UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Đorđe Todorović

PODRŠKA ZA NAPREDNU ANALIZU PROMENLJIVIH LOKALNIH ZA NITI POMOĆU ALATA GNU GDB

master rad

Beograd, 2019.

Mentor:

dr Milena Vujošević-Janičić, docent
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Filip Marić, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Miroslav Marić, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: ____________________

Bratu Daretu

Naslov master rada: Podrška za naprednu analizu promenljivih lokalnih za niti pomoću alata GNU GDB

Rezime: Potreba za razvojom softvera raste svakodnevno. Da bi se podigla efikasnost programera potrebno je razvijati alate koji služe za pronalaženje i otklanjanje grešaka. Jedan takav alat je debager *GNU GDB* koji se može koristiti na razne načine. Jedan važan slučaj njegove upotrebe je učitavanje datoteke jezgra (eng. *core dump, core file*). Ovaj način se koristi u situacijama kada se analizira program koji se izvršavao na računaru sa drugačijom arhitekturom procesora od arhitekture na kojoj se vrši analiza programa. Višenitno programiranje je veoma prisutno, ponekad i neizbežno prilikom razvoja softvera za uređaje sa ugrađenim računarom. Često takvi uređaji nemaju debager, pa je analiza programa moguća samo na ličnim računarima. Jedna od važnih mogućnosti prilikom debagovanja višenitnih aplikacija je čitanje vrednosti promenljive koje su lokalne za niti (eng. *TLS-Thread Local Storage*). Ovaj rad opisuje unapređenje debagera *GNU GDB* koje omogućava čitanje vrednosti *TLS* promenljive iz datoteke jezgra generisane na uređaju sa ugrađenim računarom.

Ključne reči: Thread Local Storage, TLS, GNU‘s Not Unix, GNU Debugger, GNU GDB, Datoteka jezgra

Sadržaj

1 Uvod **1**

2 Debageri **4**

2.1 Prevođenje programa 5

2.2 Format DWARF 5

2.3 Format ELF 9

2.4 Sistemski poziv `ptrace` 10

2.5 Realizacija osnovnih elemenata upotrebe debagera 11

3 Debager *GNU GDB* **16**

3.1 Istorija debagera *GNU GDB* 16

3.2 Pojam *arhitektura* za debager *GNU GDB* 17

3.3 Datoteke jezgra 21

3.4 Višearhitekturni debager - *Multiarch GNU GDB* 23

4 Promenljive lokalne za niti **27**

4.1 Uvod u *TLS* 27

4.2 Definisanje novih podataka u formatu ELF 29

4.3 Rukovanje TLS promenljivom tokom izvršavanja programa 31

4.4 Pokretanje i izvršavanje procesa 33

4.5 TLS modeli pristupa 36

5 Implementacija rešenja **38**

5.1 Detalji implementacije 39

5.2 Alternativno rešenje 42

5.3 Čitanje i pisanje datoteke jezgra za procesorsku arhitekturu MIPS . . 43

5.4 Testiranje 45

5.5 Upotreba alata 46

6 Zaključak **50**

Literatura **51**

Glava 1

Uvod

Softver je svuda oko nas, od automobila, aviona pa sve do kućnih aparata poput frižidera i šporeta. Veoma je važno da softver bude efikasan i kvalitetan. U težnji za efikasnijim softverom, kao i za softverom čija arhitektura oslikava logičku stukturu problema koriste se višenitne aplikacije. Da bi softver bio kvalitetan, u procesu razvoja neophodni su alati koji pomažu programeru da uoči i ispravi greške. Primer takvog alata je debager. Najpoznatiji debager je *GNU GDB* [7]. Bitno je obezbediti laku analizu višenitnih programa, što je posebno izazovan zadatak.

Razvoj softvera za uređaje sa ugrađenim računarom (npr. softver za automobile) se obično odvija na ličnim računarima, jer ugrađeni računari oskudevaju u pogledu resursa. Arhitektura procesora uređaja sa ugrađenim računarom se najčešće ne poklapa sa arhitekturom procesora ličnih računara. Programi koji su prevedeni za jednu arhitekturu računara se ne mogu izvršavati na računarima drugačijih arhitektura. Greške u programima koji se izvršavaju na uređajima sa ugrađenim računarom najčešće otklanjamo koristeći alate na ličnim računarima.

Debager *GNU GDB*, između ostalog, omogućava analiziranje i otklanjanje grešaka u programima koji se izvršavaju na računarima drugih arhitektura. Jedan način kroz koji se to može ostvariti je upotreba *GNU GDB* servera. Taj proces zovemo *udaljeno debagovanje*. *Klijent/server* model čine serverski deo koji obezbeđuje usluge i resurse, i klijentski deo koji traži usluge. Aplikacija *GNU GDB* server funkcioniše kao omotač oko programa koji se izvršava, tako da ne koristi puno resurasa uređaja na kome se izvršava. Debager na ličnom računaru predsavlja klijenta koji se uz korišćenje posebnih protokola povezuje sa *GNU GDB* serverom koji je pokrenut na udaljenom uređaju. Ograničenja udaljenog debagovanja su isti operativni sistem na ličnom računaru i udaljenom uređaju, i učitavanje iste izvršne datoteke u

debager na oba uređaja. Drugi način debagovanja programa je učitavanje datoteke jezgra (eng. *core dump, core file*) u debager. Datoteka jezgra može biti kreirana sa korisničkog nivoa, npr. baš uz pomoć debagera *GNU GDB*, ili prilikom neregularnog prekida izvršavanja programa samo jezgro operativnog sistema beleži informacije o stanju sistema i zapisuje je u nju. Datoteka jezgra sadrži stanje radne memorije procesa prilikom prekidanja rada programa na nestandardan način. Preciznije, ona sadrži informacije o vrednostima promenljivih, procesorskih registara, programskih brojača, informacije o samom procesu, informacije o nitima itd. Tako kreirana datoteka jezgra na gostujućoj arhitekturi može biti učitana i analizirana u debageru *GNU GDB* na računaru sa arhitekturom domaćina.

Ovaj rad opisuje postupak korišćenja debagera prilikom čitanja vrednosti promenljivih lokalnih za niti, ili, skraćeno, *TLS* (eng. *TLS-Thread Local Storage*) promenljivih [19]. Definisanjem takvih promenljivih svaka nit ima različitu kopiju iste promenljive i prilikom analiziranja višenitnih programa značajno je pr<očitati vrednosti te promenljive iz svih niti. Ključna reč, programskih jezika *C* i *C++*, koja se dodaje ispred definicije ili deklaracije promenljive je `__thread`. Različite arhitekture mogu imati različitu implementaciju mehanizma *TLS*, te je njihovo čitanje iz debagera *GNU GDB* dodatno otežano. Glavni doprinos rada predstavlja omogućavanje čitanja vrednosti *TLS* promenljive iz datoteke jezgra koja je generisana na različitoj arhitekturi u odnosu na arhitekturu domaćina. Time se poboljšavaju karakteristike debagera *GNU GDB* jer se proširuju dostupne informacije za analizu programa drugih arhitektura.

Rezultati opisani u ovom radu su predstavljeni i na konfereniciji TELFOR [18].

Rad se pored prvog uvodnog, sastoji još iz pet poglavlja. Drugo poglavlje detaljno opisuje rad debagera uključujući neophodnu podršku operativnih sistema i formate *DWARF* (format za predstavljanje pomoćnih informacija za debagovanje) i *ELF* (format izvršnih datoteka, deljenih biblioteka, objektnih datoteka i datoteka jezgra). Treće poglavlje opisuje uptorebu standardnog debagera *GNU GDB* kao i višearhitekturne verzije debagera *Multiarch GNU GDB* koja omogućava debagovanje programa koji se izvršavaju na arhitekturi koja je različita u odnosu na arhitekturu na kojoj se debager izvršava. Četvrto poglavlje prikazuje detalje o *TLS* promenljivama. Naveden je opis organizacije registara, memorije i uvođenja novih struktura podataka koje omogućavaju funkcionisanje mehanizma *TLS*. U petom poglavlju opisuje se implementacija poboljšanja debagera koja omogućava čitanje vrednosti promenljivih lokalnih za niti iz datoteke jezgra koja je generisana na arhi-

tekturi računara drugačijoj od one na kojoj se debager izvršava. U šestom poglavlju predstavljen je zaključak rada.

Glava 2

Debageri

Greške su sastavni deo svakog rada koji obavlja čovek, te ih i programeri prave. Greške mogu biti hardverske i softverske. One mogu imati razne poslednice. Neke greške su manje važne, kao npr. greška u korisničkom interfejsu neke kompijuterske igrice. Postoje i greške koje mogu imati daleko veće posledice, pa čak i ugroziti živote drugih, kao npr. greške u softveru ili hardveru uređaja i aplikacija avio industrije. Faza testiranja je veoma važna u ciklusu razvoja softvera. Nakon faze testiranja obično sledi faza analize i otklanjanja grešaka.

Debager (eng. *debugger*) je softverski alat koji koriste programeri za testiranje, analizu i otklanjanje grešaka u programima. Sam proces korišćenja takvih alata nazivamo debagovanjem (eng. *debugging*). Debageri mogu pokrenuti rad nekog procesa ili preuzeti kontrolu izvršavanja procesa koji je već u fazi rada. U oba slučaja, debager preuzima kontrolu nad procesom. To mu mogućava da izvršava proces instrukciju po instrukciju, da postavlja tačke prekida (eng. *breakpoints*) itd. Proces izvršavanja programa od strane debagera sekvencijalno, instrukciju po instrukciju ili liniju po liniju, nazivamo koračanje. Tačke prekida su mesta u programu na kojima se zaustavlja izvršavanje programa prilikom debagovanja. To može biti trenutak kada program izvrši određenu funkciju, liniju koda itd. Neki debageri imaju mogućnost izvršavanja funkcija programa koji se debaguje, uz ograničenje da program pripada istoj procesorskoj arhitekturi kao i sistem na kojem se debager izvršava.

Podršku debagerima, u opštem slučaju, daju operativni sistemi, kroz sistemske pozive koji omogućavaju tim alatima da pokrenu i preuzmu kontrolu nad nekim drugim procesom. Za neke naprednije tehnike debagovanja poželjna je podrška od strane hardvera. U radu će detaljno biti obrađen rad debagera u okviru *UNIX*-

olikih, posebno *Linux* operativnih sistema. *DWARF*[1] predstavlja format zapisivanja pomoćnih informacija za debagovanje (eng. *debug information*), koje opisuju izvršnu datoteku [4]. Alati za debagovanje u okviru *UNIX*-olikih operativnih sistema prate taj standard. Debageri i programski prevodioci u okviru operativnog sistema *Windows* ne prate standard *DWARF* prilikom baratanja pomoćnim informacijama za debagovanje, već standard *Microsoft CodeView*. Više informacija o ovom standardu može se pronaći u literaturi [11].

2.1 Prevođenje programa

Programi koji se debaguju se prevode uz pomoć odgovarajuće opcije programskih prevodioca (za prevodioce *GCC* [12] i *LLVM/Clang* [14], to je opcija *-g*) koja obezbeđuje generisanje pomoćnih informacija za debagovanje. Takva verzija izvršnog programa se zove *razvojna*. Ukoliko je program koji se analizira preveden bez optimizacija, pomoćne informacije za debagovanje koje prate program su potpune. Verzija programa koja se isporučuje krajnjem korisniku se zove *objavljena* (eng. *release*). Prilikom prevođenja objavljene verzije programa se koriste optimizacije. Optimizacije se koriste da bi program bio brži i zauzimao manje memorije. Postoje različiti nivoi optimizacija i oni se zadaju kao opcija prevodiocu prilikom prevođenja. Nivoi optimizacija objavljenih programa su *-O2* i *-O3*.

Prilikom optimizacija se gube razne pomoćne informacije koje su potrebne za debagovanje. Neke promenljive i funkcije programa neće biti predstavljene pomoćnim informacijama za debagovanje. Npr. promenljiva programa može biti živa samo u nekim određenim delovima programa, pa programski prevodioci generišu pomoćne informacije za debagovanje o njenim lokacijama samo u tim određenim delovima koda. Prilikom optimizacija na nivou mašinskog koda život promenljive može biti skraćen, pa čitanje vrednosti promenljive iz debagera u nekim sitacijama neće biti moguće, iako gledajući izvorni kod očekujemo da je ona živa u tom trenutku.

2.2 Format DWARF

DWARF je format za predstavljanje pomoćnih informacija za debagovanje, koji se koristi od strane programskih prevodioca (kao npr. *GCC* ili *LLVM/Clang*) i debagera (kao npr. *GNU GDB*) da bi se omogućilo debagovanje na nivou izvornog

[1] DWARF - *Debugging With Attribute Record Format*

koda. Omogućava podršku za razne programske jezike kao što su *C/C++* i *Fortran*, ali je dizajniran tako da se lako može proširiti na ostale jezike. Arhitekturalno je nezavisan i predstavlja „most" između izvornog koda i izvršne datoteke. Trenutno je poslednji realizovani standard verzija 5 formata *DWARF*.

