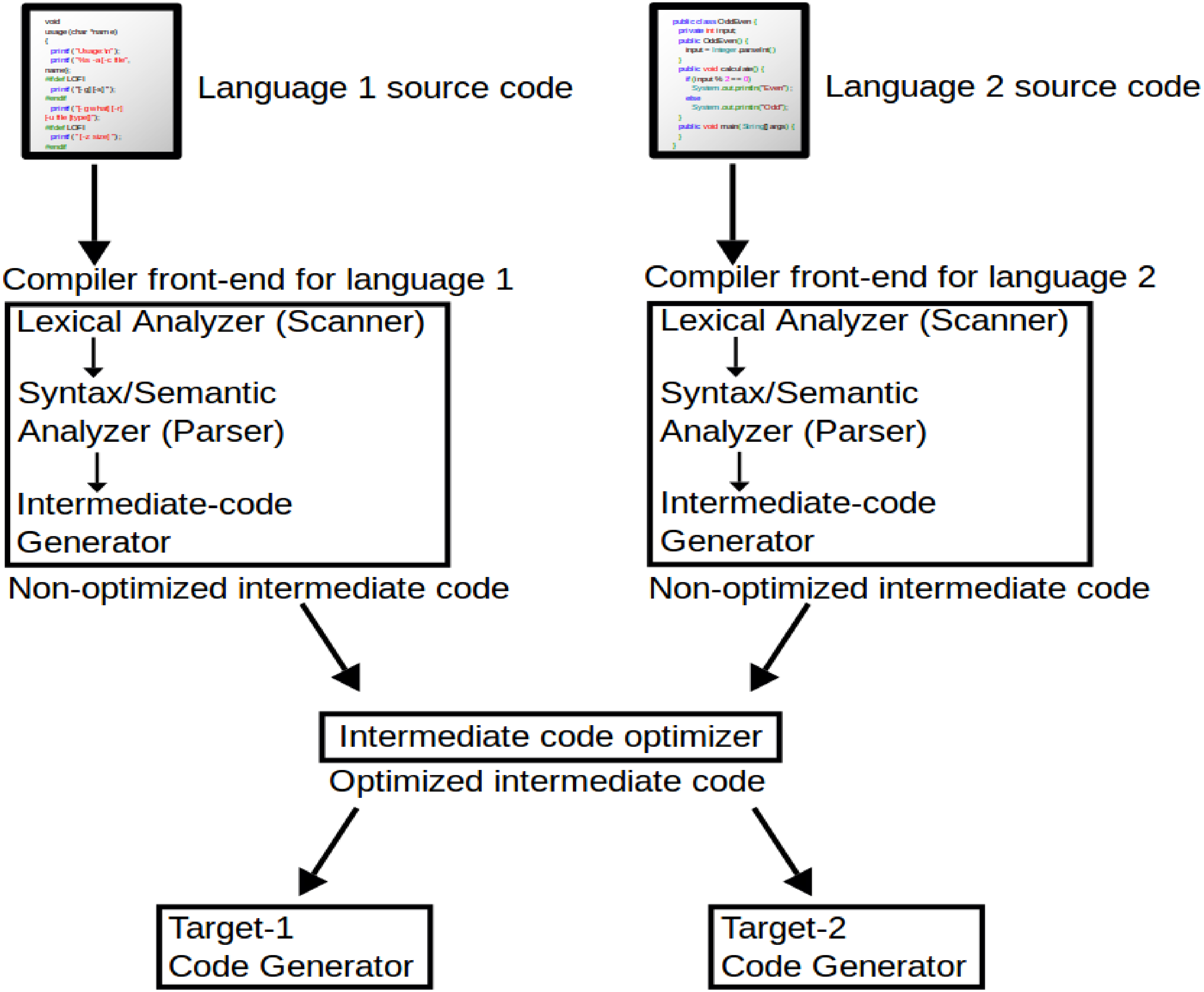
Lecture 9: From program to executable Linking and loading

Contents

- Linker vs. loader
- Linking the executable
- Libraries
- Loading executable
- ELF – UNIX format
- PE – windows program
- Dynamic libraries

Compiler structure



```
void
usage(char *name)
{
  printf("Usage:\n");
  printf("%s -a[-c file",
name);
#ifdef LOFI
  printf("[-g][-s]");
#endif
  printf("[-g what] [-r]
[-u file [type]]");
#ifdef LOFI
  printf(" [-z size] ");
#endif
```

