UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO

Jure Žbontar

POVEZOVALNIK IN DINAMIČNI NALAGALNIK ZA PROCESOR HIP

Diplomska naloga
na univerzitetnem študiju

Mentor: doc. dr. Boštjan Slivnik

Ljubljana, 2008

Besedilo je oblikovano z urejevalnikom besedil LaTeX,
slike pa so napisane v programskem jeziku METAPOST.

Namesto te strani **vstavite** original izdane teme diplomskega dela s podpisom mentorja in dekana ter žigom fakultete, ki ga diplomant dvigne v študentskem referatu, preden odda izdelek v vezavo!

Zahvala

Iskreno bi se rad zahvalil mentorju doc. dr. Boštjanu Slivniku, ki mi je pri izbiri teme pustil popolnoma proste roke in me pri delu vodil strokovno in nikoli vsiljivo.

Zahvalil bi se rad tudi vsem, ki so delo prebrali in mi pomagali odpraviti številne napake, ki so se vanj prikradle. To so, v abecednem vrsten redu: Boštjan Slivnik, Dani Žbontar, Naja Pogačar (povzetek v angleščini), Rok Bozovičar in Tomo Žbontar. Rad bi se zahvalil tudi sestri Zori, ki me je v zgodnjih jutranjih urah vedrila z igro spomina, mačku Muniju, ki mi je med pisanjem grel kolena in staršema Danici in Tomotu, ki sta ustvarila idealno okolje za izdelavo diplomske naloge (mir, hrana, postelja). Na tem mestu bi najraje naštel vse profesorje in asistente, ki so me z njihovim veseljem do dela in obširnim znanjem navdušili nad nekaterimi temami računalništva in matematike. Ker pa se bojim, da seznam ne bo vseboval vseh, bodo morali za enkrat ostati neimenovani. Ta zahvala ne bi bila popolna, če ne bi omenil tudi svojih super sošolk in sošolcev, ki so skozi vsa štiri leta v učilnicah ustvarjali prijateljsko vzdušje.

Na koncu še zahvala, ki bi jo bilo morda bolje uvrstiti v razdelek “Opravičila”, če bi ta obstajal. Zahvaljujem se Barbari Dolenc, ki je tako nesrečno izvedela komaj en dan pred odhodom, da zaradi mojih študijskih obveznosti, potovanja v Kolumbijo in Venezuelo preprosto ne bo.

Adi Byron

Kazalo

Slike

Tabele

Izpisi

Seznam uporabljenih kratic

a.out	Assembler Output
BSD	Berkeley Software Distribution
bss	Block Started by Symbol
COM	Command file - format objektne datoteke
DAG	Directed Acyclic Graph
DOS	Disk Operating Sistem
ELF	Executable and Linkable Format
GCC	GNU Compiler Collection
GOT	Global Offset Table
HIP	Hipotetični računalnik prof. dr. D. Kodeka
hiputils	veriga programov
MIPS	Microprocessor without Interlocked Pipeline Stages
PIC	Position Independent Code
RISC	Reduced Instruction Set Computer

Povzetek

V delu je predstavljena veriga programov *hiputils*, ki vsebuje zbirnik, povezovalnik, dinamični nalagalnik, simulator in program za ustvarjanje statičnih knjižnic za procesor HIP. Natančno je opisan postopek ustvarjanja, povezovanja in nalaganja statičnih in dinamičnih knjižnic s programi verige hiputils. Določen je tudi format objektnih datotek, statičnih in dinamičnih knjižnic.

Poleg izdelave razvojnega okolja je predstavljeno povezovanje in nalaganje programov in knjižnic. V delu so opisani tudi formati objektnih datotek, skupaj s primeri obstoječih formatov (COM, a.out, ELF). Natančno so opisane tri glavne naloge povezovalnikov: razporeditev segmentov, upravljanje simbolov in premestitve. Podan je tudi opis knjižnic - tako statičnih, kot tudi dinamičnih, ter opis postopka dinamičnega nalaganja in premestitve. Opisan je tudi postopek, ki omogoča da se strojna koda lahko izvaja na poljubnem začetnem naslovu (pozicijsko neodvisna koda).

Ključne besede:

povezovalnik, nalagalnik, zbirnik, HIP, objektna datoteka, knjižnica, pozicijsko neodvisna koda

Abstract

In this work a collection of programs, called *hiputils* is presented. The toolchain constitutes an assembler, linker, dynamic loader, simulator and a static library creation utility for the HIP processor. A precise description of the process of creating, linking and loading of static and dynamic libraries in hiputils is given. A format for object files, static and dynamic libraries is also defined.

Beside hiputils, linking and loading of programs and libraries is also described. Several object file formats, including COM, a.out and ELF are studied and compared. The three main tasks of linkers: storage allocation, symbol management and relocation are detailed. A description of libraries - static as well as dynamic - is also given, along with a description of dynamic loading and relocation. A mechanism, which allows code to run at an arbitrary start address is also depicted (position independent code).

Keywords:

linker, loader, assembler, HIP, object file, library, position independent code

Poglavje 1

Uvod

V tem poglavju bomo odgovorili na vprašanje, kaj so povezovalniki in nalagalniki ter kako delujejo. Ker s povezovalniki in nalagalniki nimamo neposrednega stika, bi sprva lahko pomislili, da je njihova naloga nepomembna, njihov obstoj pa ogrožen. V nadaljevanju se bomo prepričali, da to ni res. Preden navedemo ustrezne definicije pojmov, si raje oglejmo zgodovinski pregled povezovalnikov in nalagalnikov ter njihov dosedanji razvoj.

Sprva je delo povezovalnika v celoti opravljal programer. Prvi računalniki niso poznali prevajalnikov in zbirnikov, zato je bila naloga prevajanja programa v jezik, ki ga ta razume, odgovornost programerja. Ta je na list papirja napisal program, ki ga je nato ročno prevedel v strojno kodo. Simboličnim imenom, če jih je program sploh vseboval, je bilo potrebno ročno izračunati dejanski naslov, kar že samo po sebi ni bilo prijetno opravilo. Pravi problem je nastopil, ko je bilo treba program spremeniti. To je zahtevalo pregled vseh simbolov in ponoven izračun njihovih novih naslovov. Problem določanja dejanskih naslovov simbolom so, vsaj za nekaj čas, rešili zbirniki. Programerji so bili razbremenjeni nalog ročnega prevajanja strojnih ukazov in računanja dejanskih naslovov, saj so to namesto njih opravljali zbirniki.

Kmalu so programi postali tako veliki, da je bilo nepraktično vsak program prevesti kot celoto, zato so programerji svoje programe razdelili na več modulov, ki jih je zbirnik obdelal posebej, enega za drugim. Če je bilo potrebno popraviti kakšen del programa, so se ponovno prevedli le tisti moduli, ki so bili spremenjeni. To je bila na takratnih računalnikih pomembna prednost,

saj je bila procesorska moč takrat skrajno omejena[1]. Istočasno se je pojavila tudi potreba po knjižnicah. Knjižnice so zbirke programskih modulov, ki jih lahko povežemo v glavni program. Programerjem tako ni več potrebno v vsakem programu definirati pogosto uporabljanih procedur kot na primer `izpiši`, `uredi` ali `koreni`, saj so te že prisotne v standardnih knjižnicah, ki jih lahko preprosto vključimo v glavni program.

Ker se moduli programa prevedejo neodvisno od ostalih, zbirnik dejansko nikoli ne vidi vseh modulov programa kot celote. To pomeni, da zbirnik ne bo imel na voljo dovolj informacij za izračun vseh naslovov simbolov, ki se v modulu uporabljajo. Predstavljajmo si program, ki uporablja proceduro `koreni`, definirano v matematični knjižnici. Ko ta program prevedemo, zbirnik ne ve ničesar o matematični knjižnici (niti to, če knjižnica sploh obstaja), zato v izhodni datoteki ne more zamenjati simbole `koreni` z dejanskim naslovom procedure `koreni`. Potrebni so torej programi, ki na vhodu sprejmejo zbirko modulov, ki še vedno vsebujejo simbolične naslove, te naslove razrešijo in module združijo v eno samo, veliko, izvršljivo datoteko. Te programe imenujemo *povezovalniki* in so na računalniških sistemi prisotni že od samega začetka.

Z razvojem prvih operacijskih sistemov so se pojavili dodatni zapleti. Programi si po novem naslovni prostor delijo z operacijskim sistemom in morda tudi z ostalimi programi. Lahko se torej zgodi, da je začetni naslov programa, ki ga je določil povezovalnik že zaseden, saj smo predhodno na to lokacijo morda že naložili kakšen program. Iz zagate nas reši poseben program, imenovan *nalagalnik*. Njegova naloga je naložiti program v glavni pomnilnik na začetni naslov, če je ta prost, sicer je potrebno program premestiti na drugo lokacijo. Pri tem je seveda potrebno popraviti vse naslove, ki se v programu pojavljajo, podobno, kot smo to že storili pri povezovanju. Z razvojem navideznega pomnilnika so se nalagalniki poenostavili. Operacijski sistem lahko vsakemu procesu dodeli svoj lasten naslovni prostor, zato se vsak program ponovno lahko začne na konstantnem začetnem naslovu.

Skoraj vsi sodobni operacijski sistemi omogočajo sočasno izvajanje več programov. Ker se nekateri deli programa med izvajanjem ne spreminjajo (na primer strojna koda), je možno doseči, da si več procesov deli isti segment, ki ga je potrebno v pomnilnik prenesti samo prvič. Povezovalniki so zadolženi, da v modulih ločijo dele programa, ki se med izvajanjem ne spreminjajo in tiste, ki se. Nalagalniki lahko sedaj segmente, ki se ne spreminjajo, v po-

[1] Tudi danes ostaja ta prednost pomembna. Čeprav so se procesorji v zadnjih 40 letih postali neprimerno hitrejši, ostaja proces prevajanja velikih programov še vedno zelo počasen. Jedro operacijskega sistem Linux se na mojem računalniku prevaja skoraj eno uro!

mnilnik naložijo na poseben način, ki omogoča večim procesom uporabo istega segmenta.

Izkaže se, da imajo tudi različni programi veliko skupnih segmentov. Skoraj vsi programi napisani v programskem jeziku C uporabljajo podprograme (na primer `printf`), ki so definirane v dinamični knjižnici `libc.so`. Sodobni povezovalniki, nalagalniki in operacijski sistemi omogočajo, da se taka knjižnica v pomnilnik prenese samo enkrat, uporablja pa se v vseh programih, ki kličejo kakšno od podprogramov, ki so v knjižnici definirane. Postopek morda spominja na tistega, ki smo ga predstavili v prejšnjem odstavku, vendar se v primeru dinamičnih knjižnic pojavijo določeni problemi, ki jih je prej potrebno rešiti. Kako rešiti te in tudi ostale probleme, ki pri povezovanju in nalaganju nastopijo, bo seveda ena izmed glavnih tem te diplomske naloge.

1.1 Cilj diplomske naloge

Cilj diplomske naloge je izdelati razvojno okolje za procesor HIP, ki omogoča statično in dinamično povezovanje ter nalaganje programov in knjižnic. Predstavili bomo verigo programov *hiputils*[2], ki sem jo razvil posebej za namene te diplomske naloge. Veriga hiputils vsebuje zbirnik, povezovalnik, dinamični nalagalnik, simulator in program za ustvarjanje statičnih knjižnic, poleg tega določa tudi lasten format objektnih datotek. V vsakem poglavju bomo posvetili kakšen razdelek temu, da opišemo kako so pojmi, ki so bili v poglavju predstavljeni, implementirani v programih verige hiputils. Uporabljene rešitve pa bomo primerjali z rešitvami v že obstoječih implementacijah.

1.2 Povezovalniki in nalagalniki

Naloge, ki jih opravljajo povezovalniki in nalagalniki se močno prekrivajo. V grobem jih lahko razdelimo v štiri skupine:

Nalaganje. Nalaganje je proces kopiranja programa ali knjižnice iz trdega diska v pomnilnik.

[2]Ime *hiputils* (izgovorjeno "hip jutils") sem izbral, saj spominja na izraz *binutils* - znano verigo programov, ki vsebuje zbirnik, povezovalnik in ostale podporne programe.

Razporeditev segmentov. Kot bomo videli v nadaljevanju, je vsak modul sestavljen iz enega ali več segmentov. Pri povezovanju več modulov, je potrebno vse segmente istega tipa združiti v skupni segment.

Premestitev. Zbirniki prevedejo nizkonivojski zbirni jezik v strojno kodo, ki jo razume in izvršuje CPE. Vsem simbolom, ki jih je programer morda uporabil, je potrebno med procesom prevajanja določiti njihov dejanski naslov, kar žal ni vedno mogoče. V dveh primerih je potrebno računanje dejanskih naslovov preložiti na nekoliko kasneje:

- V programu so uporabljeni simboli, ki so definirani drugje. Primer je klic procedure iz knjižnice.
- Program se naloži na lokacijo, ki je drugačna od tiste, ki jo je pri prevajanju določil zbirnik.

Če zbirnik naleti na naslov, ki ga med prevajanjem ne more izračunati, mora o tem nujno obvestiti povezovalnik oziroma nalagalnik. To stori tako, da v objektno datoteko vstavi *premestitveni vnos* za vsak naslov, ki ga je potrebno spremeniti.

Razreševanje simbolov. Program je ponavadi sestavljen iz več modulov. Če je v modulu A uporabljen simbol s, ki je definiran v modulu B, je naloga povezovalnika, da razreši nedefiniran simbol s v modulu A, ter na njegovo mesto zapiše dejanski naslov simbola s.

Sedaj, ko smo spoznali osnovne naloge povezovanja in nalaganja, lahko bolj natančno definiramo pojma *povezovalnik* in *nalagalnik*.

Definicija *Nalagalnik* je program ali knjižnica, ki opravlja naloge nalaganja, premestitve in razreševanja simbolov.

Podobno lahko definiramo tudi povezovalnik.

Definicija *Povezovalnik* je program, ki opravlja naloge razporeditve segmentov, premestitve in razreševanja simbolov.

Zgornji definiciji bosta za namene te diplomske naloge povsem zadostovali. Na tem mestu omenimo še, da obstajajo tudi nalagalniki, ki ne opravljajo premestitve in razreševanja simbolov. Obstajajo pa tudi *povezovalni nalagalniki*, ki združujejo vse štiri zgoraj naštete naloge povezovanja in nalaganja.

Povezovalniki imajo posebno lastnost, ki si jo morda delijo le še z razhroščevalniki. Danes so to edini razširjeni programi, ki neposredno spreminjajo strojno

kodo, shranjeno v objektnih datotekah. Ta lastnost jih intimno poveže z arhitekturo, oziroma naborom ukazov procesorja, za katerega so bili napisani. Prostora za napake ni! Že najmanjša nepravilna sprememba ima lahko za izvajanje sicer pravilnega programa, katastrofalne posledice.

1.3 Primer povezovanja

Da bi naloge povezovanja bolje razumeli, si delovanje povezovalnika oglejmo na preprostem primeru. Primer je napisan v zbirniku hipotetičnega procesorja HIP. Prevajanje, povezovanje, nalaganje in simulacijo pa izvajajo programi verige hiputils.

Izpis 1.1: Izvorna datoteka `m.s`.

```
1  #include <hip/regdef.h>
2  #include <hip/asm.h>
3
4          .data
5  niz:    .asciz "Hello, world!\n"
6
7          .text
8          .globl _start
9  _start:
10         /* a0 <- naslov spremenljivke niz */
11         lhi     a0,%hi(niz)
12         addi    a0,a0,%lo(niz)
13
14         /* klic procedure foo v datoteki foo.s */
15         lhi     t0,%hi(foo)
16         call    ra,%lo(foo)(t0)
17
18         /* izhod iz programa */
19         syscall EXIT
```

Izpis 1.1 prikazuje glavni program. Program je sicer preprost, a je s stališča povezovalnika vseeno zanimiv. V vrsticah 11 in 12 je uporabljen simbol `niz`, ki je sicer definiran v isti datoteki, vendar njegovega naslova še ne moremo izračunati, saj ne vemo, kam bo povezovalnik premaknil segment .data. Prav tako ne moremo izračunati naslova spremenljivke `foo`, ki se pojavi v vrsticah 15 in 16. Simbol je namreč definiran drugje.

Datoteko `m.s` prevedemo z naslednjima ukazoma:

```
$ cpp m.s m.cpp
$ as.py -o m.o m.cpp
```

Rezultat prevajanja je objektna datoteka `m.o` (izpis 1.2) zapisana v formatu, ki ga bomo spoznali v poglavju 2. Datoteka vsebuje opis dveh segmentov: .data in .text, velikosti 15 in 20 bajtov, zaporedoma. Ker zbirnik med prevajanjem ne more določiti dejanskih naslovov obeh segmentov, jih začasno postavi na naslov nič (naslove bo kasneje popravil povezovalnik).

Izpis 1.2: Objektna datoteka `m.o`.

```
1  LINK
2  0 2 5 4
3
4  # needs
5
6  # segments
7  .data            00000000    F  RWP
8  .text            00000000   14   RP
9
10 # symbols
11 .data            00000000    1    S
12 .text            00000000    2    S
13 _start           00000000    2    D
14 foo              00000000    2    U
15 niz              00000000    1    L
16
17 # relocations
18 0       2       1       U2
19 4       2       1       L2
20 8       2       4       U2
21 C       2       4       L2
22
23 # data
24 48656C6C6F2C20776F726C64210A00
25 1C040000008400001C080000B51F000078000000
```

Tabeli segmentov sledi opis petih simbolov. V tabeli simbolov se med drugim nahajajo globalni simbol `_start`, lokalni simbol `niz` in nedefiniran simbol `foo`. Simbol `_start` imenujemo tudi vstopna točka programa, ki določi lokacijo, kjer se izvajanje programa začne. Morda je nekaterim programerjem čudno, da lahko izhod prevajanja vsebuje simbole, ki so nedefinirani. Navajeni so namreč, da jih prevajalniki višjenivojskih jezikov vedno opozorijo na to očitno

napako. Vendar z nedefiniranimi simboli v objektnih datotekah ni nič narobe. Zbirnik sam po sebi ne zna razrešiti vseh simbolov, ki nastopajo v programu, saj med prevajanjem vidi samo eno datoteko, ki sestavlja program in ne vseh naenkrat. Spomnimo se, da izhod zbirnika ni izvršljiva datoteka. To postane šele takrat, ko z njo opravi povezovalnik. Takrat pa seveda nedefiniran simbol v izhodu pomeni usodno napako[3].

Tabeli simbolov sledijo premestitveni vnosi. Premestitveni vnosi niso nič drugega kot navodila povezovalniku, kje in kako naj popravi naslove v strojni kodi. V našem konkretnem primeru potrebujemo za pravilno izvajanje programa štiri premestitvene vnose. Opazimo, da za razrešitev enega simbola potrebujemo dva vnosa. To je posledica arhitekture procesorja, ki ga uporabljamo. Ker so vsi ukazi procesorja HIP dolgi 32 bitov, je nemogoče v en ukaz stlačiti celoten 32-bitni naslov. Ta težava je značilna za procesorje tipa RISC, med katere sodi tudi procesor HIP. Problem rešimo tako, da naslov razdelimo na zgornjih 16 bitov in spodnjih 16 bitov, nato z ukazom `lhi`, ki mu sledi ukaz `addi`, po delih prenesemo naslov v ustrezen register. Prvi premestitveni vnos v našem primeru pomeni, da je treba na lokacijo 0, segmenta .text, vpisati zgornjih 16 bitov (U2) naslova simbola `niz`, ki je definiran v segmentu .data. Na koncu datoteke je shranjena še strojna koda segmenta .text in ustrezno predstavljeni podatki segmenta .data.