Format *DWARF* se primenjuje na programe *UNIX*-olikih operativnih sistema, kao što su *Linux* i *MacOS*. Generisane pomoćne informacije za debagovanje, prateći standard *DWARF*, su podeljene u nekoliko sekcija sa prefiksom `.debug_`. Neke od njih su `.debug_line` (sadrzi informacije o linijama izvornog koda), `.debug_loc` (sadrzi informacije o lokacijama promenljivih) i `.debug_info`. Sekcija `.debug_info` je ključna sekcija koja je organizovana u drvoliku strukturu koja sadrži pomoćne informacije za debagovanje (koje mogu referisati i na pomoćne informacije iz ostalih sekcija). Čvorovi ove drvolike strukture se sastoje od osnovnih jedinica pomoćnih informacija za debagovanje *DIE* (eng. *Debug Info Entry*). Osnovna jedinica sadrži određene atribute sa prefikosm `DW_AT_` i identifikovana je oznakom (eng. *tag*). Različitim entitetima programskog jezika odgovaraju različite osnovne jedinice. Na primer, oznaka za lokalne promenljive je `DW_TAG_local_variable`, a oznaka za funkcije je `DW_TAG_subprogram`. Osnovna jedinica dodatno može ukazivati na razne informacije o entitetu kao što su ime promenljive ili funkcije, linija deklaracije, itd. Koren svakog stabla, koje sadrži pomoćne informacije za debagovanje, je predstavljen osnovnom jedinicom, tj. oznakom `DW_TAG_compile_unit`, koja predstavlja kompilacionu jedinicu, tj. izvorni kod programa.

Listing 2.1: Primer programa napisanog u *C* programskom jeziku

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int x;
6    x = 5;
7    printf ("The value is %d\n", x);
8    return 0;
9  }
```

Deo stabla sa pomoćnim informacijama za debagovanje za primer programa napisanog u programskom jeziku *C* koji je zadat listingom 2.1 je prikazan u listingu 2.2.

Listing 2.2: Primer *DWARF* reprezentacije

```
1  <1><73>: Abbrev Number: 4 (DW_TAG_subprogram)
2    <74> DW_AT_external : 1
3    <74> DW_AT_name : (indirect string, offset: 0x68): main
4    <78> DW_AT_decl_file : 1
5    <79> DW_AT_decl_line : 3
6    <7a> DW_AT_type : <0x57>
7    <7e> DW_AT_low_pc : 0x400526
8    <86> DW_AT_high_pc : 0x2a
9    <8e> DW_AT_frame_base : 1 byte block: 9c (
         DW_OP_call_frame_cfa)
10   <90> DW_AT_GNU_all_tail_call_sites: 1
11 <2><90>: Abbrev Number: 5 (DW_TAG_variable)
12   <91> DW_AT_name : x
13   <93> DW_AT_decl_file : 1
14   <94> DW_AT_decl_line : 5
15   <95> DW_AT_type : <0x57>
16   <99> DW_AT_location : 2 byte block: 91 6c (DW_OP_fbreg: -20)
```

Funkcija `main` je predstavljena oznakom `DW_TAG_subprogram`. Atribut te osnovne jedinice predstavljen sa `DW_AT_name` ima vrednost imena funkcije. Atributi `DW_AT_low_pc` i `DW_AT_high_pc` redom predstavljaju adresu prve mašinske instrukcije te funkcije u memoriji programa i pomeraj na kojem se nalazi poslednja mašinska instrukcija te funkcije. Sledeći čvor drveta predstavlja promenljivu `x` istog test primera. Ta osnovna jedinica je dete čvora koji predstavlja funkciju `main` i ukazuje da se promenljiva `x` nalazi unutar funkcije `main`. Promenljiva `x` je predstavljena oznakom `DW_TAG_variable`. Atribut `DW_AT_name` predstavlja ime promenljive, `DW_AT_type` referiše na osnovnu jedinicu koja predstavlja tip promenljive, dok `DW_AT_location` atribut predstavlja lokaciju promenljive u memoriji programa.

Debag promenljive

Svaka promenljiva programa prevedenog sa pomoćnim informacijama za debagovanje, ukoliko se ne radi o optimizovanom programu, je predstavljena oznakom `DW_TAG_variable`. Atribut `DW_AT_location` ukazuje na lokaciju promenljive. Lokacija može biti predstavljena izrazom, kao npr. lokacija promenljive `x` prikazana

u listingu 2.3. Izraz te promenljive ukazuje da se ona nalazi na pomeraju `-20` trenutnog stek okvira `main` funkcije. U neoptimizovanom kodu sve promenljive imaju lokacije zadate nekim izrazom u formatu *DWARF*. Njihove vrednosti su dostupne debagerima u bilo kom delu koda u kom su definisane.

U optimizovanom kodu lokacija promenljive može sadržati referencu na informaciju o lokaciji u `.debug_loc` sekciji. Lokacije u toj sekciji su predstavljene listama lokacija. Jedna promenljiva u optimizovanom kodu može biti smeštena na raznim memorijskim lokacijama ili registrima. Elementi liste opisuju lokacije promenljive na mestima u kodu gde je ona živa. Ukoliko promenljiva nije živa u nekom delu koda, programski prevodioci u optimizovanom kodu neće pratiti njenu lokaciju. Naredni primer zadat u listingu 2.3 predstavlja lokaciju promenljive u optimizovanom kodu.

Listing 2.3: Primer *DWARF* reprezentacije promenljive

```
1 <2><90>: Abbrev Number: 5 (DW_TAG_variable)
2 <91> DW_AT_name : x
3 <93> DW_AT_decl_file : 1
4 <94> DW_AT_decl_line : 5
5 <95> DW_AT_type : <0x5e>
6 <99> DW_AT_location : 0x0 (location list)
```

Lokacijska lista promenljive `x` je predstavljena u listingu 2.4. U ovom konkretnom primeru, promenljiva živi samo na jednom mestu. Potencijalno je mogla imati još elemenata lokacijske liste. `Offset` predstavlja informaciju gde se lokacijska lista određene promenljive nalazi u `.debug_loc` sekciji. `Begin` i `End` predstavljaju informaciju od koje do koje adrese u programu važi data lokacija, tj. od koje do koje instrukcije je određena promenljiva živa. `Expression` predstavlja izraz koji opisuje lokaciju promenljive.

Listing 2.4: Primer *DWARF* reprezentacije lokacijske liste

```
1 Contents of the .debug_loc section:
2         Offset Begin End Expression
3         00000000 400450 40046a (DW_OP_fbreg: -20)
4         0000001c <End of list>
```

2.3 Format ELF

ELF (eng. *Executable and Linkable Format*) je format izvršnih datoteka, deljenih biblioteka, objektnih datoteka i datoteka jezgra [3].

ELF sadrži razne informacije o samoj datoteci. Podeljen je u dva dela: *ELF* zaglavlje i podaci datoteke. *ELF* zaglavlje sadrži informacije o arhitekturi za koju je program preveden i definiše da li program koristi 32-bitni ili 64-bitni adresni prostor. Zaglavlje 32-bitnih programa je dužine 52 bajta, dok kod 64-bitnih programa zaglavlje je dužine 64 bajta. Podaci datoteke mogu sadržati programsku tabelu zaglavlja (eng. *program header table*), sekcijsku tabelu zaglavlja (eng. *section header table*) i ulazne tačke prethodne dve tabele. Primer u listingu 2.5 prikazuje format *ELF* pročitan alatom *readelf*.

Listing 2.5: Primer dela zaglavlja *ELF*

```
1 ELF Header:
2 Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
3 Class: ELF64
4 Data: 2's complement, little endian
5 Version: 1 (current)
6 OS/ABI: UNIX - System V
7 Type: EXEC (Executable file)
8 Machine: Advanced Micro Devices X86-64
```

Format *ELF* definiše segment podataka koji sadrži informacije o globalnim promenljivama. Segment podataka je podeljen na dva dva dela. Tabela 2.1 opisuje `.bss` i `.data` sekcije. Sekcija `.bss` sadrži neinicijalizovane globalne promenljive, dok `.data` sadrži inicijalizovane.

Ime polja	*Vrednosti polja u okviru* `.tbss` *sekcije*	*Vrednosti polja u okviru* `.tdata` *sekcije*
`sh_name`	`.bss`	`.data`
`sh_type`	`SHT_NOBITS`	`SHT_PROGBITS`
`sh_flags`	`SHF_ALLOC + SHF_WRITE`	`SHF_ALLOC + SHF_WRITE`
`sh_addr`	Virtualna adresa sekcije	Virtualna adresa sekcije
`sh_offset`	Pomeraj sekcije	Pomeraj sekcije
`sh_size`	Veličina sekcije	Veličina sekcije
`sh_link`	`SHN_UNDEF`	`SHN_UNDEF`
`sh_info`	0	0
`sh_addralign`	Poravnanje sekcije	Poravnanje sekcije
`sh_entsize`	0	0

Tabela 2.1: Tabela vrednosti polja koji opisuju sekcije podataka

2.4 Sistemski poziv `ptrace`

Operativni sistem *Linux* pruža sistemski poziv `ptrace` [5] koji debagerima omogućava rad. Ovaj sistemski poziv omogućava jednom procesu kontrolu nad izvršavanjem nekog drugog procesa, uključujući i menjanje njegove memorije i sadržaja registara. Potpis ove funkcije je:

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

Prvi argument sistemskog poziva predstavlja informaciju kojom operativnom sistemu jedan proces, ne nužno debager, ukazuje na nameru preuzimanja kontrole drugog procesa. Ukoliko taj argument ima vrednost `PTRACE_TRACEME` to ukazuje na nameru praćenja (eng. *tracing*) određenig procesa, `PTRACE_PEEKDATA` i `PTRACE_POKEDATA` redom ukazuju na nameru čitanja i pisanja memorije, `PTRACE_GETREGS` i `PTRACE_SETREGS` se odnose na čitanje i pisanje registara. To su samo neki osnovni slučajevi korišćenja, za više informacija pogledati [5]. Drugi argument sistemskog poziva `pid` ukazuje na identifikacioni broj ciljanog procesa. Treći i četvrti argument se po potrebi koriste u zavisnosti od namere korišćenja sistemskog poziva `ptrace` za čitanje ili pisanje sa adrese koja je data trećim argumentom, pri tom baratajući podacima na adresi zadatoj četvrtim argumentom. To znači da ukoliko se koristi `PTRACE_TRACEME` poslednja tri argumenta sistemskog poziva se ignorišu. Ukoliko se

koristi `PTRACE_GETREGS` sa adrese `addr` se čita jedna reč iz memorije.

Navodimo par osnovnih primera korišćenja `ptrace` sistemskog poziva. Primeri 1 i 2 navode dva osnovna primera korišćenja `ptrace` sistemskog poziva. U nastavku teksta biće navedeno još primera.

Primer 1 *Program inicira da će biti praćen od strane roditeljskog procesa:*

```
ptrace(PTRACE_TRACEME, 0, NULL, NULL);
```

Primer 2 *Čitanje vrednosti registara procesa sa identifikatorom 8845 i upisivanje tih vrednosti na adresu promenljive* `regs`*:*

```
ptrace(PTRACE_GETREGS, 8845, NULL, &regs);
```

2.5 Realizacija osnovnih elemenata upotrebe debagera

U nastavku teksta se opisuju realizacije osnovnih elemenata upotrebe debagera.

Tačke prekida

Postoje dve vrste tačaka prekida: softverske i hardverske [7].

Osvrnimo se prvo na softverske tačke prekida. Postavljanje tačaka prekida predstavlja jednu od najkorišćenijih mogućnosti debagera, te stoga ćemo navesti nekoliko smernica kako je ista realizovana, u opštem slučaju. Ne postoji jedinstveni poziv nekog sistemskog poziva za postavljanje tačke prekida, već se ista obavlja kao kombinacija više mogućnosti sistemskog poziva `ptrace`. Opišimo ceo postupak na jednostavnom primeru.

Program je preveden za procesorsku arhitekturu *Intel x86-64* i asemblerski kôd `main` funkcije je prikazan u listingu 2.6.

Listing 2.6: Primer funkcije `main` u asemblerskom jeziku

```
1  0000000000400450 <main>:
2  400450:        48 83 ec 08          sub    $0x8,%rsp
3  400454:        ba 05 00 00 00       mov    $0x5,%edx
4  400459:        be 04 06 40 00       mov    $0x400604,%esi
5  40045e:        bf 01 00 00 00       mov    $0x1,%edi
6  400463:        31 c0                xor    %eax,%eax
7  400465:        e8 c6 ff ff ff       callq  400430 <printf>
8  40046a:        31 c0                xor    %eax,%eax
9  40046c:        48 83 c4 08          add    $0x8,%rsp
10 400470:        c3                   retq
11 400471:        66 2e 0f 1f 84       nopw   %cs:0x0(%rax,%rax,1)
12 400478:        00 00 00
13 40047b:        0f 1f 44 00 00       nopl   0x0(%rax,%rax,1)
```

Primera radi, želimo da postavimo tačku prekida na treću po redu instrukciju funkcije `main`:

`be 04 06 40 00 mov $0x400604, %esi`

Da bismo to uradili, debager menja prvi bajt instrukcije sa posebnom magičnom vrednošću, obično `0xcc`, i kada izvršavanje dostigne do tog dela koda ono će se zaustaviti na tom mestu. Pošto `0xbe` je zamenjeno sa `0xcc` i na tom mestu u kodu dobijamo instrukciju:

`cc 04 06 40 00 int3`

Ukoliko korisnik želi da nastavi dalje, instrukcija prekida se zamenjuje sa originalnom instrukcijom koja se izvršava i nastavlja se sa radom programa. Instrukcija `int3` je posebna instrukcija procesorske arhitekture *Intel x86-64*, koja izazva softverski prekid. Kada vrednost u registru programski brojač (eng. *CPU register pc*) pokaže na adresu instrukcije `int3`, izvršavanje se zaustavlja na toj tački. Debager ima informaciju da je tačka prekida postavljena te on čeka na signal koji ukazuje na to da je program dostigao do instrukcije prekida. Operativni sistem prepoznaje instrukciju `int3`, poziva se specijalni obrađivač tog signala (na *Linux* sistemima `do_int3()`), koji dalje obaveštava debager šaljući mu signal sa kodom `SIGTRAP` koji on obrađuje na željeni način. Ovo važi za procesorsku arhitekturu *Intel x86-64*, a instrukcija prekida za arhitekture kao što su *ARM, MIPS, PPC* se drugačije kodira, ali postupak implementacije tačaka prekida je isti.

Ukoliko želimo da stavimo tačku prekida eksplicitno na funkciju `main`, za to koristimo posrednike u vidu pomoćnih informacija za debagovanje. U tom slučaju de-

bager traži element stabla sa pomoćnim informacijama za debagovanje koji ukazuje na informacije o `main` funkciji, i odatle čita informaciju na kojoj adresi u memoriji se nalazi prva mašinska instrukcija date funkcije. U listingu 2.7 vidimo element koji opisuje funkciju uz pomoć atributa. Debager će pročitati atribut `DW_AT_low_pc` i na tu adresu postaviti instrukciju `int3`. Kao što je već rečeno, pomoćne simbole za debagovanje generišemo uz pomoć opcije kompajlera `-g`.

Listing 2.7: Primer *DWARF* reprezentacije funkcije `main`

