Arhitektura računala 2

prevođenje i pokretanje programa

Siniša Šegvić

Slobodan Ribarić

Zavod za elektroniku, mikroelektroniku,

računalne i inteligentne sustave

Fakultet elektrotehnike i računarstva

Sveučilište u Zagrebu

Uvod

Što se zbiva s programom u višem jeziku prije pokretanja?

Uvod

Što se zbiva s programom u višem jeziku prije pokretanja?

Prevoditelj prevodi pojedine komponente iz višeg jezika u zbirni jezik.

Uvod

Što se zbiva s programom u višem jeziku prije pokretanja?

Prevoditelj prevodi pojedine komponente iz višeg jezika u zbirni jezik.

Asembler prevodi svaku komponentu iz zbirnog jezika u takozvani premjestivi (relocatable) **objektni** modul.

Uvod

Što se zbiva s programom u višem jeziku prije pokretanja?

Prevoditelj prevodi pojedine komponente iz višeg jezika u zbirni jezik.

Asembler prevodi svaku komponentu iz zbirnog jezika u takozvani premjestivi (relocatable) **objektni** modul.

Linker (povezivač) povezuje korisničke objektne datoteke s bibliotekama i standardnim okruženjem u **izvršnu** datoteku:

Uvod

Što se zbiva s programom u višem jeziku prije pokretanja?

Prevoditelj prevodi pojedine komponente iz višeg jezika u zbirni jezik.

Asembler prevodi svaku komponentu iz zbirnog jezika u takozvani premjestivi (relocatable) **objektni** modul.

Linker (povezivač) povezuje korisničke objektne datoteke s bibliotekama i standardnim okruženjem u **izvršnu** datoteku:

Loader (punjač): učitava sadržaj izvršne datoteke u memoriju, te pokreće program što rezultira pozivanjem funkcije main()

Uvod

Što se zbiva s programom u višem jeziku prije pokretanja?

Prevoditelj prevodi pojedine komponente iz višeg jezika u zbirni jezik.

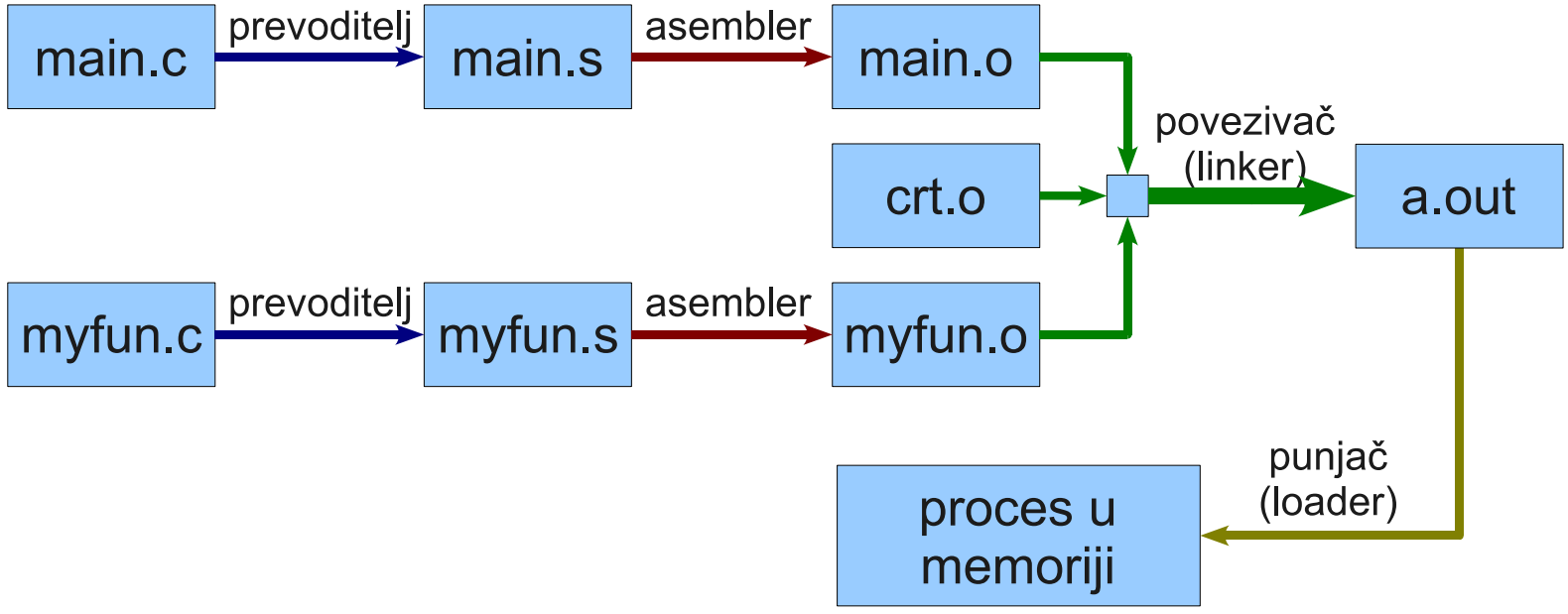
Asembler prevodi svaku komponentu iz zbirnog jezika u takozvani premjestivi (relocatable) **objektni** modul.

