Tačka prekida: GNU debager

Seminarski rad u okviru kursa
Metodologija stručnog i naučnog rada
Matematički fakultet

Kristina Pantelić, 91/2016, kristinapantelic@gmail.com
Ivana Cvetkoski, 65/2016, ivana.cvetkoski@gmail.com
Bojana Ristanović, 45/2016, bojanaristanovic97@gmail.com
Nikola Stamenić, 177/2016, nikola.stamenic@hotmail.com

20. april 2020

Sažetak

Danas je teško zamisliti da bismo mogli napraviti bilo koji značajan programerski projekat bez korišćenja naprednih alata za pronalaženje grešaka. U ovom radu čitalac će se upoznati sa terminom **debagovanje**, specifičnostima GDB-a, načinom upravljanja i njegovim korišćenjem u razvojnim okruženjima. Takođe, napravljena je paralela između GDB-a i drugih popularnih debagera.

Sadržaj

1 Uvod

Pronalaženje grešaka je složen zadatak, programeri troše dosta vremena na njihovo otklanjanje i zato je važno da budu upoznati sa alatima koji će im pomoći da to vreme skrate. GNU debager (GDB) je jedan od najznačajnijih besplatnih softvera koji služi da analizira kompjuterske programe. Jedna od mnogih stvari koje nam ovaj debager omogućava da uradimo, a koje će biti opisane u radu, jeste da vidimo šta se dešava "unutar" programa u toku izvršavanja tj. da imamo uvid u stanja izvršavanja programa. Ovaj debager zauzima prvo mesto na listi najboljih debagera koji se koriste na Linux operativnom sistemu[12]. Na samom početku ovog rada upoznaćemo se sa bagovima i debagovanjem, a u daljem tekstu ćemo detaljnije opisati specifičnosti GDB-a, kako i u kojim okruženjima se koristi i kakav je GDB debager u poređenju sa drugim debagerima.

2 Bag

Greške u programima se mogu podeliti u dve grupe, sintaksne i semantičke. Kada je program sintaksno ispravan, to još uvek ne znači da on i radi ono za šta je napisan. U tom slučaju program sadrži greške logičke prirode tj. programer je tokom pisanja programa pogrešno protumačio značenje (semantiku) pojedinih naredbi koje je napisao. Otkrivanje i ispravljanje semantičkih grešaka je daleko teže od otkrivanja i ispravljanja sintaksnih grešaka. Popularni naziv za semantičku grešku u programu je bag (*eng.* bug).

Propust (greška, bag) u razvoju softvera je sve ono što stvara probleme u funkcionisanju softvera kao završnog proizvoda. Bag predstavlja sve ono što ima za posledicu da se softver ne ponaša u skladu sa specifikacijom ili očekivanjem korisnika[20].

Jedna od uobičajenih klasifikacija bagova je prema načinu ispoljavanja[20]:

1. Nekonzistentnosti u korisničkom interfejsu
2. Neispunjena očekivanja
3. Slabe performanse
4. Padovi sistema (programa) ili oštećenja podataka

Razlozi za greške uglavnom spadaju u sledeće kategorije procesa[20]:

1. Kratki ili nemogući rokovi
2. Pristup „Prvo kodiraj, razmišljaj kasnije"
3. Pogrešno shvaćeni zahtevi
4. Neznanje inženjera ili nepravilna obuka
5. Nedostatak posvećenosti kvalitetu

3 Debagovanje

Debagovanje (*eng.* debugging) je proces pronalaženja i otklanjanja grešaka ili nedostataka koji sprečavaju tačnu operaciju računarskog softvera ili sistema. Greška se obično otkriva, jer se program neočekivano ponaša. Da bi se pronašao uzrok greške, ključno je objasniti zašto dolazi

do takvog ponašanja. Debagovanje ima tendenciju da bude teže kada su različiti podsistemi čvrsto povezani, pošto promene u jednom mogu da prouzrokuju nastanak bagova u drugom[15].

"Otklanjanje grešaka je dvostruko teže nego pisanje kôda. Stoga, ako napišete kôd što je pametnije moguće, po definiciji, niste dovoljno pametni da ga ispravite."

-Brian V. Kernighan

Debagovanje se sastoji iz četiri koraka[20]:

1. Uočavanje da postoji greška
2. Razumevanje greške
3. Lociranje greške
4. Ispravljanje greške

Obično je najteži deo posla ispravno razumevanje i tačno lociranje greške. Jednom kada se greška locira, njeno ispravljanje u većini slučajeva ne predstavlja poseban problem.

Testiranje je metod koji smanjuje verovatnoću nastajanja grešaka. Otklanjanje grešaka se razlikuje od testiranja. Debagovanje počinje nakon što je u softveru utvrđena greška, dok se testiranje koristi da bi se osiguralo da je program tačan[20].