```
1  <1><73>: Abbrev Number: 4 (DW_TAG_subprogram)
2    <74> DW_AT_external : 1
3    <74> DW_AT_name : (indirect string, offset: 0x68): main
4    <78> DW_AT_decl_file : 1
5    <79> DW_AT_decl_line : 3
6    <7a> DW_AT_type : <0x57>
7    <7e> DW_AT_low_pc : 0x400526
8    <86> DW_AT_high_pc : 0x2a
9    <8e> DW_AT_frame_base : 1 byte block: 9c
10   <90> DW_AT_GNU_all_tail_call_sites: 1
```

Podrška za hardverske tačke prekida (eng. *watchpoint*) je direktno povezana sa hardverom. Hardverske tačke prekida se postavljaju na određenu adresu u programu. Specijalni registari mogu signalizirati razne promene na toj adresi, npr. čitanje, pisanje ili izvršavanje, što predstavlja prednost u odnosu na softverske tačke prekida. Mana hardverskih tačaka prekida u odnosu na softverske tačke prekida je neophodna hardverska podrška.

Koračanje

Pod procesom koračanja (eng. *stepping*) kroz program podrazumevamo izvršavanje programa sekvencu po sekvencu. Sekvenca može biti jedna procesorska instrukcija, linija koda ili pak neka funkcija programa koji se debaguje [7].

Instrukcijsko koračanje na platformi sa *Intel x86-64* arhitekturom procesora je direktno omogućeno kroz sistemski poziv `ptrace`:

```
ptrace(PTRACE_SINGLESTEP, debuggee_pid, nullptr, nullptr);
```

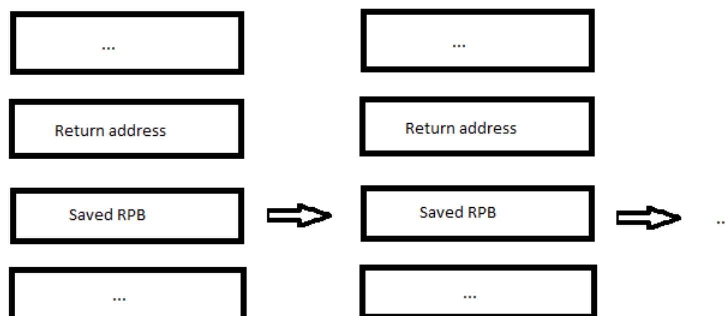
Operativni sistem će poslati debageru signal `SIGTRAP` kada je korak izvršen.

Pored instrukcijkog koračanja pomenućemo još jednu vrstu koračanja u funkciju koja je pozvana `call` ili `jump` instrukcijom. Komanda debagara *GNU GDB* koja nam to omogućava jeste `step in`.

Postoje arhitekture za koje ovo ne važi, kao npr. arhitekture u okviru platforme *ARM*. Ova arhitektura nema hardversku podršku za instrukcijsko koračanje i za njih se koračanje implementira na drugačiji način, uz pomoć emulacije instrukcija, ali u ovom radu neće biti reči o tome.

Izlistavanje pozivanih funkcija

Objasnimo komandu izlistavanje pozivanih funkcija (eng. *backtrace*) posmatrajući organizaciju stek okvira (eng. *stack frames*) na platformi sa *Intel x86-64* arhitekturom procesora [7].



Slika 2.1: Primer ređanja stek okvira na platformi sa *Intel x86-64* arhitekturom procesora

Na slici 2.1 navedeni su stek okviri za dva funkcijska poziva. Pre povratne vrednosti funkcije obično se ređaju argumenti funkcije. Sačuvana adresa u registru `RBP` jeste adresa stek okvira svog pozivaoca. Prateći sve okvire kao elemente povezane liste dolazimo do svih pozivanih funkcija do zadate tačke. Ako se pitamo

kako debager čita informaciju o imenu neke funkcije, odgovor je u tome da debager pretražuje stablo sa pomoćnim informacijama za debagovanje, tražeći oznaku `DW_TAG_subprogram` sa odgovarajućom povratnom adresom, pritom čitajući atribut `DW_AT_name` tog elementa.

Čitanje vrednosti promenljivih

Za čitanje vrednosti promenljivih u programu, debager pretražuje stablo sa pomoćnim informacijama za debagovanje tražeći promenljivu sa zadatim imenom. U slučaju lokalnih promenljivih, traži se `DW_TAG_variable` element čiji atribut `DW_AT_name` odgovara imenu navedene promenljive. Kada se ta promenljiva pronađe konsultuje se atribut `DW_AT_location`, koji ukazuje na lokaciju gde se vrednost promenljive nalazi. Ukoliko ovaj atribut nije naveden debager će vrednost takve promenljive smatrati kao optimizovanu prijavljujući informaciju o tome [7].

Glava 3

Debager *GNU GDB*

GNU GDB je alat koji omogućava uvid u događanja unutar drugog programa koji se nalazi u fazi izvršavanja, ili u slučaju neregularnog prekida izvršavanja programa uvid u to šta se dešavalo pa je do toga došlo. *GNU GDB* debager takođe omogućava uvid u to šta se dešavalo sa programima i na platformama koje imaju različitu arhitekturu od arhitekture domaćina (eng. *host architecture*). Da bi se to realizovalo koristi se *GNU GDB* server, što se naziva *udaljeno debagovanje*, jer se udaljenom uređaju pristupa preko posebnih protokola. U svrhe debagovanja programa drugih procesorskih arhitektura na ličnim računarima se takođe upotrebljava *Multiarch GNU GDB* koji koristi biblioteke namenjene drugačijim arhitekturama. Programi koji mogu biti analizirani mogu biti napisani u raznim programskim jezicima, kao što su *Ada, C, C++, Objective-C, Pascal*. *GNU GDB* debager se može pokrenuti na najpopularnijim operativnim sistemima *UNIX* i *Microsoft Windows* varijanti. U radu se podrazumeva korišćenje *UNIX*-olikog operativnog sistema.

3.1 Istorija debagera *GNU GDB*

Izvorni kod alata *GNU GDB* je originalno napisan od strane Ričarda Stalmana 1986. godine kao deo *GNU* sistema [16]. *GNU GDB* je slobodan i besplatan softver realizovan pod licencom *GNU General Public License (GPL)*. Od 1990. do 1993. godine alat je održavao Džon Gilmor, a trenutno za održavanje alata je zadužena grupa *GDB Steering Committee*, odobrena od strane fondacije *FSF* (eng. *Free Software Foundation*). Više informacija o fondaciji *FSF* se može pronaći u [6].

Poslednja realizovana verzija alata *GNU GDB* je 8.3.

3.2 Pojam *arhitektura* za debager *GNU GDB*

Za debager *GNU GDB* arhitektura je veoma labav koncept. Može se posmatrati kao bilo koje svojstvo programa koji se debaguje, ali obično se misli na procesorsku arhitekturu. Za debager su bitna dva svojstva procesorske arhitekture:

- Skup instrukcija (eng. *Instruction Set Architecture - ISA*) predstavlja specifičnu kombinaciju registara i mašinskih instrukcija.
- ABI (eng. *Application Binary Interface*) predstavlja spisak pravila koja propisuju pravilan način korišćenja skupa instrukcija.

Standardni *GNU GDB*

Standardni *GNU GDB* je preveden za istu procesorsku arhitekturu kao i računar na kome se alat izvršava, tj. arhitekturu domaćina. Korišćenje standardne verzije alata *GNU GDB* nad nekim programom ima ograničenje. Program koji se debaguje mora biti iste procesorske arhitekture kao i arhitektura domaćina.

Prevođenje standardnog *GNU GDB* debagera

Preuzimanje izvornog koda debagera se vrši komandama prikazanim na listingu 3.1.

Listing 3.1: Komande korišćene za preuzimanje izvornog koda debagera

```
1 mkdir gdb
2 cd gdb
3 git clone http://gnu.org/gnu/gdb.git
```

Prva komanda pravi direktorijum u kome se izgrađuje (prevodi) alat. Druga komanda vrši pozicioniranje trenutne putanje u taj direktorijum. Trećom komandom se preuzima izvorni kod alata.

Prvi korak prevođenja je konfiguracija direktorijuma u kome se prevođenje izvršava. Komandama iz listinga 3.2 se kreira `Makefile`.

Listing 3.2: Komande korišćene za kreiranje `Makefile`

```
1 mkdir build
2 cd build
3 ../configure
```

Prva komanda pravi direktorijum u kome se izgrađuje (prevodi) alat. Druga komanda vrši pozicioniranje trenutne putanje u taj direktorijum. Trećom komandom se kreira `Makefile`*. Skripte* `configure` *kao ulaz parsiraju skriptu* `Makefile.in` *podešavajući okruženje (putanje do deljenih biblioteka, programski prevodilac i drugo) prilikom čega je krajnji izlaz datoteka* `Makefile` *kojom se izgrađuje (prevodi) softver.*

Nakon konfiguracije direktorijuma prevođenje alata se vrši komandom `make` (listing 3.3).

Listing 3.3: Izvršavanje komande `make`

```
1 make
```

Pokretanje standardnog debagera *GNU GDB*

GNU GDB očekuje kao argument komandne linije program koji je preveden za istu procesorsku arhitekturu. Ukoliko je trenutna putanja pozicionirana u direktorijum gde je izgrađen alat, pokretanje alata nad programom pod imenom *test* se vrši sledećom komandom prikazanom u listingu 3.4.

Listing 3.4: Pokretanje debagera

```
1 ./gdb/gdb test
```

Korišćenje standardnog debagera *GNU GDB*

Pokažimo neke od osnovnih komandi debagera *GNU GDB*. Primer programa koji se debaguje je prikazan u listingu 3.5.

Listing 3.5: Primer programa u programskom jeziku C

```
1 int fn1 (int &addr) {
2   return 0;
3 }
4
5 int main()
6 {
7   int a = 5;
8   fn1(&a);
9 }
```

Postavljanje tačke prekida

Postavljanje tačke prekida na funkciju programa koji se debaguje se vrši komandom `break`. Program koji se debaguje se zaustavlja kada dostigne do određene funkcije. Primer korišćenja postavljanja tačke prekida na funkciju (u ovom slučaju `fn1`) je prikazan u listingu 3.6.

Listing 3.6: Primer postavljanja tačke prekida

```
1 (gdb) break fn1
2 Breakpoint 1 at 0x4005fe: file test.c, line 4.
3 (gdb) r
4 Starting program: /master_examples/x86_arch/test
5 7
6 Breakpoint 1, fn1 (arg=0x7fffffffdbf4) at test.c:4
7 4 (*arg)++;
```

Izvršavanje programa sekvencu po sekvencu

Izvršavanje programa sekvencu po sekvencu se radi pomoću tehnike koračanja. Komanda u okviru debagera *GNU GDB* koja nam omogućava izvršavanje instrukciju po instrukciju je `stepi`. Primer korišćenja komande `stepi`, uz pomoć korišćenja komande `disassemble` koja nam prikazuje asemblerski kod programa koji se debaguje je prikazan u listingu 3.7.

Listing 3.7: Primer korišćenja komande `stepi`

```
(gdb) disassemble
Dump of assembler code for function fn1:
0x00000000004005f6 <+0>: push %rbp
0x00000000004005f7 <+1>: mov %rsp,%rbp
0x00000000004005fa <+4>: mov %rdi,-0x8(%rbp)
=> 0x00000000004005fe <+8>: mov -0x8(%rbp),%rax
0x0000000000400602 <+12>: mov (%rax),%eax
0x0000000000400604 <+14>: lea 0x1(%rax),%edx
0x0000000000400607 <+17>: mov -0x8(%rbp),%rax
0x000000000040060b <+21>: mov %edx,(%rax)
0x000000000040060d <+23>: mov -0x8(%rbp),%rax
0x0000000000400613 <+29>: cmp $0x5,%eax
0x0000000000400616 <+32>: jle 0x400627 <fn1+49>
0x0000000000400628 <+38>: pop %rbp
0x0000000000400629 <+42>: retq
End of assembler dump.
(gdb) stepi
0x0000000000400602 4  (*arg)++;
(gdb) disassemble
Dump of assembler code for function fn1:
0x00000000004005f6 <+0>: push %rbp
0x00000000004005f7 <+1>: mov %rsp,%rbp
0x00000000004005fa <+4>: mov %rdi,-0x8(%rbp)
0x00000000004005fe <+8>: mov -0x8(%rbp),%rax
=> 0x0000000000400602 <+12>: mov (%rax),%eax
0x0000000000400604 <+14>: lea 0x1(%rax),%edx
0x0000000000400607 <+17>: mov -0x8(%rbp),%rax
0x000000000040060b <+21>: mov %edx,(%rax)
0x000000000040060d <+23>: mov -0x8(%rbp),%rax
0x0000000000400613 <+29>: cmp $0x5,%eax
0x0000000000400616 <+32>: jle 0x400627 <fn1+49>
0x0000000000400628 <+38>: pop %rbp
0x0000000000400629 <+42>: retq
End of assembler dump.
```

Izvršavanje sledeće linije programa se radi korišćenjem komande `next`. Primer

korišćenja `next` komande je prikazan u listingu 3.8.

Listing 3.8: Primer korišćenja komande `next`