Linker (povezivač) povezuje korisničke objektne datoteke s bibliotekama i standardnim okruženjem u **izvršnu** datoteku:

Loader (punjač): učitava sadržaj izvršne datoteke u memoriju, te pokreće program što rezultira pozivanjem funkcije main()

To nije sve: dinamičke biblioteke? Java?

UVOD: SHEMA



- na prevođenje utječe jezik, binarni standard (ABI), te ISA
- na asembliranje utječe ISA, te standard za objektne module
- na povezivanje utječu standardi za objektne i izvršne module
- na punjenje utječe standard za izvršne module i operacijski sustav

OBJEKTI: SVOJSTVA

Objektni modul je datoteka s izvršnim kodom koja je prikladna za **umetanje** u veći programski sustav.

- svaka komponenta višeg programskog jezika prevodi se u **zasebni** objektni modul (ušteda pri prevođenju velikih programa)
- kôd (i podatci) u svakoj komponenti počinju od **adrese 0**
- reference na podatke i potprograme iz drugih modula **nedefinirane**

OBJEKTI: SVOJSTVA

Objektni modul je datoteka s izvršnim kodom koja je prikladna za **umetanje** u veći programski sustav.

- svaka komponenta višeg programskog jezika prevodi se u **zasebni** objektni modul (ušteda pri prevođenju velikih programa)
- kôd (i podatci) u svakoj komponenti počinju od **adrese 0**
- reference na podatke i potprograme iz drugih modula **nedefinirane**

Objektni modul proizvodi asembler ili izravno prevoditelj:

- u načelu objektni moduli **ne ovise** o prevoditelju, asembleru i jeziku!
- standardi: ELF (UNIX), COFF (win32)
- ekstenzije: .o (UNIX), .obj (win32)

OBJEKTI: ELEMENTI

Tipično, objektni modul sadrži sljedeće elemente:

- **zaglavlje**: sadržaj datoteke u standardnom formatu
- **programsku sekciju** (text): prevedeni binarni program
- **podatkovnu sekciju** (data): statičke varijable (imaju konstantnu adresu tijekom života programa)
- **relokacijske informacije**: što sve treba promijeniti prilikom premještanja sekcija?
- tablica **simbola**: adrese stat. varijabli, funkcija, vanjske reference
- informacije za ispitivanje programa iz debuggera (ELF: DWARF)

Sadržaj objektnih modula možemo promotriti naredbama **objdump** (Linux) i **objconv** (Windows).

Na Linuxu mogu biti korisni i alati nm i readelf.

OBJEKTI: PRIMJER

```
### ovo je main.c
int myfun(int a, int c);
int x=42, y=1234;

int main(){
  return myfun(1,2)*x+y;
}

### prevedi main.c u zbirni jezik ($ označava unos naredbe):
$ gcc -c -masm=intel -S -mpreferred-stack-boundary=2 main.c

### napravi objektni modul
$ as main.s -o main.o

### u kojem je formatu datoteka main.o?
$ file main.o
main.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped

### pogledajmo tablicu simbola u main.o:
$ objdump --syms main.o | grep  -v '^.\{14\}d'
...
SYMBOL TABLE:
00000000 g     O .data  00000004 x
00000004 g     O .data  00000004 y
00000000 g     F .text  0000002e main
00000000         *UND*  00000000 myfun
```

OBJEKTI: PRIMJER (2)

Usporedimo originalni main.s (lijevo) s disasembliranim main.o (desno):

```
$ cat main.s
main:
        push    ebp
        mov     ebp, esp
        sub     esp, 8
        mov     DWORD PTR [esp+4], 2
        mov     DWORD PTR [esp], 1
        call    myfun
        mov     edx, eax
        mov     eax, DWORD PTR x
        imul    edx, eax
        mov     eax, DWORD PTR y
        lea     eax, [edx+eax]
        leave
        ret
```

```
$ objdump --disassembler-options=intel -d -j .text main.o
00000000 <main>:
   0:   55                      push   ebp
   1:   89 e5                   mov    ebp,esp
   3:   83 ec 08                sub    esp,0x8
   6:   c7 44 24 04 02 00 00 00 mov    DWORD PTR [esp+0x4],0x2
   e:   c7 04 24 01 00 00 00    mov    DWORD PTR [esp],0x1
  15:   e8 fc ff ff ff          call   16 <main+0x16>
  1a:   89 c2                   mov    edx,eax
  1c:   a1 00 00 00 00          mov    eax,ds:0x0
  21:   0f af d0                imul   edx,eax
  24:   a1 00 00 00 00          mov    eax,ds:0x0
  29:   8d 04 02                lea    eax,[edx+eax*1]
  2c:   c9                      leave
  2d:   c3                      ret
```

OBJEKTI: PRIMJER (2)

