Assembly Language Programming
Linkers

Zbigniew Jurkiewicz, Instytut Informatyki UW

November 14, 2017

Placement problem (relocation)

- Because there can be more than one program in the memory, during compilation it is impossible to forecast their real addresses.
- Labor division:
  - linker: allocation of memory = preparation of relative addresses in program
  - loader: final relocation

Architectural solutions

- The hardware relocation (relocation registers) and virtual memory have simplified the job of a linker.
- Each program get the whole (virtual) address space.
- But there appeared a new element: *code sharing*, which forced the physical division of programs into code and data sections.

Linker tasks

- Consolidate binary modules
  - Combine relocatable binary modules into a single executable file, which will be loaded by the loader.
- Solve external references
  - They must be solved during consolidation process.
  - External reference: reference to a symbol in another module.
- Relocate symbols
  - Connect symbolic names with relative addresses described in the contexts of modules (.o files) with final absolute addresses in executable code.
  - Example: `getline` in `iosys` module $\leftrightarrow$ address 612 bytes from the beginning of the executable code.
  - Perform *code fixups*: update all references to this symbols to make them correspond to their new addresses.

Relocation

- Assembler generated addresses are not relocated.
- For example assume that in

```
mov eax,[a]
mov [b],eax
```

`a` has local address (offset) `0x1234`, and `b` is imported.

- Code after assembly

```
A1 34 12 00 00   mov eax,[a]
A3 00 00 00 00   mov [b],eax
```

- During linkage process the linker decides, that the section containing `a` is to be relocated `0x10000` bytes, and `b` has address `0x9A12`

```
A1 34 12 01 00   mov eax,[a]
A3 12 9A 00 00   mov [b],eax
```

Relocation

- Similar modifications are necessary for data section, if it contains pointers, e.g. in the table of procedure addresses.
- RISC generate more problems, because an address is often build by two or three consequent instructions.

Formats of binary files

- In Unix binary files (and other files too) start with 32-bit long *magic number*, which determines the type of a file.
- Traditional, but now rarely used, format of binary file in Unix is `a.out`. Its magic number is `0x407`.
- It has been mostly replaced by the ELF format.

Format `a.out`

Header a.out
Section text
Section data
Other sections
Optional relocation information

Format ELF

ELF header
Program header table (dla loadera)
.text section
.data section
.bss section
.symtab
.rel.text
.rel.data
.debug
Section header table (relocation info for linker)

ELF format (*Executable and Linking Format*)

- Now basic format in Unix with magic number `0x177` = `'ELF'`
- It is used for storing executable programs, as well as object modules and libraries (and also memory dumps).
- So has different type: relocatable, executable, *shared*, *core image*. So it is possible to place there informations necessary for liker and for loader too.
- File starts with general header, then there is program segment table (description of segments for loader).

ELF format

- This is followed by proper file contents, that is sections:
  - code `.text`
  - initialized data `.data`
  - non-initialized data `.bss`
  - `.symtab`: symbol table
  - `.rel.text`: relocation info for `.text`
  - `.rel.data`: relocation info for `.data`
  - `.debug`: debugger info (if `gcc -g` was used)
- The file concludes with a section header table, which describes all sections for linker use.

Libraries

- Libraries divide into static and *shared* (aka. dynamic).
- With static libraries you have to relink program after each modification of program **or libraries**.

Shared libraries

- *Shared* libraries divide into loaded statically or dynamically.
  - for libraries loaded statically the addresses (placement) are decided during program load (at load-time, if you prefer);
  - libraries loaded dynamically can be loaded at run-time as needed, at the first call to their procedures (such procedures are sometimes called *autoloaded*).
- Shared libraries should contain *Position Independent Code* (PIC), to make it possible to load them into any area of memory (e.g. `gcc` compiler has an option for that).

Dynamic linkage: methods

- *Implicit linking*
  - With a program we associate a *linkage segment*, describing external procedures called from dynamic libraries as pairs [*name*, *address* (initially equal 0, i.e. incorrect)].
  - The code contains indirect calls through this address, except for the first time, bacause initially it is a *trap* to dynamic linker, resulting in incremental linking of necessary library and filling the address field.
- *Explicit linking*
  - Program in its prologue specifies all used shared libraries and links with them.

Creating a dynamic library

- To build a dynamic library, we use assembler as usual

```
nasm -f elf64 pakiet.asm
```

- However when declaring the exported symbols we should specify their ich type: `function` or `data`, for example

```
  global random:function,seed:data
```

- Linkage will be different

```
ld -shared -o libpakiet.so pakiet.o
```

Creating a dynamic library

- The library built that way can be used similarly to system libraries

```
ld -L . -dynamic-linker /lib/ld-linux.so.2 \
   -o program program.o -l pakiet
```

- Before running a `program` we must appropriately set the environment variable, for example

```
export LD_LIBRARY_PATH=.
```

- Utility `ldd` called with binary program name will tell us, which shared libraries are used, and which have not been found.
- Utility `nm` gives all external symbols of any binary module.