```
1 (gdb) r
2 The program being debugged has been started already.
3 Start it from the beginning? (y or n) y
4 Starting program: /master_examples/x86_arch/test
5 4
6 Breakpoint 1, fn1 (arg=0x7fffffffdbf4) at test.c:4
7 4  (*arg)++;
8 (gdb) next
9 5 if ((*arg) > 5)
```

3.3 Datoteke jezgra

Datoteka jezgra je snimak (eng. *snapshot*) memorije programa, registara i ostalih sistemskih informacija u trenutku neočekivanog prekida rada programa. Veoma važnu ulogu ima u procesu debagovanja programa sa uređaja sa ugrađenim računarom koji često pripadaju različitoj procesorskoj arhitekturi u odnosu na lični računar. Uređaji sa ugrađenim računarom obično imaju ograničene resurse, pa često na takvim platformama ne postoji debager. Najčešća procedura debagovanja ovakvih programa jeste prebacivanje datoteke jezgra i programa na lični računar na kome se analiza problema odvija koristeći debager.

Struktura datoteke jezgra

Datoteke jezgra sadrže razne informacije iz memorije programa uključujući i vrednosti lokalnih promenljivih, globalnih promenljivih, podatke lokalne za niti itd. Takođe sadrži vrednosti registara u trenutku prekida programa. U to spadaju i programski brojač i stek pokazivač.

Sadržaj datoteke jezgra je organizovan sekvencijalno sledećim redosledom:

Zaglavlje. Sadrži osnovne informacije o datoteci jezgra i pomeraje (eng. *offset*) kojima se lociraju ostale informacije iz nje.

strukture `ldinfo`. Definiše informacije relevantne za dinamički punilac[1] (eng. *dynamic loader*).

strukture `mstsave`. Definiše informacije relevantne za sistemske niti.

Korisnički stek. Sadrži kopiju korisničkog steka u trenutku neregularnog prekida rada programa.

Segment podataka. Sadrži kopiju segmenta podataka u trenutku neregularnog prekida rada programa.

Memorijski mapirani regioni [2] i strukture `vm_info`. Sadrži informacije o pomerajima i dužinama mapiranih regiona.

Generisanje datoteke jezgra

Datoteka jezgra se generiše ukoliko dođe do neregularnog prekida rada programa. To je podrazumevana akcija prilikom pojave signala operativnog sistema koji ukazuju na prekid rada programa. Datoteku jezgra generiše jezgro operativnog sistema. Jezgra *UNIX*-olikih operativnih sistema podrazumevano postavljaju dužinu datoteke jezgra na `0`. To je razlog zašto na našim sistemima nemamo datoteku jezgra nakon npr. prekidanja izvršavanja programa uz poruku *Segmentation fault*. Da bi se datoteka jezgra generisala, potrebno je eksplicitno promeniti dužinu datoteke jezgra koristeći komandu prikazanu u listingu 3.9.

Listing 3.9: Primer korišćenja komande `unlimited`

```
1 ulimit -c unlimited
```

Datoteka jezgra se može generisati i sa korisničkog nivoa koristeći debager *GNU GDB*, komandom `gcore`.

[1] Dinamički punilac preslikava relativne adrese programa u fizičke.

[2] Memorija programa je podeljena u nekoliko memorijskih regiona. Maprianjem memorijskih regiona se čuvaju relacije između virtualnih i fizičkih adresa programa.

3.4 Višearhitekturni debager - *Multiarch GNU GDB*

Multiarch GNU GDB je verzija debagera koja može da debaguje programe sa platformi različitih arhitektura. Alat na korisničkom nivou emulira instrukcije i registre ciljanih platformi. Potrebne su mu i deljene biblioteke za tu ciljanu platformu koje koristi program koji se debaguje. Podržan je samo podskup komandi standardne verzije debagera *GNU GDB*.

Prevođenje debagera *Multiarch GNU GDB*

Prvi korak je pozicioniranje u direktorijum sa izvornim kodom debagera je prikazan na 3.10.

Listing 3.10: Pozicioniranje u direktorijum *gdb*

```
1 cd gdb
```

Sledeći korak je konfiguracija direktorijuma u kome se prevođenje izvršava. Komande za kreiranje datoteke `Makefile` kojom odobravamo debagovanje svih podržanih arhitektura u alatu *GNU GDB* (kao npr. *MIPS32, MIPS64, ARM, AARCH64, x86_64, i386, SPARC*) su prikazane u listingu 3.11.

Listing 3.11: Konfiguracija direktorijuma za izgradnju debagera

```
1 mkdir build_multi
2 cd build_multi
3 ../configure --enable-targets=all
```

Prva komanda pravi direktorijum u kome se izgrađuje (prevodi) alat. Druga komanda vrši pozicioniranje trenutne putanje u taj direktorijum. Trećom komandom se kreira `Makefile`*. Pošto je prilikom konfiguracije navedena opcija* `-enable-targets=all` *time je omogućeno prevođenje alata za sve podržane arhitekture. Takođe je moguće konfigurisati prevođenje za samo određene arhitekture navodeći eksplicitno opciju kojom se navode ciljane procesorske arhitekture, npr.* `-enable-targets=mips-linux-gnu` *za arhitekturu* MIPS *ili* `-enable-targets=arm-linux-gnu` *za arhitekturu* ARM*. Osnovna razlika između* `Makefile`*-ova standardne i* `Multiarch` *verzije debagera je*

ta što je za standardni alat potrebno okruženje (deljene biblioteke, programski prevodilac, itd.) samo za arhitekturu domaćina. Za `Multiarch` *verziju alata potrebno je pronaći putanje do programskog prevodioca i deljenih biblioteka za ciljane platforme navedene opcijom* `-enable-targets=`*. Ukoliko konfiguracione skripte ne pronađu ciljano okruženje za neku od navedenih arhitektura, ta će biti ignorisana.*

Pokretanje debagera *Multiarch GNU GDB*

Ukoliko je trenutna putanja pozicionirana u direktorijum gde je izgrađen alat, pokretanje alata *Multiarch* GDB nad programom pod imenom *test* se vrši na isti način kao i standardna verzija debagera komandom prikazanom u listingu 3.12.

Listing 3.12: Pokretanje debagera

```
1 ./gdb/gdb test
```

Korišćenje debagera *Multiarch GNU GDB*

Spisak komandi koje se mogu koristiti korišćenjem *Multiarch* verzije debagera *GNU GDB* je limitiran. To se odnosi na izvršavanje programa koji se debaguje, jer program pripada drugačijem adresnom prostoru. Neke od komandi koje mogu biti upotrebljene su izlistavanje vrednosti registara programa, izlistavanje instrukcija, analiza stek okvira pozivanih funkcija. Najčešće se ova verzija alata koristi tako što se učita datoteka jezgra koja je generisana na ciljanoj platformi kada je program koji se debaguje neočekivano prekinuo sa radom. To obično prati analiza uzroka greške.

Primer učitavanja datoteke jezgra u debager *GNU GDB*

U listingu 3.13 je prikazano učitavanje datoteke jezgra programa čije izvršavanje je prekinuto od strane jezgra operativnog sistema. Komandom alata `core-file` se učitava datoteka jezgra u debager. Da bi se izazvalo generisanje datoteke jezgra u izvornom kodu programa napisanog u *C* ili *C++* programskom jeziku može se koristiti `abort()` funkcija iz standardne *C* biblioteke. U ovom primeru je pozvana ta funkcija koja je izazvala prekid programa uz signal `SIGABRT`. Tom prilikom jezgro operativnog sistema je napravilo datoteku jezgra sa imenom *core*.

Listing 3.13: Primer korišćenja komande `core-file`

```
1 (gdb) core-file core
2 [New LWP 20586]
3 [Thread debugging using libthread_db enabled]
4 Using host libthread\_db library "/lib/x86-linux-gnu/libthread_db.so.1".
5 Core was generated by './test.core'.
6 Program terminated with signal SIGABRT, Aborted.
7 #0 0x00007fb229164428 in raise (sig=6) at raise.c:54
```

U listingu 3.14 je prikazana upotreba komande debagera *GNU GDB* `bt`. Ona se koristi za izlistavanje pozivanih funkcija programa. Stek okviri programa koji su prikazani na primeru potvrđuju da je u funkciji `main()` došlo do poziva `abort()` funkcije. Što je izazvalo prekid rada programa.

Listing 3.14: Primer korišćenja komande `bt`

```
1 (gdb) bt
2 #0 0x00007fb229164428 in raise (sig=6) at raise.c:54
3 #1 0x00007fb22916602a in abort () at abort.c:89
4 #2 0x0000000000400605 in main () at tls.c:18
```

Analiza datoteke jezgra

U listingu 3.15 je prikazan primer korišćenja debagera *Multiarch GNU GDB* prilikom učitavanja datoteke jezgra generisane na uređaju sa ugrađenim računarom arhitekture *MIPS*. Uz datoteku jezgra učitavaju se izvršna datoteka i deljene biblioteke koje izvršna datoteka koristi na ciljanoj platfomi.

Listing 3.15: Primer učitavanja datoteke jezgra generisane na uređaju sa ugrađenim računarom arhitekture *MIPS*

```
(gdb) set solib-search-path ~/master_examples/mips_arch/
(gdb) core-file ~/master_examples/mips_arch/core
[New LWP 21808]
[New LWP 21813]
[New LWP 21810]
[New LWP 21809]
[New LWP 21811]
[New LWP 21812]
Core was generated by `example'.
Program terminated with signal SIGABRT, Aborted.
#0 0x00000000 in ?? ()
[Current thread is 1 (LWP 21808)]
```

Komanda `set solib-search-path` ***dir*** debagera *GNU GDB* korišćena u prethodnom primeru služi za navođenje direktorijuma iz kojeg debager treba da koristi deljene biblioteke za učitani program. Ta komanda je veoma bitna za debagovanje programa sa platformi drugih arhitektura, jer ukoliko putanja nije navedena debager koristi biblioteke sa računara domaćina.

U listingu 3.16 je prikazana upotreba komande `info registers`. *Multiarch GNU GDB* čita informaciju o arhitekturi programa koji se debaguje iz datoteke jezgra. Nakon toga čita vrednosti registara iz datoteke i ispisuje ih na standardni izlaz.

Listing 3.16: Primer korišćenja komande `info registers`

```
(gdb) info registers
          zero      at      v0      v1      a0      a1
R0    00000000 00005530 00000000 00005530 00000000 00005530
          t0      t1      t2      t3      t4      t5
R8    00000000 00000000 00000000 7fcf4ca0 00000000 00000000
          s0      s1      s2      s3      s4      s5
R16   00000000 7719a684 00000000 00000000 00000000 00000002
          k0      k1      gp      sp      s8      ra
R24    00000000 771c6490 00000000 77160634  00000000 7fcf4c20
          sr      lo      hi      bad     cause   pc
      00000000 00000000 77165000 00000000 00b24608 00000000
```

Glava 4

Promenljive lokalne za niti

Ranije je bilo podrazumevano da procesi imaju svoj memorijski prostor i da se se svi poslovi procesa izvršavaju sekvencijalno. Novi operativni sistemi obično podržavaju *niti* (eng. *threads*). Niti predstavljaju delove jednog procesa koji mogu obavljati razne poslove tog procesa koristeći njegove resurse. Svaka nit poseduje svoje registre, memoriju i stek. Pisanje višenitnih programa predstavlja fudamentalan i neizbežan koncept savremenog programiranja, jer niti podižu performanse i brzinu softvera. Većina kompleksnih korisnički programa je napisana uz korišćenje niti. Često su niti sakrivene od strane krajnjeg programera, npr. na vebu se *JavaScript* izvršava u višenitnom okruženju, ali programeri koji pišu u *JavaScript* programskom jeziku ne rade direktno sa nitima [20]. Promenljive lokalne za niti su važan mehanizam koji se koristi u okviru višenitnih aplikacija programskih jezika *C* i *C++*. One omogućavaju definisanje promenljivih koje imaju različitu vrednost u svakoj niti. Jedan primer je promenljiva koja jednoznačno identifikuje grešku koja je nastala u programu. Greška može biti izazvana u svakoj posebnoj niti iz različitog razloga, te promenljiva koja je opisuje treba da ima različitu vrednost u svakoj niti.

4.1 Uvod u *TLS*

Povećanje korišćenja niti u programiranju dovelo je do potrebe programera za boljim načinom rukovanja podacima lokalnim za niti. Skup interfejsa za rukovanje nitima *POSIX* [13] omogućava kreiranje istog `void *` objekata posebno za svaku nit. Taj interfejs je nezgrapan za korišćenje, jer objekat mora biti dinamčki alociran u vremenu izvršavanja programa. Kada se objekat više ne koristi, mora da se oslobodi. Celokupan proces zahteva dosta posla programera. Pored toga, podložan je

greškama. Iz tih razloga postoji potreba za efikasnijim rešenjem. Da bi se odgovorilo na opisane probleme, odlučeno je da se programski jezici prošire i tako prepuste težak posao programskim prevodiocima.

Za jezike *C* i *C++* ključna reč `__thread` se koristi za deklaraciju i definiciju promenljivih lokalnih za niti. Neki primeri deklaracija promenljivih lokalnih za niti su prikazani u listingu 4.1. Od *C++11* verzije jezika koristi se i ključna reč `thread_local`. Razlika između `__thread` i `thread_local` je ta što `__thread` nikada nije postao deo standarda *C* i *C++*, dok `thread_local` jeste. Što se tiče implementacije, ove dve ključne reči imaju samo jednu rezliku. Ta razlika je što programski prevodilac prilikom kreranja promenljivih deklarisanih korišćenjem `thread_local` za svaku referencu na takvu promenljivu kreira funkcijski poziv na funkciju u kojoj se vrši inicijalizacija promenljive. Za promenljive deklarisane ključnom reči `__thread` inicijalizacija se vrši unutar funkcije gde je kreirana. U nastavku teksta biće reči o implementaciji i izmenama koje su potrebne za korišćenje `__thread` varijante.

Listing 4.1: Deklaracije promenljivih lokalnih za niti