Usporedimo originalni main.s (lijevo) s disasembliranim main.o (desno):

```
$ cat main.s
main:
        push    ebp
        mov     ebp, esp
        sub     esp, 8
        mov     DWORD PTR [esp+4], 2
        mov     DWORD PTR [esp], 1
        call    myfun
        mov     edx, eax
        mov     eax, DWORD PTR x
        imul    edx, eax
        mov     eax, DWORD PTR y
        lea     eax, [edx+eax]
        leave
        ret
```

```
$ objdump --disassembler-options=intel -d -j .text main.o
00000000 <main>:
   0:   55                      push   ebp
   1:   89 e5                   mov    ebp,esp
   3:   83 ec 08                sub    esp,0x8
   6:   c7 44 24 04 02 00 00 00 mov    DWORD PTR [esp+0x4],0x2
   e:   c7 04 24 01 00 00 00    mov    DWORD PTR [esp],0x1
  15:   e8 fc ff ff ff          call   16 <main+0x16>
  1a:   89 c2                   mov    edx,eax
  1c:   a1 00 00 00 00          mov    eax,ds:0x0
  21:   0f af d0                imul   edx,eax
  24:   a1 00 00 00 00          mov    eax,ds:0x0
  29:   8d 04 02                lea    eax,[edx+eax*1]
  2c:   c9                      leave
  2d:   c3                      ret
```

```
### pogledajmo relokacijske informacije u main.o:
$ objdump --reloc main.o
...
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000016 R_386_PC32        myfun
0000001d R_386_32          x
00000025 R_386_32          y
```

OBJEKTI: PRIMJER (3)

Pogledajmo C, simbole, te disasemblirani kôd s relokacijama za myfun.o:

```
$ cat myfun.c

int limit=1;


int myfun(
  int x, int y)
{
  static int offset=8;
  int rv=offset;
  rv+= 2*x+3*y;

  if (rv<limit){
    rv=limit;
  }

  return rv;
}
```

```
$ objdump --disassembler-options=intel -r -d myfun.o

... ecx=2*'x', eax=3*'y'
15:     8d 14 01        lea    edx,[ecx+eax*1]
18:     a1 04 00 00 00  mov    eax,ds:0x4
                    19: R_386_32        .data
1d:     8d 04 02        lea    eax,[edx+eax*1]
20:     89 45 fc        mov    DWORD PTR [ebp-0x4],eax
23:     a1 00 00 00 00  mov    eax,ds:0x0
                    24: R_386_32    limit
28:     39 45 fc        cmp    DWORD PTR [ebp-0x4],eax
2b:     7d 08           jge    35 <myfun+0x35>
2d:     a1 00 00 00 00  mov    eax,ds:0x0
                    2e: R_386_32    limit
32:     89 45 fc        mov    DWORD PTR [ebp-0x4],eax
35:     8b 45 fc        mov    eax,DWORD PTR [ebp-0x4]
38:     c9              leave
39:     c3              ret
```

```
$ objdump  --syms myfun.o | grep -v '^.\{14\}d'

...
SYMBOL TABLE:
00000004 l       O .data  00000004 offset
00000000 g       O .data  00000004 limit
00000000 g       F .text  0000003a myfun
```

OBJEKTI: ZAKLJUČAK

Vidjeli smo da ABI (C, Linux, x86) predviđa premještanje (relociranje):

- pristupa statičkim podatcima
- poziva funkcija izvan objektnog modula

Grananje unutar modula je **relativno** pa ne zahtijeva premještanje.

Premještanje modula obavlja **linker**.

LINKER: ZADATCI

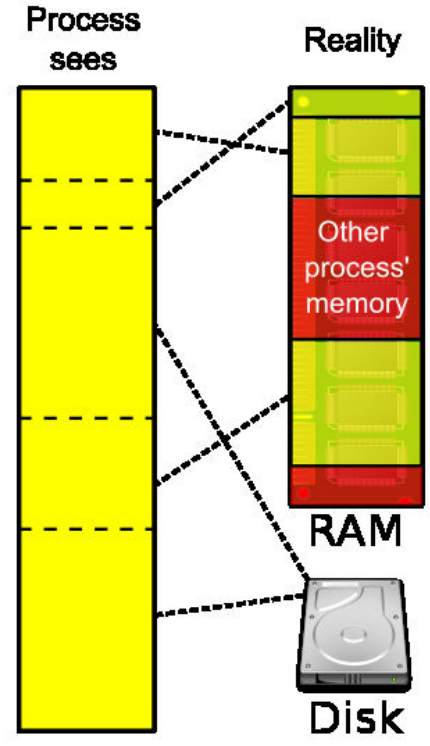
Krajnji cilj — (pokušati) generirati izvršnu datoteku:

- u igri su korisnički objektni moduli, statičke i dinamičke biblioteke, te izvršno okruženje (CRT, common runtime environment)
- **sjediniti** odgovarajuće sekcije svih modula u zajedničke super-sekcije (.text, .data, .rodata, .bss, ...)
- **uskladiti** konačne adrese statičkih podataka s konačnim položajem u izvršnoj datoteci (linker — link editor)
- **razriješiti** međuovisnosti (nedefinirane reference) sada poznatim adresama elemenata drugih modula
- objektni moduli dobiveni iz različitih jezika mogu se kombinirati, ali samo ako implementiraju isti ABI...

