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, redovan profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

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

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

Datum odbrane: _____

Daretu

Sadržaj

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 [9]. Bitno je obezbediti laku analizu višenitnih programa, što je posebno izazovan zadatak.

Razvoj softvera za uređaje sa ugrađenim računarom se obično odvija na ličnim računarima, jer uglavnom ugrađeni računari oskudevaju u pogledu resursa. Arhitektura procesora uređaja sa ugrađenim racunarom 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 racunarom 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. Načini kroz koje se to može ostvariti jesu korišćenje *GNU GDB* servera, uz korišćenje posebnih protokola, ili korišćenje datoteka jezgara. Datoteka jezgra može biti kreirana iz 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 neki nestandardni 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 dobijanja vrednosti promenljivih lokalnih za niti, ili, skraćeno, *TLS* (eng. *TLS-Thread Local Storage*) promenljivih [18]. Definisanjem takvih promenljivih svaka nit ima različitu kopiju iste promenljive i prilkom analiziranja višenitnih programa značajno je pročitati 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 *TLS* mehanizma, te je njihovo čitanje iz debagera *GNU GDB* dodatno otežano. Glavni doprinos rada predstavlja omogućavanje čitanja vrednosti *TLS* promenljive iz *GNU GDB* za sve arhitekture procesora podržanih u debageru na arhitekturi domaćina, čime se proširuju dostupne informacije za analizu programa drugih arhitektura. Rezultati opisani u ovom radu su predstavljeni i na konfereniciji TELFOR [17].

Rad se pored prvog uvodnog, sastoji još iz pet poglavlja. Drugo poglavlje opisuje kako rade debageri. Navedena je podrška operativnih sistema i opis formata izvršnih fajlova koji potpomažu rad debagera. Treće poglavlje pruža informacije o debageru *GNU GDB* i opisuje detalje implementacije osnovnih komandi debagera. Četvrto poglavlje prikazuje detalje o promenljivama lokalnih za niti. Naveden je opis organizacije registara i memorije koji potpomažu funkcionisanje *TLS* mehanizma. Takođe je dat opis novih struktura podataka definisanih za promenljive lokalne za niti. U petom poglavlju je opisana implementacija poboljšanja debagera prilikom čitanja vrednosti promenljivih lokalnih za niti iz nedomaćinske datoteke jezgra, dok je u šestom poglavlju predstavljen zaključak rada.

Glava 2

Kako rade 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 su manje važne, kao npr. korisnički interfejs aplikacije ima neočekivanu boju pozadine. 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. *debagger*) 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 se „nakačiti" na proces 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. *brakpoints*) itd. Proces izvršavanja programa od strane debagera sekvencijalno, instrukciju po instrukciju ili liniju po liniju, nazivamo koračanje. Tačke prekida su mogućnost debagera da zaustavi izvršavanje programa na određenoj tački. 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 domaćinski 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 *UNIX*-olikih, posebno *Linux* debagera. *Windows* debageri i programski prevodioci ne prate standard *DWARF*

[4] prilikom baratanja sa debag informacijama. Alati za debagovanje u okviru operativnog sistema *Windows* koriste standard *Majkrosoft 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* [7] i *LLVM/Clang* [12], to je opcija `-g`) koja obezbeđuje generisanje pomoćnih debag informacija. Ukoliko je program koji se analizira preveden bez optimizacija, debag informacije koje prate program su potpune. Za programe koji se puštaju u produkciju, da bi bili brži i zauzimali manje memorije, se prilikom prevođenja koriste optimizacije. Postoje različiti nivoi optimizacija i oni se zadaju kao opcija prevodiocu prilikom prevođenja. Nivoi optimizacija produkcijskih programa su *-O2* i *-O3*.

Prilikom optimizacija se gube razne debag informacije. Neke promenljive i funkcije programa neće biti predstavljene debag informacijama. Npr. promenljiva programa može biti živa samo u nekim određenim delovima programa, pa programski prevodioci generišu debag informacije 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 debag fajl format 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 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 izvršnog fajla. Trenutno je poslednji realizovani standard verzija 5 formata *DWARF*.

Debag fajl format *DWARF* na *UNIX*-olike operativne sisteme, kao što su *Linux* i *MacOS*. Generisane debag informacije, prateći *DWARF* standard, su podeljene u nekoliko sekcija sa prefiksom `.debug_`. Neke od njih su `.debug_line`, `.debug_loc` i `.debug_info` koje redom predstavljaju informacije o linijama izvornog koda, lokacijama promenljivih i ključna debag sekcija koja sadrži debag informacije koje referišu na informacije iz ostalih debag sekcija. *DWARF* je predstavljen kao drvolika struktura, smeštena u `.debug_info` sekciju, koja razne entitete programskog jezika opisuje osnovnom debag jedinicom *DIE* (eng. *Debug Info Entry*). Osnovna debag jedinica može opisivati lokalnu promenljivu programa, formalni parametar, funkciju itd. Svaka od njih je identifikovana *DWARF* tagom koji predstavalja informaciju o toj jedinici, gde je npr. tag za lokalne promenljive predstavljen sa `DW_TAG_local_variable`, ili tag za funkciju je obeležen sa `DW_TAG_subprogram`. Svaka debag jedinica je opisana određenim *DWARF* atributima sa prefikosm `DW_AT_`. Oni mogu ukazivati na razne informacije o entitetu kao sto su ime promenljive ili funkcije, liniju deklaracije, itd. Koren svakog *DWARF* stabla je predstavljen debag jedinicom, sa tagom `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 *DWARF* stabla za primer 2.1 programa napisanog u C programskom jeziku je prikazan u primeru 2.2.