Izpis 1.3: Izvorna datoteka `foo.s`.

```
1  #include <hip/regdef.h>
2  #include <hip/asm.h>
3
4          .text
5          .globl foo
6  foo:
7          /* klic procedure operacijskega sistema */
8          syscall PRINTF
9
10         /* skok iz procedure */
11         j       0(ra)
```

Poglejmo si sedaj še datoteko `foo.s`. V datoteki je definirana procedura `foo`, ki jo uporablja glavni program. V telesu procedure je uporabljen sistemski

[3]To seveda drži le za statično povezovanje. Pri dinamičnem povezovanju se napaka sporoči šele ob izvajanju programa. Pri nekaterih implementacijah pa šele ob prvem klicu nedefinirane procedure.

klic (ukaz `syscall`). Sistemski klici se razlikujejo od navadnih klicev procedur z ukazom `call`. Pri klicu sistemske procedure se izvajanje programa za hip prekine, nadzor prevzame operacijski sistem, ki izvrši zahtevano akcijo, nato se izvajanja programa nadaljuje iz mesta, kjer je bilo prekinjeno.

Izpis 1.4: Objektna datoteka `foo.o`.

```
1  LINK
2  0 1 2 0
3
4  # needs
5
6  # segments
7  .text            00000000    8   RP
8
9  # symbols
10 .text            00000000    1    S
11 foo              00000000    1    D
12
13 # relocations
14
15 # data
16 78000001B3E00000
```

Podobno kot datoteko `m.s`, je potrebno datoteko `foo.s` najprej prevesti.

```
$ cpp foo.s foo.cpp
$ as.py -o foo.o foo.cpp
```

Rezultat prevajanja je predstavljen v izpisu 1.4. Tokrat vsebuje objektna datoteka le segment .text. Iz tabele simbolov razberemo, da je v tej datoteki definiran simbol `foo`. Tabela premestitvenih vnosov je prazna, saj se v programu nismo sklicevali na nobene simbole[4].

Sedaj je na vrsti za nas morda še najbolj zanimiv del prevajalnega procesa: povezovanje. Iz objektnih datotek `m.o` in `foo.o` bo povezovalnik sestavil izvršljivo datoteko `a.out`.

```
$ ld.py -o a.out m.o foo.o
```

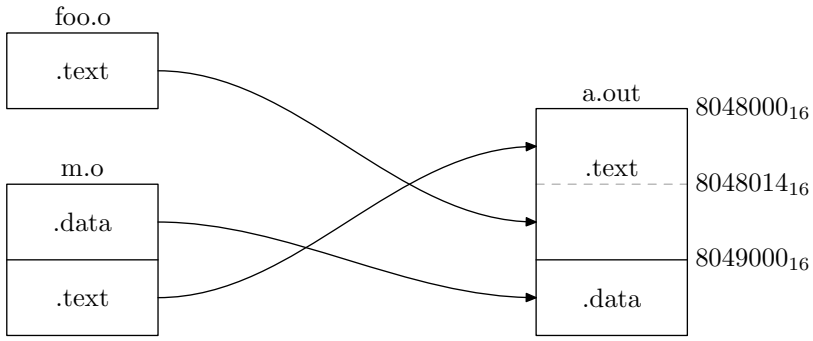
[4]Simbola `PRINTF` ne smemo šteti med prave simbole, saj gre le za makro ukaz, ki je definiran v datoteki `hip/asm.h`. Program `cpp` poskrbi, da se še pred prevajanjem razširijo vsi makro ukazi v izvorni datoteki. V trenutni implementaciji se ukaz PRINTF nadomesti s številom 1.

Izpis 1.5: Izvršljiva datoteka `a.out`.

```
1  LINK
2  0 2 4 0
3
4  # needs
5
6  # segments
7  .text            08048000   1C   RP
8  .data            08049000   10  RWP
9
10 # symbols
11 .data            08049000    2    S
12 .text            08048000    1    S
13 _start           08048000    1    D
14 foo              08048014    1    D
15
16 # relocations
17
18 # data
19 1C040805008490001C080805B51F80147800000078000001B3E00000
20 48656C6C6F2C20776F726C64210A0000
```

Izvršljiva datoteka `a.out`, ki je nastala pri povezovanju, je prikazana v izpisu 1.5. Prva stvar, ki jo opazimo je ta, da je format izvršljivih datotek identičen formatu objektnih. Opazimo tudi, da se segment .text sedaj začne na naslovu 8048000_{16}, njegova velikost je vsota velikosti segmentov .text obeh objektnih datotek. Prestavljen je bil tudi segment .data, ki se sedaj prične na naslovu 8049000_{16}, njegova velikost je navzgor zaokrožena na naslednje število, ki je deljivo s štiri. Ta proces imenujemo dodelitev segmentov in je nazorno prikazan na sliki 1.1. V tabeli simbolov je definiran simbol `foo`. Tokrat je znan tudi njegov dejanski naslov med izvajanjem (8048014_{16}). Tabela premestitev je prazna, saj je imel povezovalnik dovolj informacij, da je lahko uspešno izvedel vse štiri premestitve.

Rezultat povezovanja je torej izvršljiva datoteka `a.out`, ki jo lahko naložimo v pomnilnik in zaženemo. Nalagalnik v našem primeru ne bo imel preveč dela. Vse kar mora storiti je naložiti segment .text od naslova 8048000_{16} dalje, segment .data pa od naslova 8049000_{16} dalje. Na koncu sledi še skok na lokacijo, ki je označena s simbolom `_start` (8048000_{16}). V verigi hiputils je prisoten tudi simulator z nalagalnikom. Poskusimo sedaj zagnati datoteko `a.out`:



Slika 1.1: Preprost primer razporeditve segmentov in dodelitve začetnih naslovov.

```
$ sim.py a.out
Hello, world!
```

Nalagalniki so v resnici veliko bolj zanimivi kot to morda prikazuje naš preprost primer. V poglavju 4.2, bomo spoznali dinamične nalagalnike, ki lahko podobno kot povezovalniki, razrešijo nedefinirane simbole in izvajajo premestitve. Posebej zanimiv je primer dinamičnega nalagalnika iz operacijskega sistema Linux, ki je v celoti implementiran kot dinamična knjižnica.

Poglavje 2

Objektne datoteke

Rezultat prevajanja je objektna datoteka. Več objektnih datotek povezovalnik združi v eno samo, ki jo nato nalagalnik naloži v pomnilnik. V tem poglavju si bomo ogledali, kaj vse sestavlja tipično objektno datoteko. Sledijo tudi opisi treh formatov objektnih datotek, ki so se uporabljali v preteklosti ali pa se uporabljajo še danes. Na koncu bomo definirali še format objektne datoteke, ki jo uporabljajo programi verige hiputils.

2.1 Sestavni deli objektne datoteke

Tipično objektno datoteko sestavlja naslednjih pet odsekov:

Glava: V glavi objektne datoteke so shranjeni splošni podatki o njeni vsebini. Ponavadi so to podatki o velikosti segmentov, imenu izvorne datoteke, datumu in uri nastanka, velikosti tabele simbolov, številu premestitvenih vnosov, ...

Strojni ukazi: Vsaka objektna datoteka mora vsebovati tudi prevedene ukaze in podatke, ki jih je ustvaril zbirnik oziroma prevajalnik.

Premestitveni vnosi: Seznam vseh lokacij v strojni kodi, ki jih je treba pred izvajanjem popraviti.

Tabela simbolov: Tabela simbolov vsebuje globalne simbole, ki so definirani v tem modulu in tudi nedefinirane simbole, ki so definirani drugje. Včasih so v tabeli simbolov tudi posebni simboli, ki jih definira povezovalnik.

Podatki za razhroščevanje: V tem delu so shranjeni podatki o strojni kodi, ki se ne uporabijo pri povezovanju, ampak so koristni pri razhroščevanju. Primer takih podatkov je na primer celotna izvorna datoteka, preslikava med vrsticami strojne in izvorne kode, lokalni simboli, definicije preprocesorja, ...

Seveda ni nujno, da vsi formati objektnih datotek vsebujejo prav vse naštete tipe informacij. Kot bomo videli v nadaljevanju (razdelek 2.2), je za osnovne potrebe dovolj, če objektna datoteka vsebuje le strojne ukaze.

Format objektne datoteke, ki se uporablja samo za povezovanje, je lahko različen od formata, ki se uporablja pri nalaganju. Objektna datoteka je lahko prisotna v treh oblikah:

Povezljiva objektna datoteka vsebuje med drugim tudi tabelo simbolov in premestitvene vnose, ki so za uspešno povezovanje nujno potrebni. Strojna koda je ponavadi razdeljena na več manjših, logično povezanih enot, ki jih imenujemo segmenti.

Izvršljiva objektna datoteka vsebuje strojno kodo, ki je ponavadi poravnana z velikostjo strani navideznega pomnilnika, kar nam omogoča, da jo lahko preprosto preslikamo v naslovni prostor procesa. Tabela simbolov je lahko prazna, razen v primeru, ko uporabljamo simbole, ki so definirani v kakšni dinamični knjižnici. Tudi premestitvenih vnosov ne potrebujemo. Strojna koda je razdeljena na segmente. Ponavadi je datoteka razdeljene na segmente, ki jih lahko samo beremo in segmente, v katere lahko tudi pišemo.

Naložljiva objekta datoteka je podobna izvršljivi, s to razliko, da lahko vsebuje tudi premestitvene vnose, ki so potrebni pri dinamičnem nalaganju. Tabela simbolov pa vsebuje globalne simbole, ki so v datoteki definirani.

Objektna datoteka je lahko v eni od zgoraj navedenih oblik, lahko pa je tudi kombinacija dveh ali celo vseh treh. Nekateri formati podpirajo vse tri oblike, spet drugi morda le eno. V nadaljevanju si bomo pogledali nekaj popularnih formatov objektnih datotek.

2.2 Objektna datoteka COM

Naš pregled začnimo pri najbolj preprostem formatu. Kot smo že zgoraj namignili, je za preproste potrebe dovolj, če objektna datoteka ne vsebuje nič drugega kot samo strojno kodo. Format objektne datoteke COM je najbolj

razširjen primer tega tipa in se je uporabljal na operacijskem sistemu MS-DOS. Datoteka COM je sestavljena dobesedno samo iz strojne kode.

Nalagalnik najprej poišče dovolj velik kos pomnilnika. Na odmik 100_{16} nato naloži vsebino izvršljive datoteke, v lokacije na odmiku 0 do FF_{16} pa se prenesejo argumenti iz ukazne vrstice in drugi parametri. Vsi segmentni registri procesorja x86 kažejo na začetek dodeljenega kosa pomnilnika, register `sp` pa na konec segmenta, saj sklad na procesorjih intel raste navzdol proti naslovu 0. Nalagalnik na koncu skoči na začetni naslov in s tem se izvajanje programa začne.

Naslovi znotraj segmenta se interpretirajo relativno glede na segmenti register, zato programi, ki jih lahko stlačimo v en 64KB velik segment, ne potrebujejo nobene premestitve. Za programe, ki so preveliki, da bi jih lahko naložili v en segment, je za premestitev zadolžen programer. Ta pomanjkljivost je odpravljena v formatu EXE, ki je nadomestil format COM, a ga ne tem mestu ne bomo predstavili. Njegov opis lahko najdete v literaturi [4].

2.3 Objektna datoteka a.out

Operacijski sistemi danes vsakemu procesu ustvarijo njegov lasten naslovni prostor. V tem primeru se proces povezovanja in nalaganja poenostavi, saj lahko povezovalnik predpostavi fiksen začetni naslov, ki bo med nalaganjem gotovo na voljo. Na ta način se vse premestitve lahko izvršijo že v procesu povezovanja. Format objektne datoteke a.out je bil zasnovan ravno za tak scenarij.

2.3.1 Izvršljiv format objektne datoteke a.out

V najbolj osnovni izvedbi je datoteka a.out sestavljena iz glave, ki ji sledi strojna koda in začetne vrednosti podatkov. Kmalu so razvijalci strojno kodo in podatke ločili v dva segmenta, imenovana text in data. Tako je sedaj možno, da si več procesov istega programa med seboj deli skupno kopijo segmenta text. To lahko storimo, saj se segment text med izvajanjem programa ne spreminja in se pred izvajanjem vedno naloži na isti naslov. Na ta način lahko prihranimo kar nekaj pomnilnika, saj se na modernih operacijskih sistemi pogosto dogaja,

Izpis 2.1: Glava datoteke a.out

```
struct exec {
        u_int32_t a_midmag; // identifikacijska stevilka
        u_int32_t a_text;   // velikost segmenta text v bajtih
        u_int32_t a_data;   // velikost segmenta data v bajtih
        u_int32_t a_bss;    // velikost segmenta bss v bajtih
        u_int32_t a_syms;   // velikost tabele simbolov
        u_int32_t a_entry;  // naslov vstopne tocke programa
        u_int32_t a_trsize; // stevilo premestitvenih vnosov za segment text
        u_int32_t a_drsize; // stevilo premestitvenih vnosov za segment data
};
```

da se hkrati izvaja več kopij istega programa[1].

Glava datoteke a.out

Format glave se od ene verzije UNIX-a do druge spreminja. Na sliki 2.1 je prikazana verzija, ki je bila uporabljena v operacijskem sistemu OpenBSD [10].

Prvi štirje bajti datoteke (polje `a_midmag`) vsebujejo identifikacijsko številko (magic number), ki pove, da imamo opravka z datoteko v formatu a.out. Za identifikacijsko število je vedno uporabljeno pravilo debelega konca, zato lahko objektne datoteke formata a.out uporabljamo na vseh procesorjih, ne glede na to, kako so v pomnilniku shranjeni sestavljeni operandi. Iz identifikacijskega števila lahko razberemo tudi shemo nalaganja. V nadaljevanju si bomo ogledali sheme NMAGIC, ZMAGIC in QMAGIC. Vsebina ostalih polj glave je razložena v komentarjih na sliki 2.1.

NMAGIC

Sedaj si bomo ogledali shemo nalaganja NMAGIC. Ko nalagalnik odpre datoteko v formatu a.out, stori naslednje korake:

1. Preberi glavo objektne datoteke in preveri identifikacijsko številko.

[1] Trenutno se na mojem računalniku izvaja devet lupin bash, prav toliko emulatorjev terminala xterm, štirje urejevalniki teksta vi, dva pregledovalnika dokumentov xpdf in dva brskalnika.

2. Preveri ali se morda v pomnilniku že nahaja segment text datoteke, ki jo nalagamo. V tem primeru je potrebno ta segment le preslikati v naš naslovni prostor, sicer ustvari nov segment, v katerega iz datoteke skopira strojne ukaze in ga preslika v naslovni prostor. Označi segment samo za branje, da ga bomo lahko delili z drugimi procesi.

3. V pomnilniku ustvari segment data in ga preslikaj v naslovni prostor. V ta segment iz datoteke skopiraj podatke in z vrednostjo nič zapolni preostali del segmenta, ki predstavlja segment bss. Segment bss uporabljamo za podatke z začetno vrednostjo nič. Ker je začetna vrednost podatkov v segmentu bss znana, ga ni potrebno vključiti v objektno datoteko.

4. Ustvari nov segment, kjer bo živel sklad in vanj prenesi argumente iz ukazne vrstice.

5. Nekaterim registrom nastavi začetne vrednosti in skoči na naslov, ki je shranjen v polju `a_entry`.

ZMAGIC

Shema NMAGIC sicer deluje pravilno, na sistemih z navideznim pomnilnikom pa lahko postopek izboljšamo. Pri shemi NMAGIC za vsak segment data in text dodelimo nov kos navideznega pomnilnika. Ker je datoteka a.out že shranjena na disku, jo lahko neposredno preslikamo v naslovni prostor. S tem privarčujemo nekaj prostora na trdem disku, pa tudi nekaj časa, saj mora navidezni pomnilnik naložiti le tiste strani, ki jih proces dejansko naslavlja, ne pa cele datoteke.

Shema ZMAGIC zahteva nekaj sprememb v formatu a.out. Ker datoteko neposredno preslikamo v naslovni prostor procesa, mora struktura objektne datoteke odražati dejansko razporeditev segmentov v pomnilniku. Konkretno to pomeni, da moramo velikosti segmentov poravnati z velikostjo strani navideznega spomina. Tipične velikosti strani na današnjih računalnikih se gibljejo med 4KB in 16KB. To pomeni, da se na računalnikih z velikostjo strani 4KB, glava, ki je sicer velika le nekaj bajtov, napihne na 4KB. Tudi velikost segmenta text moramo povečati, da bo deljiv s 4K. V povprečju torej izgubimo 2KB. Velikost segmenta data, lahko ostane enaka, saj mu logično sledi segment bss, ki ga pred izvajanjem nalagalnik tako ali tako zapolni z bajti z vrednostjo nič.

QMAGIC

Shema ZMAGIC sicer odpravi nekatere pomanjkljivosti sheme NMAGIC, vendar to stori na račun veliko izgubljenega prostora na trdem disku.

Izboljšava, ki jo lahko naredimo je, da prestavim glavo datoteke kar v segment text, pred kodo. To pomeni, da se sedaj koda ne bo začela na naslovu nič, ampak nas to prav nič ne moti. Sploh je naslov nič posebej slaba lokacija za začetek kode, saj si želimo, da bi procesor pri dostopu do naslova nič sprožil napako (glej izpis 2.2). Zato bomo celotno prvo stran, skupaj z glavo, preslikali raje v drugo stran naslovnega prostora na naslov 1000_{16}.

Izpis 2.2: Ker je spremenljivka *foo* definirana v segmentu bss, bo njena začetna vrednost enaka 0. Želeli bi, da bi pri izvajanju programa prišlo do napake (Segmentation fault), saj dostopamo do naslova 0.

```
1 int bar(void) {
2         static int *foo;
3         return *foo;
4 }
```

2.3.2 Povezljiv format objektne datoteke a.out

V prejšnjem razdelku smo opisali zgradbo izvršljive objektne datoteke a.out. Sedaj je čas, da si ogledamo povezljivo različico. Format izvršljivih objektnih datotek mora biti čimbolj preprost, saj je namenjen izvajanju na strojni opremi in zato praviloma ne vsebuje tabele simbolov in premestitvenih vnosov. Na drugi strani to ne velja za povezljive datoteke, ki jih obdelamo v programski opremi in lahko zato izvajamo bolj sofisticirane transformacije.

Tradicionalno se na UNIX sistemih za izvršljive in povezljive objektne datoteke uporablja isti format, le da izvršljive objektne datoteke izpustijo nekatere dele, ki za nalaganje niso potrebni. Tako je zgradba povezljive objektne datoteke a.out enaka tisti, ki smo jo že opisali v prejšnjem razdelku, le da segmentu text in data sledita tabeli premestitvenih vnosov - ena za segment text, druga za segment data. Na koncu sta prisotni še tabela simbolov in tabela nizov.

