funkcije za delo s sistemskim klicem `ptrace`.

4.2 Pregled vzorčnega programa

4.2.1 Funkcije za delo s ptrace

Funckije za delo s `ptrace` se nahajajo v datoteki `src/ptrace/ptrace.c`. Za vsako izmed uporabljenih nastavitev obstaja svoja funkcija, v kateri je poskrbljeno tudi za osnovno obravnavo napak. Izjemi sta funkciji `print_registers(Tracee* process)` ter `ptrace_set_options(Tracee* tracee, unsigned int options)`. Prva izpiše registre procesa, druga pa nastavi `ptrace` glede na nastavitve, podane v parametru `options`.

4.2.2 Nastavitve ptrace

Pred vstopom v zanko najprej počakamo ciljni program, da sproži signal `SIGSTOP`, ki smo ga sprožili po klicu `fork()`. Nato nastavimo delovanje s klicem `ptrace(PTRACE_SETOPTIONS, process->pid, NULL, PT_OPTIONS)`. Spremenljivka `process` je tipa `Tracee`, ki hrani `pid` ciljnega procesa. `PTRACE_SETOPTIONS` pa je globalna spremenljivka, ki hrani nastavitve, ki so potrebne za izvajanje `ptrace`. Te so:

`PTRACE_O_TRACEEXIT` - Ustavi ciljni program ob klicu `exit()`. Ciljni program je

ustavljen pred ustavitvijo procesa, zato je kontekst izvajanja (registri ter izhodni status) še vedno na voljo. Kljub temu, pa na tej točki izhoda iz podprocesa ne moremo preprečiti. [2]

`PTRACE_O_TRACESYSGOOD` - Ob sprejemu pasti, sproženih iz sistemskih klicev, nastavi 7. bit v številki signala na 1 (torej `SIGTRAP|0x80`). To omogoča razlikovanje med pastjo iz sistemskega klica ter navadno pastjo.

4.2.3 Glavni program

V datoteki `src/debugger.c` se nahaja glavni program s funkcijo `main`. V njej najprej preberemo argumente, ki so bili podani programu - to je ciljni program ter njegovi argumenti. Nato izvedemo sistemski klic `fork`. V otroku s `ptrace_traceme()` staršu sporočimo, da bomo programu sledili. Program se ne zaustavi, zato v naslednjem koraku pošljemo prekinitev `SIGSTOP`, da se program ustavi, saj si izvajanje ciljnega programa želimo začeti iz starša. Nato izvedemo še klic `execve(argv[1], args, NULL)`, ki začne izvajanje podprograma.
V staršu medtem inicializiramo podatkovno strukturo `Tracee`, ki vsebuje `pid` procesa, v katerem se izvaja ciljni program ter `struct user_regs_struct*`, ki je kazalec na strukturo, v katero se zapišejo vrednosti registrov ob klicu `ptrace(PTRACE_GETREGS, ...)`. Nato vstopimo v glavno zanko programa.

4.2.4 Glavna zanka

Pred vstopom v glavno zanko najprej nastavimo delovanje `ptrace` na način, podan v spremenljivki `PT_OPTIONS`. Nato v neskončni zanki beremo ukaze uporabnika prek ukazne vrstice. Ko uporabnik poda ukaz s pomožno funkcijo `resolve_request` ugotovimo željo uporabnika ter pokličemo `ptrace` z ustreznimi parametri. Z dvema pogojnima stavkoma preverimo pogoja za zaustavitev. Prvi je, če se je ciljni program zaustavil. Pomagamo si z makrom `WIFEXITED(status)`, ki pove, ali se program še izvaja. Če se ne, s pomočjo `WEXITSTATUS` pridobimo izhodni status, prekinemo zanko ter program zaključimo.

Drugi izhodni pogoj je signal za zaustavitev programa. Z `WIFSIGNALED` preverimo, ali se je program končal, ko je prejel signal. Če je pogoj izpoljen, z `WTERMSIG` pridobimo številko signala, ki je povzročil zaustavitev ter prekinemo zanko.

Nazadnje pa še preverimo, če se je program ustavil. Če je pogoj izpoljnjen ter velja, da je 7. bit številke signala postavljen (`SIGTRAP|0x80`), potem smo znotraj sistemskega klica. V nasprotnem primeru pa smo prejeli le navaden signal za zaustavitev.

5 Zaključek

Razhroščevalnik GDB je kompleksno, a hkrati relativno enostavno za uporabo, če smo vešči dela z ukazno vrstico. Ciljnemu programu sledi z uporabo sistemskih klicev, primarno s sistemskim klicem `ptrace`, ki jedru operacijskega sistema pošilja signale in pasti ter se odziva na pasti ter signale prejete iz jedra. Ta, v principu preprost mehanizem, pa omogoča razvoj močnih ter uporabnih orodij, kot je GDB.

References

[1] (2021, oct) Debugger. [Online]. Available: https://en.wikipedia.org/wiki/Debugger

[2] (2021, oct) ptrace(2) — linux manual page. [Online]. Available: https://man7.org/linux/man-pages/man2/ptrace.2.html

[3] (2021, oct) Gdb: The gnu project debugger. [Online]. Available: https://www.gnu.org/software/gdb/

[4] (2021, jul) binutils-gdb. [Online]. Available: https://github.com/bminor/binutils-gdb