```
1 __thread int j;
2 __thread struct state s;
3 extern __thread char *p;
```

Prednost korišćenja promenljivih lokalnih za niti nije ograničena samo na korisničke programe. Okruženje izvršavanja programa, između ostalih standardna biblioteka, takođe koristi pogodnosti ovog mehanizma. Npr. globalne promenljive `errno`, koje jednoznačno označavaju nastalu grešku u programu, moraju biti lokalne za niti, jer u različitim nitima može doći do različitih grešaka. Napomenimo da navođenje `__thread` pri deklaraciji ili definiciji neke automatske promenljive nema smisla i to nije dozvoljeno, jer automatske promenljive su uvek lokalne za niti. Statičke promenljive funkcija su takođe kandidati za korišćenje *TLS* promenljivih.

Osnovne operacije nad promenljivama koje su lokalne za niti su intuitivne. Npr. adresni operator vraća adresu promenljive za trenutnu nit. Memorija alocirana za promenljivu lokalnu za nit u dinamički učitanom modulu se oslobađa kada se taj modul oslobodi iz memorije.

Implementacija ovog mehanizma zahteva promenu okruženja izvršavanja programa. Format izvršnih datoteka je proširen kako bi definisao promenljive lokalne za niti odvojeno od standardnih promenljivih, što je opisano u okviru poglavlja 4.2. Dinamički punilac je nadograđen kako bi ispravno inicijalizovao te nove sekcije.

Standardna biblioteka koja rukuje nitima je promenjena kako bi alocirala nove podatke lokalne za niti za svaku novu nit. U okviru poglavlja 4.3 je dat detaljan opis izvršavanja programa koji sadrže promenljive lokalne za niti. Pokretanje i priprema izvršavanja procesa koji sadrže *TLS* promenljive su opisani u poglavlju 4.4. Referisanje *TLS* promenljive se ne odvija na standardan način, već koristeći *TLS* modele pristupa opisane u poglavlju 4.5.

4.2 Definisanje novih podataka u formatu ELF

Izmene u formatu izvršnih datoteka, potrebne za emitovanje *TLS* objekata su minimalne. Umesto smeštanja inicijalizovanih promenljivih u sekciju `.data` ili neinicijalizovanih promenljivih u `.bss` sekciju, *TLS* promenljive se smeštaju u `.tdata` i `.tbss` sekcije [19]. Nove sekcije se od originalnih razlikuju u samo jednom dodatnom sekcijskom flegu. Tabela 4.1 prikazuje nove sekcije. Jedina razlika u odnosu na sekcije podataka koje ne koriste višenitno okruženje jeste fleg `SHF_TLS`.

Ime polja	*Vrednosti polja u okviru `.tbss` sekcije*	*Vrednosti polja u okviru `.tdata` sekcije*
`sh_name`	`.tbss`	`.tdata`
`sh_type`	`SHT_NOBITS`	`SHT_PROGBITS`
`sh_flags`	`SHF_ALLOC + SHF_WRITE + SHF_TLS`	`SHF_ALLOC + SHF_WRITE + SHF_TLS`
`sh_addr`	Virtualna adresa sekcije	Virtualna adresa sekcije
`sh_offset`	0	Pomeraj inicijalizacione slike
`sh_size`	Veličina sekcije	Veličina sekcije
`sh_link`	`SHN_UNDEF`	`SHN_UNDEF`
`sh_info`	0	0
`sh_addralign`	Poravnanje sekcije	Poravnanje sekcije
`sh_entsize`	0	0

Tabela 4.1: Tabela vrednosti polja koji opisuju nove TLS sekcije

Imena novih sekcija, kao ni ostalih u formatu *ELF*, nisu bitna. Dinamički povezivač[1] (eng. *dynamic linker*) će svaku sekciju tipa `SHT_PROGBITS` sa dodatnim flegom `SHF_TLS` tretirati kao `.tdata`, dok će sekcije tipa `SHT_NOBITS` sa dodatnim `SHF_TLS`

[1] Dinamički povezivač povezuje sve objektne module u izvršni program.

Polje	*Vrednost*
`p_ptype`	`PT_TLS`
`p_offset`	Pomeraj TLS inicijalizacione slike
`p_vaddr`	Virtualna adresa TLS inicijalizacione slike
`p_paddr`	Rezervisano
`p_filesize`	Veličina TLS inicijalizacione slike
`p_memsz`	Ukupna veličina TLS šablona
`p_flags`	`PF_R`
`p_aligment`	Poravnanje TLS šablona

Tabela 4.2: Tabela koja predstavlja nove vrednosti programskog zaglavlja

tretirati kao `.tbss` sekciju. Odgovornost proizvođača ovakvih sekcija, obično programskih prevodioca, je da pravilno generiše sva polja prikazana u tebeli 4.1.

Za razliku od standardnih `.data` sekcija, program koji se izvršava ne koristi `.tdata` sekciju direktno. Ta sekcija može da bude modifikovana u vreme pokretanja programa, od strane dinamičkog punioca. Prilikom pokretanja programa, prva akcija jeste realokacija koju izvršava dinamički punilac. Nakon toga podaci lokalni za niti se smeštaju u deo koji se naziva inicijalizaciona slika (eng. *initialization image*) i ona se ne modifikuje više nakon toga. Za svaku nit, uključujući i inicijalnu, nova memorija se alocira na mestu gde se kopira inicijalizaciona slika. Ovim se omogućava da svaka nit ima identičan početni sadržaj. Kako ne postoji samo jedna adresa koja ukazuje na simbol *TLS* promenljive, standardna tabela simbola ne može biti iskorišćena. U izvršnoj datoteci polje `st_value` ne sadrži apsolutnu adresu promenljive prilikom izvrašavanja programa, jer apsolutna adresa nije poznata prilikom prevođenja programa. Iz tog razloga je uveden novi tip simbola (`STT_TLS`). Svaki simbol koji referiše na *TLS* ima takav tip simbola. Izvršne datoteke u polju `st_value` imaju vrednost pomeraja promenljive u *TLS* inicijalizacionoj slici. Prilikom standardnih realokacija ne sme se pristupati simbolima tipa `STT_TLS`. Takvim simbolima se može pristupati samo prilikom procesa realokacija uvedenih za rukovanje promenljivama lokalnim za niti. Takođe, realokacije namenjene rukovanju *TLS* promenljivama ne smeju pristupati simbolima ostalih tipova.

Da bi dinamički povezivač mogao da izvrši inicijalizaciju inicijalizacione slike, njena pozicija prilikom izvršavanja programa mora biti zapisana u programskom zaglavlju. Originalno zaglavlje programa nije moglo da bude iskorišćeno, pa je novo, prošireno, zaglavlje definisano. Proširenje koje opsuje *TLS* inicijalizacionu sliku je prikazano u tabeli 4.2.

Svaka *TLS* promenljiva je identifikovana pomoću pomeraja od početka *TLS* sekcije. U memoriji, `.tbss` sekcija je alocirana odmah nakon `.tdata` sekcije. Nijedna virtualna adresa ne može biti izračunata prilikom povezivanja (eng. *link time*).

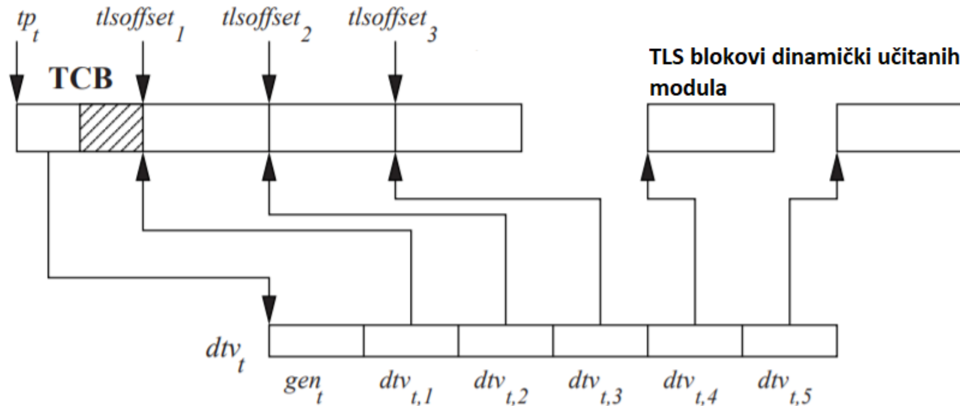
4.3 Rukovanje TLS promenljivom tokom izvršavanja programa

Obrada podataka lokalnih za niti nije prosta kao obrada običnih podataka. Standardni pristup kreiranja i korišćenja segmenta podataka od strane procesa se ne može iskoristiti. Umesto toga, nekoliko kopija jednog istog podatka mora biti kreirano, svi inicijalizovani iz iste inicijalizacione slike.

Mehanizami koji omogućavaju izvršavanje programa bi trebalo da zaobiđu kreiranje podataka lokalnih za niti ako to nije neophodno. Npr. samo jedna od više kreiranih niti koje čine taj određeni proces može koristiti učitani modul. Dosta memorije i vremena bi se gubilo prikom alociranja tih podataka za sve niti. Za ovakve situacije je poželjan lenji metod.

Standardna pravila pretrage simbola (eng. *symbol lookup*) u izvršnim datotekama sa *ELF* formatom ne mogu biti primenjena na simbole koji opisuju promenljive lokalne za niti. Standardan proces povezivanja ne može biti primenjen kada se u programu koriste *TLS* promenljive.

Promenljiva lokalna za nit se identifikuje referencom na objekat i pomerajem te promenljive unutar prostora lokalnog za tu nit. Da bi se ove vrednosti mapirale u virtualne adrese, mehanizam izvršavanja programa zahteva nove strukture podataka, tj. stukture podataka koje se ne koriste ukoliko nemamo višenitno izvršavanje i promenljive lokalne za niti. One mapiraju referencu objekta u neku adresu u određenom prostoru lokalnom za tu nit. Da bi se to omogućilo definisane su dve varijante struktura podataka. Različite arhitekture procesora mogu odabrati jedan od ova dva pristupa, ali to mora biti propisano *ABI*-jem za tu arhitekturu. Arhitekture, bilo da koriste prvu ili drugu varijantu, rezervišu jedan registar koji pokazuje na prostor lokalan za niti. Takav registar nazivamo *nitni registar*. Za procesorsku arhitekturu *Intel x86-64* to je registar `fs`. Jedan od razloga za korišćenje druge varijante modela je istorijski. Neke arhitekture su dizajnirale sadržaj nitne memorije na koju pokazuje nitni registar tako da nisu kompatibilne za korišćenje prve varijante *TLS* strukture.



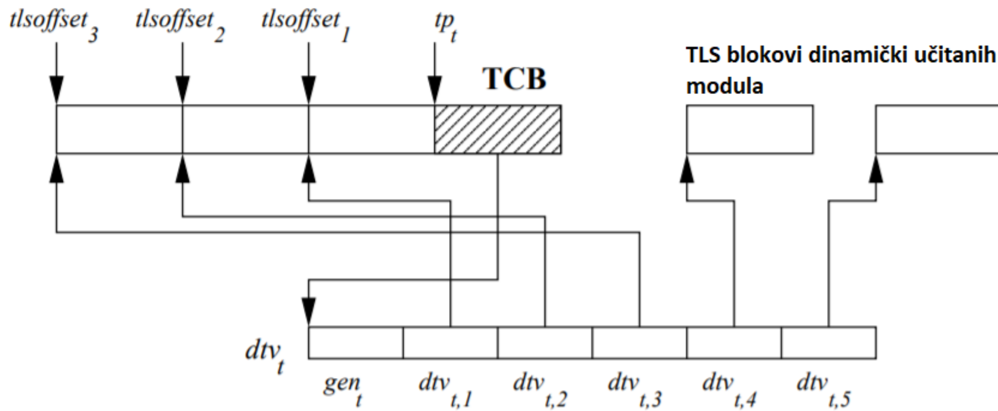
Slika 4.1: TLS struktura podataka (*varijanta 1*)

Na slici 4.1 je prikazan primer prve varijante *TLS* strukture podataka. Nitni registar za nit `t` je označen sa tp_t. Početak memorije svake niti je određen *nitnim kontrolnim blokom* TCB (eng. *Thread Control Block*). Nitni registar pokazuje na nitni kontrolni blok, koji na pomeraju nula sadrži pokazivač na *nitni dinamički vektor* dtv_t za tu određenu nit. Nitni dinamički vektor predstavlja niz koji sadrži informacije o podacima lokalnim za niti različitih učitanih modula. Memoriju lokalnu za nit u okviru jednog učitanog modula nazivamo *TLS blok*. Ukoliko je modul inicijalno dostupan, tj. učitan u program prilikom pokretanja, element nitnog dinamičkog vektora pokazuje na fiksni pomeraj podataka lokalnog za tu nit u okviru tog modula. Ukoliko se modul dinamički učitava u program element dinamičkog nitnog vektora pokazuje na memoriju lokalnu za nit u okviru tog modula koji se učitava. *TLS šablon* (eng. *template*) je sastavljen od svih *TLS* blokova koji su učitani prilikom pokretanja procesa.

Na slici 4.1 nitni dinamički vektor dtv_t kao svoje prvo polje sadrži generacioni broj gen_t koji se koristi pri promeni veličine nitnog dinamičkog vektora i alokacije *TLS* blokova. Ostala polja sadrže pokazivače na *TLS* blokove za različite učitane module. *TLS* blokovi za module koji se učitavaju pri pokretanju programa su smeštena direktno nakon *TCB* bloka i stoga ima arhitekturalno specifičan, fiksni pomeraj od adrese na nitni pokazivač. Za sve inicijalno dostupne module pomeraj svakog *TLS* bloka, uz to i pomeraj *TLS* promenljive, u odnosu na *TCB* mora biti fiksan nakon pokretanja programa.

Druga varijanta *TLS* strukture podataka ima sličnu strukturu kao prva. Pri-

kazana je na slici 4.2. Jedina razlika je ta što nitni pokazivač pokazuje na nitni kontrolni blok za koji je nepoznata veličina i sadržaj. Negde svakako taj nitni kontrolni blok sadrži pokazivač na dinamički nitni vektor, ali nije navedeno gde. To je kontrolisano od strane mehanizma za izvršavanje programa.



Slika 4.2: TLS struktura podataka (*varijanta 2*)

U trenutku pokretanja programa *TCB*, zajedno sa dinamičkim nitnim vektorom, se kreira za glavnu nit. Pozicija tog *TLS* bloka, za svaki pojedinačan modul, se računa koristeći arhitekturalno specifične formule, zasnovane na veličini i poravnanju *TLS* bloka propisanih *ABI*-jem za tu specifičnu procesorsku arhitekturu.

4.4 Pokretanje i izvršavanje procesa

Za programe koji koriste *TLS* promenljive mehanizam operativnog sistema koji služi za pokretanje procesa mora podesiti memoriju za inicijalnu nit pre nego što preda kontrolu tog procesa drugim mehanizmima. *Statičko povezivanje* predstavlja proces kreiranja izvršne datoteke povezivanjem objektnih modula odmah nakon kompilacije, dok se *dinamičko povezivanje* vrši tokom izvršavanja programa. Podrška za korišćenje *TLS* promenljive u statičko povezanim programima je limitirana. Neke procesorske arhitekture (kao npr. *IA-64*) ne definišu uopšte statičko povezivanje. Neke druge platforme obeshrabruju korišćenje statičkog povezivanja pružanjem samo određenog broja funkcionalnosti. Rukovanje *TLS* promenljivom u statičko povezanom programu je prosto, zbog postojanja samo jednog modula, tj. samo taj program.

Zanimljiviji je slučaj rukovanja *TLS* promenljivom u dinamičko povezanom programu. U ovom slučaju dinamički povezivač mora uključiti podršku za rukovanje takvih segmenata podataka. U nastavku teksta je opisano učitavanje i pokretanje dinamičkog koda.

Da bi se podesila memorija lokalna za nit, dinamički povezivač čita sve potrebne informacije o svakom modulu, i o njegovim *TLS* blokovima, iz `PT_TLS` polja tabele *TLS* programskog zaglavlja prikazanog u tabeli 4.2. Informacije o svim modulima moraju biti prikupljene. Ovaj proces se ostvaruje koristeći povezanu listu čiji element sadrži:

- pokazivač na *TLS* inicijalizacionu sliku,
- veličinu *TLS* inicijalizacione slike,
- *TLS* pomeraj ($tlsoffset_m$) za module `m`,
- fleg koji daje informaciju o tome da li modul koristi statički *TLS* model.

Ove informacije će biti proširene kada se učitaju dodatni dinamički moduli. Te informacije se koriste od strane standardne biblioteke za rukovanje nitima prilikom podešavanja *TLS* blokova za novokreiranu nit.

Promenljiva unutar lokalnog prostora za nit, *TLS*, je identifikovana po referenci na modul i pomeraju u okviru tog *TLS* bloka. Ukoliko imamo strukturu podataka dinamički nitni vektor, možemo definisati referencu na neki modul kao ceo broj (eng. *integer*), počevši od broja `1`. To može biti korišćeno kao indeks u dtv_t nizu. Identifikacione brojeve koji svaki modul dobija određuje mehanizam izvršavanja programa, obično neki modul standardne biblioteke za rukovanje nitima. Samo izvršna datoteka dobija fiksan broj, `1`, a svi ostali učitani moduli dobijaju različite brojeve.

Računanje specifične adrese neke *TLS* promenljive je prosta operacija koja može biti izvršena ukoliko je programski prevodilac koji je preveo kôd korsitio varijantu 1 *TLS* strukture podataka. Ali to ne može biti tako lako odrađeno u kodu koji je generisan od strane kompajlera za procesorske arhitekture koje koriste varijatnu 2 *TLS* strukture podataka. Umesto toga, definisana je funkcija `__tls_get_addr()`, koja je prikazana u listingu 4.2.

Listing 4.2: Implementacija funkcije `__tls_get_addr()`