Izpis 2.3: Zgradba premestitvenega vnosa formata a.out.

```
struct relocation_info {
        int             r_address;
        unsigned int    r_symbolnum : 24,
                        r_pcrel : 1,
                        r_length : 2,
                        r_extern : 1,
                        not_used : 4;
};
```

Premestitveni vnosi

Zgradba premestitvenega vnosa objektne datoteke a.out je prikazana v izpisu 2.3. Sedaj si poglejmo, kaj točno pomenijo polja, definirana v podatkovni strukturi `relocation_info`:

r_address V tem polju je shranjen naslov na lokacijo, ki jo je potrebno popraviti. Vrednost polja pomeni odmik v bajtih od začetka segmenta.

r_symbolnum Na tem mestu je zapisan indeks simbola v tabeli simbolov, na katerega se premestitev nanaša.

r_pcrel Če je bit postavljen, lahko predpostavimo, da popravljamo naslov v ukazu, ki uporablja pc-relativno naslavljanje.

r_length Polje vsebuje dvojiški logaritem dolžine naslova v bajtih.

r_extern Bit je postavljen, če je simbol, na katerega kaže polje `r_symbolnum`, definiran v drugi izvorni datoteki.

Zadnji štirje biti strukture `relocation_info` se na različnih arhitekturah in operacijskih sistemih uporabljajo za različen naloge. V operacijskem sistemu OpenBSD se na primer uporabljajo za implementacijo dinamičnih knjižnic.

Tabela simbolov in nizov

Na koncu datoteke a.out je shranjena tabela simbolov. Posamezen vnos je velik 12 bajtov in je prikazan v izpisu 2.4, sestavljen je iz naslednjih polj:

Izpis 2.4: Format simbola datoteke a.out.

```
struct nlist {
        long            n_strx;
        unsigned char   n_type;
        char            not_used;
        short           n_desc;
        unsigned long   n_value;
};
```

n_strx Ker so imena lahko poljubno dolga, jih ne moremo hraniti neposredno v tabeli simbolov, ampak so spravljena v tabeli nizov. Polje `r_strx` vsebuje indeks v tabeli nizov, kjer je shranjeno ime simbola.

n_type V tem polju je spravljena informacija o tipu simbola. Datoteka a.out pozna več tipov simbolov:

- *text*, *data* in *bss*: Simbol je lokalen in se nahaja v segmentu text, data ali bss.
- *abs*: Absoluten simbol, katerega naslov se med premestitvijo ne spremeni.
- *undefined*: Simbol je definiran v drugem modulu. Vrednost polja `n_value` je ponavadi 0.

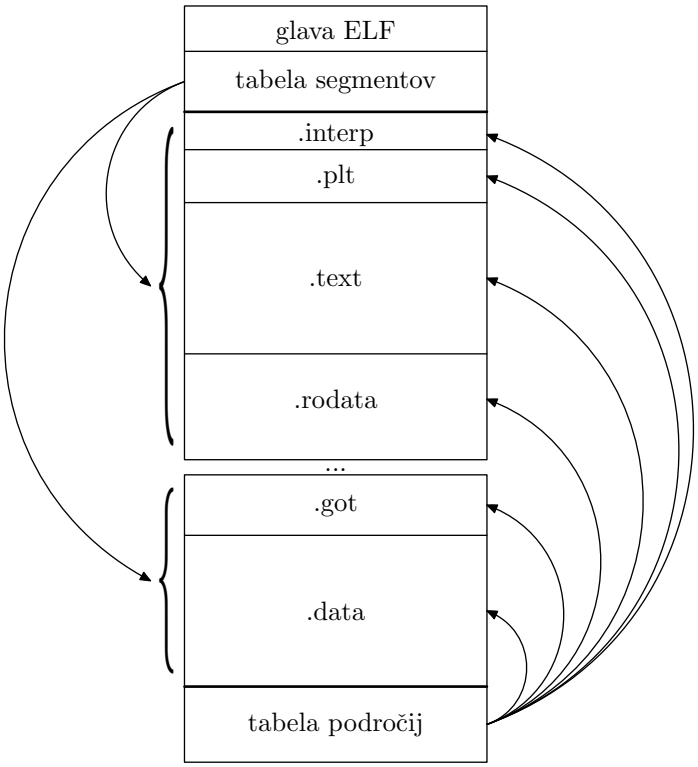
n_desc Polje se uporablja za razhroščevanje. Povezovalnik vrednosti polja ne spreminja. Različni razhroščevalniki uporabljajo to polje za različne namene.

n_value Če je simbolov tipa *text*, *data* in *bss* polje vsebuje naslov.

2.4 Objektna datoteka ELF

Format a.out je preprost in učinkovit format objektnih datotek. Vseeno se na današnjih sistemih skoraj ne pojavlja več. Najbrž je glavni krivec dejstvo, da ga je težko prilagoditi potrebam dinamičnega povezovanja.

ELF je trenutno eden izmed najbolj razširjenih formatov objektnih datotek in se uporablja na večini UNIX sistemov, med drugim tudi na operacijskih



Slika 2.1: Zgradba datoteke ELF.

sistemih Linux in BSD[2]. Format ELF je formalno definiran v specifikaciji [8], podroben opis zgradbe pa se nahaja tudi v [11].

Zgradba datoteke ELF je zanimiva in je prikazana na sliki 2.1. Datoteko ELF lahko interpretiramo na dva načina. Zbirniki in povezovalniki vidijo datoteko ELF kot množico logično povezanih področij, ki jih opisuje tabela področij. Na drugi strani nalagalniki interpretirajo datoteko ELF kot množico segmentov, ki so opisani v tabeli segmentov. En segment je ponavadi sestavljen iz več področij. Tako je na primer segment, kjer je dovoljeno branje in pisanje sestavljen iz področij .got, .data, .bss,... segment, kjer je dovoljeno samo branje, pa iz področij .interp, .plt, .text, .rodata,...

Datoteka ELF se začne z glavo, ki se lahko dekodira na računalnikih, ki uporabljajo pravilo debelega konca, kot tudi tistih, ki uporabljajo pravilo tankega konca. V prvih štirih bajtih je shranjena identifikacijska številka, ki označuje,

[2]ELF se uporablja celo na igralnih konzolah PlayStation 2, PlayStation 3, Wii in GP2X ter tudi na nekaterih mobilnih telefonih.

da gre za datoteko v formatu ELF ("\177ELF"). Naslednji trije bajti opisujejo format in preostanek glave. Samo zgradbo glave ELF si na tem mestu ne bomo podrobno ogledali. Njena definicija je na operacijskem sistemu Linux shranjena v datoteki `/usr/include/elf.h`. Glavo objektne datoteke ELF lahko preučujemo z ukazom `readelf -h <datoteka>`.

2.4.1 Povezljiv format objektne datoteke ELF

Izpis 2.5: Tabela področij.

```
typedef struct {
        uint32_t   sh_name;   // indeks niza v tabeli simbolov
        uint32_t   sh_type;
        uint32_t   sh_flags;
        Elf32_Addr sh_addr;   // zacetni naslov
        Elf32_Off  sh_offset; // odmik v objektni datoteki do zacetka podrocja
        uint32_t   sh_size;   // velikost podrocja v bajtih
        uint32_t   sh_link;   // zaporedna stevilka podrocja
        uint32_t   sh_info;   // vec podatkov o podrocju
        uint32_t   sh_addralign; // poravnaj zacetek podrocja
        uint32_t   sh_entsize; // stevilo vnosov, ce je podrocje tabela
} Elf32_Shdr;
```

Povezljiva objektna datoteka je sestavljena iz množice segmentov, ki jih opisuje tabela segmentov. Definicija tabele segmentov je prikazana v izpisu 2.5. Vsako področje vsebuje določen tip podatkov, kot na primer strojno kodo, podatke, premestitvene vnose, tabelo simbolov, ... Natančneje, format ELF definira naslednje tipe področij:

PROGBITS Področje vsebuje strojno kodo, podatke ali informacije za razhroščevalnik.

NOBITS Podobno kot tip PROGBITS, le da področja tega tipa v objektni datoteki ne zasedajo nič prostora. Uporabljajo se predvsem za implementacijo področja .bss.

SYMTAB, DYNSYM Področje vsebuje tabelo simbolov. Tabela SYMTAB definira simbole, ki se uporabljajo v procesu povezovanja. Tabela DYNSYM pa vsebuje simbole, ki jih uporablja dinamični nalagalnik. Ta tabela se skupaj s programom preslika v naslovni prostor procesa, zato bi si želeli, da je njena velikost čim manjša.

STRTAB Podobno kot pri formatu a.out imamo tudi v datotekah ELF spravljene tabele nizov, s to razliko, da je v datoteki ponavadi prisotna več kot samo ena tabela.

REL, RELA Področje vsebuje tabelo premestitvenih vnosov. Vsaka arhitektura definira svoje tipe premestitvenih vnosov.

DYNAMIC, HASH V tem področju so shranjeni podatki, ki jih potrebuje dinamični nalagalnik.

Polje `sh_flags` lahko postavi naslednje tri zastavice:

ALLOC Za področja, ki imajo postavljeno zastavico ALLOC, moramo pri nalaganju rezervirati nekaj prostora v pomnilniku.

WRITE Ko je področje naloženo, lahko vanj tudi pišemo.

EXECINSTR Področje vsebuje strojno kodo, ki jo lahko izvajamo.

Tipična povezljiva datoteka formata ELF vsebuje kar nekaj področij. Povezovalnik pri svojem delu prebere področja, ki jih pozna. Področja, ki jih ne pozna ali ne potrebuje, lahko preskoči. Nekaj najbolj pogostih področij so:

.text (`sh_type = PROGBITS; sh_flags = ALLOC | EXECINSTR`):
Področje vsebuje strojne ukaze in je ekvivalent segmenta .text v formatu a.out.

.data (`sh_type = PROGBITS; sh_flags = ALLOC | WRITE`):
Podobno kot v segment .data v formatu a.out.

.rodata (`sh_type = PROGBITS; sh_flags = ALLOC`):
Podatki, ki se med izvajanjem programa ne spreminjajo.

.bss (`sh_type = NOBITS; sh_flags = ALLOC | WRITE`):
Področje .bss je tipa NOBITS, ker v objektni datoteki ne zaseda prostora in je ekvivalent segmentu .bss formata a.out.

.rel.text, .rel.data, .rel.rodata (`sh_type = (REL ali RELA)`):
V teh področjih so shranjeni premestitveni vnosi za področja .text, .data in .rodata.

.symtab (`sh_type = SYMTAB`):
Tabela simbolov, ki se uporablja pri povezovanju.

.dyntab (`sh_type = DYNTAB; sh_flags = ALLOC`):
Tabela simbolov, ki se uporablja pri dinamičnem nalaganju. Ker se pri dinamičnem nalaganju tabela simbolov preslika v naslovni prostor, ima postavljeno zastavico ALLOC.

.strtab (`sh_type = STRTAB`):
Tabela nizov, ki predstavljajo imena simbolov področij.

.dynstr (`sh_type = STRTAB; sh_flags = ALLOC`):
Tabela nizov, ki predstavljajo imena simbolov segmentov. Podobno kot področje .dyntab, si pri nalaganju področje .dynstr preslika v naslovni prostor procesa.

.interp (`sh_type = PROGBITS; sh_flags = ALLOC`):
Področje vsebuje ime programa, ki naj se uporabi kot interpreter. Ponavadi področje vsebuje ime dinamičnega nalagalnika (na operacijskem sistemu Linux je to `/lib/ld-linux.so.X`)

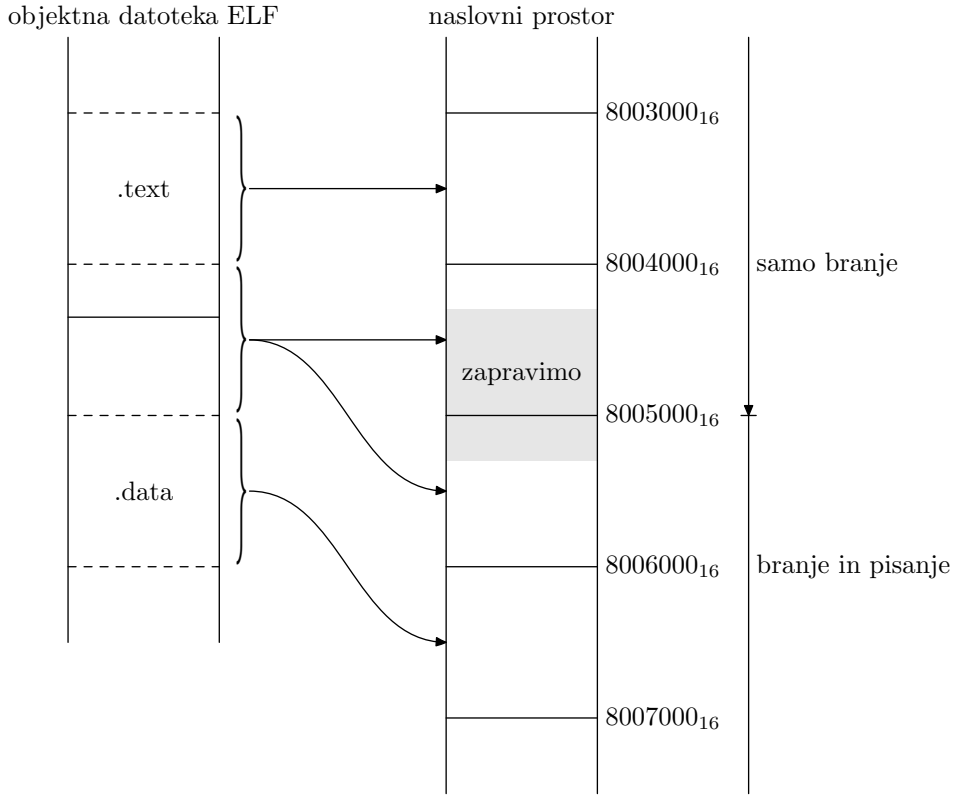
Opise področji lahko, iz datoteke v formatu ELF, preberemo z ukazom `readelf -S <datoteka>`

Tabela simbolov datoteke ELF se bistveno ne razlikuje od tiste, ki jo uporablja format a.out, zato je na tem mestu ne bomo podrobno opisali. Raje si poglejmo, kako izgleda izvršljiva datoteka ELF.

2.4.2 Izvršljiv format datoteke ELF

Izvršljiva datoteka ELF ima podobno zgradbo kot povezljiva, le da so podatki shranjeni tako, da jih je hitro mogoče preslikati v naslovni prostor procesa. Datoteka vsebuje tabelo segmentov, ki vsebuje navodila kam preslikati določene segmente objektne datoteke. Izvršljiva datoteka vsebuje le malo segmentov. Skoraj vedno pa je prisoten segment, ki ga lahko med izvajanjem programa spreminjamo ter segment, ki ga lahko samo beremo. Format ELF privarčuje nekaj prostora na trdem disku v zameno za nekaj prostora v naslovnem prostoru. Tako kot pri shemi QMAGIC formata a.out, je glava ELF datoteke združena s področjem .text in se skupaj z njim preslika v naslovni prostor. Podobno se lahko znebimo omejitve o poravnanosti naslovov z velikostjo strani. V

datoteki ELF se lahko segment konča in začne na poljubnem odmiku. V splošnem pa moramo zaradi tega prvo in zadnjo stran segmenta preslikati dvakrat. Primer prikazuje slika 2.2.



Slika 2.2: Recimo, da področju .text, ki se zaključi na naslovu $80045FF_{16}$, sledi področje .data (prikazano na levi). Zadnja stran segmenta, ki vsebuje področje .text, se bo preslikala dvakrat. Prvič na naslove 8004000_{16} - 8005000_{16} (stran je označena samo za branje), drugič pa na naslove 8005000_{16} - 8006000_{16} (stran je označena za branje in pisanje). Področje .data se torej začne na navideznem naslovu 8005600_{16}. Na sliki je označen tudi del naslovnega prostora, ki ga pri tem zapravimo.

2.5 Objektna datoteka procesorja HIP

Na tem mestu bomo definirali preprost format objektne datoteke, ki ga uporabljajo programi verige hiputils. Za razliko od obeh formatov, ki smo jih

spoznali v tem poglavju (najbrž tudi vseh formatov, ki se v praksi uporabljajo), bo naš format zapisan v ASCII datoteki. To storimo zato, da nam ni potrebno pisati posebnih orodij za branje objektnih datotek, kot na primer program `readelf`.

```
LINK
<št. knjižnic> <št. segmentov> <št. simbolov> <št. premestitev>

<dinamične knjižnice, ki naj se pred izvajanjem naložijo>
<tabela segmentov>
<tabela simbolov>
<tabela premestitvenih vnosov>

<vsebina segmentov>
```

Slika 2.3: Osnovna zgradba objektne datoteke.

Osnovna zgradba objektne datoteke je prikazana na sliki 2.3. Leksemi so med seboj ločen z enim ali več znaki za presledek, tabulator ali novo vrstico (whitespace). Prazne vrstice in vrstice, ki se začnejo z znakom '#', se ignorirajo. Datoteka se začne z identifikacijsko številko - nizom "LINK". V naslednji vrstici so navedena štiri števila, ki predstavljajo glavo formata in določajo velikosti štirih segmentov, ki sledijo.

Glavi sledi segment, v katerem navedemo imena vseh dinamičnih knjižnic, ki se morajo preslikati v naslovni prostor, preden program zaženemo. Imena dinamičnih knjižnic navedemo vsakega v svoji vrstici.

V naslednjem delu so shranjene definicije segmentov. Vsak vnos v tabeli ima obliko:

```
<ime segmenta> <začetni naslov> <velikost segmenta> <zastavice>
```

Zastavice lahko vsebujejo znake 'R' (readable), 'W' (writable) in 'P' (present). Segmenti so oštevilčeni v vrstnem redu, kot so v datoteki navedeni. Prvi segment je oštevilčen s številom 1.

Tabeli segmentov sledi tabela simbolov. Posamezen vnos izgleda tako:

```
<ime simbola> <vrednost> <številka segmenta> <tip simbola>
```

Številka segmenta je indeks segmenta v tabeli segmentov, v katerem je simbol definiran. Tip simbola je lahko:

'L', če je simbol definiran v tej datoteki.

'D', če je simbol definiran v tej datoteki in izvožen s psevdo ukazom `.globl`.

'U', če je simbol definiran drugje.

'S', če gre za poseben simbol, ki kaže na začetek segmenta.

Tabeli simbolov sledijo premestitveni vnosi:

```
<lokacija> <segment> <referenca> <tip> [<sumand>]
```

Polje `lokacija` kaže na mesto, ki ga je potrebno popraviti. Naslov se interpretira relativno, glede na začetek segmenta, ki je naveden v drugem polju. `referenca` je številka simbola, na katerega se premestitev nanaša. `tip` nam pove, za kakšno vrsto premestitvenega vnosa gre. Polje `sumand` ni obvezno in se uporablja samo skupaj z določenimi tipi premestitvenih vnosov.

Na koncu je zapisana še vsebina vseh segmentov, ki imajo v polju `zastavice` prisoten znak 'P'. Vsebina je zakodirana kot dolg niz znakov, v katerem znaka $2i$ in $2i + 1$ skupaj interpretiramo kot število v šestnajstiškem zapisu in predstavlja vrednost i-tega bajta.