LINKER: DETALJI

Današnji linkeri tipično proizvode izvršni program koji **nije premjestiv** (prije izvršavanja ga treba postaviti na **fiksnu** adresu u memoriji)

- ovo može izgledati problematično, jer ne znamo koji dio fizičkog memorijskog prostora će biti slobodan u trenutku poziva!
- problema ipak nema jer virtualna memorija **preslikava** stranice adresnog prostora procesa u proizvoljne fizičke stranice
- u načelu, relokacija se može odgoditi do trenutka učitavanja, ali danas za to nema potrebe (na računalima opće namjene)



[Wikipedia]

LINKER: PRIMJER

```
### poveži main.o i myfun.o u izvršnu datoteku a.out
$ gcc main.o myfun.o

### pogledajmo tablicu simbola u a.out:
objdump --syms a.out | grep  -v '^.\{14\}d'
...
0804a010 g     O .data  00000004              x
0804a014 g     O .data  00000004              y
08048394 g     F .text  00000040              main
080483d4 g     F .text  0000003a              myfun
...

### relokacijska tablica sada je prazna ...
$ objdump --reloc a.out

### disasemblirani kod ($80483b9 + $1b = $080483d4):
objdump --disassembler-options=intel -d -j .text main.o

08048394 <main>:
...
80483a5:        c7 44 24 04 02 00 00 00 mov    DWORD PTR [esp+0x4],0x2
80483ad:        c7 04 24 01 00 00 00    mov    DWORD PTR [esp],0x1
80483b4:        e8 1b 00 00 00          call   80483d4 <myfun>
80483b9:        89 c2                   mov    edx,eax
80483bb:        a1 10 a0 04 08          mov    eax,ds:0x804a010
80483c0:        0f af d0                imul   edx,eax
80483c3:        a1 14 a0 04 08          mov    eax,ds:0x804a014
80483c8:        8d 04 02                lea    eax,[edx+eax*1]
...
```

LINKER: BIBLIOTEKE

Osim izvršnog programa, linker može napraviti i **biblioteku**

Biblioteka je kolekcija premjestivih objektnih modula:

- koriste se pri povezivanju drugih objektnih ili izvršnih modula
- statičke biblioteke koristi linker, o dinamičkima ćemo pričati kasnije

LINKER: BIBLIOTEKE

Osim izvršnog programa, linker može napraviti i **biblioteku**

Biblioteka je kolekcija premjestivih objektnih modula:

- koriste se pri povezivanju drugih objektnih ili izvršnih modula
- statičke biblioteke koristi linker, o dinamičkima ćemo pričati kasnije

Npr, o kojim dinamičkim bibliotekama ovisi naš `a.out`?

```
ldd a.out
      linux-gate.so.1 =>  (0xb7fe8000)
      libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e78000)
      /lib/ld-linux.so.2 (0xb7fe9000)
```

LINKER: BIBLIOTEKE

Osim izvršnog programa, linker može napraviti i **biblioteku**

Biblioteka je kolekcija premjestivih objektnih modula:

- koriste se pri povezivanju drugih objektnih ili izvršnih modula
- statičke biblioteke koristi linker, o dinamičkima ćemo pričati kasnije

Npr, o kojim dinamičkim bibliotekama ovisi naš `a.out`?

```
ldd a.out
      linux-gate.so.1 =>  (0xb7fe8000)
      libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e78000)
      /lib/ld-linux.so.2 (0xb7fe9000)
```

Npr, kako bismo provjerili gdje je definiran `fopen`?

```
$ objdump /usr/lib/libc.a --syms | grep -e 'F .text.* fopen$' -B 25

iofopen.o:     file format elf32-i386

SYMBOL TABLE:
...
00000140  w    F .text  00000022 fopen
```