Language 1 source code

```
public class OddEven {
  private int input;
  public OddEven() {
    input = Integer.parseInt()
  }
  public void calculate() {
    if (input % 2 == 0)
      System.out.println("Even");
    else
      System.out.println("Odd");
  }
  public void main( String[] args) {
  }
}
```

Language 2 source code

Compiler front-end for language 1

Lexical Analyzer (Scanner)

Syntax/Semantic Analyzer (Parser)

Intermediate-code Generator

Non-optimized intermediate code

Compiler front-end for language 2

Lexical Analyzer (Scanner)

Syntax/Semantic Analyzer (Parser)

Intermediate-code Generator

Non-optimized intermediate code

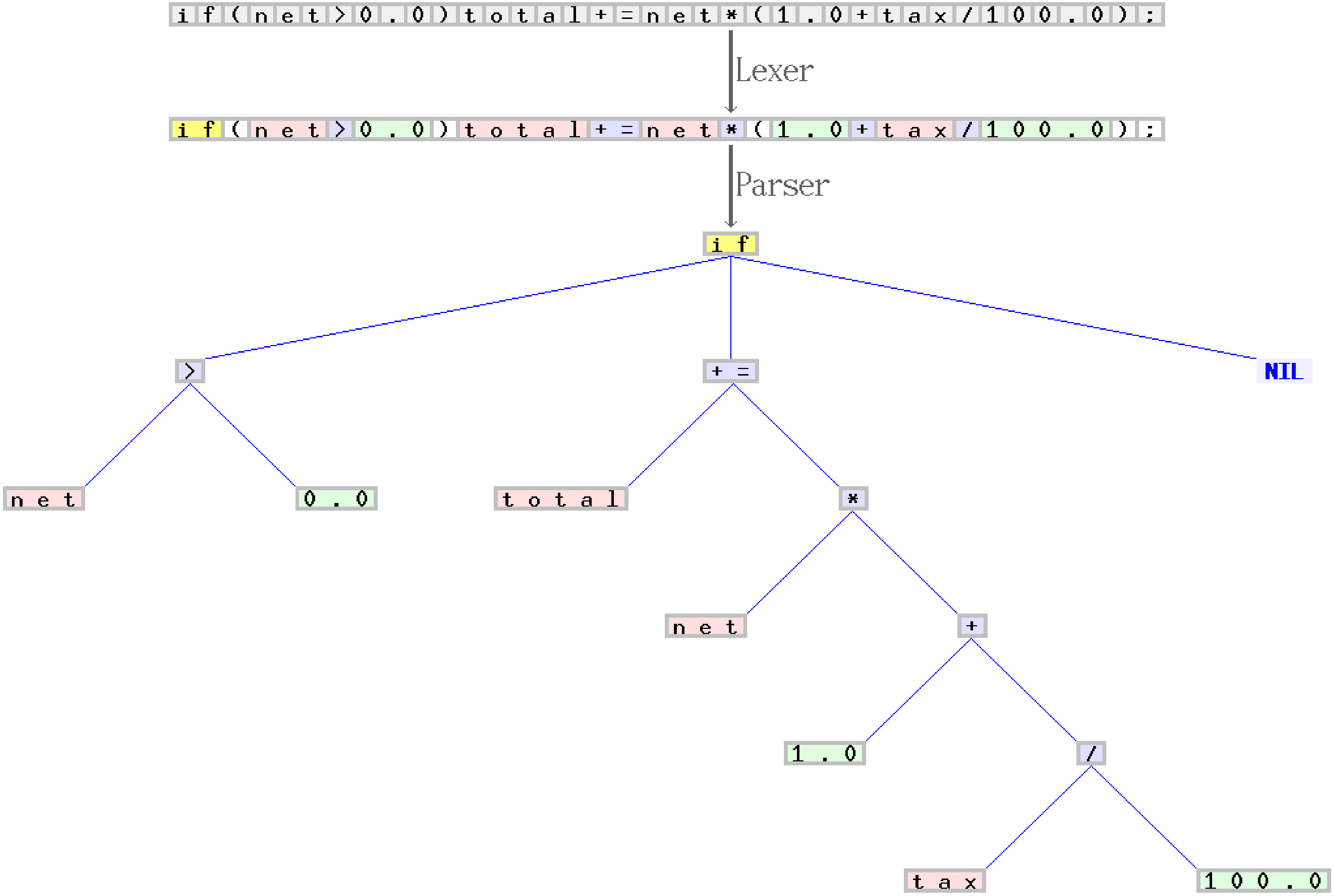
Intermediate code optimizer

Optimized intermediate code

Target-1 Code Generator

Target-2 Code Generator

Lexical and syntax analysis



if(net>0.0)total+=net*(1.0+tax/100.0);
Lexer
if(net>0.0)total+=net*(1.0+tax/100.0);
Parser
if
>
+=
NIL
net
0.0
total
*
net
+
1.0
/
tax
100.0

Syntactic and semantic analysis

- Parser is doing syntactic and semantic analysis
- Parser can be generated automatically using tools: yacc or bison
  - yacc = Yet Another Compiler Compiler; bison is animal similar to yacc
- Parser are generated according to Grammar usually in Backus-Naur Form (BNF)

- BNF – definition of address in BNF:

```
<postal-address> ::= <name-part> <street-address> <psc-part>
<name-part>      ::= <personal-part> <last-name> <EOL>