Primer preproste objektne datoteke je predstavljen v izpisu 2.6.

Izpis 2.6: Objektna datoteka za procesor HIP

```
1  LINK
2  1 3 6 1
3
4  # needs
5  m.so
6
7  # segments
8  .text            08048000   10   RP
9  .bss             08049004    8   RW
10
11 # symbols
12 .bss             08049004    3    S
13 .text            08048000    1    S
14 _start           08048000    1    D
15 foo              08048000    1    U
16
17 # relocations
18 0       2       4       A4
19
20 # data
21 0B3C10009B990000B73F000078000000
22 08048000
```

Poglavje 3

Povezovanje

3.1 Razporeditev segmentov

Prva naloga povezovalnika je razporeditev segmentov. Tipičen vhod povezovalnika je množica objektnih datotek, ki so sestavljene iz enega ali več segmentov. Povezovalnik mora združiti segmente istega tipa in jih v izhodno datoteko zapisati kot en segment. Kot bomo videli v nadaljevanju, mora povezovalnik včasih tudi ustvariti nove segmente, ki jih združi z tistimi iz vhodnih modulov in zapiše v izhodno datoteko. Primer takšnega segmenta je .got, kjer je shranjena globalna tabela odmikov. V procesu povezovanja se ponavadi pridruži segmentoma .data in .bss ter ostalim, ki jih lahko med izvajanjem programa spreminjamo. Primer segmenta, ki ga ustvari povezovalnik je tudi segment .plt, ki se med izvajanjem ne spreminja, zato ga moramo v izhodni datoteki zapisati med segmente, ki nimajo postavljene zastavice za branje. Več o segmentih .got in .plt bomo spoznali, ko se bomo pogovarjali o dinamičnih knjižnicah in pozicijsko neodvisni kodi.

Ker so naslovi simbolov podani z odmikom od začetka segmenta, morajo biti dejanski naslovi vseh segmentov znani še preden se lahko lotimo razreševanja simbolov in premestitve.

3.1.1 Preprost model

V preprostem modelu bomo vzeli, da je vhod povezovalnika množica modulov $M := \{M_1, M_2, \ldots M_n\}$. Vsak modul M_i se začne na naslovu 0 in ima en segment dolžine L_i bajtov. Povezovalnik po vrsti obišče vse module. Modulu

M_i določi začetni naslov $S_i := \sum_{j=1}^{i-1} L_j$, dolžina izhodne datoteka je seveda enaka $L = \sum_{j=1}^{n} L_j$.

Ker večina arhitektur (med njimi tudi procesor HIP) zahteva, da so naslovi operandov poravnani, povezovalniki ponavadi zaokrožijo velikost segmentov L_i, da so deljivi s 4 oziroma 8 (odvisno od arhitekture procesorja). Definirajmo funkcijo align_n:

$$\text{align}_n(x) := \min\{i : i \in \mathbb{N} \wedge i \geq x \wedge n|i\}$$

Enačbi za S_i in L lahko spremenimo, tako da bodo začetni naslovi poravnani.

$$\begin{aligned} S_i &= \sum_{j=1}^{i-1} \text{align}_n(L_j) \\ L &= \left[\sum_{j=1}^{n-1} \text{align}_n(L_j)\right] + L_n \end{aligned} \tag{3.1}$$

Ker dolžine zadnjega segmenta ne poravnamo, je morda enačba (3.1) bolj zapletena, kot smo sprva pričakovali. Funkcijo align_n se da implementirati elegantno in učinkovito:

```
def align(x, n):
    return (x - 1 + n) & -n
```

Konkreten primer preprostega modela je prikazan v tabeli 3.1.

3.1.2 Realen model

V praksi skoraj vsi objektni formati definirajo več kot le en segment kot smo predpostavili v preprostem modelu. Naloga povezovalnika je, da združi vse segmente istega tipa v en sam segment. Povezovalnik mora opraviti dva prehoda čez vhodne objektne datoteke, saj je začetni naslov segmenta nemogoče določiti, dokler ne poznamo velikosti vseh segmentov, ki v izhodni datoteki nastopajo pred njim. Naj bo vhod povezovalnika množica modulov $M := \{M_1, M_2, \ldots M_n\}$. Vsak modul M_i se začne na naslovu 0 in ima natanko m segmentov. Velikosti segmentov so shranjeni v matriki $L \in \mathbb{N}^{n \times m}$. Element matrike $L_{i,j}$ predstavlja velikost segmenta j v modulu M_i. Podobno so začetni

ime	velikost segmenta	lokacija
foo.o	253_{16}	1000_{16} - 1252_{16}
bar.o	$12C_{16}$	1254_{16} - $137F_{16}$
baz.o	$FA5_{16}$	1380_{16} - 2324_{16}
qux.o	1390_{16}	2328_{16} - $36B7_{16}$
quux.o	121_{16}	$36B8_{16}$ - $37D8_{16}$

Tabela 3.1: Primer povezovanja petih objektnih datotek v preprostem modelu. Začetni naslov naj bo 1000_{16}, velikosti segmentov pa naj bodo poravnane s 4. Zaradi zahteve o poravnanosti izgubimo štiri bajte (enega pri datoteki foo.o, ostale tri pa pri datoteki baz.o). Čeprav velikost datoteke quux.o ni deljiva s štiri, se velikost izhodne datoteke ne poveča, ker je objektna datoteka quux.o zadnja po vrsti. (glej enačbo (3.1)).

naslovi segmentov shranjeni v matriki $S \in \mathbb{N}^{n \times m}$. Element matrike $S_{i,j}$ predstavlja začetni naslov segmenta j v modulu M_i. Elemente matrike S lahko izračunamo z naslednjo enačbo:

$$S_{i,j} = \sum_{l=1}^{j-1} \sum_{k=1}^{n} L_{k,l} + \sum_{k=1}^{i-1} L_{k,j}$$

Podobno kot v preprostem modelu lahko tudi tukaj vpeljemo zahtevo o poravnanosti naslovov. Nekateri formati (recimo a.out) zahtevajo tudi, da so velikosti segmentov izhodne datoteke deljive z velikostjo strani navideznega pomnilnika. Če upoštevamo obe zahtevi o poravnanosti, dobimo novo enačbo za matriko S:

$$S_{i,j} = \sum_{l=1}^{j-1} \text{align}_{\text{page}} \left(\sum_{k=1}^{n} \text{align}_{\text{word}}(L_{k,l}) \right) + \sum_{k=1}^{i-1} \text{align}_{\text{word}}(L_{k,j}) \tag{3.2}$$

Povezovalnik verige hiputils računa naslove segmentov tako, kot je prikazano v enačbi (3.2).

Primer realnega modela razporeditve segmentov je prikazan v tabeli 3.3, ki je rezultat razporeditve segmentov petih objektnih datotek, predstavljenih v tabeli 3.2.

ime	.text	.data	.bss
foo.o	253_{16}	320_{16}	$5E_{16}$
bar.o	$12C_{16}$	$1A7_{16}$	150_{16}
baz.o	$FA5_{16}$	$3F0_{16}$	$8D0_{16}$
qux.o	1390_{16}	$1B13_{16}$	1400_{16}
quux.o	121_{16}	255_{16}	ACD_{16}

Tabela 3.2: Opis petih objektnih datotek.

ime	.text	.data	.bss
foo.o	1000_{16} - 1252_{16}	4000_{16} - $431F_{16}$	6624_{16} - 6681_{16}
bar.o	1254_{16} - $137F_{16}$	4320_{16} - $44C6_{16}$	6684_{16} - $67D3_{16}$
baz.o	1380_{16} - 2324_{16}	$44C8_{16}$ - $48B7_{16}$	$67D4_{16}$ - $70A3_{16}$
qux.o	2328_{16} - $36B7_{16}$	$48B8_{16}$ - $63CA_{16}$	$70A4_{16}$ - $84A3_{16}$
quux.o	$36B8_{16}$ - $37D8_{16}$	$63CC_{16}$ - 6620_{16}	$84A4_{16}$ - $8F70_{16}$

Tabela 3.3: Tabela prikazuje razporeditev segmentov petih datotek, ki so opisani v tabeli 3.2. Začetni naslov je 1000_{16}, operandi so poravnani s 4, segmenti pa s 4K. Ker .data in .bss spadata v isti segment, ni potrebno, da se .bss začne na naslovu, ki je deljiv z 4K.

3.2 Tabela simbolov

Razreševanje simbolov je ena najpomembnejših nalog povezovalnika in nalagalnika. Če se iz enega modula ne bi dalo dostopati do podatkov ali procedur definiranih drugje, bi bili povezovalniki danes veliko manj koristni.

Vsak modul definira svojo tabelo simbolov, ki jo ustvari zbirnik. Tabela vključuje naslednje tipe simbolov:

- Globalni simboli, ki so definirani v tem modulu in izvoženi s psevdo ukazom zbirnika `.globl`.
- Globalni simboli, do katerih v tem modulu samo dostopamo, definicija pa je (upamo) prisotna drugje. Takim simbolom pravimo tudi zunanji ali nedefinirani.
- Lokalni simboli, definirani v tem modulu, vendar nedostopni izven njega. Če se v različnih modulih pojavi lokalni simbol z istim imenom, do konflikta ne pride. Uporabljajo se tudi za razhroščevanje.

- Imena segmentov, katerih vrednost kaže na začetek ustreznega segmenta. Z njihovo pomočjo lahko poenotimo postopek premestitve lokalnih in globalnih simbolov.
- Zaporedne številke vrstic, ki jih razhroščevalniki uporabljajo za preslikovanje med ukazi strojne kode in številko vrstice v izvorni datoteki.

Povezovalnik prebere tabele simbolov vseh modulov, ki se pojavijo na vhodu. Včasih si shrani vse simbole, bolj pogosto le tiste, ki so pri povezovanju dejansko potrebni. Vse te tabele povezovalnik združi v eno samo veliko tabelo, ki jo imenujemo globalna tabela simbolov. Globalna tabela simbolov mora biti zgrajena še preden se lotimo procesa premestitve. Povezovalnik na koncu shrani vse ali samo nekatere simbole v izhodno objektno datoteko, odvisno od formata izhoda.

Nekateri formati lahko v eni objektni datoteki vsebujejo več tabel simbolov. Format ELF hrani ločeni tabeli simbolov, ki sta zapisani v področjih .dynsym in .symtab. Tabela .dynsym je manjša od obeh in se uporablja pri dinamičnem nalaganju, medtem ko je tabela .symtab običajna tabela simbolov, ki jo uporablja povezovalnik.

3.2.1 Zgradba tabele simbolov

Tabele simbolov so podobne tistim, ki jih najdemo v prevajalnikih, le da so bolj preproste. Največkrat so implementirane s podatkovno strukturo odprte zgoščene tabele. To pomeni, da so simboli shranjeni v tabeli, do katere dostopamo preko zgoščevalne funkcije. Pri odprti zgoščeni tabeli v polju hranimo kazalec na seznam elementov, ki sovpadajo.

Naj bo s simbol, shranjen v tabeli simbolov T, velikosti l, do katerega želimo dostopati. S $h(s)$ označimo zgoščeno vrednost simbola s. Simbol s se tedaj nahaja v seznamu, katerega začetni naslov je shranjen v polju $ll := T[h(s) \bmod l]$. Elemente seznama ll sedaj enega za drugim primerjamo s simbolom s, dokler ne najdemo iskanega simbola s.

Simbole med seboj primerjamo tako, da primerjamo njihova imena - znak za znakom. Naj bo l_1 dolžina imena simbola s_1 in l_2 dolžina imena simbola s_2, potem za primerjanje simbolov s_1 in s_2 potrebujemo $\mathcal{O}(\min(l_1, l_2))$ časa. V preteklosti so povezovalniki omejevali največje dovoljeno število znakov v imenu simbola. Ponavadi je bila ta meja nastavljena na približno osem znakov. Danes morajo povezovalniki podpirati imena poljubne dolžine - posledično

lahko primerjanje dveh simbolov traja poljubno dolgo. To na prvi pogled ne predstavlja večjega problema. Nenazadnje se najbrž ne spomnimo, kdaj smo nazadnje v kakšnem programu zasledili ime spremenljivke, ki bi bilo večje od na primer 50 znakov. Če pa se je to že zgodilo, se najbrž spomnimo tudi, da smo bili v tistem trenutku tako jezni na avtorja programa, da smo si tako ali tako želeli, da bi se njegov program povezoval čim počasneje[1].

V resnici vsi C++ programerji uporabljajo dolga imena, čeprav se tega morda ne zavedajo. Prevajalniki jezika C++ ponavadi vsebujejo kodirno shemo, ki imena metod, skupaj s podatkovnim tipom argumentov in podatki o imenskem prostoru, preslikajo v niz znakov. Pri tem se imena simbolov precej napihnejo. Poglejmo si primer: funkcija `f` z dvema argumentoma, ki je definirana z: `f(Pair, First::Second::Third)` se po preslikavi napihne v niz `f__F4PairQ35First6Second5Third`. To je bil preprost primer. V "zglednem" C++ projektu, z veliko imenskimi prostori, template-i in ugnezdenimi razredi lahko napihnjena imena presežejo 1000 znakov! Ta problem je še posebej pereč pri dinamičnem nalaganju, saj se čas, ki je potreben za dinamično nalaganje, neposredno prišteje času za zagon programa.

Problem v praksi rešujemo tako, da v tabeli simbolov hranimo poleg simbola še njegovo zgoščeno vrednost, torej hranimo par $(s, h(s))$. Simbola s_1 in s_2 lahko sedaj primerjamo tako, da najprej preverimo ali se ujemata zgoščeni vrednosti. To lahko storimo v času $\mathcal{O}(1)$. V primeru, da $h(s_1) \neq h(s_2)$ lahko nemudoma odgovorimo z ne, sicer moramo, tako kot prej, opraviti drago primerjanje nizov imen.

Namesto ene zgoščevalne funkcije lahko uporabimo več zgoščevalnih funkcij $h_1, h_2, \ldots, h_n$. Če za vsak simbol v tabeli simbolov hranimo $(n+1)$-terico $(s, h_1(s), h_2(s), \ldots, h_n(s))$, lahko posplošimo postopek, ki smo ga opisali v prejšnjem odstavku. Imena simbolov s_1 in s_2 je potrebno primerjati natanko tedaj, ko velja $\forall i \in \{1, 2, \ldots, n\} : h_i(s_1) = h_i(s_2)$, sicer je odgovor ne.

3.2.2 Globalna tabela simbolov

Povezovalnik si pri svojem delu gradi globalno tabelo simbolov, ki vsebuje globalne simbole vseh vhodnih modulov.

Vsakič ko povezovalnik iz vhoda prebere nov modul, iz objektne datoteke izlušči vse globalne simbole, ki so v modulu definirani ali uporabljeni in jih zapiše

[1] Vrag naj vzame razvijalce GTK API-ja! Naj rečejo kar hočejo, ampak ime procedure `gtk_dialog_set_alternative_button_order_from_array` je preprosto predolgo.

v globalno tabelo simbolov. Za vsak globalni simbol si zapomni še na katerem mestu je definiran in kje je uporabljen. V primeru, da globalna tabela simbolov že vsebuje simbol z istim imenom, je to usodna napaka in proces povezovanja se zaključi neuspešno. Ko povezovalnik obdela vse vhodne module, se sprehodi čez globalno tabelo simbolov in preveri če so vsi globalni simboli, ki so uporabljeni, tudi definirani. Če v tabeli obstaja globalni simbol brez definicije, je podobno kot prej, za proces povezovanja to usodna napaka.

Pozorni moramo biti še na eno malenkost. Na posamezen simbol se v premestitvenih vnosih sklicujemo z njegovim indeksom v lokalni tabeli simbolov. V splošnem ne velja, da sta ista simbola v različnih modulih označena z isto številko. Problema se lahko znebimo tako, da za vsak modul hranimo tabelo kazalcev na simbole v globalni tabeli simbolov.

3.3 Premestitev

Sedaj je na vrsti gotovo najbolj pomemben del procesa povezovanja - premestitev. Z besedo *premestitev* označujemo dve različni opravili:

1. Segmenti se v splošnem ne začnejo na naslovu 0, kot to predvidevajo zbirniki, zato je potrebno določene naslove v objektni datoteki popraviti, da bodo odsevali dejanske naslove med izvajanjem.

2. Zbirniki dovoljujejo, da se v modulu pojavijo simboli, ki so definirani drugje. Pri procesu povezovanja je potrebno vse reference na nedefinirane simbole razrešiti in v izhodno objektno datoteko zapisati ustrezne spremembe.

Povezovalniki morajo poznati tako absolutne kot relativne naslove, saj so na večini procesorskih arhitektur naslovi v podatkih absolutni, naslovi v ukazih pa so lahko tako absolutni kot relativni.

3.3.1 Premestitev med nalaganjem

Na veliko sistemih se premestitev izvaja tako med povezovanjem kot tudi med nalaganjem.

Izhod povezovalnika je izvršljiva datoteka s konstantnim začetnim naslovom. Če se slučajno zgodi, da je pri nalaganju ta naslov že zaseden, mora nalagalnik

v programu popraviti vse absolutne naslove, da se bo program pravilno izvajal na novi lokaciji. Na operacijskem sistemu DOS so vsi programi povezani tako, da se začnejo na naslovu 0. Dejanski naslov določi nalagalnik, ki potem tudi opravi potrebne premestitve. Na ostalih sistemih povezovalnik ponavadi izbere fiksen začetni naslov, ki je ponavadi prost. V primeru, da je izbran naslov že zaseden, je potrebno poiskati novega, nalagalnik pa mora opraviti ustrezne premestitve. Nekako tako delujeta sistema Windows in UNIX. Naslov, ki ga izbere povezovalnik, je skoraj vedno na voljo, razen v primeru dinamičnih knjižnic, ko so potrebne dodatne premestitev med nalaganjem.

Premestitve med nalaganjem so bolj preproste kot premestitve med povezovanjem. Vse kar moramo storiti je, da vsakemu absolutnemu naslovu prištejemo razliko med dejanskim začetnim naslovom in naslovom začetka, ki ga je določil povezovalnik.

3.3.2 Premestitev med povezovanjem

Povezovalnik se lahko loti premestitve, ko razpredi vse segmente in zgradi globalno tabelo simbolov. Odvisno od lokacije simbola, ki ga naslavljamo, je potrebno storiti eno izmed naslednjih opravil (v nadaljevanju predpostavljamo, da je povezovalnik vse segmente postavil na naslov 0).

- Absolutnim naslovom znotraj enega segmenta moramo prišteti dejanski naslov segmenta.

- Popraviti je potrebno tudi vse naslove, ki kažejo na nek drug segment. Absolutnim naslovom je treba prišteti začetni naslov segmenta, v katerega kažejo. Relativnim naslovom je potrebno prišteti razliko med začetnim naslovom segmenta, v katerem so definirani in začetnim naslovom segmenta, na katerega kažejo.

- Razrešiti je potrebno naslove globalnih simbolov. Če je naslov absoluten, je v izhodno datoteko potrebno zapisati le vrednost simbola (vrednost simbola je v tem delu povezovanja enaka dejanskemu naslovu simbola med izvajanjem), sicer je tej vrednost potrebno odšteti še začetni naslov segmenta, v katerem se nahajamo.