Debagovanje je jedan od najkreativnijih aspekata programiranja, zahteva od programera iskustvo, inteligenciju, razmišljanje "van kutije", sagledavanje problema sa različitih strana, logičko zaključivanje, odlučnost; ali može biti i jedan od najzahtevnijih aspekata programiranja. Iskustvo i veština debagovanja programera su bitni faktori u procesu debagovanja. Međutim, težina debagovanja softvera najviše varira zbog složenosti sistema, ali takođe, u određenoj meri, zavisi i od programskog jezika koji se koristi, kao i dostupnih alata poput debagera[20].

3.1 Vrste debagovanja

Klasično debagovanje se zasniva na tehnici praćenja kôda korišćenjem funkcije za ispis, tako što ispisujemo vrednosti promenljivih. Pre svega, ovaj način podrazumeva konstantno pozivanje funkcije za ispis, rekompajliranje i pokretanje programa, analizu dobijenog izlaza i konačno uklanjanje poziva funkcije za ispis kada uspemo da popravimo bag. Navedeni koraci se ponavljaju svaki put kada otkrijemo novi bag. Ovaj način debagovanja oduzima previše vremena, stvara umor i najvažnije, odvlači pažnju od pravog zadatka[17].

Sistem za interaktivno debagovanje zahteva mogućnost kontrole toka izvršavanja programa: postavljanjem tačaka prekida (engl. breakpoints) izvršavanje programa se pauzira, korišćenjem komandi debagera analizira se progres programa, a zatim se ponovo nastavlja sa izvršavanjem programa. Moguće je postaviti i uslovne izraze koji se proveravaju tokom izvršavanja programa i ukoliko se uslovi ispune, izvršavanje programa se zaustavlja i vrše se analize.

Udaljeno (daljinsko) debagovanje je postupak debagovanja programa koji se izvršava na sistemu udaljenom od debagera. Da bi proces započeo, debager mora da se poveže sa udaljenim sistemom preko mreže. Tada debager može da kontroliše izvršavanje programa na udaljenom sistemu i da sakuplja informacije o njegovom stanju. Ovo je moguće ukoliko

je udaljeni sistem iste arhitekture kao i arhitektura na kojoj se debager pokreće ili ukoliko debager poseduje podršku za arhitekturu udaljenog sistema[16]. Udaljeno debagovanje se koristi za debagovanje ugradnih uređaja, uređaja na kojima se neposredno debagovanje ne može izvršiti ili debagovanje jezgra operativnog sistema[21].

Post-mortem debagovanje je postupak debagovanja programa nakon njegovog prekida. Tačan momenat prekida procesa može se ustanoviti automatski od strane sistema (npr. kada je proces završen zbog nekog odstupanja), preko instrukcija napisanih od strane programera ili eksplicitno od strane korisnika. Za ovu vrstu debagovanja često se koriste datoteke jezgra koje sistem generiše[16].

3.2 Debager

Debager ili alat za debagovanje je računarski program koji se koristi da testira i debaguje druge programe ("metaprogram"), dajući mogućnost da se nezavisno pokrene izabrana grupa instrukcija (simulator grupe instrukcija). Simulator grupe instrukcija se zaustavlja na određenim tačkama programa ukoliko su određeni uslovi ispunjeni. Obično se programi koji se prevode u režimu za debagovanje sporije izvršavaju nego kada se isti program izvršava u normalnom režimu rada čak i ako je u pitanju rad na istom procesoru[1]. Kada program usled baga ili netačnog podatka ne može da nastavi normalno sa radom, napredniji debager pokazuje lokaciju problema u originalnom kôdu. Takođe, debager nam omogućava da postavimo tačke posmatranja koje nam mogu reći u kom trenutku tokom izvođenja programa vrednost određene promenljive postaje sumnjiva; omogućava nam da pratimo izvršavanje programa, da ga zaustavimo, restartujemo, postavimo mesta prekida i da izmenimo vrednosti u memoriji.

4 GNU debager (GDB)

GNU debager (*eng.* GNU Debugger), kog često srećemo pod nazivom GDB, je alat koji služi za pronalaženje i otklanjanje grešaka tj. debagovanje. Originalno ga je razvijao Ričard Stalman 1986. godine, kao i mnoge druge programe za GNU sistem[21]. Danas održavanjem upravlja GDB upravni odbor koga je formirala Fondacija slobodnog softvera (*eng.* Free Software Foundation). GDB je pisan na programskom jeziku C i standardni je debager za GNU operativni sistem. Međutim, njegova upotreba nije isključivo ograničena na GNU operativni sistem. To je prenosivi debager koji radi na mnogim Uniksolikim (eng. *unixlike*) operativnim sistemima ali i na Microsoft Windows operativnim sistemima. Koristi se za mnoge programske jezike, uključujući Adu, C, C++, Objective-C, Free Pascal, Fortran, Javu. Poslednja realizovana verzija alata GDB u vreme pisanje ovog rada je 9.1[11].

Jedna od specifičnosti GNU debagera, pored svoje standardne namene, jeste da omogućava i pronalaženje, analiziranje i otklanjanje grešaka u programima koji se izvršavaju na računarima drugih arhitektura (udaljeno debagovanje)[22].