Listing 2.2: Primer *DWARF* reprezentacije.

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

Funkcija `main` je predstavljena *DWARF* tagom `DW_TAG_subprogram`. Atribut te debag 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 debag jedinica je dete čvora koji predstavlja funkciju `main` i ukazuje da se promenljiva `x` nalazi unutar funkcije `main`. Promenljiva `x` je predstavljena *DWARF* tagom `DW_TAG_variable`. Atribut `DW_AT_name` predstavlja ime promenljive, `DW_AT_type` referiše na debag jedinicu koja predstavlja tip promenljive, dok `DW_AT_location` atribut predstavlja lokaciju promenljive u memoriji programa.

Debag promenljive

Svaka promenljiva programa prevedenog sa debag informacijama, ukoliko se ne radi o optimizovanom programu, je predstavljena *DWARF* tagom `DW_TAG_variable`.

Atribut `DW_AT_location` ukazuje na lokaciju promenljive. Lokacija može biti predstavljena *DWARF* izrazom, kao npr. lokacija promenljive `x` prikazana sledećim primerom. *DWARF* izraz te promenljive ukazuje da se ona nalazi na pomeraju `-20` trenutnog stek okvira `main` funkcije. U neoptimizovanom kodu sve promenljive imaju lokacije zadate *DWARF* izrazom. 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 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 primerom 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 *DWARF* 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*) [3] je format izvršnih fajlova, deljenih biblioteka, objektnih fajlova i datoteka jezgara.

ELF sadrži razne informacije o samom fajlu. Podeljen je u dva dela: *ELF* zaglavlje i podaci fajla. *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 fajla mogu sadržati programsku tabelu zaglavlja (eng. *Program header table*), sekcijsku tabelu zaglavlja (eng. *Section header table*) i ulazne tačke prethodne dve tabele. Primer 2.5 prikazuje *ELF* fajl format počitan alatom *readelf*:

Listing 2.5: Primer *ELF* zaglavlja.

```
ELF Header:
Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Class: ELF64
Data: 2's complement, little endian
Version: 1 (current)
OS/ABI: UNIX - System V
ABI Version: 0
Type: EXEC (Executable file)
Machine: Advanced Micro Devices X86-64
Version: 0x1
Entry point address: 0x400480
Start of program headers: 64 (bytes into file)
Start of section headers: 8992 (bytes into file)
Flags: 0x0
Size of this header: 64 (bytes)
Size of program headers: 56 (bytes)
Number of program headers: 9
Size of section headers: 64 (bytes)
Number of section headers: 38
Section header string table index: 35
Section Headers:
...
```

Polje	*.bss*	*.data*
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.

U fajl formatu *ELF* definisan je 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.

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 argu-

ment se po potrebi koriste u zavisnosti od namere korišćenja sistemskog poziva `ptrace` za čitanje ili pisanje sa adrese datom trećim argumentom, pri tom baratajući podacima na adresi zadatoj četvrtim argumentom. To znači 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 jos 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 implementacije osnovnih elemenata upotrebe debagera.

Tačke prekida

Postoje dve vrste tačaka prekida (eng. *breakpoints*): softverske i hardverske [9].

Osvrnimo se prvo na softverske tačke prekida. Postavljanje tačaka prekida predstavlja jednu od najkorišćenijih mogućnosti debagera, te stoga navedimo par smernica kako je ista realizovana, u opštem slučaju. Ali takođe treba napomenuti da 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 primeru 2.6.