3.3.3 Proces premestitve

Povezljive (in naložljive) objektne datoteke vsebujejo premestitvene vnose, ki opisujejo mesta v strojni kodi in podatkih, ki jih je potrebno popraviti. Če se pri procesu premestitve zgodi, da je nov naslov prevelik, je naloga povezovalnika, da prekine povezovanje in uporabnika opozori na napako. Ker je proces premestitve povezan z arhitekturo procesorja, si ga bomo sedaj ogledali na dveh konkretnih primerih.

x86

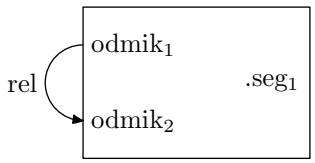
Čeprav je kodiranje ukazov na procesorjih x86 zelo zapleteno, je s stališča povezovalnika arhitektura x86 presenetljivo preprosta. Posamezen ukaz lahko vsebuje kvečjemu absolutni ali relativni naslov. Ukazi, ki naslavljajo podatke vsebujejo 32-bitne naslove, ki jim je preprosto potrebno prišteti naslov začetka segmenta, v katerega kažejo. Ukaz za klic procedure in ukazi za skoke uporabljajo pc-relativno naslavljanje. Za skoke in klice znotraj istega segmenta premestitev ni potrebna, če kličemo ali skačemo na lokacijo v drugem segmentu pa mora povezovalnik naslovu prišteti razliko med začetnim naslovom segmenta, v katerega skačemo in začetnim naslovom segmenta, kjer se trenutno nahajamo.

HIP

Proces premestitve pri procesorju HIP je bolj zapleten kot tisti pri arhitekturi x86. Kot je to pri procesorjih tipa RISC v navadi, je 32-bitni naslov nemogoče spraviti v en strojni ukaz, zato pri procesorju HIP uporabljamo zaporedje dveh ukazov: `lhi` in `addi`, ki po delih preneseta vrednost 32-bitnega naslova v register. Povezovalnik moramo prilagoditi, da bo premestitev pravilno opravljal tudi na teh polovičnih naslovih. Vsi ukazi load/store in ukaza `call` in `j` uporabljajo bazno naslavljanje, zato premestitev ni potrebna. Ukaza za pogojni skok `bne` in `beq` uporabljata pc-relativno naslavljanje in jih popravimo podobno kot tiste pri procesorjih x86.

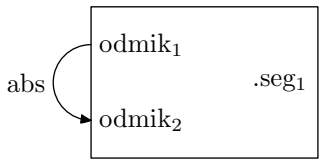
Dejansko se proces premestitve prične že pri prevajanju izvorne datoteke. Zbirnik je namreč tisti, ki mora ugotoviti kateri ukazi vsebujejo naslov, ki ga je potrebno pred izvajanjem popraviti. Zbirnik to informacijo v obliki premestitvenih vnosov posreduje povezovalniku. Kako zbirnik verige hiputils zapiše premestitvene vnose, je podrobno prikazano na sliki 3.1.

Povezovalnik za procesor HIP pozna kar nekaj različnih tipov premestitve-



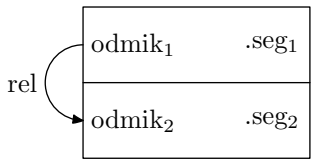
Premestitev ni potrebna.

Relativni naslov simbola definiranega v istem segmentu.



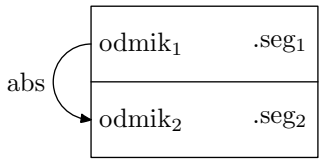
$$\begin{aligned} \text{mem}[\text{odmik}_1] &= \text{odmik}_2 \\ \text{segment} &= .\text{seg}_1 \\ \text{referenca} &= .\text{seg}_1 \end{aligned}$$

Absolutni naslov simbola definiranega v istem segmentu.



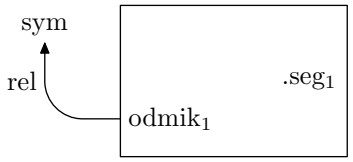
$$\begin{aligned} \text{mem}[\text{odmik}_1] &= \text{odmik}_1 - \text{odmik}_2 \\ \text{segment} &= .\text{seg}_1 \\ \text{referenca} &= .\text{seg}_2 \end{aligned}$$

Relativni naslov simbola definiranega v drugem segmentu.



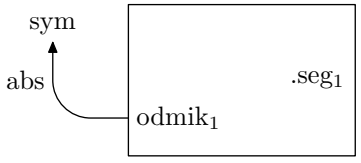
$$\begin{aligned} \text{mem}[\text{odmik}_1] &= \text{odmik}_2 \\ \text{segment} &= .\text{seg}_1 \\ \text{referenca} &= .\text{seg}_2 \end{aligned}$$

Absolutni naslov simbola definiranega v drugem segmentu.



$$\begin{aligned} \text{mem}[\text{odmik}_1] &= \text{odmik}_1 \\ \text{segment} &= .\text{seg}_1 \\ \text{referenca} &= \text{sym} \end{aligned}$$

Relativni naslov simbola definiranega v drugem modulu.



$$\begin{aligned} \text{mem}[\text{odmik}_1] &= 0 \\ \text{segment} &= .\text{seg}_1 \\ \text{referenca} &= \text{sym} \end{aligned}$$

Absolutni naslov simbola definiranega v drugem modulu.

Slika 3.1: Premestitveni vnosi za procesor HIP. V diagramu se "segment" in "referenca" nanašata na ustrezni polji v premestitvenem vnosu.

nih vnosov (Spomnimo se, da je premestitveni vnos sestavljen iz petih polj: `lokacija`, `segment`, `referenca`, `tip` in `sumand`.):

A4: Štirje bajti na naslovu `lokacija` so absolutni naslov simbola `referenca`.

A2: Podobno kot tip A4, le da je velikost naslova dva bajta.

R2: Dva bajta na naslovu `lokacija` sta relativni naslov simbola `referenca`.

U2: Dva bajta na naslovu `lokacija` sta zgornjih 16-bitov naslova simbola `referenca`. V polju `sumand` je shranjen celoten 32-bitni naslov.

L2: Dva bajta na naslovu `lokacija` sta spodnjih 16-bitov naslova simbola `referenca`.

DR2: V bajta na naslovu `lokacija` naj se shrani razlika med naslovom simbola `referenca` in simbolom `_GP`, ki ga definira povezovalnik. Premestitveni vnos se uporablja za implementacijo gp-relativnega naslavljanja.

GA2: Bajta na naslovu `lokacija` vsebujeta razliko med naslovom simbola `referenca` in začetkom globalne tabele odmikov.

GP2: Bajta na naslovu `lokacija` vsebujeta lokacijo simbola `referecna` v globalni tabeli odmikov, relativno glede na simbol `_GOT`.

GR2: V bajta na naslovu `lokacija` naj se shrani naslov simbola `referenca`, relativno glede na simbol `_GOT`.

ER4: V štirih bajtih na naslovu `lokacija` je shranjen absolutni naslov simbola `referenca`. Premestitveni vnos se uporablja pri dinamičnem nalaganju.

Premestitvene vnose GA2, GP2 in GR2 uporabljamo za implementacijo pozicijsko neodvisne kode. Več o pozicijsko neodvisni kodi bomo povedali v poglavju 4.2.1.

Premestitvena vnosa U2 in L2 se v praksi zelo pogosto pojavljata, zato si zaslužita, da ju preučimo malo bolj podrobno. Najprej si poglejmo primer kode v zbirniku HIP, ki ustvari premestitvena vnosa U2 in L2:

```
/* t0 <- naslov spremenljivke foo */ | /* t0 <- vrednost spremenljivke foo */
        lhi     t0,%hi(foo)          |         lhi     t0,%hi(foo)
        addi    t0,t0,%lo(foo)       |         lw      t0,%lo(foo)(t0)
```

Najprej si poglejmo kaj sta in kako delujeta funkciji zbirnika %hi in %lo. Funkcija %lo je preprosta in vrne spodnjih 16 bitov argumenta. Implementira jo funkcija `lo`:

```
def lo(n):
    return n & 0xffff
```

Smiselno je pričakovati, da je funkcija %hi prav tako preprosta in da jo lahko implementiramo z:

```
def hi_naive(n):
    return (n & 0xffff0000) >> 16
```

Zgornja definicija je žal nepravilna. Da je v implementaciji res napaka, se lahko prepričamo, če vzamemo, da je naslov spremenljivke `foo` enak $1234FFFF_{16}$. Potem, po zgornjih definicijah funkcij %hi in %lo, velja $\%\text{hi}(1234FFFF_{16}) = 1234_{16}$ in $\%\text{lo}(1234FFFF_{16}) = FFFF_{16}$. Ampak ker procesor HIP interpretira odmik v ukazih z baznim naslavljanjem kot predznačeno število, bo število $FFFF_{16}$ interpretirano kot -1, naslov, ki se bo naložil pa bo $1233FFFF_{16}$ [2]. Pravilna implementacija funkcije %hi je:

```
def hi(n):
    return ((n & 0xffff0000) >> 16) + (1 if n & 0xffff > 0x7fff else 0)
```

Vrnimo se sedaj nazaj k premestitvenima vnosoma U2 in L2. Naj se spremenljivka `foo` tokrat nahaja na naslovu $12347FFF_{16}$. Velja $\%\text{hi}(12347FFF_{16}) = 1234_{16}$ in $\%\text{lo}(12347FFF_{16}) = 7FFF_{16}$. Sedaj pa si predstavljajmo, da se segment, v katerem je spremenljivka `foo` definirana, premakne za $A000_{16}$. Zgornjih 16 bitov novega naslova je potem 1235_{16}. Ta izračun ne bi bil mogoč, če ne bi poznali celotnega naslova. Ker je lokacija, na katero kaže premestitveni vnos U2 velika le 2 bajta, vanjo ne moremo neposredno shraniti naslova velikega 4 bajte. Zato potrebujemo novo polje v premestitvenem vnosu `sumand`, v katerega shranimo celotni naslov.

[2] Kot posledica se med izvajanjem na ekranu najbrž izpišeta dve besedi, ki sta med programerji tako zelo osovraženi - Segmentation fault.

Poglavje 4

Knjižnice

Knjižnica je množica objektnih datotek, ki jih po potrebi vključimo v glavni program. Skoraj vsi sodobni povezovalniki vsebujejo podporo za knjižnice, med njimi tudi povezovalnik verige hiputils. Knjižnice se po načinu povezovanja in nalaganja med seboj zelo razlikujejo in jih delimo v dve glavni skupini: statične knjižnice in dinamične knjižnice. V tem poglavju si bomo podrobno ogledali obe omenjeni skupini knjižnic, opisali bomo tudi kako so knjižnice v povezovalniku verige hiputils implementirane.

4.1 Statične knjižnice

Statične knjižnice vedno vsebujejo eno ali več objektnih datotek, ponavadi vsebujejo tudi imenik, s katerim pospešimo iskanje in dostop do teh datotek. Proces vključevanja statične knjižnice v glavni program se v celoti izvrši že v času povezovanja. Pri povezovanju se iz statične knjižnice preberejo vse tiste objektne datoteke, ki razrešijo sicer nedefinirane simbole. Te objektne datoteke se nato povežejo skupaj z ostalimi objektnimi datotekami na vhodu v izvršljiv program, natanko tako, kot smo to opisali v poglavju 3. Ker so izvršljivi programi, ki so povezani s statičnimi knjižnicami identični tistim, ki so sestavljeni samo iz objektnih datotek, nalagalniku sploh ni treba poznati koncepta statične knjižnice.

4.1.1 Preiskovanje knjižnic

Ko povezovalnik na vhodu namesto navadne objektne datoteke prebere statično knjižnico, mora v knjižnici poiskati vse tiste objektne datoteke, ki razrešijo kakšen nedefiniran simbol. Če v knjižnici obstaja takšna objektna datoteka, jo povezovalnik iz knjižnice prebere, nato pa jo obravnava kot vsako drugo objektno datoteko. Da bi proces preiskovanja knjižnic pospešili, vsebuje večina sodobnih formatov imenik, ki ni nič drugega kot združena tabela simbolov vseh objektnih datotek, ki knjižnico sestavljajo.

Preiskovanje statične knjižnice je iterativni proces. To vidimo, če vzamemo, da je knjižnica A sestavljena iz objektnih datotek a_1 in a_2 (v tem vrstnem redu). Nadalje naj velja, da objektna datoteka a_2 uporablja enega ali več simbolov, ki so definirani v a_1, glavni program pa uporablja simbole, ki so definirani v objektni datoteki a_2, ne pa tudi v a_1. Ko povezovalnik na vhodu prebere knjižnico A, po vrsti pregleda vse njene elemente. Povezovalnik ugotovi, da objektno datoteko a_1 ni potrebno vključiti, saj po predpostavki ne razreši nobenih nedefiniranih simbolov. Ker se glavni program sklicuje na simbole definirane v datoteki a_2, jo je potrebno vključiti v glavni program. Če bi na tem mestu zaključili s preiskovanjem knjižnice A, bi na koncu procesa povezovanja ostali nekateri simboli nedefinirani. Spomnimo se, da se objektna datoteka a_2 sklicuje na simbole iz datotek a_1, ki ni vključena v glavni program. Zato je potreben ponoven prelet čez objektne datoteke modula A. Tokrat povezovalnik pravilno ugotovi, da mora v glavni program vključiti še objektno datoteko a_1.

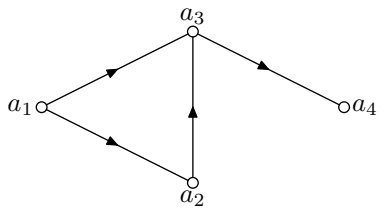
V splošnem moram knjižnico preiskati tolikokrat, da v zadnjem preletu objekti knjižnice ne razrešijo nobenega nedefiniranega simbola. V najslabšem primeru potrebujemo $\mathcal{O}(n)$ preletov, kjer z n označimo število objektnih datotek v knjižnici. S pametno ureditvijo objektnih datotek znotraj knjižnice lahko število preletov v večini primerov zmanjšamo na 2 ali 3.

Programa lorder in tsort

Kot smo videli, lahko ureditev objektnih datotek znotraj knjižnice usodno vpliva na hitrost povezovanja statičnih knjižnic. Če bi, na primer v prejšnjem primeru zamenjali vrstni red objektnih datotek a_1 in a_2, bi pri povezovanju potrebovali le en prelet.

Nekateri UNIX sistemi vsebujejo par programov `lorder` [13] in `tsort` [12], ki

nam pomagata pri ureditvi objektnih datotek v statičnih knjižnicah[1]. Odvisnost med objektnimi datotekami lahko predstavimo z grafom odvisnosti kot to prikazuje slika 4.1. Pri procesu povezovanja bomo potrebovali en sam prelet natanko tedaj, ko bo v grafu odvisnosti obstajala topološka ureditev.



Slika 4.1: Na sliki je prikazan graf odvisnosti štirih objektnih datotek: a_1, a_2, a_3 in a_4. Velja $a_i a_j \in E(G) \Leftrightarrow a_i$ uporablja simbole definirane v a_j. Graf na sliki lahko topološko uredimo. Topološka ureditev je v tem primeru: a_1, a_2, a_3, a_4.

Ukaz `lorder` definira graf odvisnosti objektnih datotek na vhodu. Izhod je sestavljen iz več vrstic. Vrstica "a_i a_j" je na izhodu prisotna natanko tedaj, ko se v modulu a_i sklicujemo na simbole definirane v modulu a_j. V izhodni datoteki lahko vsako vrstico interpretiramo kot povezavo v grafu odvisnosti. Program `lorder` bi na objektnih datotekah iz slike 4.1 vrnil:

```
$ lorder a1.o a2.o a3.o a4.o
a1.o a1.o
a2.o a2.o
a3.o a3.o
a4.o a4.o
a1.o a2.o
a1.o a3.o
a2.o a3.o
a3.o a4.o
```

Prve štiri vrstice so potrebne, saj sicer vozlišča stopnje nič ne bi bili prisotna v izhodu. Ostale štiri pa definirajo graf odvisnosti.

Program `tsort` na vhodu sprejme graf odvisnosti in ga, če je to možno, topološko uredi. Topološka ureditev grafa G obstaja natanko tedaj, ko je graf G usmerjen aciklični graf (DAG). Poglejmo si primer:

[1]Na moji namestitvi operacijskega sistema Linux omenjena programa ne obstajata. Na srečo sem jih, skupaj z izvorno kodo, našel na operacijskem sistemu OpenBSD.

```
$ lorder a1.o a2.o a3.o a4.o | tsort
a1.o
a2.o
a3.o
a4.o
```

Program `tsort` je našel pravilno topološko ureditev. Morda nekatere zanima, kaj se zgodi, če graf odvisnosti vsebuje cikle. V tem primeru program `tsort` izbriše naključno povezavo iz cikla. Ta postopek ponavlja toliko časa, dokler ne pridela grafa, ki ga je mogoče topološko urediti. V skrajnem primeru je potrebno izbrisati vse, razen ene povezave. Če v graf odvisnosti iz slike 4.1 dodamo povezavo a_4a_1, dobimo cikel (a_1, a_3, a_4):

```
$ lorder a1.o a2.o a3.o a4.o | tsort
tsort: cycle in data
tsort: a1.o
tsort: a3.o
tsort: a4.o
a1.o
a2.o
a3.o
a4.o
```

Na UNIX sistemu bi torej knjižnico z objekti `a1.o`, `a2.o`, `a3.o` in `a4.o` zgradili z ukazom:

```
$ ar cr liba.a ‘lorder a1.o a2.o a3.o a4.o | tsort‘
```

4.1.2 Primeri statičnih knjižnic

V nadaljevanju si bomo ogledali dva možna primera implementacij statičnih knjižnic, na koncu pa bomo opisali, kako delujejo statične knjižnice v verigi hiputils.

Preprost format statične knjižnice

V najbolj preprostem primeru je statična knjižnica le stik ene ali več objektnih datotek in ne vsebuje nobenega imenika:

```
$ cat a1.o a2.o a3.o a4.o > liba.a
```

Takšne knjižnice se na sodobnih sistemih ne uporabljajo več prav pogosto. Ker ne vsebujejo imenika, moramo, če želimo izluščiti kakšno objektno datoteko, prebrati celotno knjižnico. Knjižnice tega tipa so se zaradi svoje preprostosti uporabljale predvsem na magnetnih trakovih in luknjanih karticah, saj nam imenik na zaporednem mediju tako ali tako ne koristi.

Možna implementacija statičnih knjižnic na sistemu UNIX

V literaturi [4] je predstavljena zelo zanimiva ideja implementacije statičnih knjižnic na operacijskem sistemu UNIX. Čeprav se tak sistem v praksi ne uporablja, je zanimiv, zato si ga bomo v nadaljevanju ogledali.

Knjižnica je navaden direktorij v datotečnem sistemu operacijskega sistema. Direktorij vsebuje vse objektne datoteke, ki sestavljajo knjižnico. V direktoriju se za vsak izvožen simbol nahaja simbolična povezava z imenom simbola, ki kaže na objektno datoteko, v katerem je simbol definiran.