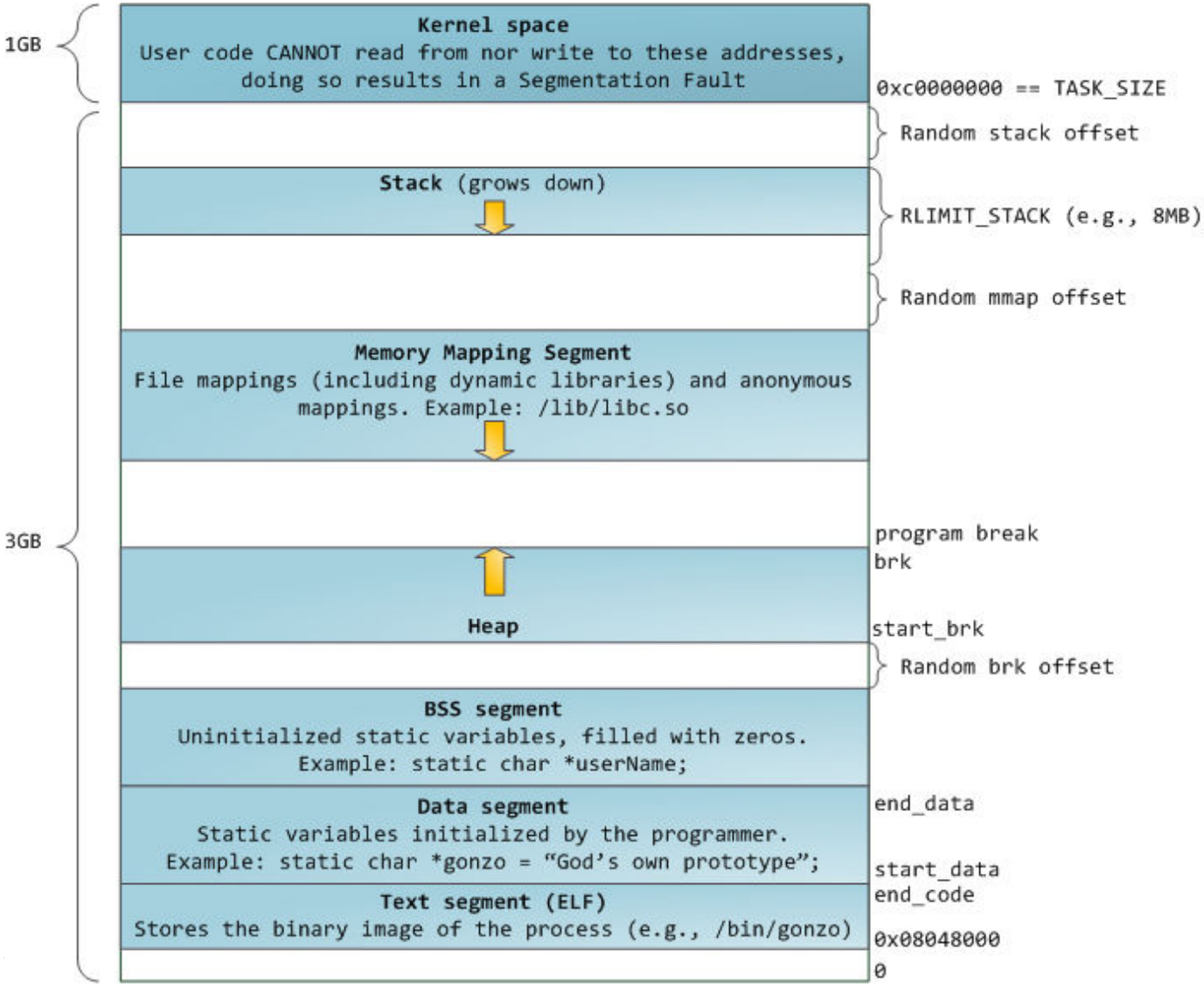
LOADER: AKCIJE

Cilj — pripremiti izvršnu datoteku za izvođenje:

- pročitati zaglavlje izvršne datoteke, odrediti veličine memorijskih segmenata
- kreirati virtualni adresni prostor procesa (alocirati stavke u tablici preslikavanja, to korisnički programi ne mogu raditi)
- kopirati segmente izvršne datoteke u memoriju
  (ili postaviti elemente tablice preslikavanja da se učitaju na zahtjev)
- inicijalizirati registre ESP, EBP
- postaviti argumente naredbenog retka na stog
- skočiti na inicijalizacijski potprogram
  - gcc: poziva se _start() koji priprema argv, argc te poziva main()
  - `http://linuxgazette.net/issue84/hawk.html`

LOADER: REZULTAT

Standardni raspored memorije procesa na x86 pod Linuxom:



http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory

DLL,SO: UVOD

Dinamičke biblioteke se povezuju s glavnim programom tijekom **izvođenja** programa (engl. *at run time*)

Prednosti dinamičkog povezivanja su:

- takva biblioteka se može održavati **neovisno** o glavnom programu (pod uvjetom očuvanja binarne kompatibilnosti; **ušteda** diska)
- ako je biblioteka PIC (Position Independent Code), virtualna memorija istu instancu biblioteke može mapirati u memorijski prostor **različitih** procesa (**ušteda** memorije, **brže** pokretanje)

DLL,SO: UVOD

Dinamičke biblioteke se povezuju s glavnim programom tijekom **izvođenja** programa (engl. *at run time*)

Prednosti dinamičkog povezivanja su:

- takva biblioteka se može održavati **neovisno** o glavnom programu (pod uvjetom očuvanja binarne kompatibilnosti; **ušteda** diska)
- ako je biblioteka PIC (Position Independent Code), virtualna memorija istu instancu biblioteke može mapirati u memorijski prostor **različitih** procesa (**ušteda** memorije, **brže** pokretanje)

S obzirom na to da radimo s izvršnom datotekom, dinamičku biblioteku nije moguće integrirati u glavni program kao statičku biblioteku

DLL,SO: UVOD

Dinamičke biblioteke se povezuju s glavnim programom tijekom **izvođenja** programa (engl. *at run time*)

Prednosti dinamičkog povezivanja su:

- takva biblioteka se može održavati **neovisno** o glavnom programu (pod uvjetom očuvanja binarne kompatibilnosti; **ušteda** diska)
- ako je biblioteka PIC (Position Independent Code), virtualna memorija istu instancu biblioteke može mapirati u memorijski prostor **različitih** procesa (**ušteda** memorije, **brže** pokretanje)

S obzirom na to da radimo s izvršnom datotekom, dinamičku biblioteku nije moguće integrirati u glavni program kao statičku biblioteku

Dinamičke biblioteke se obično učitavaju u posebni dio memorije (pogledati sliku na prethodnoj stranici)

DLL,SO: DETALJI

U načelu ne znamo gdje će se biblioteka učitati:

- redosljed učitavanja biblioteka ne mora biti isti
- binarni raspored (*layout*) biblioteke može se promijeniti između prevođenja i pokretanja
- $\Rightarrow$ dinamička biblioteka treba biti ili **premjestivi** ili **PIC** modul!