Listing 2.6: Primer funkcije `main` u asemblerskom jeziku.

```
1 0000000000400450 <main>:
```

```
400450:        48 83 ec 08           sub    $0x8,%rsp
400454:        ba 05 00 00 00        mov    $0x5,%edx
400459:        be 04 06 40 00        mov    $0x400604,%esi
40045e:        bf 01 00 00 00        mov    $0x1,%edi
400463:        31 c0                 xor    %eax,%eax
400465:        e8 c6 ff ff ff        callq  400430 <printf>
40046a:        31 c0                 xor    %eax,%eax
40046c:        48 83 c4 08           add    $0x8,%rsp
400470:        c3                    retq
400471:        66 2e 0f 1f 84        nopw   %cs:0x0(%rax,%rax,1)
400478:        00 00 00
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 registar programski brojač (eng. *CPU register pc*) stigne do `int3` instrukcije izvršavanje se zaustavalja na toj tački. Debager je već upoznat od strane korisnika 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. Treba napomenuti da ovo važi za procesorsku arhitekturu *Intel x86-64*, instrukcija prekida za arhitekture kao što su *ARM, MIPS, PPC* itd., se drugačije kodira, ali postupak implementacije tačaka prekida je isti.

Ukoliko želimo da stavimo tačku prekida eksplicitno na funkciju `main`, za to koristimo posrednika u vidu *DWARF* debag informacija. U tom slučaju debager traži element *DWARF* stabla koji ukazuje na informacije o `main` funkciji, i odatle čita informaciju na kojoj adresi u memoriji se nalazi prva mašinska instrukcija date

funkcije. Na primeru 2.7 vidimo *DWARF* element koji opisuje funkciju uz pomoć atributa. Debager će pročitati `DW_AT_low_pc` atribut i na tu adresu postaviti `int3` instrukciju. Napomenimo da *DWARF* debag simbole generišemo uz pomoć `-g` opcije kompajlera.

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

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

Hardverske tačke prekida su direktno povezane sa hardverom u vidu specijalnih registara. Postavlja se na određenu adresu i hardverski *watchpoint* monitori za zadatu adresu mogu signalizirati razne promene, npr. čitanje, pisanje ili izvršavanje, što im daje prednost u odnosu na softverske tačke prekida. Mane u odnosu na softverske tačke prekida su performanse, koje su neuporedivo sporije, i takođe neophodna hardverska podrška za korišćenje hardverskih tačaka prekida.

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 [9].

Instrukcijsko koračanje na platformi Intel x86-64 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.

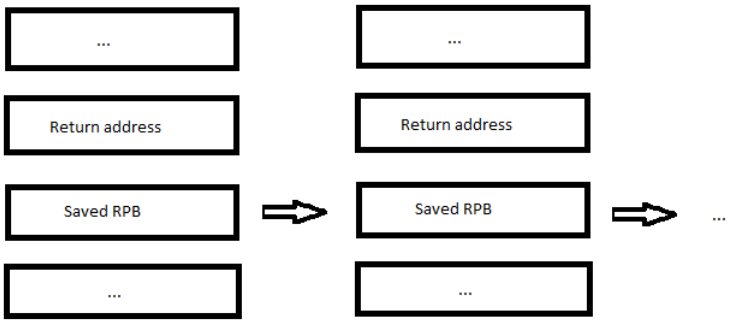
Pored instrukcijkog koračanja pomenućemo još jednu vrstu koračanja u neku funkciju koja je pozvana `call` ili `jump` instrukcijom. Komanda debagara *GNU GDB* koja nam to omogućava jeste `step in`.

Treba napomenuti da postoje arhitekture za koje ovo ne važi, kao npr. platforma *ARM*, koja nema hardversku podršku za instrukcijsko koračanje i za njih se kora-

čanje 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 Intel x86-64 [9].



Slika 2.1: Primerređanja stek okvira na x86-64 platformi.

Na slici 2.1 navedeni su stek okviri za dva funkcijska poziva. Pre povratne vrednosti funkcije obično seređaju 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 ima informaciju o imenu funkcije odgovor je u tome što pretražuje DWARF stablo sa debag informacijama, tražeći `DW_TAG_subprogram` sa odgovarajućom povratnom adresom, pritom čitajući `DW_AT_name` atribut tog elementa.

Čitanje vrednosti promenljivih

Za čitanje vrednosti promenljivih u programu, debager pretražuje DWARF stablo tražeći promenljivu sa zadatim imenom. U slučaju lokalnih promenljivih, traži se

`DW_TAG_variable` element čiji `DW_AT_name` odgovara navedenoj promenljivoj. Kada se ista pronađe konsultuje se `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 [9].

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 domaćinske arhitekture (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 domaćinskim platformama se takođe upotrebljava *Multiarch GNU GDB* koji koristi biblioteke namenjene ciljanim 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 *GNU GDB* debagera

Izvorni kod alata *GNU GDB* je originalno napisan od strane Ričarda Stolmana 1986. godine kao deo *GNU* sistema [15]. *GNU GDB* je slobodan i besplatan softver realizovan pod *GNU General Public License (GPL)*. Od 1990. do 1993. godine alat je održavao Džon Gilmor, a trenutno za održavanje alata je zadužena *GDB Steering Committee* grupa, odobrena od strane *FSF* (eng. *Free Software Foundation*) [6].

Poslednja realizovana verzija alata *GNU GDB* je 8.2.1.

3.2 Šta je *arhitektura* za 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.

Domaćinski GNU GDB

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

Prevođenje domaćinskog GNU GDB debagera

Preuzimanje izvornog koda debagera se vrši komandama prikazanim na 3.1.

Listing 3.1: Komande korišćene za preuzimanje izvornog koda debagera.