                   | <personal-part> <name-part>
<personal-part>  ::= <first-name>
                   | <initial> "."
<street-address> ::= <street-name> <house-num> <EOL>
<psc-part>       ::= <PSC> “  “ <town-name> <EOL>
```

Intermediate code

- .class is example of intermediate code for java
- Lot of different types: - three-address code (often abbreviated to TAC or 3AC)

  Example of TAC representation

```
t1 := b * b       x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)
t2 := 4 * a
t3 := t2 * c
t4 := t1 - t3
t5 := sqrt(t4)
t6 := 0 - b
t7 := t5 + t6
t8 := 2 * a
t9 := t7 / t8
x := t9
```

```
     t1 := 0            ; for (i = 0; i < 10; ++i) {b[i] = i*i;}
L1:  if t1 >= 10 goto L2    ; conditional jump
     t2 := t1 * t1          ; square of i
     t3 := t1 * 4           ; word-align address
     t4 := b + t3           ; address to store i*i
     *t4 := t2              ; store through pointer
     t1 := t1 + 1           ; increase i
     goto L1                ; repeat loop
L2:
```

Background

- Operating system is responsible for starting programs
- Program must be brought into memory and placed within a process memory space for it to be executed
- User programs go through several steps before being run
- Linkers and loaders prepare program to execution
- Linkers and loaders enable to binds programmer's abstract names to concrete numeric values – addresses