Recimo, da objektna datoteka `a1.o` izvozi simbola `foo` in `bar`, datoteka `a2.o` izvozi simbol `foobar`, datoteka `a3.o` izvozi simbola `baz` in `qux`, datoteka `a4.o` pa izvozi simbol `quux`. Knjižnica z imenom `liba` je shranjena tako:

```
$ ls -l liba/
total 16
-rw-r--r--  1 jure  jure  600 Sep 13 15:58 a1.o
-rw-r--r--  1 jure  jure  568 Sep 13 15:58 a2.o
-rw-r--r--  1 jure  jure  568 Sep 13 15:58 a3.o
-rw-r--r--  1 jure  jure  568 Sep 13 15:58 a4.o
lrwxr-xr-x  1 jure  jure    4 Sep 13 15:59 bar -> a1.o
lrwxr-xr-x  1 jure  jure    4 Sep 13 15:59 baz -> a3.o
lrwxr-xr-x  1 jure  jure    4 Sep 13 15:58 foo -> a1.o
lrwxr-xr-x  1 jure  jure    4 Sep 13 15:59 foobar -> a2.o
lrwxr-xr-x  1 jure  jure    4 Sep 13 15:59 quux -> a4.o
lrwxr-xr-x  1 jure  jure    4 Sep 13 15:59 qux -> a3.o
```

Opisan sistem je zelo eleganten, saj za ustvarjanje in preiskovanje knjižnice ne potrebujemo posebnih programov, ampak samo standardne programe za delo z direktoriji in datotekami.

Statične knjižnice verige hiputils

Statične knjižnice za procesor HIP sem implementiral podobno kot to počne program `ar` na UNIX-u. Statično knjižnico `liba.a` sestavimo iz objektnih datotek `a1.o` in `a2.o` z naslednjim ukazom:

```
$ ar.py -o liba.a a1.o a2.o
```

Izhodna datoteka `liba.a` je prikazana v izpisu 4.1.

Knjižnica se začne z vrstico:

```
LIBRARY nnnn pppppp
```

`nnnn` je število objektnih datotek v knjižnici, `pppppp` pa je odmik v bajtih od začetka datoteke do začetka imenika. Glavi knjižnice sledijo konkatenirane objektne datoteke. Na koncu datoteke (oziroma na odmiku `pppppp`) pa se nahaja imenik. Imenik vsebuje `nnnn` vrstic oblike:

```
pppppp llllll simbol1 simbol2 simbol3 ...
```

`ppppp` in `llllll` hranita lokacijo in dolžino objektne datoteke. Simboli `simbol1`, `simbol2`, `simbol3`, ... pa so simboli, ki jih objektna datoteka izvozi.

4.2 Dinamične knjižnice

Druga vrsta knjižnic, ki se danes zelo pogosto uporablja, so dinamične knjižnice[2]. Pri dinamičnih knjižnicah ostaneta glavni progam in knjižnica v ločenih datotekah, ki ju povežemo šele pri nalaganju.

Eden izmed načinov, kako lahko implementiramo dinamične knjižnice, je pozicijsko neodvisna koda (Position Independent Code - PIC). Spoznali bomo, kaj vse moramo storiti, če hočemo, da se knjižnica izvaja na poljubnem začetnem naslovu in zakaj je to sploh potrebno. Nato bomo primerjali statične in dinamične knjižnice in našteli njihove prednosti in slabosti. Na koncu si bomo ogledali še kako so dinamične knjižnice implementirane v verigi hiputils in kako jih ustvarjamo ter uporabljamo. Dinamične knjižnice formata ELF so lepo razložene tudi v literaturi [2].

[2] Z izrazom *dinamične knjižnice* v tem poglavju označujemo ELF dinamične knjižnice.

Izpis 4.1: Primer statične knjižnice verige hiputils

```
1  LIBRARY
2  0002 000238
3  LINK
4  0 2 4 0
5
6  # needs
7
8  # segments
9  .data            00000000    4  RWP
10 .text            00000000    8   RP
11
12 # symbols
13 .data            00000000    1    S
14 .text            00000000    2    S
15 bar              00000004    2    D
16 foo              00000000    1    D
17
18 # relocations
19
20 # data
21 00000001
22 C0430800D8210800
23 LINK
24 0 1 3 1
25
26 # needs
27
28 # segments
29 .text            00000000    8   RP
30
31 # symbols
32 .text            00000000    1    S
33 foo              00000000    1    U
34 foobar           00000004    1    D
35
36 # relocations
37 4       1       2       U2
38
39 # data
40 C04118001C010000
41 000014 000136 bar foo
42 00014A 0000EE foobar
```

4.2.1 Pozicijsko neodvisna koda

Ena izmed glavnih prednosti dinamičnih knjižnic pred statičnimi je ta, da je potrebno segment knjižnice, ki se med izvajanjem ne spreminja, v pomnilnik naložiti samo enkrat. Vsak program, ki knjižnico uporablja, mora le preslikati ustrezen segment v svoj naslovni prostor. S tem lahko prihranimo ogromno pomnilnika med izvajanjem.

Ker v splošnem programi ne preslikajo segmentov knjižnice vedno na isti začetni naslov, morajo obstajati mehanizmi, ki omogočajo, da se knjižnica lahko izvaja na poljubnem začetnem naslovu. Ampak te mehanizme že poznamo. V poglavju 3.3 smo se naučili, da premestitev rešuje točno ta problem. Zakaj torej ne uporabimo premestitev? Glavni razlog je, da bi proces premestitve fizično spremenil področje .text knjižnice in bi zato vsak program potreboval svojo lastno kopijo celotne knjižnice. S tem žal izgubimo eno izmed najbolj pomembnih prednosti dinamičnih knjižnic. Potrebna bo drugačna rešitev.

Iščemo torej način, kako lahko isto knjižnico naložimo na poljuben začetni naslov, brez da bi pri tem spremenili področja .text. Problem so seveda naslovi, ki se v kodi pojavljajo. Rešitev, ki se uporablja pri dinamičnih knjižnicah formata ELF in ki sem jo uporabil tudi sam je, da vse naslove premaknemo iz področja .text v posebno področje .got, ki se nahaja med podatki (torej ga lahko pred izvajanjem spreminjamo). Namesto, da bi premestitve opravil povezovalnik, kot je to v navadi pri povezljivih objektnih datotekah, to stori kar nalagalnik. Zaradi te dodatne zadolžitve si zasluži tudi drugačno ime - *dinamični nalagalnik*.

Globalna tabela odmikov

Globalna tabela odmikov (Global Offset Table - GOT) je shranjena v posebnem področju .got, ki ga lahko beremo in pišemo. V globalni tabeli odmikov so med izvajanjem shranjeni vsi naslovi globalnih simbolov, ki jih v modulu uporabljamo. V kodi se lahko sedaj znebimo vseh ukazov s problematičnimi naslovi tako, da dostope do globalnih simbolov opravimo neposredno preko tabele GOT. Za te spremembe je zadolžen prevajalnik višjenivojskih jezikov oziroma programer v zbirniku. V vsakem primeru se bo sedaj izhodna datoteka lahko naložila na poljuben naslov, brez da bi pri tem morali popravljati področje .text.

Prostor za tabelo GOT ustvari povezovalnik, vendar tabele še ne napolni z naslovi. V izhodno datoteko raje zapiše premestitvene vnose, ki služijo kot

navodila dinamičnemu nalagalniku kako naj ustvari posamezne vnose tabele. Tipa premestitvenih vnosov, ki se tu pojavita sta:

ER4: Vse kar mora dinamični nalagalnik storiti je prišteti nov začetni naslov programa. Za ta tip premestitve potrebujemo relativno malo časa. Bolj nas skrbi naslednji tip premestitve.

A4: Dinamični nalagalnik mora razrešiti simbol, na katerega kaže premestitveni vnos. To ponavadi zahteva zamudno iskanje po zgoščenih tabelah, kar je počasna operacija.

Preden uporabimo tabelo GOT, moramo najprej v register naložiti njen naslov. To moramo seveda storiti na pozicijsko neodvisen način. Tega problema se na različnih arhitekturah lotimo na različne načine. Kako je to rešeno na procesorju HIP, si lahko ogledate v dodatku A.5. Na procesorjih x86 pa s sledčimi ukazi:

```
        ...
        call    __i686.get_pc_thunk.bx
        addl    $_GLOBAL_OFFSET_TABLE_, %ebx
        ...

__i686.get_pc_thunk.bx:
        movl    (%esp), %ebx
        ret
```

Procedura `__i686.get_pc_thunk.bx` prenese vrednost programskega števca v register `%ebx`, ukaz `addl` pa prišteje ustrezen odmik do tabele GOT.

4.2.2 Nalaganje dinamične knjižnice

Sedaj bomo opisali poenostavljen opis postopka nalaganja dinamične knjižnice v formatu ELF.

Ko operacijski sistem zažene program, preslika njegove segmente v novo ustvarjen naslovni prostor. Ponavadi imajo vsi programi prisotno področje .interp, ki vsebuje ime dinamičnega nalagalnika. Na operacijskem sistemu Linux je to običajno dinamična knjižnica `ld-linux.so`. Operacijski sistem, namesto da bi začel izvajanje programa, preslika vsebino dinamičnega nalagalnika v naslovni prostor programa in skoči na vstopno točko nalagalnika.

Ker je dinamični nalagalnik implementiran kot dinamična knjižnica, ki med drugim vsebuje tudi premestitvene vnose, mora nalagalnik vsebovati na začetku kodo, ki opravi vse potrebne premestitve na svojem lastnem segmentu.

Nalagalnik najprej ustvari globalno tabelo simbolov, ki je implementirana kot seznam, katerega elementi so tabele simbolov vseh do sedaj naloženih dinamičnih knjižnic. Nato iz glave programa razbere katere dinamične knjižnice potrebuje, jih preslika v naslovni prostor in doda nov vnos v globalno tabelo simbolov. Lahko se zgodi, da je katera dinamična knjižnica odvisna od druge. Seveda moramo v tem primeru v naslovni prostor preslikati obe knjižnici.

Ko so enkrat vse dinamične knjižnice naložene in je globalna tabela simbolov dokončno zgrajena, pride na vrsto proces premestitve. Premestitev je pri dinamičnem nalaganju nekoliko bolj preprosta kot pri povezovanju, saj imamo opravka le z nekaj različnimi tipi premestitvenih vnosov. Ko se proces premestitve zaključi, so vse potrebne dinamične knjižnice naložene in pripravljene za zagon. Vse kar je še potrebno, je skok na vstopno točko in izvajanje programa se lahko prične.

4.2.3 Prednosti in slabosti

Dinamične knjižnice imajo svoje prednosti in slabosti. Prednosti v večini primerov odtehtajo slabosti, zato se dinamične knjižnice danes uporabljajo zelo pogosto. Prednosti dinamičnih knjižnic so:

- Več različnih programov, ki uporabljajo isto dinamično knjižnico, si lahko delí segment knjižnice, ki se med izvajanjem ne spreminja. To pomeni, da ga je potrebno v spomin naložiti le enkrat, navidezni pomnilnik pa poskrbi, da se segment preslika v naslovni prostor vsakega programa, ki knjižnico uporablja. Na ta način lahko zmanjšamo porabo pomnilnika in izvajanje programov pohitrimo, saj se napake strani pojavljajo manj pogosto.
- V primeru, da v dinamični knjižnici odkrijemo ranljivost, je dovolj že to, da staro knjižnico nadomestimo z novo (in ponovno zaženemo programe, ki jo uporabljajo). Pri statičnih knjižnicah moramo knjižnico ponovno povezati z vsemi programi, ki knjižnico uporabljajo. To pa je nočna mora vsakega sistemskega administratorja.
- Pri statičnih knjižnicah se knjižnica pri povezovanju fizično skopira v izhodno objektno datoteko. Če veliko programov uporablja isto knjižnico,

bi zaradi tega izgubili nekaj prostora na trdem disku. Pri dinamičnih knjižnica tega problema ni, saj se koda knjižnice in programa združita šele pri nalaganju.

- Z uporabo dinamičnih knjižnic lahko na preprost način implementiramo podporo za tako imenovane dodatke (plugin).

Uporaba dinamičnih knjižnic prinese tudi nekaj negativnih posledic:

- Ker mora dinamični nalagalnik pred vsakim zagonom programa najprej poiskati vse knjižnice, ki jih program potrebuje, nato pa še opraviti vse premestitve, je začetni čas programov, ki uporabljajo veliko dinamičnih knjižnic daljši, kot če bi uporabili statične knjižnice.
- Ker dinamične knjižnice zahtevajo pozicijsko neodvisno kodo, je na nekaterih arhitekturah koda prevedena za dinamične knjižnice občutno počasnejša kot sicer. Vsaka metoda, ki uporablja globalne simbole, mora najprej naložiti naslov tabele GOT, poleg tega mora rezervirati še en register, v katerem se naslov tabele hrani. Na procesorju HIP, ki ima 32 splošnonamenskih registrov, to ni velik problem[3].
- Pri dinamičnih knjižnicah imamo lahko probleme, če pogosto menjamo stare verzije z novimi, oziroma če to za našim hrbtom počne namestitveni program. Nekateri programi imajo težave, če se pri nalaganju povežejo s knjižnico, ki je novejša ali starejša od tiste, za katero so bili napisani.

4.2.4 Dinamične knjižnice verige hiputils

Za konec si poglejmo še kako so dinamične knjižnice implementirane v verigi hiputils. Dinamične knjižnice verige hiputils se zgledujejo po formatu ELF. Implementirane so z pozicijsko neodvisno kodo, ki uporablja globalno tabelo odmikov. Razlika je tudi v tem, da dinamičen nalagalnik verige hiputils ni implementiran v obliki dinamične knjižnice, ampak je kar neposredno vgrajen v simulator.

Dinamične knjižnice ustvarimo z ukazom `ld.py -shared`. Recimo da želimo ustvariti dinamično knjižnico `liba.so` iz datotek objektnih `a1.o`, `a2.o`, `a3.o` in `a4.o`. To lahko storimo z ukazom:

[3]Prav tako to ni problem na procesorju MMIX, ki ima kar 256 splošnonamenskih registrov.

```
$ ld.py --shared -o liba.so a1.o a2.o a3.o a4.o
```

Format dinamičnih knjižnic je enak formatu za povezljive in izvršljive objektne datoteke, le da je prva vrstica namesto `LINK` enaka `LINKLIB`, kar označuje, da gre za dinamično knjižnico. Dinamično knjižnico lahko sedaj povežemo z glavno objektno datoteko:

```
$ ld.py -o main.out main.o liba.so
```

V izhodni datoteki `main.out` je zapisano, da je potrebno pred izvajanjem v spomni naložiti knjižnico `liba.so`. Poglejmo si samo del datoteke `main.out`.

```
1 LINK
2 1 3 6 1
3
4 # needs
5 m.so
6
7 # segments
8 ...
```

Poglavje 5

Zaključek

V diplomski nalogi smo si ogledali osnovne principe povezovalnikov in nalagalnikov. Začeli smo z opisom objektnih datotek, kjer smo podrobno predstavili formate datotek COM, ELF in a.out ter format objektne datoteke verige hiputils. Sledil je opis procesa povezovanja, kjer smo se seznanili s tremi glavnimi nalogami vsakega povezovalnika: razporeditev segmentov, upravljanje s simboli in premestitev. Končali pa smo z opisom statičnih in dinamičnih knjižnic in procesom dinamičnega nalaganja.

Za namene te diplomske naloge sem izdelal verigo programov z imenom hiputils, ki implementirajo večino tem, ki smo jih obravnavali. Verigo programov hiputils sestavljajo:

- zbirnik *as.py*
- povezovalnik *ld.py*
- dinamični nalagalnik *ld_so.py*
- program za ustvarjanje statičnih knjižnic *ar.py*
- program za ogled strojne kode (disassembler) *objdump.py*
- simulator *sim.py*

Posamezne programe verige sem opisal na mestih, kjer je bilo to primerno. Opis programov, ki niso neposredno povezani z glavno temo diplomske naloge (*ar.py* in *sim.py*), sem izpustil.

5.1 Nadaljnje delo

V tej diplomski nalogi smo si ogledali kar nekaj različnih tem povezanih s povezovanjem in nalaganjem programov in knjižnic. Jasno je, da smo morali nekatere teme in ideje tudi izpustiti. Med bolj zanimivimi so:

- *Naključna razporeditev naslovnega prostora* (Address Space Layout Radomization - ASLR) je tehnika dinamičnega nalaganja, kjer začetne naslove segmentov izbere generator naključnih števil. Prednost tehnike ASLR je, da onemogoči ali vsaj oteži nekatere klasične napade ("return to libc" in "shellcode injection", glej [1]) na slabo napisane programe, ponavadi v programskem jeziku C.
- *Globalna optimizacija*: ker je povezovalnik edini del sistema, ki vidi program v celoti, lahko izvaja določene optimizacije, ki sicer (na primer s prevajalnikom) niso možne. To postane še posebej zanimivo, če so bili posamezni moduli programa napisani v različnih programskih jezikih (eden v Pascl-u, drugi v C++-u, tretji v Fortran-u,...). Primer takšnega povezovalnika je predstavljen v literaturi [6].
- V knjigi [4] me je navdušila ideja o formatu statičnih knjižnic (glej poglavje 4.1.2), ki so implementirane kot navadni direktoriji v datotečnem sistemu. Idejo bi bilo dobro implementirati, hitrost preiskovanja pa primerjati s programom `ar`, ki se na sistemih UNIX ponavadi uporablja za izgradnjo statičnih knjižnic.

V diplomi sem predstavili programe verige hiputils. Delo na teh programih še ni končano. V prihodnje bo treba še:

- *Testirati, testirati in testirati!* Programi verige hiputils sicer delujejo pravilno na vseh testnih primeri, ki sem si jih uspel domisliti, vendar to ne pomeni, da ne vsebujejo nobene napake. Tako kot pri vsakem drugem programu, je tudi tukaj potrebno poiskati čim več hroščev in jih odpraviti.
- Ker je format ELF danes tako zelo razširjen, bi bilo zaželeno, če bi ga programi verige hiputils podpirali. Podporo za ELF lahko vgradimo tako, da na novo napišemo funkciji `read` in `write` iz datoteki `obj.py`, ali pa uporabimo knjižnico BFD, ki jo uporabljajo tudi programi verige binutils.

Dodatek A

Zbirnik za procesor HIP

Na prvi pogled se morda zdi, da zbirniki niso vključeni v proces povezovanja, vendar to ni res. Med prevajanjem si zbirnik gradi tabelo naslovov, za katere je v tistem trenutku še nemogoče določiti dejanski naslov. Ta tabela se v obliki premestitvenih vnosov shrani v izhodno objektno datoteko. Če sedaj za trenutek na proces povezovanja pogledamo samo iz vidika premestitve, dobimo o zbirnikih popolnoma drugačno mnenje. Zbirniki so namreč tisti, ki vodijo proces premestitve. Na premestitvene vnose lahko gledamo kot na posebne ukaze, ki se izvršijo v povezovalniku, ki ima v tem razmerju položaj podrejenega. Povezovalnik ni torej nič več kot pasivna enota, njegova edina zadolžitev pa je izvrševanje ukazov, ki mu jih je v objektni datoteki pustil zbirnik.