```
1  void *
2  __tls_get_addr (size_t m, size_t offset)
3  {
4    char *tls_block = dtv[thread_id][m];
5    return tls_block + offset;
6  }
```

Smeštanje vektora `dtv[thread_id]` u memoriju je arhitekturalno specifično. Sa `m` se označava identifikacioni broj modula, koji mu je dodeljen od strane dinamičkog punionca prilikom učitavanja. Korišćenje `__tls_get_addr()` funkcije ima i dodatne prednosti kako bi se olakšala implementacija dinamičkog modela, gde je alokacija nekog *TLS* bloka odložena do njegove prve upotrebe. Da bi se to podržalo potrebno je popuniti `dtv[thread_id]` vektor sa specijalnom vrednošću kojom će biti prepoznate situacije kada je taj vektor trenutno prazan, tj. da u datom trenutku određeni blok nije u upotrebi. To je omogućeno malom promenom u izvornom kodu `__tls_get_addr()` funkcije koja je prikazana u listingu 4.3.

Listing 4.3: Promena implementacije funkcije `__tls_get_addr()`

```
1  void *
2  __tls_get_addr (size_t m, size_t offset)
3  {
4    char *tls_block = dtv[thread_id][m];
5    if (tls_block == UNALLOCATED_BLOCK)
6      tls_block = dtv[thread_id][m] = allocate_tls(m);
7    return tls_block + offset;
8  }
```

Funkcija `allocate_tls()` određuje memorijske zahteve *TLS* modula `m` i inicijalizuje ga ispravno. Postoje dva tipa podataka: inicijalizovani i neinicijalizovani. Inicijalizovani podaci moraju biti kopirani iz inicijalizacione nitne slike, podešenih prilikom učitavanja modula `m`. Neinicijalizovani podaci se postavljaju na vredosti `0`. Primer implementacije funkcije `allocate_tls()` je prikazan u listingu 4.4.

Listing 4.4: Implementacija funkcije `allocate_tls()`

```
void *
allocate_tls (size_t m)
{
  void *mem = malloc (tlssize[m]);
  memset (memcpy (mem, tlsinit_img[m], tlsinit_size[m]), '\0',
          tlssize[m] - tlsinit_size[m]);
  return mem;
}
```

Vrednosti `tlssize[m]`, `tlsinit_size[m]` i `tlsinit_img[m]` su poznate nakon učitavanja modula `m`. Primetimo da se ista inicijalizaciona slika `tlsinit_img[m]` koristi za sve niti modula, bilo kada da se one kreiraju. Novonapravljena nit ne nasleđuje podatke od svog roditelja (eng. *parent*), već dobija samo kopiju inicijalnih podataka.

4.5 TLS modeli pristupa

Svako referisanje *TLS* promenljive prati jedan od modela pristupa. Modeli pristupa se dele na dva tipa: dinamički i statički. Različite arhitekture *ABI*-jem propisuju koji od modela pristupa će koristiti kao podrazumevani. Postoje različiti modeli pristupa, a četiri najpoznatija su opisana u nastavku teksta.

Generalni dinamički TLS model

Generalni model pristupa *TLS* promenljivoj dozvoljava referisanje svih *TLS* promenljivih, bilo da je to iz deljene biblioteke ili dinamičke izvršne datoteke. Ovaj model takođe podržava odloženo alociranje *TLS* bloka do trenutka kada se prvi put taj blok referiše iz specifične niti.

Lokalni dinamički TLS model

Lokalni dinamički model pristupa *TLS* promenljivoj predstavalja optimizaciju generalnog dinamičkog modela. Programski prevodilac može odrediti da je neka promenljiva definisana samo lokalno, ili zaštićeno (eng. *protected*) u objektu koji je napravljen u programu. U tom slučaju, programski prevodilac daje instrukcije dinamičkom povezivaču da statički poveže dinamički *TLS* pomeraj i da korsiti ovaj

model. Iz tog razloga predstavlja hibridnu kombinaciju statičkog i dinamičkog modela pristupa, te prema tome predstavlja model koji diže performanse u odnosu na generalni dinamički model. Samo jedan poziv `tls_get_addr()` po funkciji je potreban za određivanje adrese $dtv_{0,m}$[2].

Statički TLS model sa dodeljenim pomerajima

Ovaj model pristupa dopušta referisanje samo na one *TLS* promenljive koje su dostupne kao deo inicijalnog statičkog *TLS* šablona. U ovom modelu, relativni pomeraj pokazivača na nit date promenljive `x` je smešten u polju globalne tabele sa pomeraijma (eng. *Global offset table-GOT*) za promenljivu `x`.

Deljene biblioteke uobičajeno koriste dinamički model pristupa, jer statičkim modelom mogu referisati samo na određeni broj *TLS* promenljivih.

Statički TLS model

Ovaj model pristupa dopušta referisanje samo na one *TLS* promenljive koje su dostupne kao deo *TLS* bloka od te dinamičke izvršne datoteke. Dinamički povezivač računa relativni pomeraj pokazivača na nit statički, bez potrebe za dinamičkim realokacijama ili dodatnih inforamcija iz *GOT*-a. Ovaj model pristupa ne dopušta referisanje *TLS* promenljivih izvan te dinamički povezane izvršne datoteke.

[2] Prvi broj (`0`) ukazuje na identifikacioni broj niti, a drugi broj (`m`) ukazuje na identifikacioni broj modula.

Glava 5

Implementacija rešenja

TLS mehanizam može biti različito implementiran za različite procesorske arhitekture. *TLS* promenljiva se može smeštati u posebne registre, u određene delove memorije itd. Debager mora ispravno raditi na svim arhitekturama, a arhitekturalne razlike anulirati na neki način. Implementacija realizovanog rešenja za proširenje alata se upravo i zasniva na prethodnoj činjenici. Zbog toga su detaljno istražene implementacije *TLS* mehanizma raznih arhitektura, kako u funkcijama biblioteke *GNU C*, poznatije kao *glibc* [17], tako i karakteristike koje su *ABI*-jem propisane.

U radu je uspešno poboljšana verzija debagera *Multiarch*. Izvorni kôd alata i instrukcije za prevođenje i upotrebu debagera sa pobljšanjem za pristupanje vrednosti *TLS* promenljive se mogu pronaći na strani [8]. Ukoliko je program koji je učitan u alat *Multiarch GNU GDB* iste arhitekture kao arhitektura domaćina, zadržava se način čitanja vrednosti *TLS* promenljive kao u slučaju debagera *GNU GDB* arhitekture domaćina, jer je ta funkcionalnost već implementirana. Rešenje obuhvata arhitekture procesora *MIPS*, *ARM*, *AARCH64*, *i386* i *PowerPC*. Kao odgovor na komentare *GNU* zajednice razvijeno je i alternativno rešenje koje je takođe opisano u nastavku teksta.

Još jedno rešenje predstavlja projekat *Infinity* [9] koji razvija kompanija *Red Hat* [15]. Taj projekat ima ideju da obuhvati rešenja za razne probleme o debagovanju niti, ne samo obradu mehanizma *TLS*. Prednost rešenja koji se predstavlja u ovom master radu je što je cela funkcionalnost prepuštena samom debageru *GNU GDB*, te nema potrebe za instalacijom ili prevođenjem eksternih projekata, čime se dobija na uštedi korisničkog napora pri korišćenju debagera.

5.1 Detalji implementacije

U nastavku teksta opisani su detalji implementacije poboljšanja debagera.

Izmena sistema izgradnje alata

Datoteke *gdb/configure* i *gdb/Makefile.in* su zaduženi za izgradnju alata. U najopštijem slučaju na *UNIX*-olikim operativnim sistemima, *configure* skripta je zadužena za pravljenje konačnog *Makefile* od ulaznog *gdb/Makefile.in*.

Izvorni kôd koji u sebi sadrži funkcije za čitanje *TLS* promenljive (*gdb/linux-thread-db.c*) se prevodio samo za standardnu verziju *GNU GDB* debagera, pa stoga u proširenje za *Multiarch GNU GDB* takođe treba uvrstiti izmenu u pomenutoj datoteci prilikom prevođenja. Pored toga, postojeće funkcije koje su zadužene za pristupanje vrednosti *TLS* promenljivih je trebalo izmeniti te je u projekat dodat direktorijum *gdb/glibc-dep/* koji sadrži funkcije koje barataju sa mehanizmom *TLS* različite procesorske arhitekture od arhitekture domaćina. Da bi se te datoteke prevodile samo u slučaju *Multiarch* verzije definisan je makro pod imenom `CROSS_GDB` koji se definiše prilikom kreiranja datoteke *Makefile*. Da bi se ispratila konvencija pisanja skripti za izgradnju alata, u direktorijum *gdb/config/* je dodata datoteka *glibc.mh* koja definiše spisak objektnih datoteka za pristupanje vrednosti *TLS* promenljive u tom posebnom slučaju i takođe u toj datoteci se definiše gore pomenuti makro `CROSS_GDB`.

Sadržaj datoteke *gdb/config/glibc.mh* je prikazan u listingu 5.1.

Listing 5.1: Sadržaj datoteke *gdb/config/glibc.mh*