```
1 mkdir gdb
2 cd gdb
3 git pull 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 sa 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* `Makefile.in` *skriptu podešavajući okruženje (putanje do deljenih biblioteka, programski prevodilac i drugo) prilikom čega je krajnji izlaz fajl* `Makefile` *kojim se izgrađuje (prevodi) softver.*

Nakon konfiguracije direktorijuma prevođenje alata se vrši komandom:

Listing 3.3: Izvršavanje komande `make`.

```
1 make
```

Pokretanje domaćinskog GNU GDB debagera

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 na 3.4.

Listing 3.4: Pokretanje debagera.

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

Korišćenje domaćinskog GNU GDB debagera

Pokažimo neke od osnovnih komandi debagera *GNU GDB*.

Primer programa koji se debaguje je prikazan na 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 na 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 na 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 na 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 (eng. *core dump file*) 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. Ugrađeni uređaji 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.

`ldinfo` **strukture.** Definiše informacije relevantne za dinamički loader.

`mstsave` strukture. 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 pucanja programa.

Memorijski mapirani regioni [1] i `vm_info` strukture. Sadrži informacije o pomerajima i dužinama mapiranih regiona.

Generisanje datoteke jezgra

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

Listing 3.9: Primer korišćenja komande `unlimited`.

```
1 ulimit -c unlimited
```

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

3.4 *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. Skup komandi alata je limitiran u odnosu na domaćinski *GNU GDB*.

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

Prevođenje *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. Ovim komandama kreiramo `Makefile` kojim odobravamo debagovanje svih podržanih arhitektura u alatu GBD (kao npr. *MIPS32, MIPS64, ARM, AARCH64, x86_64, i386, SPARC*) je prikazan na 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 domaćinske i* `Multiarch` *verzije debagera je ta što je za domaćinski alat potrebno okruženje (deljene biblioteke, programski prevodilac, itd.) samo za domaćinsku arhitekturu. 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 *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 domaćinska verzija debagera komandom prikazanom na 3.12.

Listing 3.12: Pokretanje debagera.

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

Korišćenje *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, itd. 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 primeru 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 primeru ispod 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`.

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

U primeru 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`.

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

Analiza datoteke jezgra

U primeru 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 izvršni fajl i deljene biblioteke koje izvršni fajl 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 na domaćinskoj mašini.