4.1 Osnovne operacije GDB-a

GDB komanda **run** pokreće izvršavanje programa od prve linije izvornog kôda. Izvršavanje programa teče do trenutka dok ga GDB ne pauzira.

Razlog pauziranja može biti na programskoj ili programerskoj strani. Pod programskom stranom podrazumeva se pauziranje izvršavanja programa zbog greške nastale u izvršavanju, a pod programerskom pauziranje na mestima specifikovanim od strane programera kako bi se mogle ispitati vrednosti promenljivih u cilju otkrivanja grešaka.

4.1.1 Metode upravljanja debagovanjem u GDB-u

Da bi se GDB koristio nad određenim programom, neophodno je program kompajlirati na određen način. Za programske jezike C i C++ to je zastavicom *-g*, odnosno naredbom *gcc program.c -g -o program* za C, dok je za C++ naredba *g++ program.cpp -g -o program*.

Jedna veoma korisna opcija je navođenje *--tui* zastavice pri pokretanju samog debagera. Korisna je iz razloga što korisnik dobija izvorni kod programa koji debaguje, što dalje olakšava posao postavljanja tačaka prekida i eventualno uočavanje nekih od grešaka[19].
Neke od metoda kojima programer može upravljati GDB debagerom[17]:

1. Tačke zaustavljanja (*eng.* breakpoints)
   Komandom **break**, uz koju se navodi broj linije na kojoj GDB treba da pauzira izvršavanje, uvodi se nova tačka zaustavljanja. Cilj zaustavljanja izvršavanja programa je ispitivanje vrednosti promenljivih u programu kako bi se otkrila greška.
2. Pojedinačni koraci (*eng.* single-stepping)
   GDB komanda **next** omogućava izvršavanje jedne po jedne linije programa. Nakon jedne izvršene linije kôda, GDB pravi pauzu u izvršavanju programa dok se ponovo ne pozove komanda **next**. Upotreba komande **step** je slična, razlika je u tome što ukoliko je naredna naredba za izvršavanje funkcija, komandom **step** se ulazi u funkciju i korak po korak prolazi kroz nju, a komanda **next** izvršava čitavu funkciju u jednom koraku, vraća njenu povratnu vrednost i zaustavlja se na liniji u kôdu nakon izvršene funkcije.
3. Nastavi rad (*eng.* resume operation)
   GDB komandom **continue** nastavlja se izvršavanje programa do naredne tačke zaustavljanja.
4. Privremene tačke zaustavljanja (*eng.* temporary breakpoints)
   GDB komandom **tbreak** postavlja se tačka zaustavljanja u programu koja će važiti sve do njenog prvog dostizanja u izvršavanju programa. Nakon njenog prvog dostizanja, ta tačka zaustavljanja prestaje da važi.

4.1.2 Tačke zaustavljanja

Postoje tri razloga zbog kojih GDB može pauzirati izvršavanje programa[17]:

1. Tačka zaustavljanja (*eng.* breakpoint)
   GDB pauzira izvršavanje programa kada se stigne do naznačenog mesta u programu.
2. Tačka nadgledanja (*eng.* watchpoint)
   GDB pauzira izvršavanje programa kada se promeni vrednost memorijske lokacije koju programer želi da prati.
3. Tačka hvatanja (*eng.* catchpoint)
   GDB pauzira izvršavanje programa kada se određeni događaj desi.

U dokumentaciji ova tri mehanizma se zajedničkim imenom nazivaju tačke zaustavljanja.

4.1.3 Kretanje kroz stek pozive naviše i naniže

Podaci o izvršavanju poziva funkcije smešteni su u stek frejmu. Frejm sadrži vrednosti lokalnih promenljivih, vrednosti parametara funkcije kao i memorisanu lokaciju u programu odakle je izvršen poziv funkcije. Svaki put kada se pozove funkcija, stvara se novi stek frejm i postavlja se na sistemski stek. Na vrhu steka nalazi se funkcija koja se trenutno izvršava, a nakon završetka funkcije njen stek frejm se skida. Postoje funkcionalnosti uz pomoć kojih možemo da se "šetamo" kroz stek pozive. To uspevamo narednim komandama:
Pozivanjem GDB komande **frame**, okviri na steku se numerišu od nule, počevši od vrha steka. GDB komanda **up** vodi do narednog roditeljskog stek frejma, dok komanda **down** vodi u suprotnom smeru. Navedene operacije mogu biti veoma korisne, jer vrednosti lokalnih promenljivih u nekim od ranijih poziva mogu dati rešenje o tome šta je tačno izazvalo bag. GDB komanda **backtrace** pokazuje ceo stek tj. celu kolekciju stek frejmova koja trenutno postoji na steku[17].
Jedan primer ispisa greške, kao i stek poziva se može naći u 1.

```
(gdb) run
Starting program: /home/user/Desktop/MSNR/a.out

Program received signal SIGSEGV, Segmentation fault.
0x0000555555555918 in f () at msnr.cpp:18
18          cout << vec->at(i);

(gdb) backtrace
#0  0x0000555555555918 in f () at msnr.cpp:18
#1  0x00005555555559be in main () at msnr.cpp:26
```

Listing 1: Primer ispisa greške u konzoli

Neke od najbitnijih komandi GDB-a date su u dodatku (tabela 3), dok se više informacija može naći u literaturi [2], kao i u MAN stranama Linux operativnih sistema.