Če želimo napisati svoj lasten povezovalnik, nam torej ne preostane drugega, kot da napišemo tudi lasten zbirnik ali pa vsaj prilagodimo že obstoječega. Zbirnik za procesor HIP sicer že obstaja, vendar sem se v tem primeru raje odločil za prvo opcijo. Na koncu se je izkazalo, da sem največ časa porabil ravno za programiranje procedur, ki so zadolžene za ustvarjanje premestitvenih vnosov - nekaj kar bi tako ali tako moral napisati na novo, če bi želel uporabiti že obstoječ zbirnik. Samo programiranje zbirnika se je izkazalo kot zelo zabavna in poučna izkušnja.

V tem poglavju si bomo ogledali zbirnik za procesor HIP in našteli njegove glavne značilnosti. Ker zbirniki niso glavna tema te diplomske naloge, bomo opis zgradbe in izvorno kodo zbirnika verige hiputils izpustili.

Najprej si bomo ogledali ukaze, psevdo ukaze in makro ukaze zbirnika. Nato bomo spregovorili nekaj besed o rabi registrov in o dogovoru za klicanje procedur. Na koncu si bomo ogledali še, kako lahko do podatkov dostopamo z

gp-relativnim naslavljanjem, oziroma preko globalne tabele odmikov. Pri implementaciji zbirnika in tudi programov verige hiputils sem se največ zgledoval po procesorju MIPS in literaturi [5, 7].

A.1 Ukazi

Vsi ukazi procesorja HIP in njihova zgradba so lepo razloženi že v [3, 9], zato bi bilo njihovo ponovno opisovanje na tem mestu nesmiselno. Zaradi preprostosti sem izpustil ukaze `TRAP`, `RFE`, `EI`, `DI`, `MOVER` in `MOVRE`. V tem razdelku si bomo raje ogledali psevdo ukaze našega zbirnika in razložili kako je možno z uporabo preprostega trika, v izvorni kodi uporabiti tudi makro ukaze.

A.1.1 Psevdo ukazi

Psevdo ukazi so ukazi zbirnika, ki se, za razliko od navadnih ukazov, ne prevedejo neposredno v strojno kodo. Služijo le kot dodatna navodila zbirniku, kako in predvsem kam naj prevede določen odsek kode. V izvorni datoteki lahko psevdo ukaze preprosto ločimo od navadnih, saj se vsi začnejo z znakom '.'. Vsi psevdo ukazi, ki jih zbirnik verige hiputils podpira so navedeni v tabeli A.1.

psevdo ukaz	opis
`.ascii "s"`	ASCII niz
`.asciz "s"`	ASCII niz zaključen z bajtom 0
`.space n`	preskoči `n` bajtov
`.align n`	poravnaj naslov, da bo deljiv z `n`
`.word n1, n2,...`	zapiši eno ali več 32 bitnih števil
`.short n1, n2,...`	zapiši eno ali več 16 bitnih števil
`.byte n1, n2,...`	zapiši eno ali več 8 bitnih števil
`.globl s`	simbol `s` izvozi kot globalni simbol
`.data`	začetek segmanta .data
`.text`	začetek segmanta .text
`.bss`	začetek segmanta .bss

Tabela A.1: Psevdo ukazi zbirnika verige hiputils.

A.1.2 Makro ukazi

Da bi zmanjšali število vrstic izvorne kode in povečali berljivost programov v zbirnem jeziku, je potrebno ukaze, ki se v skupini pojavljajo na večih mestih, zamenjati z makro ukazi. Makro ukazi se ne izvršujejo neposredno na procesorju, ampak se v procesu prevajanja razširijo v enega ali več strojnih ukazov. Večina problemov, ki jih makro ukazi rešujejo je prikaza na sliki A.1.

```
1  /* nalozi naslov foo v register r2 */
2  lhi     r2,%hi(foo)                    LA(r2, foo)
3  addi    r2,r2,%lo(foo)
4
5  /* nalozi naslov bar v register r3 */
6  lhi     r3,%hi(bar)                    LA(r3, bar)
7  addi    r3,r3,%lo(bar)
8
9  /* nalozi naslov baz v register r4 */
10 lhi     r4,%hi(baz)                    LA(r4, baz)
11 addi    r4,r4,%lo(bar)
12
13 /* poklici proceduro foobar */
14 lhi     r1,%hi(foobar)                 CALL(foobar)
15 call    r31,%lo(foobar)(r1)
16
17 /* ukaz NOP */
18 addi    r0,r0,0                        NOP
```

Slika A.1: V levem delu izpisa je prikazan prvoten program. Na desni strani pa je identičen program, napisan s pomočjo makro ukazov. Iz primera se lepo vidi, da so programi, ki vsebujejo makro ukaze, krajši in bolj razumljivi.

Makro ukaze bi sicer lahko implementirali tako, da bi jih neposredno vključili v kodo zbirnika. Obstaja pa elegantnejši način: programerji so hitro ugotovili, da se potreba po makro ukazih pogosto pojavlja. Nastali so številni programi (na primer m4 ali cpp), ki so namenjeni izključno izvrševanju makro ukazov. Konkreten primer programa tega tipa je *"The C Preprocessr" (cpp)*, ki ga bomo uporabili tudi mi. Prednost uporabne zunanjega programa za izvrševanje makro ukazov so jasne in se ujemajo z UNIX filozofijo.

Vse makro ukaze bomo torej implementirali kot ukaze preprocesorju `cpp` in jih shranili v datoteko `hip/asm.h`. Izpis A.1 prikazuje definicije vseh makro ukazov, ki so trenutno implementirani. Prijetna posledica dejstva, da smo se odločili za zunanji program je tudi to, da lahko uporabnik na preprost način

Izpis A.1: Definicije makro ukazov iz datoteke `hip/asm.h`.

```
1  /* system calls */
2  #define EXIT 0
3  #define PRINTF 1
4  #define READ_INT 2
5  #define READ_STRING 3
6  #define DUMP_MEM 4
7  #define DUMP_REG 5
8
9  /* pseudo ops */
10 #define NOP addi r0,r0,0
11
12 #define LOAD_GP                              \
13   lhi   gp,%hi(_GP);                         \
14   addi  gp,gp,%lo(_GP)
15
16 #define LOAD_GOT(label)                      \
17   addui gp,t9,label@TOGOT
18
19 #define LA(reg, addr)                        \
20   lhi   reg,%hi(addr);                       \
21   addi  reg,reg,%lo(addr)
22
23 #define CALL(addr)                           \
24   lhi   at,%hi(addr);                        \
25   call  ra,%lo(addr)(at)
26
27 #define CALL_SO(addr)                        \
28   LA(t9,addr);                               \
29   call  ra,0(t9)
```

doda nove makro ukaze. Vse kar mora storiti, je dodati ustrezne definicije v datoteko `hip/asm.h`.

A.2 Sistemski klici

V simulator je vgrajen tudi zelo preprost operacijski sistem. Do funkcij, ki nam jih ta ponuja, dostopamo z ukazom `syscall n`, kjer je `n`, zaporedna številka storitve, ki jo želimo uporabiti. Zaenkrat pozna operacijski sistem le nekaj osnovnih sistemskih klicev, vendar dodajanje novih ne bi smel biti prevelik problem. Podprti sistemski klici so opisani v tabeli A.2.

#	ime	opis
0	EXIT	Zaključi izvajanje programa.
1	PRINTF	Izpiši niz na naslovu `a0`, z argumenti na naslovih `a1`, `a2` in `a3`.
2	READ_INT	Preberi število in ga shrani v register `a0`.
3	READ_STRING	Preberi niz in ga skopiraj na naslov `a0`.
4	DUMP_MEM	Izpiši vrednosti vseh uporabljenih pomnilniških lokacij.
5	DUMP_MEM	Izpiši vrednosti vseh registrov.

Tabela A.2: Sistemski klici preprostega operacijskega sistema.

A.3 Dogovor o uporabi registrov

Pri prevajanju izvorne kode višjenivojskih jezikov mora prevajalnik vedeti katere registre lahko uporablja in kateri registri so rezervirani za ostale procedure. To je še posebej pomembno, če se med seboj kličeta proceduri, ki sta bili prevedeni z različnima prevajalnikoma, ali če iz zbirnika kličemo proceduro, ki je bila prevedena s prevajalnikom višjenivojskega jezika. Potreben je dogovor, ki za vsak register določa kdaj in kako ga lahko uporabimo. Raje, kot da si izmislimo novo, bomo uporabili že preverjeno shemo, ki jo uporablja prevajalnik GCC na arhitekturi MIPS.

- Register `zero` (0) vedno vsebuje konstanto 0. Pri pisanju v register `zero` se ne zgodi nič.
- Registri `at` (1), `k0` (26) in `k1` (27) so rezervirani za izvrševanje makro ukazov in procedur operacijskega sistema, zato jih programerji in prevajalniki ne smejo uporabljati.
- Registri `a0-a3` (4-7) se uporabljajo za prenos prvih štirih argumentov pri klicu procedur. Če procedura sprejme več kot štiri argumente se ostali argumenti prenesejo prek sklada.
- Registra `v0` (2) in `v1` (3) uporabljamo za vračanje vrednosti iz procedur.
- `t0-t9` (8-15, 24, 25) so registri, ki jih uporabljamo za hranjenje začasnih vrednosti. Predpostaviti moramo, da se vrednost teh registrov po klicu procedure spremeni. Če želimo, da se njihova vrednost po klicu ohrani, smo za to odgovorni sami.

- `s0-s7` (16-23) so registri, kjer lahko hranimo spremenljivke, ki živijo dalj časa. Njihova vrednost naj se ohrani tudi po klicu procedure.

- `gp` (28) naj kaže nekam v bližino segmenta .data. Z njegovo pomočjo lahko hitro naslovimo 64K bajtov velik blok pomnilnika. Več o gp-relativnem naslavljanju bomo spregovorili v razdelku A.4

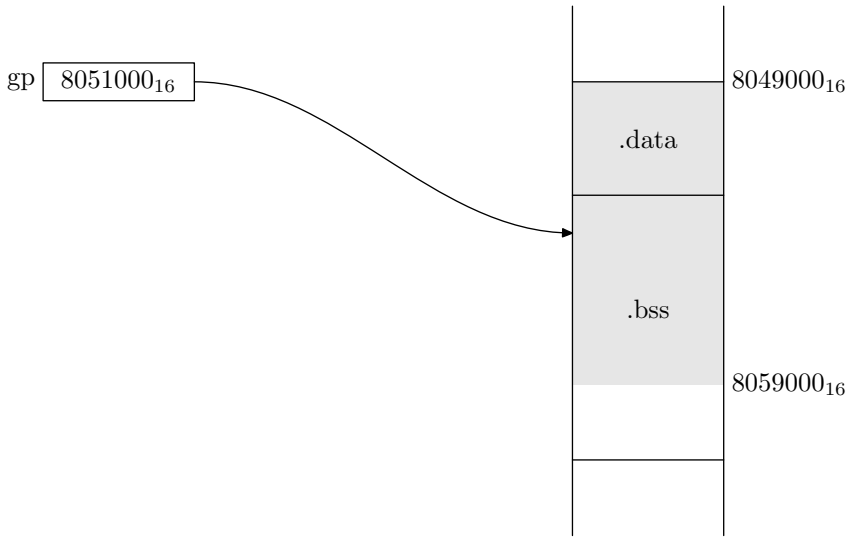
- V registru `sp` (29) naj bo shranjen naslov sklada.

- Register `fp` (30) naj vsebuje naslov okvirja. S pomočjo naslova okvirja lahko prevajalniki implementirajo lokalne spremenljivke.

- Register `ra` (31) naj hrani naslov, kamor se moramo vrniti, ko zaključimo izvajanje procedure.

Definicije simboličnih imen registrov se nahajajo v datoteki `hip/regdef.h`.

A.4 Gp-relativno naslavljanja



Slika A.2: Register `gp` kaže na naslov 8051000_{16}. Če je uporabljen kot bazni register, je z njegovo pomočjo mogoče nasloviti naslove od 8049000_{16} do 8058000_{16} (zatemnjen del pomnilnika).

Vsi ukazi procesorja HIP so veliki 32 bitov, zato potrebujemo natanko dva ukaza, če želimo v register prenesti 32 bitni naslov[1]. Pri programih, ki pogosto uporabljajo globalne in statične spremenljivke, se te dodatni ukazi poznajo tako na času izvajanja, kot tudi na velikosti izhodne datoteke. Rešitev tega problema je gp-relativno naslavljanje. Pri gp-relativnem naslavljanju najprej nastavimo register `gp`, da kaže nekam na sredino segmenta .data, nato ga lahko pri dostopu do globalnih in statičnih spremenljivk uporabimo kot bazni register in tako pri vsakem dostopu privarčujemo en ukaz. Takojšnji operand je pri procesorju HIP velik 16 bitov, zato lahko z gp-relativnim naslavljanjem naslovimo 64K kos pomnilnika okoli naslova, ki je shranjen v registru `gp` (slika A.2). Če naslov pade izven tega kosa, je to usodna napaka, ki jo zazna povezovalnik. Slika A.3 primerja navaden program s programom, ki uporablja gp-relativno naslavljanje.

A.5 Globalna tabela odmikov

Dinamične knjižnice se lahko naložijo na katerikoli naslov, zato mora bit njihova koda pozicijsko neodvisna. Eden izmed načinov kako implementiramo pozicijsko neodvisno kodo je s pomočjo globalne tabele odmikov (GOT). Velik del realizacije globalne tabele odmikov opravita povezovalnik in nalagalnik, vseeno so spremembe potrebne tudi v zbirniku. Rešiti je namreč potrebno naslednje probleme:

Kako naložiti naslov tabele GOT? Najlažje bi bilo, če bi naslov lahko z ukazoma `lhi` in `addi` kar prenesli v register `gp`, podobno kot smo to storili pri že gp-relativnem naslavljanju. Ta naiven pristop ima v sebi skrito veliko pomanjkljivost. Ker bi absolutni naslov tabele GOT zapekli v segment .text, bi s tem zgubili želeno lastnost pozicijske neodvisnosti. Prav zaradi pozicijske neodvisnosti smo se sploh začeli ukvarjati s tabelo GOT. Očitno bo potrebna drugačna rešitev.

Opazimo, da pri dinamičnem nalaganju razdalja med segmenti ostane enaka, zato lahko že pri povezovanju določimo odmik med začetkom klicane procedure in tabelo GOT. Računanje naslova globalne tabele odmikov si olajšamo tako, da se dogovorimo za poseben način klicanja

[1]Nekateri procesorji potrebujejo še več ukazov. Procesor MMIX profesorja D. Knuth-a potrebuje zaporedje kar štirih ukazov - `SETH`, `INCMH`, `INCML` in `INCL` za prenos sicer 64 bitnega naslova.

```
1  #include <hip/regdef.h>
2  #include <hip/asm.h>
3
4          .data
5  foo:    .word 0xDEAD
6  bar:    .word 1337
7  baz:    .word 0xCAFE
8
9          .text
10         .globl _start
11 _start:
12
13
14
15
16         lhi     t0,%hi(foo)
17         lw      t0,%lo(foo)(t0)
18
19         lhi     t1,%hi(bar)
20         lw      t1,%lo(bar)(t1)
21
22         lhi     t2,%hi(baz)
23         addi    t2,t2,%lo(baz)
24         ...
```

(a) Prvoten program.

```
#include <hip/regdef.h>
#include <hip/asm.h>

        .data
foo:    .word 0xDEAD
bar:    .word 1337
baz:    .word 0xCAFE

        .text
        .globl _start
_start:
        /* nastavi gp */
        lhi     gp,%hi(_GP)
        addi    gp,gp,%lo(_GP)

        lw      t0,foo@GPOFF(gp)


        lw      t1,bar@GPOFF(gp)


        addi    t2,gp,baz@GPOFF

        ...
```

(b) Prirejen program, ki uporablja gp-relativno naslvljanje

Slika A.3: S pomočjo gp-relativnega naslavljanja lahko do podatkov, definiranih v segmentu .data, dostopamo z enim ukazom. Prav tako potrebujemo le en ukaz, če želimo v register prenesti naslov simbola.

procedur. Dogovorimo se, da pred klicem procedure, v register `t9` shranimo njen naslov. Naslov tabele GOT bomo hranili v registru `gp`. Do njega lahko pridemo z naslednjim ukazom:

```
        .globl foo
foo:
        addui   gp,t9,foo@TOGOT
        ...
```

Ukaz `foo@TOGOT` se v zbirniku razširi v preprosto odštevanje.

```
        .globl foo
foo:
        addui   gp,t9,_GOT - foo
        ...
```

Kako dostopamo do naslovov shranjenih v tabeli GOT? Ko imamo v registru `gp` shranjen naslov globalne tabele odmikov, lahko do posameznega elementa v tabeli dostopamo z ukazom `@GOT`. Naj bo `bar` simbol, ki je spravljen v tabeli GOT. Njegov naslov lahko v register `t0` naložimo z naslednjim ukazom:

```
        lw      t0,bar@GOT(gp)
```

Kako dostopamo do podatkov v segmentu .data in .bss? Postopek je skoraj identičen kot pri gp-relativnem naslavljanju. Naj bo `baz` simbol, ki je definiran v segmentu .data. Njegovo vrednost bi želeli prenesti v register `a0`. Razmak med simbolom `baz` in tabelo GOT izvemo z ukazom `baz@GOTOFF`.

```
        .data
baz:    .byte 42

        .text
        ...
        lw      a0,baz@GOTOFF(gp)
        ...
```

Dodatek B

Izvorna koda

B.1 Splošne datoteke

Izpis B.1: `common.py`

```
1  import sys
2
3  def die(msg):
4      print >> sys.stderr, msg
5      sys.exit(1)
6
7  def pack_int(n, size):
8      '''pack a signed integer n into size bits. pack_int(-1,4) = 0xF'''
9      msg = 'integer %#x out of range' % n
10     size = 1 << (size - 1)
11     if n < 0:
12         -size <= n < size or die(msg)
13         n += (size << 1)
14     else:
15         0 <= n < (size << 1) or die(msg)
16     return n
17
18 def unpack_int(n, size):
19     '''inverse of pack_int. unpack_int(pack_int(n, size), size) = n'''
20     size = 1 << size
21     0 <= n < size or die('integer out of range')
22     if n >= (size >> 1):
23         n -= size
24     return n
25
26 def hi(n):
```

```
27      '''corresponds to the assembler function %hi'''
28      return ((n & 0xffff0000) >> 16) + (1 if n & 0xffff > 0x7fff else 0)
29
30  def lo(n):
31      '''corresponds to the assembler function %lo'''
32      return n & 0xffff
33
34  def align(address, boundry):
35      return (address - 1 + boundry) & -boundry
36
37  def id(x):
38      return x
```