U primeru 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
          fsr      fir
      00000000 00000000
```

Glava 4

Promenljive lokalne za niti

Pisanje višenitnih programa predstavlja fudamentalan i neizbežan koncept savremenog programiranja. Gotovo nijedan kompleksan korisnički program ne može biti napisan bez korišćenja niti. Preciznije, niti podižu performanse i brzinu programa, te se sve češće koriste u pisanju softvera. 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 Motivacija

Povećanje korišćenja niti u programiranju dovelo je do potrebe programera za boljim načinom rukovanja podacima lokalnih za niti. *POSIX* [13], skup interfejsa za rukovanje nitima, definiše interfejse koji omogućavaju kreiranje jednog 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. Zbog toga nastaje ozbiljan problem kada se kombinuje sa dinamičko učitanim kodom. Iz tih razloga postoji potreba za efikasnijim rešenjem.

Da bi se odgovorilo na sve 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. 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. Jedina razlika je ta 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 `__thread` klučnom reči inicijalizacija se vrši unutar funkcije gde je kreirana. U nastavku teksta biće reči o implementaciji i izmenama koje su potrebne za korišćenje `__thread` varijante.

Neki primeri deklaracija promenljivih lokalnih za niti su prikazani u okviru listinga 4.1.

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. Promenljive statičkih 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 fajlova je proširen kako bi definisao promenljive lokalne za niti odvojeno od normalnih promenljivih. Dinamički punilac (eng. *dynamic loader*) 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 nastavku poglavlja su opisane izmene u fajl formatu *ELF* i dat

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.

je detaljan opis izvršavanja programa koji sadrže promenljive lokalne za niti.

4.2 Definisanje novih podataka u fajl formatu ELF

Izmene u fajl formatu izvršnih fajlova, 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 [18]. 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`.

Imena novih sekcija, kao ni ostalih u fajl formatu *ELF*, nisu bitna. 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` 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 normalnih `.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

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.

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, normalna tabela simbola ne može biti iskorišćena. U izvršnom fajlu 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. Izvršni fajlovi u polju `st_value` imaju vrednost pomeraja promenljive u *TLS* inicijalizacionoj slici. Nijedna realokacija ne sme pristupati simbolima tipa `STT_TLS`, osim onih koji su uvedene za rukovanje *TLS* promenljivih. Takođe, te nove realokacije ne smeju koristi simbole ostalih tipova.

Da bi dinamički linker mogao da izvrši inicijalizaciju inicijalizacione slike, njena pozicija prilkom 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

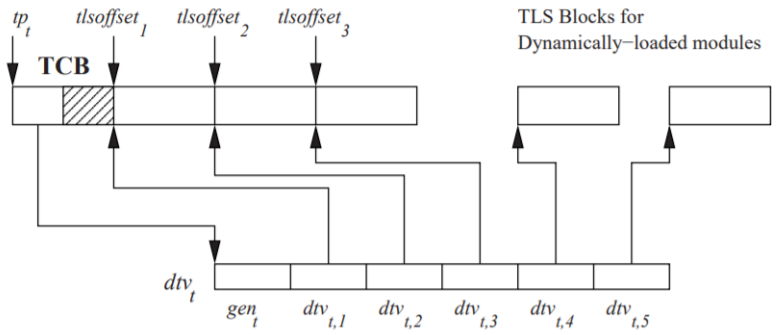
Obrada podataka lokalnih za niti nije prosta kao obrada običnih podataka. Standardni pristup kreiranja i korišćenja segmenta podataka od strane procesa se ne može iskoristiti. Umesto toga, nekoliko kopija jednog istog podatka mora biti kreirano, svi inicijalizovani iz iste inicijalizacione slike.

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

Nije samo alociranje memorije problem za korišćenje *TLS* promenljivih. Standardna pravila pretrage simbola (*eng. symbol lookup*) u izvršnim fajlovima sa *ELF* formatom ne mogu biti primenjena na simbole koji opisuju promenljive lokalne za niti. Prema tome, standardan proces povezivanja ne može biti korišćen za ovakve situacije.

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.

Na slici 4.1 je prikazan primer prve varijante *TLS* strukture podataka. Nitni registar za nit `t` je označen sa tp_t. Početak memorija svake niti je određena *nitnim kontrolnim blokom* `TCB` (*eng. Thread Control Block*). Nitni registar pokazuje na nitni kontrolni blok, koji na pomeraju nula sadrži pokazivač na *nitni dinamički*



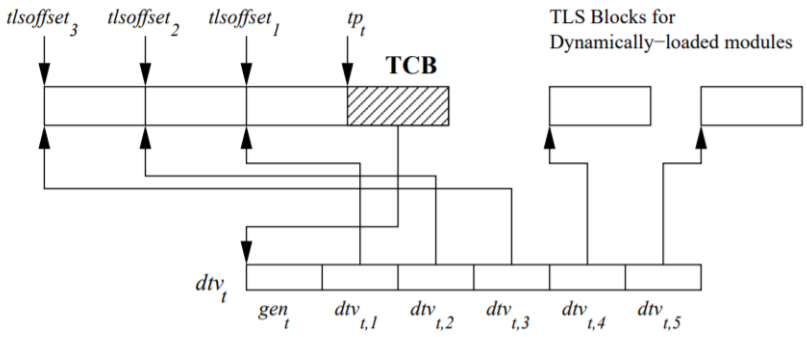
Slika 4.1: TLS struktura podataka. Varijanta 1.

vektor dtv_t za tu određenu nit. Nitni dinamički vektor predstavlja niz koje sadrže informacije o podacima lokalnih za niti koje pripadaju različitim modulima. Memoriju lokalnu za nit u okviru jednog modula nazivamo *TLS blok*. Ukoliko je modul inicijalno dostupan, element nitnog dinamičkog vektora pokazuje na fiksni pomeraj podataka lokalnih za tu nit koje odgovaraju tom modulu. Ukoliko se modul dinamički učitava u program element dinamičkog nitnog vektora pokazuje na tu memoriju.

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. Prikazana 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.

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.



Slika 4.2: TLS struktura podataka. Varijanta 2.

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. Podrška za korišćenje *TLS* promenljive u statičko povezanim programima je limitirana. Neke procesorske arhitekture (kao npr. *IA-64*) ne definišu uopšte statičko povezivanje (iako je podržano to je nestandardizovano). 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 izvršni fajl 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 okviru listinga 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 okviru listinga 4.3.

Listing 4.3: Promena implementacije funkcije `__tls_get_addr()`.

```
void *
__tls_get_addr (size_t m, size_t offset)
{
  char *tls_block = dtv[thread_id][m];
  if (tls_block == UNALLOCATED_BLOCK)
    tls_block = dtv[thread_id][m] = allocate_tls(m);
  return tls_block + offset;
}
```

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 okviru listinga 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;
}
```

`tlssize[m]`, `tlsinit_size[m]` i `tlsinit_img[m]` su poznati 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 dva modela pristupa: dinamički ili statički. Različite arhitekture ABI-jem propisuju koji od modela pristupa će koristiti kao podrazumevani. Razni modeli pristupa se mogu sresti, ali četiri najpoznatija pristupa 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 dinamičkog izvršnog fajla. 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 linkeru 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 $dtv_{0,m}$.

Statički TLS model sa dodeljenim ofsetima

Ovaj model pristupa dopušta referisanje samo na one *TLS* promenljive koje su dostupne kao deo inicijalnog statičkog *TLS šablona* (*eng. template*). Ovaj šablon je sastavljen od svih *TLS* blokova koji su sačinjeni prilikom pokretanja procesa. U ovom modelu, relativni pomeraj pokazivača na nit date promenljive `x` je smešten u polju *GOT*-a (*eng. Global offset table*) 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 tog dinamičkog izvršnog fajla. Linker 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 tog dinamički povezanog izvršnog fajla.