4.1.4 Udaljeno debagovanje

GDB pruža mogućnost udaljenog debagovanja. Jedan način uspostavljanja komunikacije za debagovanje udaljenog uređaja je kreiranje udaljenog posrednika (podrške) koji je specifičan za konkretnu arhitekturu računara koji se debaguje. Udaljen posrednik je programski kôd koji se izvršava na udaljenom uređaju i omogućava komunikaciju sa GDB-om. Specifičnost ovog načina komunikacije je neophodnost pravljenja udaljenog posrednika pri svakom debagovanju[21].
Opisani koncept koristi KGDB za debagovanje Linux jezgra na nivou izvornog kôda. Velika prednost KGDB-a je u tome što programeri koji razvijaju jezgro mogu debagovati jezgro na sličan način kao što se debaguje bilo koja druga programska aplikacija. Moguće je postavljati tačke zaustavljanja u kôdu jezgra, prolaziti korak po korak kroz kôd, ispitivati vrednosti promenljivih[5].

Alternativno, može se koristiti GNU GDB server (*eng.* gdbserver) za udaljeno debagovanje programa bez potrebe da se bilo šta menja na obe strane komunikacije. GDB server nije u potpunosti zamena za udaljene posrednike, jer nameće ograničenje da operativni sistemi klijenta i servera moraju biti isti. Način debagovanja GDB serverom sastoji se od klijentske strane (korisnički računar sa koga se debagovanje izvršava ka udaljenom računaru) koja traži usluge i serverske strane koja usluge i resurse nudi[22].

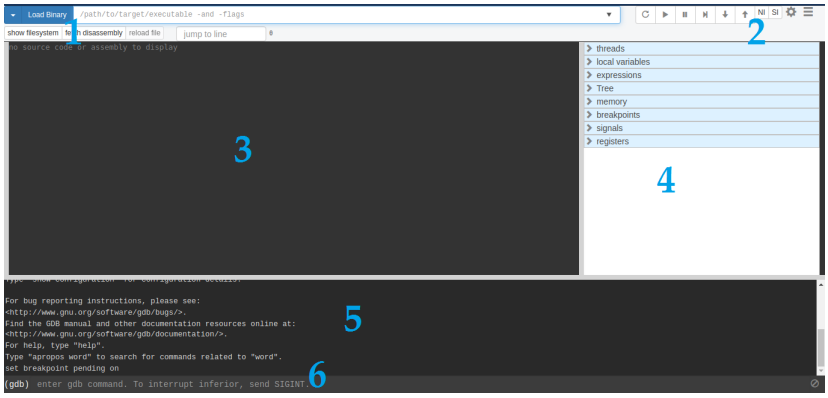
5 Upotreba GDB-a kroz grafički pristup

GDB je konzolni alat (pokreće se iz konzolne linije). Međutim, zbog popularnosti grafičkih korisničkih interfejsa (*eng.* GUI) razvijen je veliki broj GUI zasnovanih (*eng.* GUI-based) debagera koji rade pod Unix sistemom. Većina njih su grafički interfejsi za debagere. Jedan od najpoznatijih grafičkih interfejsa za debagere je DDD (*eng.* Data Display Debugger) [17]. DDD podržava grafički prikaz za više debagera, među kojima je GDB, a pored njega i DBX, WDB, Ladebug, JDB, XDB. Sam DDD se više ne razvija, od 2011-e, zaključno sa verzijom 3.3.12. [4] Pored DDD-a, KDbg je još jedan vid grafičkog okruženja za GDB koji se više ne razvija. KDbg je bio namenjen Linux operativnim sistemima sa KDE grafičkim okruženjem. Jedan od trenutno aktuelnih grafičkih interfejsa jeste gdbgui, o kojem će više reči biti u poglavlju 5.1.

Sa druge strane, danas je sve veća upotreba integrisanih razvojnih okruženja (*eng.* IDE), koji predstavljaju više od alata za debagovanje. IDE obuhvata editor, alat za izgradnju kôda, debager i druga razvojna pomagala. Ideja je u osnovi pružiti programsko okruženje u kojem se uređivanje kôda i izvršavanje odvijaju istovremeno - unutar okruženja za uređivanje kôda, a pomoću alata za praćenje stanja promenljivih[18]. Jedno od najpoznatijih IDE okruženja koje koristi GDB je Eclipse sistem, a pored njega GDB je prisutan i u CLion-u, Qt Creator-u, Code::Blocks-u[11].