Izpis B.2: `obj.py`

```
1  from common import die
2
3  class Segment:
4      '''represent a segment entry'''
5      def __init__(self, name, base=0, size=0, data=''):
6          seg_flags = {
7              '.text': 'RP',
8              '.got':  'RWP',
9              '.data': 'RWP',
10             '.bss':  'RW'
11             }
12         self.name = name
13         self.base = base
14         self.size = size
15         self.data = data
16         self.num = 0
17         self.flags = seg_flags[name]
18
19     def __repr__(self):
20         return '%-16s %08X %4X %4s' \
21             % (self.name,self.base,self.size,self.flags)
22
23 class Symbol:
24     '''represent a simbol entry'''
25     def __init__(self, name, value, seg, type):
26         self.name = name
27         self.value = value
28         self.seg = seg
29         self.type = type
30         self.linksym = None
31         self.num = 0
32
33     def __repr__(self):
34         return '%-16s %08X %4X %4s' \
35             % (self.name,self.value,self.seg.num,self.type)
36
37 class Relocation:
38     '''represents a relocation entry'''
39     class Type:
40         '''relocation type'''
41         def __init__(self, name, size, abs):
42             self.name = name
43             self.size = size
44             self.abs = abs
45
46     # define all the relocation types
```

```
47      types = [
48          Type("A4", 4, True),
49          Type("A2", 2, True),
50          Type("R2", 2, False),
51          Type("U2", 2, True),
52          Type("L2", 2, True),
53          # GP-relative
54          Type("DR2", 2, True),
55          # GOT
56          Type("GA2", 2, False),
57          Type("GP2", 2, False),
58          Type("GR2", 2, False),
59          # load time
60          Type("ER4", 4, True)
61          ]
62      type_name = {}
63      type_name.update((type.name, type) for type in types)
64      # ugly way to implement enums
65      A4,A2,R2,U2,L2,DR2,GA2,GP2,GR2,ER4 = types
66
67      def __init__(self, loc, seg, ref, type, addend = 0):
68          self.loc = loc
69          self.seg = seg
70          self.ref = ref
71          self.type = type
72          self.addend = addend
73
74      def __repr__(self):
75          s = '%X\t%X\t%X\t%s' % (self.loc, self.seg.num,
76                                  self.ref.num if self.ref else 0,
77                                  self.type.name)
78          return s + ('\t%X' % self.addend) if self.addend != 0 else s
79
80  class Object:
81      '''represents an linkable or executable file or a shared library'''
82      def __init__(self, name, shared):
83          self.needs, self.segs, self.syms, self.rels = [], [], [], []
84          self.name = name
85          self.shared = shared
86
87      def write(self, f):
88          '''write the object file to disk'''
89          segs_out = [s for s in sorted(self.segs, key = lambda s: s.base) \
90                          if s.size > 0]
91
92          f.write('LINKLIB\n' if self.shared else 'LINK\n')
93          f.write('%X %X %X %X\n\n' % (len(self.needs), len(segs_out),
```

```
94                                      len(self.syms), len(self.rels)))
95
96          f.write('# needs\n')
97          f.write(''.join(n + '\n' for n in self.needs) + '\n')
98
99          f.write('# segments\n')
100         i = 1
101         for s in segs_out:
102             f.write(repr(s) + '\n')
103             s.num = i
104             i += 1
105         f.write('\n')
106
107         f.write('# symbols\n')
108         i = 1
109         for s in sorted(self.syms, key=lambda s: s.name):
110             f.write(repr(s) + '\n')
111             s.num = i
112             i += 1
113         f.write('\n')
114
115         f.write('# relocations\n')
116         f.write(''.join(repr(r) + '\n' for r in self.rels) + '\n')
117         f.write('# data\n')
118         f.write(''.join(s.data + '\n' for s in segs_out if 'P' in s.flags))
119
120
121 class Ar:
122     '''represents statically linked libraries'''
123     syms = {}
124     objs = []
125     def __init__(self, name):
126         self.name = name
127
128     def write(self, f):
129         '''write the linked library to file'''
130         start = f.tell()
131         f.seek(len('LIBRARY\nnnnnn pppppp\n'))
132         offset = {}
133         objlen = {}
134         for o in self.objs:
135             offset[o] = f.tell()
136             o.write(f)
137             objlen[o] = f.tell() - offset[o]
138         end = f.tell()
139
140         for o in self.objs:
```

```
141             f.write('%06X %06X ' % (offset[o], objlen[o]))
142             f.write(' '.join(s.name for s in o.syms if s.type == 'D'))
143             f.write('\n')
144 
145         f.seek(start)
146         f.write('LIBRARY\n%04X %06X\n' % (len(objs), end))
147 
148 def getl(f):
149     '''read a line from file f, skipping over empty lines and comments'''
150     while True:
151         line = f.readline()
152         if line == '':
153             return None
154         line = line.strip()
155         if len(line) > 0 and line[0] != '#':
156             return line
157 
158 def read(fname, offset=0):
159     '''read an object file from fname, start at offset'''
160     try:
161         f = open(fname, 'r')
162     except IOError:
163         die('file %s not found' % fname)
164     f.seek(offset)
165     magic = f.readline().strip()
166     if magic in ['LINK', 'LINKLIB']:
167         o = readobj(f, magic == 'LINKLIB')
168     elif magic == 'LIBRARY':
169         o = readar(f)
170     else:
171         die('%s not an object or link file' % fname)
172     f.close()
173     return o
174 
175 def readar(f):
176     '''read a statically linked library'''
177     o = Ar(name = f.name)
178     nmembs, dirpos = [int(n, 16) for n in f.readline().split()[:2]]
179     f.seek(dirpos)
180     for _ in xrange(nmembs):
181         line = f.readline().split()
182         interval = int(line[0], 16)
183         for s in line[2:]:
184             o.syms[s] = interval
185     return o
186 
187 def readobj(f, shared):
```

```
188     '''read an object file f'''
189     o = Object(name=f.name, shared=shared)
190
191     # counters
192     nneeds, nsegs, nsyms, nrels = (int(n,16) for n in getl(f).split()[:4])
193
194     # shared object dependencies
195     o.needs = [getl(f) for _ in xrange(nneeds)]
196
197     # segments: name start length flags
198     for _ in xrange(nsegs):
199         name, base, size, flags = getl(f).split()[:4]
200         o.segs.append(Segment(
201                 name = name,
202                 base = int(base, 16),
203                 size = int(size, 16)))
204
205     # symbols: name value seg type
206     for _ in xrange(nsyms):
207         name, value, seg, type = getl(f).split()[:4]
208         o.syms.append(Symbol(
209                 name = name,
210                 value = int(value, 16),
211                 seg = o.segs[int(seg, 16) - 1],
212                 type = type))
213
214     # relocations: loc seg ref type [addend]
215     for _ in xrange(nrels):
216         l = getl(f).split()
217         loc, seg, ref, type = l[:4]
218         ref = int(ref, 16) - 1
219         ref = o.segs[ref] if o.syms[ref].type == 'S' else o.syms[ref]
220         addend = int(l[4], 16) if len(l) >= 5 else 0
221         o.rels.append(Relocation(
222                 loc = int(loc, 16),
223                 seg = o.segs[int(seg, 16) - 1],
224                 ref = ref,
225                 type = Relocation.type_name[type],
226                 addend = addend))
227
228     # data
229     for s in o.segs:
230         if 'P' in s.flags and s.size > 0:
231             s.data = getl(f)
232             s.data != None and len(s.data) == 2 * s.size \
233                 or die('Data for %s has wrong size' % s.name)
234     return o
```

B.2 Povezovalnik

Izpis B.3: `ld.py`

```
1  #! /usr/bin/python
2
3  import obj, sys, getopt
4  from common import *
5
6  syms = {}
7  so_syms = {}
8  segs = {
9      '.text': obj.Segment(name='.text'),
10     '.got':  obj.Segment(name='.got'),
11     '.data': obj.Segment(name='.data'),
12     '.bss':  obj.Segment(name='.bss')
13 }
14 rels = []
15 needs = []
16 got = {}
17 BASE = 0x08048000
18 create_got = False
19 shared = False
20
21 class Symbol:
22     '''represent a Symbol in the linker'''
23     def __init__(self, sym, refs):
24         self.sym = sym
25         self.refs = refs
26
27 def insert(objs):
28     '''insert object or library'''
29     global syms, so_syms, needs
30
31     arobjs = []
32     needs = [o.name for o in objs if o.__class__ == obj.Object and o.shared]
33     for o in objs:
34         if o.__class__ == obj.Object:
35             insertobj(o, so_syms if o.shared else syms)
36             if o.shared:
37                 for n in o.needs:
38                     insertobj(obj.read(n), so_syms)
39         elif o.__class__ == obj.Ar:
40             loop = True
41             while loop:
42                 loop = False
43                 for (s, offset) in o.syms.items():
```

```
44                      if s in syms and syms[s].sym.type == 'U':
45                          arobj = read(o.name, offset)
46                          arobjs.append(arobj)
47                          insertobj(arobj)
48                          loop = True
49      objs = [o for o in objs if o.__class__ == obj.Object and not o.shared]
50      objs.extend(arobjs)
51
52      return objs
53
54  def insertobj(o, tbl):
55      '''insert object'''
56      for s in o.syms:
57          if s.type == 'L':
58              continue
59          if s.name not in tbl:
60              s.linksym = tbl[s.name] = Symbol(s, [o])
61              continue
62          sym = tbl[s.name]
63          sym.refs.append(o)
64          s.linksym = sym
65          if s.type == 'D':
66              sym.sym.type == 'U' or die('%s: multiple definitions of %s' %
67                                         (sym.sym.name, s.name))
68              sym.sym = s
69
70  def init_got(objs):
71      '''initate the GOT table'''
72      global got, create_got
73
74      for o in objs:
75          for r in o.rels:
76              if r.type == obj.Relocation.GP2:
77                  if r.ref not in got:
78                      create_got = True
79                      key = r.ref if r.ref.type == 'L' else syms[r.ref.name].sym
80                      got[key] = len(got) * 4
81      segs['.got'].size = len(got) * 4
82
83
84  def allocate(objs):
85      '''allocate space for segments'''
86      for o in objs:
87          for s in o.segs:
88              assert s.base == 0
89              s.base = segs[s.name].size
90              padded = align(s.size, 4)
```

```
91              segs[s.name].size += padded
92              if 'P' in s.flags:
93                  s.data += '00' * (padded - s.size)
94
95      segs['.text'].base = 0 if shared else BASE
96      segs['.got'].base  = align(segs['.text'].base + segs['.text'].size, 2**12)
97      segs['.data'].base = segs['.got'].base  + segs['.got'].size
98      segs['.bss'].base  = segs['.data'].base + segs['.data'].size
99
100     for o in objs:
101         for s in o.segs:
102             s.base += segs[s.name].base
103
104
105 def check_symbols(objs):
106     '''check for undefined and correct offset'''
107     for o in objs:
108         for s in o.syms:
109             s.value += s.seg.base
110             s.seg = segs[s.seg.name]
111
112     if create_got:
113         syms['_GOT'].sym.value = segs['.got'].base
114         syms['_GOT'].sym.type = 'L'
115
116     # define _GP
117     if '_GP' in syms:
118         syms['_GP'].sym.value = segs['.data'].base + 2**15
119         syms['_GP'].sym.type = 'L'
120
121     msg = 'undefined reference to symbol %s'
122     def check(tbl1, tbl2):
123         for s in tbl1.values():
124             s.sym.type != 'U' or \
125                 (s.sym.name in tbl2 and tbl2[s.sym.name].sym.type == 'D') \
126                 or die(msg % s.sym.name)
127     if not shared:
128         check(syms, so_syms)
129         check(so_syms, syms)
130
131 def build_got(objs):
132     '''initial values of all the GOT tabel entries'''
133     for (var, off) in sorted(got.items(), key = lambda i: i[1]):
134         segs['.got'].data += '%08X' % var.value
135
136 def relocation(objs):
137     '''process relocation entries - the hart of the linking process'''
```

```
138     for o in objs:
139         for r in o.rels:
140             if r.type.size == 4:
141                 off = 0
142                 fmt = '%08X'
143             else:
144                 off = 4
145                 fmt = '%04X'
146 
147             not shared or 'W' in r.seg.flags or \
148                 r.type.name in ['R2','GA2','GP2','GR2'] or \
149                 die('relocation %s not allowed in pic' % r.type.name)
150 
151             # interpret reference
152             if r.ref.__class__ == obj.Segment:
153                 ref = r.ref.base
154             elif r.ref.linksym == None:
155                 ref = r.ref.value
156             else:
157                 ref = r.ref.linksym.sym.value
158 
159             loc, size = r.loc * 2, r.type.size * 2
160             data      = r.seg.data
161             val       = int(data[loc+off : loc+size+off], 16)
162 
163             if shared:
164                 if r.type == obj.Relocation.A4:
165                     rels.append(obj.Relocation(
166                             loc = r.loc + r.seg.base - segs[r.seg.name].base,
167                             seg = segs[r.seg.name],
168                             ref = None,  # ignored
169                             type = obj.Relocation.ER4))
170 
171             if r.type == obj.Relocation.A4:
172                 val += ref
173             elif r.type == obj.Relocation.A2:
174                 val += ref
175                 -32768 <= val < 32768 or die('address out of range')
176             elif r.type in [obj.Relocation.R2, obj.Relocation.GA2]:
177                 val = pack_int(ref - r.seg.base - unpack_int(val, 16), 16)
178             elif r.type == obj.Relocation.L2:
179                 val = lo(val + ref)
180             elif r.type == obj.Relocation.U2:
181                 val = hi((val << 16) + r.addend + ref)
182             elif r.type == obj.Relocation.GP2:
183                 key = r.ref if r.ref.type == 'L' else syms[r.ref.name].sym
184                 val = got[key]
```

```
185             elif r.type == obj.Relocation.GR2:
186                 r.ref.type == 'L' or die('variable %s is not local'%r.ref.name)
187                 val = ref - segs['.got'].base
188             elif r.type == obj.Relocation.DR2:
189                 val = pack_int(ref - syms['_GP'].sym.value, 16)
190             else:
191                 die('unknown relocation %s' % r.type.name)
192             r.seg.data = data[:loc+off] + (fmt % val) + data[loc+size+off:]
193 
194     # got
195     for (var, off) in got.items():
196         er4 = var.type != 'U'
197         rels.append(obj.Relocation(
198                 loc = off,
199                 seg = segs['.got'],
200                 ref = None if er4 else var,
201                 type = obj.Relocation.ER4 if er4 else obj.Relocation.A4))
202 
203 
204 def paste_code(objs):
205     '''concatenate the code from all objs'''
206     for o in objs:
207         for s in o.segs:
208             if s.name in segs:
209                 segs[s.name].data += s.data
210 
211 def link(objs, ofile):
212     '''main program'''
213     objs = insert(objs)
214     init_got(objs)
215     allocate(objs)
216     check_symbols(objs)
217     if create_got:
218         build_got(objs)
219     relocation(objs)
220     paste_code(objs)
221 
222     # write output
223     o = obj.Object(name=ofile.name, shared=shared)
224     o.segs = segs.values()
225     o.syms = [s.sym for s in syms.values()]
226     o.rels = rels
227     o.needs = needs
228     o.write(ofile)
229 
230 if __name__ == '__main__':
231     try:
```

```
232         opts, args = getopt.getopt(sys.argv[1:], 'o:', ['shared'])
233     except getopt.GetoptError, err:
234         die(err)
235     len(args) >= 1 or die('provides a file to link')
236     ofname = 'a.out'
237     for o, a in opts:
238         if o == '-o':
239             ofname = a
240         elif o == '--shared':
241             shared = True
242     ofile = open(ofname, 'w')
243     link(map(obj.read, args), ofile)
244     ofile.close()
```

B.3 Dinamični nalagalnik

Izpis B.4: `ld_so.py`

```
1  import obj
2  from common import align, die
3
4  # Class Hip is defined in the file sim.y. Two important members are:
5  # - mem: dictionary representing the processors main memory
6  # - brk: end of the read/write segment
7  # - r: a list with 32 entries corresponding to the 32 registers of HIP
8
9  syms = {}
10
11 def loadobj(hip, addr, o):
12     '''load object o at adress addr'''
13     global syms
14
15     o.base = addr
16     hip.brk = align(max(addr + s.base + s.size for s in o.segs), 2**12)
17     # load into memory
18     for s in o.segs:
19         if 'P' in s.flags:
20             vals = (int(s.data[i:i+2],16) for i in xrange(0, s.size*2, 2))
21         else:
22             vals = (0 for _ in xrange(s.size))
23         hip.mem.update((addr + s.base + i, v) for (i,v) in enumerate(vals))
24         s.base += addr
25
26     # collect symbols
27     if o.shared:
28         for s in o.syms:
29             s.value += s.seg.base
30     syms.update((s.name, s) for s in o.syms if s.type == 'D')
31
32 def relocate(hip, objs):
33     '''load time relocation'''
34     for o in objs:
35         for r in o.rels:
36             'W' in r.seg.flags or die('can not touch read only segment')
37             data = r.seg.data
38             if r.type == obj.Relocation.A4:
39                 val = syms[r.ref.name].value
40             elif r.type == obj.Relocation.ER4:
41                 val = hip.fetch(r.seg.base + r.loc, 4) + o.base
42             else:
43                 die('unknown load-time relocation %s' % r.type.name)
```

```
44              hip.store(r.seg.base + r.loc, val, 4)
45
46  def load(hip, o):
47      '''Load object o and all the shared libraries that it depends on,
48      handle load time relocation and initiate the registers.'''
49      q = [o]
50      loaded = {}
51      while len(q) > 0:
52          curr = q.pop()
53          loadobj(hip, hip.brk, curr)
54          loaded[curr.name] = curr
55          q.extend(obj.read(n) for n in curr.needs)
56      relocate(hip, loaded.values())
57      start = syms['_start']
58      hip.r[25] = hip.pc = start.value
```

Literatura

[1] Aleph One, *Smashing The Stack For Fun And Profit*, Dostopno na: `http://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf`

[2] U. Drepper, *How To Write Shared Libraries*, Dostopno na: `http://people.redhat.com/drepper/dsohowto.pdf`

[3] D. Kodek, *Arhitektura in organizacija računalniških sistemov*, Šenčur: Bitim, 2008.

[4] J. R. Levine, *Linkers and loaders*, San Francisco: Morgan Kaufmann, 2000.

[5] D. A. Patterson, J. L. Hennessy, *Computer organization and design: the hardware/software interface*, Amsterdam: Elsevier/Morgan Kaufmann, 2007, dodatek A.

[6] A. Srivastava, D. Wall, *A Practical System for Intermodule Code Optimization at Link Time*, Dostopno na: `www.hpl.hp.com/techreports/Compaq-DEC/WRL-92-6.pdf`

[7] D. Sweetman, *See MIPS run*, San Francisco: Morgan Kaufmann, 1999.

[8] *System V Application Binary Interface*, Prentice Hall, 3rd edition, December 1993.

[9] *Nabor ukazov procesorja HIP*, Dostopno na: `http://laps.fri.uni-lj.si/ars/ars_files/hip_IS.pdf`

[10] *Stran man:* `a.out(1)`, `http://www.openbsd.org/cgi-bin/man.cgi?query=a.out`

[11] *Stran man:* `elf(1)`,
`http://www.openbsd.org/cgi-bin/man.cgi?query=elf`

[12] *Stran man:* `tsort(1)`,
`http://www.openbsd.org/cgi-bin/man.cgi?query=tsort`

[13] *Stran man:* `lorder(1)`,
`http://www.openbsd.org/cgi-bin/man.cgi?query=lorder`

Izjava

Izjavljam, da sem diplomsko nalogo izdelal samostojno pod vodstvom mentorja doc. dr. Boštjana Slivnika. Izkazano pomoč drugih sodelavcev sem v celoti navedel v zahvali.

Ljubljana, september 2008 Jure Žbontar