Glava 5

Implementacija rešenja

TLS može biti različito implementiran za različite procesorske arhitekture. Može se smeštati u posebne registre, u određene delove memorije itd. *GNU GDB* alat mora poznavati sve arhitekturne razlike i anulirati ih na neki način. Implementacija ovog proširenja alata se upravo i zasniva na prethodnoj pretpostavci, te upravo to i realizuje. Istražene su implementacije *TLS*-a raznih arhitektura, kako u funkcijama *GNU C* biblioteke (poznatije kao *glibc*) [16], tako i karakteristike koje su *ABI*-jem propisane. Poboljšanje alata koje je uspešno realizovano se odnosi na verziju *Multiarch*. Ukoliko je program koji je učitan u *Multiarch GNU GDB* alat iste arhitekture kao arhitektura domaćina, zadržava se način čitanja vrednosti *TLS* promenljive kao u slučaju *GNU GDB* debagera arhitekture domaćina, jer je ta funkcionalnost već implementirana. Treba napomenuti da je u razvoju i alternativno rešenje od strane programera iz kompanije *Red Hat* [14] u vidu projekta *Infinity* [10]. Taj projekat ima ideju da obuhvati rešenja za razne probleme o debagovanju niti, ne samo obradu *TLS*-a. Prednost rešenja koji se predstavlja u ovom master radu jeste svakako ta što je cela funkcionalnost prepuštena samom *GNU GDB* debageru, te nema potrebe za instalacijom ili prevođenjem eksternih projekata. Sa tim u vidu se svakako dobija na uštedi korisničkog napora pri korišćenju debagera. Izvorni kôd alata i instrukcije za prevođenje i upotrebu debagera sa pobljšanjem za pristupanje vrednosti TLS promenljive se može pronaći na strani [8].

5.1 Detalji implementacije

U nastavku teksta opisani su detalji implementacije poboljšanja debagera.

Izmena sistema izgradnje alata

Fajlovi *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*-a (*gdb/linux-thread-db.c*) se prevodio samo za domaćinsku verziju *GNU GDB* debagera, pa stoga u proširenju za *Multiarch GNU GDB* takođe treba uvrstiti pomenuti fajl prilikom prevođenja. Pored toga, postojeće funkcije koje su zadužene za pristupanje vrednosti *TLS* promenljivih je trebalo izmeniti na neki način te je u projekat dodat direktorijum *gdb/glibc-dep/* koji sadrži funkcije koje barataju sa *TLS*-om različite procesorske arhitekture od arhitekture domaćina. Da bi se ti fajlovi prevodili samo u slučaju *Multiarch* verzije definisan je makro pod imenom `CROSS_GDB` koji se definiše prilikom kreiranja *Makefile*. Da bi se ispratila konvencija pisanja skripti za izgradnju alata, u direktorijum *gdb/config/* je dodat fajl *glibc.mh* koji definiše spisak objektnih fajlova koji služe sa pristupanje vrednosti *TLS* promenljive u tom slučaju i takođe u tom fajlu se definiše gore pomenuti makro `CROSS_GDB`.

Sadržaj *gdb/config/glibc.mh* fajla je prikazan na listingu 5.1.

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

```
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 novokreiranog *gdb/config/glibc.mh* fajla bio ispisan u krajnji *Makefile* trebalo je uvesti promenljivu koja proverava da li je u 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 *gdb/config/glibc.mh* fajla ukoliko je pomenuta opcija navedena, a u suprotnom ona ostaje prazna.

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

Listing 5.2: Izmena *gdb/configure* fajla.

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

Implementacija funkcija za pristupanje TLS promenljivoj

U direktorijumu *gdb/glibc-dep/* se nalazi srž funkcopnalnosti čitanja vrednsoti *TLS* promenljive iz datoteke jezgara sa platformi drugih procesorskih arhitektura. Kako standardna C biblioteka već ima implementirane svaku arhitekturalnu zavisnost u vezi *TLS*-a 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 se služe za pristupanje *TLS* promenljivih, očekuju da se na arhitekturi domaćina prirodno barata sa programima iste arhitekture. Izbegavanje modifikacije *libthread_db* biblioteke moguće je izbeći tako što se pomenute funkcije implementiraju u *GNU GDB*-u, na isti način kao u *glibc* biblioteci, 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 *TLS*-om. Ta funkcionalnost je dodata u funkciji `gdb_td_thr_tlsbase()` koja je dodata u fajl `gdb/glibc-dep/nptl_db/td_thr_tlsbase.c`. Sve funkcije koje bi trebalo da emu-

liraju ponašanje *TLS* funkcija standardne biblioteke koje su dodate u projekat *GNU GDB* imaju prefiks u imenu `gdb_`. Npr. GDB pandam funckiji `td_ta_new()` je `gdb_td_ta_new()`. Pomenuta funkcija je promenjena da bi alat *GNU GDB* imao inforamciju 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* se drugačije čita. Nije novo da za debagovanje višenitnih programa *GNU GDB* debagerom, čak i upotrebom domaćinske verzije, je neophodno da *GNU GDB* koristi istu veriziju *libthread_db* biblioteke kao i program koji se debaguje. To znači ako je program preveden sa verzijom *2.x* standardnom bibliotekom, alat mora korsiti istu verziju *2.x* tokom svog debagovanja. To se postiže eksplicitnim navođenjem putanje do *libthread_db* biblioteke komandom `set libthread-db-search-path`. Ukoliko se verzije *libthread_db* biblioteke ne poklapaju debager će ispisati upozorenje.

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

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

Listing 5.3: Inicijalizacija arhitekturalno yavisnih osobina o TLS.