5.1 Grafičko okruženje gdbgui

Alat **gdbgui** je grafičko okruženje za GDB, koji radi na nivou internet pretraživača. Program se pokreće naredbom *gdbgui* u terminalu, dok se za više informacija o samim komandama programa može dobiti naredbom *gdbgui --help*[3]. Samo korišćenje alata može biti identično kao i iz konzole, s obzirom da program sadrži terminal u sebi za ispis koji GDB inače vrši. Ono što olakšava posao i što je prevashodno razlog postojanja gdbgui-a, je lakše upravljanje tačkama prekida, lakši pregled informacija od značaja, kao i prikaz samog izvornog kôda, koji se kod konzolnog korišćenja dobija zastavicom *--tui*.



Slika 1: gdbgui: 1) Load Binary 2) Dugmići za kontrolu 3) Kôd 4) Informacije o programu u tačkama prekida 5) Konzola 6) Promt

Pojednostavljen prikaz gdbgui-a dat je slikom 1 uz smernice za šta koji deo služi. Duga je lista mogućnosti koje poseduje alat gdbgui, a neke najznačajnije su sledeće[8]:

1. Debagovanje različitih programa u različitim karticama (eng. *tab*)
2. Postavljanje/uklanjanje tačaka prekida
3. Pregled steka ili niti
4. Prelazak sa jednog stek okvira u drugi, kao i prelazak iz jedne niti u drugu
5. Intuitivno istraživanje lokalnih promenljivih pri pauziranom izvršavanju programa
6. Prelazak preko promenljive u kôdu za dobijanje njene vrednosti
7. Izračunavanje proizvoljnih izraza i plotovanje njihovih vrednosti kroz vreme
8. Istraživanje memorije u heksadekadnoj/karakterskoj formi
9. Pregled svih registara
10. Istraživanje izvornog kôda sa mogućnošću prelaska na određenu liniju

5.2 GDB u Qt Creator razvojnom okruženju

Qt Creator je prenosivo radno okruženje koje je pravljeno prevashodno za Qt radni okvir (eng. *framework*). Kako je i sam Qt prenosiv kako među različitim vidovima arhitektura, tako i među samim sistemima, Qt Creator mora da podržava različite prevodioce i debagere koje ti prevodioci koriste. GNU debager je podrazumevani debager ukoliko se koristi GCC prevodilac na Linux-u, Unix-u, Windows-u u kombinaciji sa MinGW-om, dok je na macOS-u eksperimentalne prirode[10].

Sam Qt Creator podržava neki vid statičke analize kôda, tako da u toku kreiranja kôda programer može da uvidi neke moguće greške, kao što su nekompatibilnost tipova koji se koriste, izlaženje izvan granica niza (vektora) i sličnih koje su uočljive u tom momentu. Nažalost, dosta grešaka ostaje neprimećeno. Tu nam pomaže debager, u ovom slučaju GDB, čija će upotreba na nekom jednostavnijem nivou biti prikazana u dodatku B. Qt Creator se može koristiti u **Debug** režimu kako bi se ispitivalo stanje aplikacije u toku njenog izvršavanja, dok se sa samim debagerom može interagovati na više načina, između ostalog[10]:

1. Prolaskom kroz program liniju po liniju ili instrukciju po instrukciju
2. Zaustavljanjem tekućeg programa
3. Postavljanjem tačaka prekida
4. Ispitivanjem sadržaja steka
5. Ispitivanjem i menjanjem sadržaja lokalnih i globalnih promenljivih
6. Ispitivanjem i menjanjem registara i sadržaja memorije tekućeg programa
7. Ispitivanjem spiska učitanih deljenih biblioteka
8. Isključivanjem delova kôda

6 Poređenje sa drugim popularnim debagerima

Postoji mnogo debagera koji se mogu koristiti kako na različitim operativnim sistemima tako i za različite programske jezike. Neke od stvari na koje treba obratiti pažnju prilikom izbora debagera[7]:

1. Debagovanje u fazi razvoja
2. Efikasno praćenje toka vrednosti
3. Debagovanje grešaka u višenitnim procesima
4. Da li program za otklanjanje grešaka brzo i lako šalje detaljne informacije o otkrivenim greškama

6.1 GDB i LLDB

LLDB je program za otklanjanje grešaka koji se koristi u LLVM (*eng.* Low Level Virtual Machine) projektima. To je besplatan softver sa otvorenim kôdom (*eng.* open source) pod licencom Univerziteta Ilionis / NCA Open Source Licence. Napravljen je kao skup komponenata za višekratnu upotrebu[6].

LLDB je napravljen od strane LLVM razvojne grupe, dok je GDB realizacija GNU projekta. Druga razlika je u tome što je LLDB pisan u C++-u, a GDB u C-u. Što se operativnih sistema tiče, LLDB radi na macOS i386 i x86-64, Linux-u, FreeBSD-u, MC Windows-u, dok je GDB prenosiv program za otklanjanje grešaka koji radi na mnogim Unix sistemima i MC Windows-u. Jedna od glavnih razlika između ova dva programa predstavljaju programski jezici u kojima se koriste. LLDB može biti korišćen da otkloni greške u C, Objective C i C++ programima, dok se GDB može koristiti za jezike Ada, C, C++, Objective C, Pascal, FORTRAN i Go.