```
1  # GLIBC fragment comes in here
2  GLIBCFILES = td_symbol_list.o \
3          fetch-value.o gdb_td_ta_new.o \
4          td_thr_tlsbase.o td_thr_tls_get_addr.o \
5          td_ta_map_lwp2thr.o native_check.o
6  INTERNAL_CFLAGS += -DCROSS_GDB
```

Da bi sadržaj iz novokreirane datoteke *gdb/config/glibc.mh* bio ispisan u krajnji *Makefile* trebalo je uvesti promenljivu koja proverava da li je prilikom navođenja opcija *gdb/configure* skripti navedena opcija *–enable_ targets*, koja zapravo označava da korisnik ima nameru da alat prevede kao *Multiarch* verziju. Vrednost promenljive

`cross_makefile_frag` u *gdb/configure* skripti postaje putanja do datoteke *gdb/config/glibc.mh* ukoliko je pomenuta opcija navedena, a u suprotnom ona ostaje prazna.

Deo koda u *gdb/configure* skripti kojim se to postiže je prikazan u listingu 5.2.

Listing 5.2: Izmena datoteke *gdb/configure*

```
1 if test "${gdb_native}" = "no" ||
2     test "${enable_targets}" != ""; then
3   cross_makefile_frag=${srcdir}/config/glibc.mh
4 else
5   cross_makefile_frag=/dev/null
6 fi
```

Implementacija funkcija za pristupanje TLS promenljivoj

U direktorijumu *gdb/glibc-dep/* se nalazi srž funkcionalnosti čitanja vrednosti *TLS* promenljive iz datoteke jezgra sa platformi drugih procesorskih arhitektura. Kako standardna C biblioteka već ima implementiranu svaku arhitekturalnu zavisnost u vezi mehanizma *TLS* za svaku procesorsku arhitekturu podržanu u alatu *GNU GDB*, potrebno je na neki način izopštiti tu funkcionalnost iz same biblioteke unutar alata. Preciznije, za baratanjem nitima *GNU GDB* koristi *libthread_db* biblioteku iz standardne biblioteke. Funkcije `td_ta_new()`, `td_thr_tls_get_addr()` i `td_th_tlsbase()`, koje služe za pristupanje *TLS* promenljivama, očekuju da se na arhitekturi domaćina prirodno barata sa programima iste arhitekture. Izbegavanje modifikacije *libthread_db* biblioteke moguće je tako što se pomenute funkcije implementiraju u *GNU GDB*-u, na isti način kao u biblioteci *glibc*, prilikom čega se sve arhitekturno zavisne vrednosti promenljivih i makroa koje primaju te funkcije postave na vrednosti koje bi trebalo da imaju u slučaju da je ta ciljna arhitektura zapravo arhitektura domaćina.

Prvi korak ovog dela implementacije je izmeštanje funkcija iz standardne biblioteke unutar GDB projekta. Unutar *gdb/glibc-dep/* je kreiran direktorijum *nptl_db* koji sadrži pomenute funkcije. Zapravo, ukoliko alat koristi bilo koju verziju standardne biblioteke do *2.22* funkcije iz *nptl_db* nisu ni potrebne za pristupanje vrednosti *TLS* promenljive. Sve potrebne informacije za pristupanje vrednosti *TLS* promenljive se mogu pročitati iz same datoteke jezgra. Ukoliko alat koristi verziju standardne biblioteke 2.22 i ili veću, funckcije iz *nptl_db* direktorijuma su neophodne kako bi se anulirale novonastale izmene prilikom baratanja mehanizmom *TLS*. Ta

funkcionalnost je dodata u funkciji `gdb_td_thr_tlsbase()` koja je dodata u datoteku `gdb/glibc-dep/nptl_db/td_thr_tlsbase.c`. Sve funkcije koje bi trebalo da emuliraju ponašanje *TLS* funkcija standardne biblioteke koje su dodate u projekat *GNU GDB* imaju prefiks u imenu `gdb_`. Npr. funkcija unutar debagera koja odgovara funckiji `td_ta_new()` je `gdb_td_ta_new()`. Pomenuta funkcija je promenjena da bi alat *GNU GDB* imao informaciju o tačnoj verziji standardne biblioteke koju je program koji se debaguje koristio na platformi na kojoj se izvršavao, jer kao što je napomenuto od verzije 2.22 *TLS* promenljiva se drugačije čita. Nije novo da za debagovanje višenitnih programa debagerom *GNU GDB*, čak i upotrebom standardne verzije, je neophodno da *GNU GDB* koristi istu veriziju biblioteke *libthread_db* kao i program koji se debaguje. To znači da ako je program preveden sa verzijom *2.x* standardne biblioteke, alat mora koristiti istu verziju bibilitoeke (*2.x*) tokom debagovanja. To se postiže eksplicitnim navođenjem putanje do biblioteke *libthread_db* komandom `set libthread-db-search-path`. Ukoliko se verzije biblioteke *libthread_db* ne poklapaju debager će ispisati upozorenje.

Takođe u `gdb_td_ta_new()` funkciji se poziva funkija `init_target_dep_constants()`, zadužena za inicijalizaciju arhitekturalno zavisnih osobina o mehanizmu *TLS*.

Deo kojim se različite osobine inicijalizuju za arhitekture *MIPS* i *ARM* u funkciji `init_target_dep_constants()` je prikazan u listingu 5.3.

Listing 5.3: Inicijalizacija arhitekturalno zavisnih osobina o TLS

```
case bfd_arch_mips:
  tls_tcb_at_tp = 0;
  tls_dtv_at_tp = 1;
  forced_dynamic_tls_offset = -2;
  no_tls_offset = -1;
  tcb_alignment = 16;
  break;
case bfd_arch_arm:
  tls_tcb_at_tp = 0;
  tls_dtv_at_tp = 1;
  forced_dynamic_tls_offset = -2;
  no_tls_offset = -1;
  tcb_alignment = 0;
  break;
```

U ovom konkretnom slučaju sve osobine mehanizma *TLS* su identične, osim poravnanja nitnog kontrolnog bloka.

Funkcija `native_check()` definisana u datoteci *gdb/glibc-dep/native-check.c* daje informaciju da li se *Multiarch* verzijom alata debaguje program arhitekture domaćina. Ako je to slučaj, pristupanje vrednosti *TLS* promenljive se vrši kao do sada, uz pomoć funkcija iz standardne C biblioteke. Ukoliko to nije slučaj, pozivaju se novokreirane funkcije iz direktorijuma *gdb/glibc-dep/*.

Cela funkcionalnost se zapravo dodaje u datoteku *gdb/linux-thread-db.c*, preciznije, u funkciju koja je zadužena za pristupanje vrednosti *TLS* promenljive. Izmena nije direktno u funkciji `thread_db_get_thread_local_address` koja vraća adresu *TLS* promenljive. Naime, promena je izvedena u funkciji `try_thread_db_load_1()` koja uspostavlja konekciju između alata *GNU GDB* i biblioteke *libthread_db*. Promenjeni su pokazivači na *TLS* funkcije u slučaju datoteka jezgra drugačijih arhitektura od arhitekture domaćina.

Deo koda koji implementira pomenutu funkcionalnost je prikazan u listingu 5.4.

Listing 5.4: Postavljanje *callback* funkcija za drugačijih arhitektura od arhitekture domaćina

```
1  #ifdef CROSS_GDB
2   if (native_check(arch) != 0) {
3    info->td_thr_tls_get_addr_p=gdb_td_thr_tls_get_addr;
4    info->td_thr_tlsbase_p=gdb_td_thr_tlsbase;
5   } else {
6    //if it's host we want to keep old way of counting tls address
7    TDB_DLSYM (info, td_thr_tls_get_addr);
8    TDB_DLSYM (info, td_thr_tlsbase);
9   }
10 #else
11   TDB_DLSYM (info, td_thr_tls_get_addr);
12   TDB_DLSYM (info, td_thr_tlsbase);
13 #endif
```

Iako je u pitanju *Multiarch* verzija alata, ako je datoteka jezgra kreirana na platformi domaćina zadržava se normalno pristupanje mehanizma *TLS*.

5.2 Alternativno rešenje

Drugi način implementacije je modifikacija funkcija biblioteke *libthread_db* eksplicitno u standardnoj biblioteci, modifikujući ih tako da rade sa različitim arhitekturama. Naime, formula koja služi za pristupanje *TLS* promenljive se računala

u vremenu prevođenja biblioteke, sada bi se računala u vremenu izvršavanja programa, i u zavisnosti od arhitekture promenljive koje učestvuju u pomenutoj formuli će dobiti odgovarajuće vrednosti. Ukoliko se radi o programu arhitekture domaćina u funkciji `td_ta_new()` promenljive dobijaju vrednosti koje odgovaraju arhitekturi domaćina, dok ako se radi o programu ciljne arhitekture, tj. različite od arhitekture domaćina, iz *GNU GDB*-a će biti pozvana nova funckija u standardnoj biblioteci `td_ta_init_target_consts()` koja će postaviti vrednosti promenljivih koje odgovaraju ciljnoj arhitekturi. Ideja je upisati ove vrednosti u datoteku jezgra na ciljnoj arhitekturi i kasnije ih pročitati na arhitekturi domaćina korišćenjem *GNU GDB* debagera. U ovom slučaju izmena u samom *GNU GDB*-u bi bila minorna, te zahteva samo proveru da li je učitani program arhitekture domaćina ili ne, i u zavisnosti od toga se poziva `td_ta_init_target_consts()` ili ne. Ovo rešenje je nastalo kao odgovor na komentare zajednice *GNU* na predhodno rešenje. Izmena standardne biblioteke se može pronaći na strani [10].

5.3 Čitanje i pisanje datoteke jezgra za procesorsku arhitekturu MIPS

Kako bi funkcionalnost bila moguća za svaku podržanu procesorsku arhitekturu, potrebno je da alat *GNU GDB* ima potpunu podršku za čitanje informacija iz datoteka jezgra. Podrška je za sve arhitekrure potpuna, ali u trenutku razvoja ove funkcionalnosti uočeno je da alat nema potpunu podršku za čitanje identifikacionog broja procesa (eng. *process ID*) iz datoteke jezgra za arhitekture *MIPS*, neophodnog za pristupanje baz̆ičnih informacija o nitima. Koristeći komandu alata `gcore` moguće je kreirati datoteku jezgra i sa korisničkog nivoa, gde je takođe uočeno da alat nema potpunu podršku za ispisivanje nekih informacija o procesu za procesorsku arhitekturu *MIPS*. Da bi podrška za *TLS* promenljive bila potpuna i za arhitekturu *MIPS*, implementirani su i uočeni nedostaci. Ova implementacija je prihvaćena od strane zajednice *GNU* i ušla je u izvršnu verziju alata 8.0.0.

Deo izvornog koda (u datoteci *bfd/elf32-mips.c*) izmene za čitanje informacija o procesu iz datoteke jezgra je prikazan na listinigu 5.5. Bitna informacija za čitanje vrednosti *TLS* promenljive je identifikacioni broj procesa. Bez izmene prikazane u listingu 5.5 to ne bi bilo moguće.

Listing 5.5: Deo izmene za čitanje informacija o procesu iz datoteke jezgra

```
case 128:                   /* Linux/MIPS elf_prpsinfo */
  elf_tdata (abfd)->core->pid
   = bfd_get_32 (abfd, note->descdata + 16);
  elf_tdata (abfd)->core->program
   = _bfd_elfcore_strndup (abfd, note->descdata + 32, 16);
```

Ukoliko se koristi datoteka jezgra generisana sa korisničkog nivoa koristeći debager, bez upisivanja u nju informacija o procesu (kao npr. identifikacioni broj procesa) čitanje vrednosti *TLS* promenljive nije moguće. Deo izvornog koda (u datoteci *bfd/elf32-mips.c*) izmene za pisanje informacija o procesu u datoteku jezgra je prikazan na listinigu 5.6.

Listing 5.6: Deo izmene za pisanje informacija o procesu u datoteku jezgra

```
  ...
    case NT_PRSTATUS:
      {
       char data[256];
       va_list ap;
       long pid;
       int cursig;
       const void *greg;

       va_start (ap, note_type);
       memset (data, 0, 72);
       pid = va_arg (ap, long);
       bfd_put_32 (abfd, pid, data + 24);
       cursig = va_arg (ap, int);
       bfd_put_16 (abfd, cursig, data + 12);
       greg = va_arg (ap, const void *);
       memcpy (data + 72, greg, 180);
       memset (data + 252, 0, 4);
       va_end (ap);
       return elfcore_write_note (abfd, buf, bufsiz,
                                  "CORE", note_type, data,
                                     sizeof (data));
      }
  ...
```

Primeri izvornog koda izmena su prikazani samo za 32-bitnu verziju arhitekture *MIPS*. Napomenimo da izmena obuhvata i 64-bitnu verziju procesorske arhitekture *MIPS*.

5.4 Testiranje

Faza testiranja je jako važan deo životnog ciklusa razvoja softvera, te je veliki značaj dat ovoj sekvenci. Testiranje je izvršeno na raznim verzijama procesorskih arhitektura *ARM* i *MIPS*.

Takođe, implementirani su testovi *DejaGNU* [1] koji testiraju pristupanje vrednosti *TLS* promenljive iz datoteka jezgra raznih arhitektura. Zbog prirode problema koji se testira datoteke jezgra su generisane na različitim platformama i kompresovane zajedno sa izvršnom datotekom u direktorijumu *gdb/testsuite/gdb.multi/*, te se prilikom izvršavanja konkretnih testova te datoteke raspakuju i učitavaju u debager. Testovi obuhvataju izvršne datoteke sa mehanizmom *TLS* generisane koristeći *2.19* i *2.22* verzije standardne biblioteke.

Komanda alata kojom pokrećemo testove, ukoliko smo pozicionirani u direktorijumu gde je alat izrađen je prikazana u listingu 5.7.

Listing 5.7: Komanda za pokretanje testova