```
1  case bfd_arch_mips:
2    tls_tcb_at_tp = 0;
3    tls_dtv_at_tp = 1;
4    forced_dynamic_tls_offset = -2;
5    no_tls_offset = -1;
6    tcb_alignment = 16;
7    break;
8  case bfd_arch_arm:
9    tls_tcb_at_tp = 0;
10   tls_dtv_at_tp = 1;
11   forced_dynamic_tls_offset = -2;
12   no_tls_offset = -1;
13   tcb_alignment = 0;
14   break;
```

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

Funkcija `native_check()` definisana u fajlu *gdb/glibc-dep/native-check.c* daje informaciju da li se *Multiarch* verzijom alata debaguje program domaćinske arhitekture. Ako je to slučaj, pristupanje vrednosti *TLS* promenljive se vrši kao do sada

uz pomoć funkcija iz standardne C biblioteke. Ukoliko to nije slučaj, pozovaju se novokreirane funkcije iz direktorijuma *gdb/glibc-dep/*.

Cela funkcionalnost se zapravo dodaje u *gdb/linux-thread-db.c* fajl 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 GDB i *libthread_db* biblioteke. Promenjeni su pokazivači na *TLS* funkcije u slučaju nedomaćinskih datoteka jezgara.

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

Listing 5.4: Postavljanje *callback* funkcija za nedomaćinske arhitekture.

```
#ifdef CROSS_GDB
 if (native_check(arch) != 0) {
  info->td_thr_tls_get_addr_p=gdb_td_thr_tls_get_addr;
  info->td_thr_tlsbase_p=gdb_td_thr_tlsbase;
 } else {
  //if it's host we want to keep old way of counting tls address
  TDB_DLSYM (info, td_thr_tls_get_addr);
  TDB_DLSYM (info, td_thr_tlsbase);
 }
#else
  TDB_DLSYM (info, td_thr_tls_get_addr);
  TDB_DLSYM (info, td_thr_tlsbase);
#endif
```

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

5.2 Alternativno rešenje

Drugi način implementacije je modifikacija *libthread_db* funkcija 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.

5.3 Unapređenje alata GNU GDB prilikom čitanja/pisanja 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 jezgara. 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 *MIPS* arhitekture, neophodnog za pristupanje bazičnih informacija o nitima. Kao što je napomenuto, 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 *MIPS* procesorsku arhitekturu. Te izmene su odrađene i prihvaćene od strane GNU zajednice. Takođe treba napomenuti da su te izmene ušle u izvršnu verziju alata.

Deo izvornog koda (u fajlu *bfd/elf32-mips.c*) izmene za čitanje informacija o procesu iz datoteke jezgra je prikazan na listingu 5.5.

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

```
1
2 case 128:                /* Linux/MIPS elf_prpsinfo */
3   elf_tdata (abfd)->core->pid
4    = bfd_get_32 (abfd, note->descdata + 16);
5   elf_tdata (abfd)->core->program
6    = _bfd_elfcore_strndup (abfd, note->descdata + 32, 16);
```

Deo izvornog koda (u fajlu *bfd/elf32-mips.c*) izmene za pisanje inforamcija o procesu u datoteku jezgra je prikazan na listinigu 5.6

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

```
...
  case NT_PRPSINFO:
    BFD_FAIL ();
    return NULL;

  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 *MIPS* 32-bitnu verziju. Napomenimo da izmena obuhvata i *MIPS* 64-bitnu verziju procesorske arhitekture.

5.4 Testiranje

Faza testiranja je jako važan deo životnog ciklusa razvoja softvera, te je veliki značaj pridodat ovoj sekvenci. Testiranje je izvršeno na raznim verzijama procesor-

skih arhitektura *ARM* i *MIPS*.