Iako je veliki deo komandi sličan, postoje razlike u nekim od najčešće korišćenih[6]. Razlike su date u tabeli 1:

	GDB	LLDB
Pokreni program	run	process launch
Prikaži vrednosti u registrima	info registers	registers read
Prikaži tačke prekida	info breakpoints	breakpoint list
Obriši sve tačke prekida	delete	breakpoint delete
Obriši tačku prekida označenu brojem	delete (broj)	breakpoint delete (broj)
Prikaži vrednosti svih lokalnih promenljivih	info locals	frame variable
Prikaži vrednosti lokalne promenljive prom	p prom	frame variable prom
Prikaži stanje steka za trenutnu nit	bt	thread backtrace
Izlistaj glavne izvršne biblioteke i sve zavisne	info shared	image list

Tabela 1: Razlike između GDB i LLDB komandi

6.2 GDB i VALGRIND

Valgrind je programski alat za pronalaženje grešaka u memoriji, otkrivanje curenja memorije i profajliranje. To je besplatan softver, otvorenog kôda koji je pod GNU General Public licencom. Uz njega dolazi nekoliko

alata. Osnovni i najviše korišćen je Memcheck, koji može da otkrije i prijavi sledeće vrste grešaka u memoriji: korišćenje neinicijalizovane memorije, čitanje/pisanje u memoriju nakon što je oslobođena, čitanje/pisanje na kraj alociranog bloka memorije, curenje memorije i mnoge druge. Valgrind će greške koje se teško pronalaze naći lako. Vrlo je temeljit. Iskustvo programera pokazuje da će otklanjanje svih grešaka koje Valgrind pronađe uštedeti vreme na duže staze. Ponaša se poput virtualnog x86 prevodioca, pa će program raditi 10 do 30 puta sporije od uobičajenog[13].

Razlike između Valgrind-a i GDB-a[13]:

- GDB je program za pronalaženje grešaka u kôdu, Valgrind između ostalog proverava i memoriju
- GDB nam dozvoljava da vidimo šta se dešava unutar programa dok on radi
- Valgrind nam neće dozvoliti da interaktivno prolazimo kroz program
- GDB ne proverava da li se koriste neinicijalizovane vrednosti ili je preplavljena dinamička memorija
- I GDB i Valgrind pokazuju broj linije u kojoj se desio Segmentation fault, ali Valgrind često pokazuje i uzrok Segmentation fault-a
- Često se greške pronalaze i ispravljaju brže koristeći Valgrind nego GDB

Još neka poređenja GDB-a sa drugim debagerima data su u narednoj tabeli (tabela 2) [9].

Odlika/Debager	GDB	PGDB	IDB	MVSD
Ispitivanje stanja niti	✔	✔	✔	✔
Specifične tačke prekida	✔	✔	✔	✔
Događaji deljenja podataka u nitima	✘	✔	✔	✔
Automatsko obaveštavanje o novim nitima	✔	✔	✔	✔
Funkcije za evidentiranje	✘	✔	✘	✘

Tabela 2: Poređenje debagera

7 Zaključak

GNU GDB predstavlja jedan od najpoznatijih debagera. Za njegovu popularnost zasluzene su odlične performanse. Njegovu snagu predstavljaju njegove karakteristike, mogućnost da se primeni na mnogim platformama kao i stepen do koga njegovo ponašanje može da se prilagodi specifičnim zahtevima[14]. Programi koji mogu biti analizirani mogu biti napisani na raznim programskim jezicima, dok se debager može pokrenuti na najpopularnijim operativnim sistemima. Još jedan od razloga zašto je GDB popularan jeste i jednostavan način njegovog korišćenja.

Literatura

[1] degugger basics. on-line at: https://worddisk.com/wiki/Debugger.

[2] GDB Commands. on-line at: http://www.yolinux.com/TUTORIALS/GDB-Commands.html#GDB_COMMAND_LINE_ARGS.

[3] gdbgui. on-line at: https://www.gdbgui.com/gettingstarted.

[4] GNUOrg. on-line at: https://www.gnu.org/software/ddd.

[5] KGDB. on-line at: https://www.kernel.org/doc/html/v4.14/dev-tools/kgdb.html.

[6] LLDB. on-line at: https://lldb.llvm.org/.

[7] .NET debugging tools comparison. on-line at: https://blog.revdebug.com/net-debugging-tools-comparison.

[8] NoWhere.net. on-line at: https://n0where.net/web-gdb-gui-gdbgui.

[9] Poredjenje debagera. on-line at: https://www.researchgate.net/figure/Comparison-of-PGDB-with-Existing-Debuggers_tbl1_310463717.

[10] Qt. on-line at: https://doc.qt.io/qtcreator/creator-debugging.html.

[11] Soureceware. on-line at: https://sourceware.org.

[12] ubuntupit. on-line at: https://www.ubuntupit.com/best-linux-debuggers-for-modern-software-engineers/?fbclid=IwAR1HcFGsB8b5ww7D1omgYs4OgrQ55Rjq3QE-ZFuWHtfeF3HGEp1kZ2J551U.