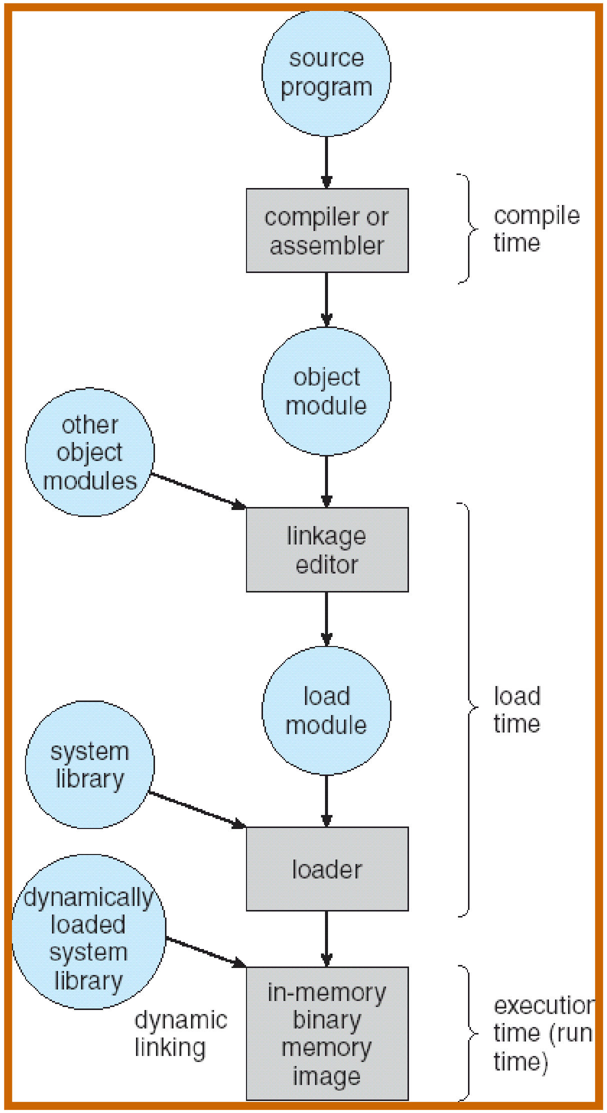
Linker vs. Loader

- Program loading – copy program from secondary storage into main memory so it’s ready to run
  - In some cases it is copying data from disk to memory
  - More often it allocate storage, set protections bits, arrange virtual memory to map virtual addresses to disk space
- Relocation
  - each object code program address started at 0
  - If program contains multiple subprograms all subprograms must be loaded at non-overlapping addresses
  - In many systems the relocation is done more than once
- Symbol resolution
  - The reference from one subprogram to another subprogram is made by using symbols

- Linker and loader are similar
- Loader does program loading and relocation
- Linker does symbol resolution and relocation
- There exists linking loaders

Binding of Instructions and Data to Memory

- **Compile time**: If memory location is known a priori, *absolute code* can be generated; must recompile code if starting location changes
- **Load time**: Must generate *relocatable code* if memory location is not known at compile time
- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., *base* and *limit registers*).



Two pass linking

- Linker’s input is set of object files, libraries, and command files.
- Output of the linker is executable file, link/load map and/or debug symbol file
- Linker uses two-pass approach
- Linker first pass
  - Scan from input files segment sizes, definitions and references
  - Creates symbol table of definitions and references
  - Determine the size of joined segments
- Linker second pass
  - Assign numeric location to symbols in new segments
  - Reads and relocates the object code, substituting numeric address for symbol references
  - Adjusting memory address according new segments
  - Create execution file with correct:
    - Header information
    - Relocated segments
    - New symbol table information
    - For dynamic linking linker generates “stub” code or an array of pointers that need

Object code

- Compilers and assemblers create object files from source files

- Object files contains:
  - Header information – overall information about file, like size of the code, size of the data, name of the source file, creation date
  - Object code – binary instructions and data
  - Relocation – list of places in object code, that have to be fixed up, when the linker or loader change the address of the object code
  - Symbols – global symbols defined in this object file, this symbols can be used by other object files
  - Debugging information – this information is optional, includes information for debugger, source file line numbers and local symbols, description of data structures

Library

- Library is sequence of object modules
- UNIX files use an “archive” format of file which can be used for collection of any types of files
- Linking library is iterative process:
  - Linker reads object files in library and looks for external symbols from program
  - If the linker finds external symbol it adds the concrete object file to program and adds external symbols of this library object to external symbols of program
  - The previous steps repeat until new external symbols and objects are added to program
- There can be dependencies between libraries:
  - Object A from lib A needs symbol B from lib B
  - Object B from lib B needs symbol C from lib A
  - Object C from lib A needs symbol D from lib B
  - Object D from lib B needs symbol E from …………

UNIX ELF

- Structure for object and executable programs for most UNIX systems
- Successor of more simple format a.out
- ELF structure is common for relocatble format (object files), executable format (program from objects), shared libraries and core image (core image is created if program fails)
- ELF can be interpreted as a set of sections for linker or set of segments for loader

- ELF contains:
  - ELF header – magic string \177ELF, attributes - 32/64 bit, little-endian/big-endian, type – relocatable/executable/shared/core image, architecture SPARC/x86/68K,….
  - Data – list of sections and segments depending on ELF type

ELF relocatable

- Created by compiler and is prepared for linker to create executable program
- Relocatable files – collection of section defined in header. Each section is code, or read-only data, or rw data, or relocation entries, or symbols.
- Attribute alloc means that loader must allocate space for this section
- Sections:
  - .text – code with attribute alloc+exec
  - .data – data with initial value, alloc+write
  - .rodata – constants with only alloc attribute
  - .bss – not initialized data – nobits, alloc+write
  - .rel.text, .rel.data, .rel.rodata – relocation information
  - .init – initialization code for some languages (C++)
  - .symtab, .dynsym – linker symbol tables (regular or dynamic)
  - .strtab, .dynstr – table of strings for .symtab resp. .dynsym (.dynsym has alloc because it's used at runtime)