```
1 make check
```

Pre izmene koja dodaje funkcionalnost *TLS*, rezultati izvršavanja testova *Multiarch* verzije debagera je prikazana u listingu 5.8

Listing 5.8: Rezultati izvršavanja testova

```
1                === gdb Summary ===
2                # of expected passes  33715
3                # of expected failures  196
4                # of known failures  66
5                # of unresolved testcases 5
6                # of untested testcases  67
7                # of unsupported tests  219
```

Posle izmene koja dodaje funkcionalnost *TLS*, rezultati izvršavanja testova *Multiarch* verzije debagera je prikazana u listingu 5.9.

Listing 5.9: Rezultati izvršavanja testova

```
1  === gdb Summary ===
2  # of expected passes  33718
3  # of expected failures  196
4  # of known failures  66
5  # of unresolved testcases 5
6  # of untested testcases  67
7  # of unsupported tests  219
```

Rezultati testiranja prikazani u listinzima 5.8 i 5.9 pokazuju da nema regresije prilikom testiranja alata. Tri testa koja dodatno prolaze jesu testovi koji su dodati izmenama koji se opisuju u ovom radu. Dodati testovi sadrže primere čitanja vrednosti *TLS* promenljive, koji se mogu naći u realnoj upotrebi.

5.5 Upotreba alata

Da bi *GNU GDB* uspešno koristio naprednije tehinke analize višenitnih programa, uključujući i analizu mehanizma *TLS*, potrebno je da mu se prosledi putanja do biblioteke *libthread_db* koja pripada istoj verziji standardne biblioteke kao i program koji se debaguje. To isto važi i za *Multiarch* verziju alata. Nakon toga potrebno je zadati putanju do biblioteka koje je program koristio na tom uređaju sa ugrađenim računarom na kome se program izvršavao. Primetimo da te biblioteke pripadaju toj ciljanoj platformi. Debager neće izvršavati program koji se debaguje, već će samo rekonstruisati sliku stanja tog procesa kada je neočekivano prekinuo sa radom, čitajući informacije iz datoteke jezgra i deljenih biblioteka.

Izvorni kod višenitnog programa koji se debaguje u primeru je napisan u programskom jeziku *C* i prikazan je u listingu 5.10.

Listing 5.10: Višenitni test primer napisan u C kodu

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

__thread int foo=0xdeadbeef;
pthread_t threads[5];

void* thread(void *e) {
  int *i = (int*)e;
  foo+=*i;
  printf("foo is %x\n", foo);
  sleep(10);
  return (void*)0;
}

int main()
{
  printf("init %x\n", foo);
  int i;
  for (i=0; i<5; i++) {
    pthread_create(&threads[i], NULL, thread, &i);
  }

  for (i=0; i<5; i++) {
    pthread_join(threads[i], NULL);
  }

  sleep(5);
  abort();
}
```

Korišćenje debagera pre implementacije čitanja vrednosti *TLS* promenljive iz datoteke jezgra generisane na uređaju sa ugrađenim računarom arhitekture *MIPS* je prikazano u listingu 5.11.

Listing 5.11: Korišćenja debagera pre implementacije

```
1 (gdb) add-auto-load-safe-path /home/glibc/build_22/INSTALL/lib
2 (gdb) set libthread-db-search-path /home/glibc/build_22/INSTALL/lib
3 (gdb) set solib-search-path ~/master_examples/mips_arch
4 (gdb) file ~/master_examples/mips_arch/example222
5 Reading symbols from ~/master_examples/mips_arch/example...done.
6 (gdb) core-file ~/master_examples/mips_arch/core
7 [New LWP 21808]
8 [New LWP 21813]
9 [New LWP 21810]
10 [New LWP 21809]
11 [New LWP 21811]
12 [New LWP 21812]
13 [Thread debugging using libthread_db enabled]
14 Using host libthread_db library "/home/glibc/build_22/INSTALL/lib/
      libthread_db.so.1".
15 Core was generated by `./example'.
16 Program terminated with signal SIGABRT, Aborted.
17 #0 0x00000000 in ?? ()
18 [Current thread is 1 (LWP 21808)]
19 (gdb) p/x foo
20 Cannot find user-level thread for LWP 21808: generic error
```

Prvom i drugom komandom alata (linije 1 i 2) se podešavaju putanje do standardne biblioteke za debagovanje niti koja ima istu verziju kao i izvršna datoteka koja se debaguje. Trećom komandom (linija 3) se zadaje putanja do deljenih biblioteka koje je izvršna datoteka koja se debaguje koristila na uređaju sa ugrađenim računarom. Četvrta komanda (linija 4) učitava izvršnu datoteku koja se debaguje. Petom komandom (linija 6) se učitava datoteka jezgra koja je generisana prilikom neočekivanog prekidanja izvršavanja datoteke koja se debaguje. Poslednjom komandom (linija 19) se bezuspešno pokušava čitanje vrednosti TLS *promenljive.*

Korišćenje debagera nakon implementacije čitanja vrednosti *TLS* promenljive iz datoteke jezgra generisane na nekom uređaju sa ugrađenim računarom je prikazano u listingu 5.12.

Listing 5.12: Korišćenja debagera posle implementacije

```
1 (gdb) add-auto-load-safe-path /home/glibc/build_22/INSTALL/lib
2 (gdb) set libthread-db-search-path /home/glibc/build_22/INSTALL/lib
3 (gdb) set solib-search-path ~/master_examples/mips_arch
4 (gdb) file ~/master_examples/mips_arch/example222
5 Reading symbols from ~/master_examples/mips_arch/example...done.
6 (gdb) core-file ~/master_examples/mips_arch/core
7 [New LWP 21808]
8 [New LWP 21813]
9 [New LWP 21810]
10 [New LWP 21809]
11 [New LWP 21811]
12 [New LWP 21812]
13 [Thread debugging using libthread_db enabled]
14 Using host libthread_db library "/home/glibc/build_22/INSTALL/lib/
     libthread_db.so.1".
15 Core was generated by `./example'.
16 Program terminated with signal SIGABRT, Aborted.
17 #0 0x00000000 in ?? ()
18 [Current thread is 1 (LWP 21808)]
19 (gdb) p/x foo
20 $1 = 0xdeadbeef
```

Prvom i drugom komandom alata (linije 1 i 2) se podešavaju putanje do standardne biblioteke za debagovanje niti koja ima istu verziju kao i izvršna datoteka koja se debaguje. Trećom komandom (linija 3) se zadaje putanja do deljenih biblioteka koje je izvršna datoteka koja se debaguje korsitila na uređaju sa ugrađenim računarom. Četvrta komanda (linija 4) učitava izvršnu datoteku koja se debaguje. Petom komandom (linija 6) se učitava datoteka jezgra koja je generisana prilikom neočekivanog prekidanja izvršavanja datoteke koja se debaguje. Poslednjom komandom (linija 19) se uspešno čita vrednosti TLS *promenljive.*

Glava 6

Zaključak

U okviru master rada, prikazan je uvod u pomoćne informacije za debagovanje, i rad debagera u opštem slučaju. Pored opisa formata pomoćnih informacija *DWARF*, u radu je dat kratak uvod o formatu izvršnih datoteka *ELF*. U radu je takođe naveden detaljan opis implementacije mehanizma *TLS*, kako hardverski tako i softverski deo.

Funkcionalnost debagera prilikom čitanja vrednosti promenljive lokalne za nit iz datoteke jezgra generisane na uređaju sa ugrađenim računarom podiže kvalitet debagera. U radu su opisana dva moguća rešenja problema čitanja vrednosti *TLS* promenljive iz datoteke jezgra i njihove implementacije. Oba rešenja su poslata zajednici *GNU* i u toku je pregled i analiza implementacije, čija se diskusija može videti na [2]. U ovom radu je takođe prikazana evaluacija testiranja debagera sa pomenutim izmenama. Naveden je i precizan postupak korišćenja debagera kako bi se vrednost *TLS* promenljive iz datoteke jezgra uspešno pročitala.

Pored implementacije čitanja vrednosti promenljive lokalne za nit iz datoteke jezgra, poboljšan je kvalitet čitanja i generisanja datoteke jezgra, sa korisničkog nivoa od strane debagera, namenjene platformama sa arhitekturom procesora *MIPS*. Nedostajala je podrška za čitanje i pisanje osnovnih informacija o procesu, jedna od njih je jedinsveni broj procesa, neophodnih za korišćenje raznih funkcionalnosti debagera prilikom analize datoteke jezgra. Te izmene su prihvaćene od strane *GNU* zajednice i uvrštene u izvršnu verziju debagera.

Kao mogući pravac daljeg razvoja, planirano je rešenje koje ne zavisi od standardne biblioteke. Ta zavisnost bi, uz pomoć upisivanja dodatnih informacija u datoteku jezgra, bila zaobiđena. Da bi se to omogućilo potrebne su promene u jezgru operativnih sistema prilikom generisanja datoteke jezgra.

Literatura

[1] *DejaGNU*. on-line at: https://www.gnu.org/software/dejagnu/.

[2] *Diskusija zajednice GNU*. on-line at: http://sourceware-org.1504.n7.nabble.com/PATCH-TLS-access-support-in-cross-Linux-GDB-td452155.html.

[3] *ELF Format*. on-line at: http://www.cs.cmu.edu/afs/cs/academic/class/15213-s00/doc/elf.pdf. 1992.

[4] Free Software Foundation. *DWARF Format*. on-line at: http://dwarfstd.org/. 1992.

[5] Linux Foundation. *ptrace*. on-line at: http://man7.org/linux/man-pages/man2/ptrace.2.html.

[6] *Free Software Foundation*. on-line at: https://www.fsf.org/.

[7] *GNU GDB*. on-line at: https://www.gnu.org/software/gdb/.

[8] *GNU GDB sa Multiarch TLS funkcionalnosti*. on-line at: https://github.com/djolertrk/gdb_tls.

[9] *Infinity*. on-line at: https://gbenson.net/.

[10] *Izmena standardne biblioteke*. on-line at: https://marc.info/?l=glibc-alpha&m=149976643921098&w=2.

[11] Reid Kleckner. *CodeView*. on-line at: https://llvm.org/devmtg/2016-11/Slides/Kleckner-CodeViewInLLVM.pdf.

[12] *Kompajler GCC*. on-line at: https://www.gnu.org/software/gcc/.

[13] *POSIX*. on-line at: http://www.csc.villanova.edu/~mdamian/threads/posixthreads.html.

[14] *Projekat LLVM*. on-line at: http://llvm.org/.

[15] *RedHat*. on-line at: https://www.redhat.com/en/.

[16] Richard Stallman. *The GNU GDB documentation*. on-line at: https://www.gnu.org/software/gdb/documentation/.

[17] *The GNU C Library (glibc)*. on-line at: https://www.gnu.org/software/libc/.

[18] Đorđe Todorović. *Improvement Of GNU GDB For Analyzing TLS Variable From Core file Of Target Architecture*. on-line at: http://2017.telfor.rs/.

[19] Red Hat Inc. Ulrich Drepper. *ELF Handling For Thread-Local Storage*. on-line at: https://www.uclibc.org/docs/tls.pdf.

[20] *Zvaniča stranica programskog jezika JavaScript*. on-line at: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

Biografija autora

Đorđe Todorović (*Užice, 12. Avgust 1993.*) Rođen je u Užicu. Završio je Gimnaziju u Požegi, Informatički smer, 2012. godine i iste godine upisao Matematički fakultet u Beogradu. 2016. godine je završio osnovne studije Matematičkog fakulteta i iste upisao master studije. Položio je sve ispite master studija u septembru 2018. godine. Od novembra 2015. pa do sada radi kao inženjer u Naučno-istraživačkom institutu RT-RK. Do sada je objavio nekoliko radova iz oblasti debagera i generisanja pomoćnih informacija za debagovanje od strane programskih prevodilaca. Na temu opisanu u master radu je objavio rad na konferenciji TELFOR [18]. Trenutno radi na LLVM projektu, tačnije na poboljšanju tog prevodioca prilikom generisanja pomoćnih informacija za debagovanje.

Radovi:

1. *Ananthakrishna Sowda, Đorđe Todorović, Nikola Prica i Ivan Baev: Improving Debug Information in LLVM to Recover Optimized-out Function Parameters, 2019 European LLVM developers' Meeting (Brussels, Belgium)*

2. *Đorđe Todorović i Nikola Prica: Debug info in optimized code - how far can we go? (Improving LLVM debug info with function entry values), 2019 FOSDEM (Brussels, Belgium)*

3. *Ananthakrishna Sowda, Đorđe Todorović, Nikola Prica i Ivan Baev: Improving Debug Information in LLVM to Recover Optimized-out Function Parameters, 2018 Bay Area LLVM Developers' Meeting (San Jose, USA)*

4. *Nikola Prica, Đorđe Todorović i Petar Jovanović: Improving debug information generation inside LLVM compiler, 2018 Conference for electronics, communication, computing, automatizing and nuclear technique (Kladovo, Serbia)*

5. *Đorđe Todorović, Nikola Prica, Petar Jovanović i Nemanja Popov: Improvement Of GNU GDB For Analyzing TLS Variable From Core file Of Target Architecture, publication description, 2017 Telecommunications forum TELFOR (Belgrade, Serbia)*