[13] Valgrind Tutorial. on-line at: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=6&cad=rja&uact=8&ved=2ahUKEwiShPLgvPXoAhWok4sKHdmWBdYQFjAFegQIAxAB&url=ftp%3A%2F%2Fftp.sara.nl%2Fpub%2Foutgoing%2Ftam2012%2F03_-_Tutorial_Valgrind.pdf&usg=AOvVaw3quUQiBYM3UQSmkMSJA9BI.

[14] Bill Gatliff. *Embedding with GNU: GNU Debugger*. Embedded Systems Programming, 1999.

[15] Koen V. Hindriks. Debugging is Explaining. Master's thesis, Delft University of Technology, The Netherlands.

[16] Milena Vujošević Janičić. *Dinamička analiza*. Matematički fakultet, Univerzitet u Beogradu. on-line at: http://www.verifikacijasoftvera.matf.bg.ac.rs/vs/predavanja/03_dinamicka_analiza/03_dinamicka_analiza.pdf.

[17] N. Matloff, P. Pesch, and P. J.. Salzman. *The Art of Debugging with GDB, DDD and Eclipse*. William Pollock, 2008.

[18] Rohan Pearce. Taking the pain out of debugging with live programming. *COMPUTERWORLD, The Voice of Business Technology*, 2015.

[19] Arnold Robbins. *GDB Pocket Reference*. O'Reilly Media, 2005.

[20] John Robbins. *Debugging Applications*. Microsoft Press, 2000.

[21] R. Stallman, P. Pesch, S. Shebs, and al. *Debugging with GDB: The GNU Source-Level Debugger*. Free Software Foundation, 2002.

[22] Đorđe Todorović. Podrška za napredu analizu promenljivih lokalnih za niti pomoću alata GNU GDB. Master's thesis, Univerzitet u Beogradu, Matematički fakultet, 2019.

A Dodatak

Komanda	Značenje
help <klasa|komanda|all>	Izlistavanje pomoći za klasu komandi, samu komandu ili sve klase komandi
run	Pokretanje samog programa, sa svim navedenim tačkama prekida
appropos *reč*	Pretraga komande vezane za datu reč
info args	Izlistavanje argumenata komandne linije
info breakpoints	Izlistavanje tačaka prekida
info registers	Izlistavanje registara u upotrebi
info threads	Izlistavanje niti koje program koristi
break +|- broj-linija	Postavljanje tačke prekida za dati broj linija od trenutne pozicije pri prekidu
delete	Brisanje svih tačaka prekida
delete broj-tačke-prekida	Brisanje tačke prekida sa određenim brojem
disable broj-tačke-prekida | raspon	Isključivanje tačke prekida sa datim brojem ili više tačaka u nekom rasponu
enable broj-tačke-prekida	Uključivanje tačke prekida sa određenim brojem
continue | c	Nastavak izvršavanja do naredne tačke prekida
finish	Dovršavanje izvršavanja započete funkcije
step <broj-koraka>	Prelazak na narednu liniju kôda ili liniju za određen broj koraka udaljenu
next <broj-koraka>	Izvršavanje naredne linije kôda ili narednih <broj-koraka> linija kôda
where	Prikaz trenutne pozicije u kôdu, broj linije
backtrace	Prikaz trenutne pozicije, ime funkcije, kojim putem se došlo do tog stanja
<up|down><broj-okvira>	Pomeraj gore ili dole kroz stek okvire, za onoliko okvira koliko zadamo, ili jedan ako ne navedemo ništa

Tabela 3: Tabela nekih od najbitnijih komandi [2].

B Upotreba GDB-a u Qt Creator-u

Samo debagovanje u Qt Creator-u će biti objašnjeno kroz naredni kôd.

```
1 #include <iostream>
2 #include <vector>
3
4 using namespace std;
5
6 int main()
7 {
```

```
8
9       string hello = "hello world";    //prva tacka prekida
10      cout << hello;                   //druga tacka prekida
11
12      auto vec = new vector<int>{1,2,3,4,5,6,7,8,9,10};
13
14      for(size_t i = 0; i < vec->size(); i++)
15          cout << vec->at(i);          //treca tacka prekida
16
17      delete vec;
18
19      for(size_t i = 0; i < vec->size(); i++){
20          cout << vec->at(i);          //cetvrta tacka prekida
21      }
22
23  }
```

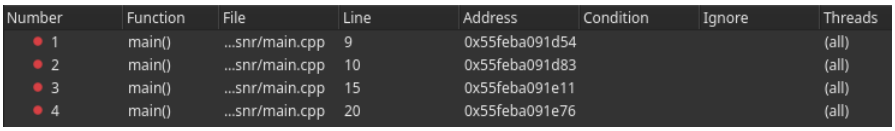
Listing 2: Primer jednostavnog programa za prikaz rada GDB-a u Qt Creator-u