ELF - exucutable

- Similar to ELF-relocatable but the data are arranged so that are ready to be mapped into memory and run
- Sections are packed into segments, usually code and read-only data into read-only segment and r/w data into r/w segment
- Segments are prepared to be loaded at defined address
- Usually it is:
  - Stack from 0x8000000
  - Text with ro-data from 0x8048000 – 0x48000 is stack size
  - Data behind text
  - Bss behind data
- Relocation is necessary if dynamic library is colliding with program – Relocated is dynamic library
- Segments are not align to page size, but the offset is used and some data are copied twice

Microsoft Portable Executable format

- Portable executable (PE) is Microsoft format for Win NT. It is mix of MS-DOS executable, Digital’s VAX VMS, and Unix System V. It is adapted from COFF, Unix format between a.out and ELF
- PE is based on resources – cursors, icons, bitmaps, menus, fonts that are shared between program and GUI
- PE is for paged environment, pages from PE can be mapped directly into memory
- PE can be executable file (EXE) or shared libraries (DLL)
- PE starts with small DOS.EXE program, that prints “This program needs Microsoft Windows”
- Then contains PE header, COFF header and “optional” headers
- Each section is aligned to memory page boundary

PE sections

- Each section has address in file and size, memory address and size (not necessarily same, because disk section use usually 512bytes, page size 4kB)
- Each section is marked with hardware permissions, read, write, execute
- The linker creates PE file for a specific target address – imagebase
- If the address space is free than loader do no relocation
- Otherwise (in few cases) the loader has to map the file somewhere else
- Relocation is done by fix-ups from section .reloc. The PE is moved as block, each pointer is shifted by fixed offset (target address – image address). The fix-up contains position of pointer inside page and type of the pointer.
- Other sections – Exports (mainly for DLL, EXE only for debugging), Imports (DLL that PE needs), Resources (list of resources), Thread Local Storage (Thread startup data)

Shared libraries - static

- It is efficient to share libraries instead linking the same library to each program
- For example, probably each program uses function printf and if you have thousands of programs in computer there will be thousands of copy printf function.
- The linker search library as usual to find modules that resolve undefined external symbols. Rather than coping the contents of module into output file it creates the table of libraries and modules into executable
- When the program is started the loader finds the libraries and map them to program address space
- Standards systems shares pages that are marked as read-only.
- Static shared libraries must used different address.
- Assigning address space to libraries is complicated.

Dynamic Libraries

- Dynamic Libraries can be relocated to free address space
- Dynamic Libraries are easier to update. If dynamic library is updated to new version the program has no change
- It is easy to share dynamic libraries
- Dynamic linking permits a program to load and unload routines at runtime, a facility that can otherwise be very difficult to provide
- Routine can be loaded when it is called
- Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases

ELF dynamic libraries

- ELF dynamic libraries can be loaded at any address, it uses position independent code (PIC)
- Global offset table (GOT) contains pointer to all static data referenced in program
- Lazy procedure linkage with Procedure Linkage Table (PLT)
  - For each dynamic function PLT contain code that use GOT to find address of this function
  - At program load all addresses point to stub – dynamic loader
  - After loading dynamic library entry in GOT is changed to real routine address
- Dynamic loader (library ld.so) finds the library by library name, major and minor versions numbers. The major version number guarantee compatibility, the minor version number should be the highest.
- Dynamic loading can be run explicitly by dlopen(),dlsym(), … functions

Dynamic Linking Libraries - DLL

- Similar to ELF dynamic libraries
- Dynamic linker is part of the windows kernel
- DLL is relocated if the address space is not free (windows call it rebasing)
- Lazy binding postpones binding until execution time
- Each function exported by DLL is identified by a numeric ordinal and by name
- Addresses of functions are defined in Export Address table

Architectural Issues

- Linkers and loaders are extremely sensitive to the architectural details of CPU and OS
- Mainly two aspects of HW architecture affect linkers
  - Program addressing
  - Instruction format
- Position independent code – enable to implement dynamic libraries
  - Separate code from data and generate code, that won't change regardless of the address at which it is loaded
  - ELF – PIC group of code pages followed by group of data pages
  - Regardless of where the in the address space the program is loaded, the offset from the code to the data doesn't change
  - Linker creates Global Offset Table containing pointers to all of the global data
  - Advantage – no load relocation, share memory pages of code among processes even though they don't have the same address
  - Disadvantage – code is bigger and slower than non-PIC

End of Lecture 8

Questions?