Učitavanje biblioteke rješava dinamički linker/loader
(u uskoj vezi s operacijskim sustavom):

- po potrebi, prilagodba dinamičke biblioteke zadanoj logičkoj adresi (popunjavanje relokacijskih podataka $\Rightarrow$ moramo kopirati biblioteku)
- razriješavanje međuovisnosti među bibliotekama
- zauzimanje virtualne memorije, fizičko učitavanje biblioteke

DLL,SO: IZVEDBE

DLL-ovi (win32) imaju fiksnu početnu adresu kao i izvršni programi:

- ako je adresa DLL-a u trenutku učitavanja zauzeta DLL se mora premjestiti $\Rightarrow$ moramo napraviti kopiju DLL-a u memoriji
- da se to ne bi događalo često, Microsoft sistemskim dll-ovima početne adrese dodjeljuje ručno!
- `http://en.wikipedia.org/wiki/Position-independent_code#Windows_DLLs`

DLL,SO: IZVEDBE

DLL-ovi (win32) imaju fiksnu početnu adresu kao i izvršni programi:

- ako je adresa DLL-a u trenutku učitavanja zauzeta DLL se mora premjestiti $\Rightarrow$ moramo napraviti kopiju DLL-a u memoriji
- da se to ne bi događalo često, Microsoft sistemskim dll-ovima početne adrese dodjeljuje ručno!
- `http://en.wikipedia.org/wiki/Position-independent_code#Windows_DLLs`

ELF format podržava PIC (position independent code):

- statičkim podatcima se pristupa ili relativno na PC (x86-64) ili indirektno (tablica GOT)
- biblioteku je potrebno prevesti s -fPIC

DLL,SO: PRISTUP

Danas dinamičkim bibliotekama tipično pristupamo na **dva** načina:

1. eksplicitno učitavanje biblioteke i pristupanje članskim funkcijama
   - UNIX: dlopen, dlsym
   - Win32: LoadLibrary, GetProcAddress
   - zgodno za dodatke (plug-in), nezgodno za standardne biblioteke
2. implicitno i zakašnjelo učitavanje (slika je na sljedećoj stranici)
   - ciljna funkcija iz biblioteke se učitava tek nakon prvog poziva!
   - **indirektan poziv** preko skretnice (stub) i tablice vektora **TV**
   - početno, vektori pokazuju na f.ju za **dinamičko punjenje** koja:
     (a) **učita** i po potrebi relocira ciljnu funkciju
     (b) **preusmjeri** odgovarajući vektor iz TV na ciljnu funkciju
     (c) **skoči** na ciljnu funkciju
   - skretnice i f.ju za dinamičko učitavanje priprema **linker**!