Takođe, implementirani su *DejeGNU* [1] testovi koji testiraju pristupanje vrednosti *TLS* promenljive iz datoteka jezgara razniih arhitektura. Zbog prirode problema koji se testira datoteke jezgra su generisane na različitim platformama i kompresovane zajedno sa izvršnim fajlom u *gdb/testsuite/gdb.multi/*, te se prilikom izvršavanja konkretnih testova ti fajlovi raspakuju i učitavaju u debager. Testovi obuhvataju izvršne fajlove sa *TLS*-om generisanih sa *2.19* i *2.22* verzijama standardne biblioteke.

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

Listing 5.7: Komanda za pokretanje testova.

```
make check
```

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

Listing 5.8: Rezultati izvršavanja testova.

```
                === gdb Summary ===
                # of expected passes  33715
                # of expected failures  196
                # of known failures  66
                # of unresolved testcases 5
                # of untested testcases  67
                # of unsupported tests  219
```

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

Listing 5.9: Rezultati izvršavanja testova.

```
                === gdb Summary ===

                # of expected passes  33718
                # of expected failures  196
                # of known failures  66
                # of unresolved testcases 5
```

```
7                    # of untested testcases  67
8                    # of unsupported tests  219
```

5.5 Upotreba alata

Da bi *GNU GDB* uspešno koristio naprednije tehinke analize višenitnih programa, uključujući i analizu *TLS*-a, potrebno je da mu se prosledi putanja do *libthread_db* biblioteke koja pripada istoj verziji standardne biblioteke kao i program koji se debaguje. To isto važi i za *Multiarch GNU GDB* 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 na listingu 5.10.

Listing 5.10: Višenitni test primer napisan u C kodu.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <pthread.h>
5
6  __thread int foo=0xdeadbeef;
7  pthread_t threads[5];
8
9  void* thread(void *e) {
10   int *i = (int*)e;
11   foo+=*i;
12   printf("foo is %x\n", foo);
13   sleep(10);
14   return (void*)0;
15 }
16
17 int main()
18 {
19   printf("init %x\n", foo);
20   int i;
```

```
21    for (i=0; i<5; i++) {
22      pthread_create(&threads[i], NULL, thread, &i);
23    }
24
25    for (i=0; i<5; i++) {
26      pthread_join(threads[i], NULL);
27    }
28
29    sleep(5);
30    abort();
31  }
```

Korišćenje debagera pre implementacije čitanja vrednosti *TLS* promenljive iz datoteke jezgra generisane na nekom uređaju sa ugrađenim računarom je prikazano na 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 se podešavaju putanje do standardne biblioteke za debagovanje niti koja ima istu verziju kao i izvršni fajl koji se debaguje. Trećom komandom se zadaje putanja do deljenih biblioteka koje je izvršni fajl koji se debaguje korsitio na uređaju sa ugrađenim računarom. Četvrta komanda učitava izvršni fajl koji se debaguje. Petom komandom se učitava datoteka jezgra koja je generisana prilikom neočekivanog prekidanja izvršavanja fajla koji se debaguje. Poslednjom komandom se pokušava čitanje vrednosti TLS *promenljive.* Kkorišćenje debagera nakon implementacije čitanja vrednosti *TLS* promenljive iz datoteke jezgra generisane na nekom uređaju sa ugrađenim računarom je prikazano na 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 se podešavaju putanje do standardne biblioteke za debagovanje niti koja ima istu verziju kao i izvršni fajl koji se debaguje. Trećom komandom se zadaje putanja do deljenih biblioteka koje je izvršni fajl koji se deba-

guje korsitio na uređaju sa ugrađenim računarom. Četvrta komanda učitava izvršni fajl koji se debaguje. Petom komandom se učitava datoteka jezgra koja je generisana prilikom neočekivanog prekidanja izvršavanja fajla koji se debaguje. Poslednjom komandom se uspešno čita vrednosti TLS *promenljive.*

Glava 6

Zaključak

Oba predloga implementacije su poslata *GNU* zajednici i u toku je pregled i analiza implementacije, čija se diskusija može videti na [2]. I prvi i drugi način imaju svoje prednosti i mane. Mana prvog načina jeste ta što svaka promena arhitekturno zavisnih vrednosti promenljivih koje učestvuju u računanju lokacije *TLS* promenljive u *glibc*-u zahteva izmenu i u *GNU GDB*-u što dodatno otežava održavanje koda. Drugi način zahteva izmene i u izvornom kodu *GNU GDB*-a i *glibc*-a, ali ukoliko želimo da na sistemima arhitekture domaćina baratamo sa programima koji su prevedeni i izvršavani na nekim ciljnim arhitektura, što je za arhitekturu domaćina neprirodno, kompromisi jesu neophodni.

Literatura

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

[2] *Diskusija GNU zajednice*. on-line at: https://www.gnu.org/software/dejagnu/.

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

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

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

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

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

[12] *LLVM projekat*. on-line at: http://llvm.org/.

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

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

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

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

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

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

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 debag informacija od strane programskih prevodilaca. Na temu opisanu u master radu je objavio rad na konferenciji TELFOR [17]. Trenutno radi na LLVM projektu, tačnije na poboljšanju tog prevodioca prilikom generisanja debag informacija.

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)*