Za početak, postavljanje tačaka prekida. Vrši se skoro isto kao i kod gdbgui-a, jedina je razlika što se kod Qt Creator-a ne klikće na sam broj linije, već malo levo od broja, te se tu prikazuje kružić koji označava da je tačka prekida postavljena. Ona se može onemogućiti/omogućiti ili uređivati desnim klikom na kružić i biranjem određene funkcionalnosti. Slika 2 prikazuje informacije o samoj tački prekida kada se kursorom pozicioniramo na dobijeni kružić. Tu imamo informacije o internom id-u tačke prekida (*Internal ID*), koji se dodeljuje po redosledu postavljanja, a ne po redosledu u kodu počevši od prve linije; zatim stanje tačke prekida odnosno da li je omogućena ili ne (*State*), tip (*Breakpoint Type*), fajl u kojem se nalazi (*Marker File*), broj linije u fajlu na kojoj se nalazi (*Marker Line*) i broj koliko smo se puta zaustavili u toj tački prekida (*Hit Count*). Na samom vrhu slike se nalazi žuta strelica. Ona se ne nalazi tu stalno, već samo u momentu kada je program zaustavljen u toj tački i ispitujemo informacije za nju.

Slika 2: Informacije o jednoj tački prekida

Prilikom dodavanja tačaka prekida, one se dodaju u listu svih tačaka prekida. Za ovako jednostavan program to nije od krucijalnog značaja, dok kod kompleksnijih programa može biti od velike pomoći. U tom spisku one se mogu uređivati, omogućiti ili onemogućiti, brisati. Kako spisak izgleda, dato je slikom 3.

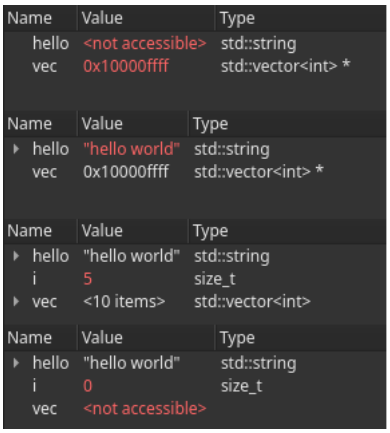
Kada smo postavili tačke prekida, vreme je da pokrenemo program u debag režimu. To se može postići na više načina, najlakši od njih je klikom na **play** dugme u donjem levom uglu, sa nacrtanom bubom na sebi, moguće je i klikom na dugme **F5** na tastaturi, kao i odlaskom u meni **Debug->Start Debugging**[10].



Number	Function	File	Line	Address	Condition	Ignore	Threads
1	main()	...snr/main.cpp	9	0x55feba091d54			(all)
2	main()	...snr/main.cpp	10	0x55feba091d83			(all)
3	main()	...snr/main.cpp	15	0x55feba091e11			(all)
4	main()	...snr/main.cpp	20	0x55feba091e76			(all)

Slika 3: Spisak tačkaka prekida i neke dodatne informacije o njima

Program se pri izvršavanju zaustavlja na svakoj od tačaka prekida i daje informacije o svim promenljivim iz tog dosega. Informacije o promenljivama su date slikom 4. Ono što primećujemo u tom delu je da GDB daje informacije o imenu promenljive, tipu i njenoj vrednosti, ukoliko ju je moguće koristiti, u suprotnom stoji poruka *<not accessible>*. Ta poruka nam šalje informaciju da tu promenljivu ne bi trebalo koristiti u tom delu kôda. U našem primeru, u liniji 17, smo pokazivač "poništili" i on više nije validan, što nam četvrta tačka prekida i govori, ali u toj liniji pokušavamo da ispišemo vrednost prvog elementa tog vektora. Ta greška je prouzrokovala **Segmentation Fault**, dok nam GDB izbacuje obaveštenje dato slikom 5. Problem u ovom vidu ispisa jeste i što nema informacija gde se ta greška dogodila, ali se ta informacija nalazi u drugom prostoru, pored spiska svih tačaka prekida, sa informacijom u kojoj liniji je trenutno program, odnosno gde se zaustavio kao i koja funkcija je u pitanju i u kom fajlu.



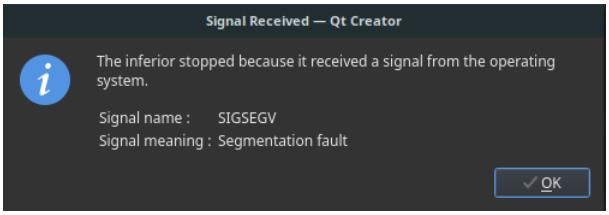
Name	Value	Type
hello	<not accessible>	std::string
vec	0x10000ffff	std::vector<int> *

Name	Value	Type
hello	"hello world"	std::string
vec	0x10000ffff	std::vector<int> *

Name	Value	Type
hello	"hello world"	std::string
i	5	size_t
vec	<10 items>	std::vector<int>

Name	Value	Type
hello	"hello world"	std::string
i	0	size_t
vec	<not accessible>	

Slika 4: Informacije o promenljivama u tačkama prekida



Slika 5: Signal pri nepravilnom završavanju programa