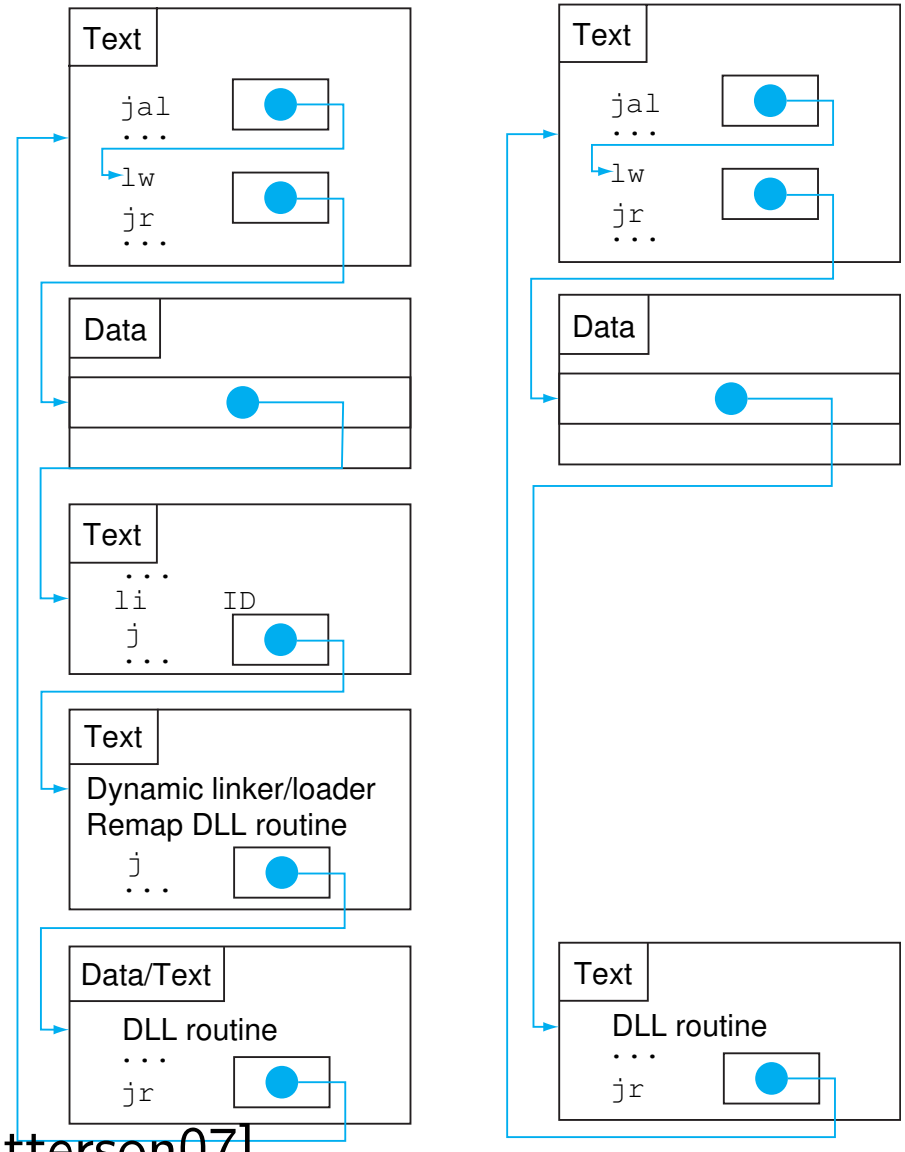
DLL,SO: IMPLICITNI POZIV

Implicitno učitavanje biblioteke zbiva se pri prvom pozivu neke od pripadajućih funkcija (lazy loading)!

- cijena kasnijih poziva: samo jedan dodatni pristup TV!

Elementi na slici, odozgo dolje:

1. glavni program (`jal`) i skretnica koja proziva TV (`lw`, `jr`)
2. tablica vektora (ELF:PLT)
3. priprema argumenta ID koji definira ciljnu funkciju
4. f.ja za dinamičko učitavanje
5. ciljna funkcija učitana iz dll-a [Patterson07]



DLL,SO: PRAKSA

Pozivanje funkcija iz biblioteke može se zakomplicirati, posebno ako:

- zaboravimo ažurirati prototip sučeljne funkcije
- nismo uskladili konvenciju prenošenja parametara
- ne uzmemo u obzir da C++ mora dekorirati imena funkcija
- sučeljne funkcije nismo kao takve definirali: `__declspec(dllexport)`, datoteka .def na Windowsima

DLL,SO: PRAKSA

Pozivanje funkcija iz biblioteke može se zakomplicirati, posebno ako:

- zaboravimo ažurirati prototip sučeljne funkcije
- nismo uskladili konvenciju prenošenja parametara
- ne uzmemo u obzir da C++ mora dekorirati imena funkcija
- sučeljne funkcije nismo kao takve definirali: `__declspec(dllexport)`, datoteka .def na Windowsima

Opisane probleme teško dijagnosticirati zbog binarne prirode biblioteka!

DLL,SO: PRAKSA

Pozivanje funkcija iz biblioteke može se zakomplicirati, posebno ako:

- zaboravimo ažurirati prototip sučeljne funkcije
- nismo uskladili konvenciju prenošenja parametara
- ne uzmemo u obzir da C++ mora dekorirati imena funkcija
- sučeljne funkcije nismo kao takve definirali: `__declspec(dllexport)`, datoteka .def na Windowsima

Opisane probleme teško dijagnosticirati zbog binarne prirode biblioteka!

Pri naguravanju sa sučeljnim funkcijama biblioteka mogu nam pomoći:

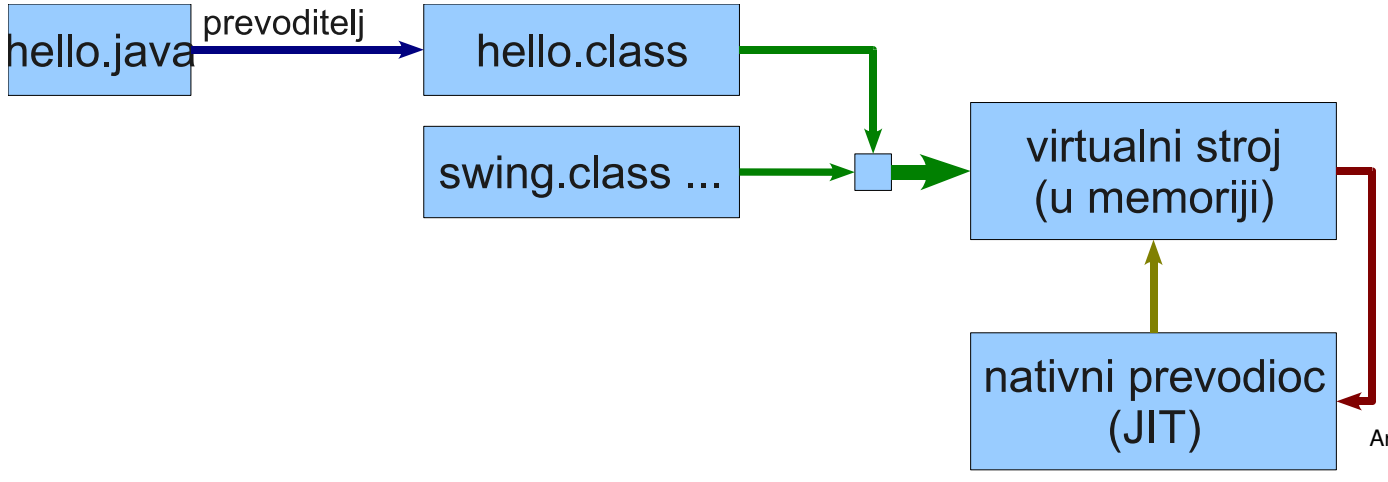
- Windows: Dependency walker (depends.exe), texe
- Linux: objdump, readelf, nm, ldd

JAVA: UVOD

Prethodna priča odnosi se na "prevođene" jezike, čiji se izvorni kod prevodi u izvršni kod kojeg razumije OS

Druga klasa su interpretirani jezici: Java, Perl, Python, C#...

- izvorni kôd se prevodi u takozvani bajtni međukod (*byte code*)
- bajtni kôd izražen u jednostavnom i prenosivom "strojnom" jeziku
- virtualni stroj (*virtual machine*) učitava, izvodi i profilira međukod
- za postizanje bolje performanse, često pozivane metode dinamički se prevode u nativni kôd ("materinji" jezik računala domaćina)



JAVA: PREDNOSTI

Interpretirani jezici lakši za korištenje od prevedenih:

- lakše eksperimentiranje (npr, interaktivan rad u Pythonu)
- lakša prenosivost (zbog virtualnog stroja)
- veća robustnost na pogreške (testiranje granica, recikliranje memorije)
- veći izbor biblioteka (GUI, regex, TCP/IP, ...)

Prevođenje u nativni kod (JIT) omogućava relativno dobru performansu (ako imamo vremena čekati rezultate profiliranja)

JIT za Javu postoji već nekoliko godina, a za Python je blizu realizacije

JAVA: NEDOSTATCI

Ipak, interpretirani jezici nisu rješenje svih problema:

- pokazuje se da je razvoj programa intrinsično težak
- dominantan faktor težine razvoja programa prema mnogima su inherentna složenost, te nedovoljno poznati i promjenljivi zahtjevi (Frederic Brooks: "*no silver bullet*")

JAVA: NEDOSTATCI

Ipak, interpretirani jezici nisu rješenje svih problema:

- pokazuje se da je razvoj programa intrinsično težak
- dominantan faktor težine razvoja programa prema mnogima su inherentna složenost, te nedovoljno poznati i promjenljivi zahtjevi (Frederic Brooks: "*no silver bullet*")

Memorijski zahtjevi tipično veći od prevođenih jezika

JAVA: NEDOSTATCI

Ipak, interpretirani jezici nisu rješenje svih problema:

- pokazuje se da je razvoj programa intrinsično težak
- dominantan faktor težine razvoja programa prema mnogima su inherentna složenost, te nedovoljno poznati i promjenljivi zahtjevi (Frederic Brooks: "*no silver bullet*")

Memorijski zahtjevi tipično veći od prevođenih jezika

Nedeterminizam pri izvođenju može biti problematičan
(recikliranje memorije, JIT)

JAVA: NEDOSTATCI

Ipak, interpretirani jezici nisu rješenje svih problema:

- pokazuje se da je razvoj programa intrinsično težak
- dominantan faktor težine razvoja programa prema mnogima su inherentna složenost, te nedovoljno poznati i promjenljivi zahtjevi (Frederic Brooks: "*no silver bullet*")

Memorijski zahtjevi tipično veći od prevođenih jezika

Nedeterminizam pri izvođenju može biti problematičan
(recikliranje memorije, JIT)

Performansa JIT prevodioca teško može prestići kombinaciju:
statičko prevođenje + profiliranje + ručno poboljšavanje

JAVA: PERFORMANSA

Language	Execution method	Optimization	Bubble Sort relative performance	Quicksort relative performance	Speedup Quicksort vs. Bubble Sort
C	Compiler	None	1.00	1.00	2468
	Compiler	O1	2.37	1.50	1562
	Compiler	O2	2.38	1.50	1555
	Compiler	O3	2.41	1.91	1955
Java	Interpreter	–	0.12	0.05	1050
	JIT compiler	–	2.13	0.29	338

[Patterson07]

$$P_{\text{Java}}(C, \text{Bubblesort}) = 0.88$$
$$P_{\text{Java}}(C, \text{Quicksort}) = 0.15$$

Gubitak performanse u općem slučaju teško ocijeniti:

- varijacija po problemima i programerima veća od varijacije po jezicima i prevodiocima
- još i danas postoje razlozi za "jačanje" programa u C-u inline assemblerom (u Javi je to teže izvesti)
- prednost JIT-a ako veliki program nema istaknuto usko grlo

Kraj: literatura

1. Computer Organization and Design, 4th ed, David Patterson and John Hennessy, Morgan Kaufmann, 2008
2. Linkers and Loaders, John R Levine, Morgan Kaufmann, 2000

KRAJ