

CS 24

# Introduction to Computer Systems

11: Compiler, Assembler, Linker

# Questions?

# Midterm Exam

- Yes, there will be a midterm.
- Out Wednesday after class; due following Wednesday before class.

# Building a Program

- Code for a program takes up several files.
- Can compile all at once with gcc:  
gcc -o exename file1.c file2.c file3.s ...
- Alternatively, can compile each separately:

```
gcc -c file1.c
```

```
gcc -c file2.c
```

```
gcc -c file3.s
```

```
...
```

Each source file translated to an **object file**.

Put them together with another run of gcc:

```
gcc -o progname file1.o file2.o file3.o ...
```

# Rebuilding

- Compiling source files separately makes rebuilding after changes faster.
  - Recompile the file(s) that changed.
  - Recompile files that `#include` files that changed.
  - Regenerate executable (i.e. **relink**).

# Make

- Recompiling programs after changes can get confusing.
- Popular tool for automation: make.
  - Reads description of program in Makefile
  - Finds out which files changed since last build.
  - Recompiles as necessary.
- I have been providing makefiles for the lab assignments.

# A Compiler “Driver”

- GCC command handles many different kinds of input files:  
C source (.c); Assembler source (.s); Object (.o); C++ (.cc); Java (.java); Fortran (.f); ...
- It’s really a driver that calls other programs to process files.
  - .o: ld (linker).
  - .s: as (assembler), ld.
  - .c: cpp (preprocessor), cc1 (compiler), as, ld.
  - .java: gcj (Java compiler, if installed\*), as, ld.
  - .f: f951 (Fortran compiler, if installed\*), as, ld.
  - .ads: gnat1 (Ada compiler, if installed\*), as, ld.
- Basic process:  
Compile C to assembly; assemble to object code; link to executable.

\*CS machines seem to have gcj installed, but not the Fortran and Ada compilers.

# What's in a binary file?

- One or more **sections**: blocks of binary data.
  - .**text**: Executable code.
  - .**rdata**: Read-only data.
  - .**data**: Initialized statically-allocated data.
  - .**bss**: Uninitialized statically-allocated data.
  - .**relo.\***: Descriptions of relocations (later).
- Executable object file:
  - .exe in MS Windows, no extension in Unix.
  - When run, contents loaded into memory.
    - Each section gets a contiguous block of memory.
    - Placed at well-defined addresses.
  - File header has start address; loader jumps to it.

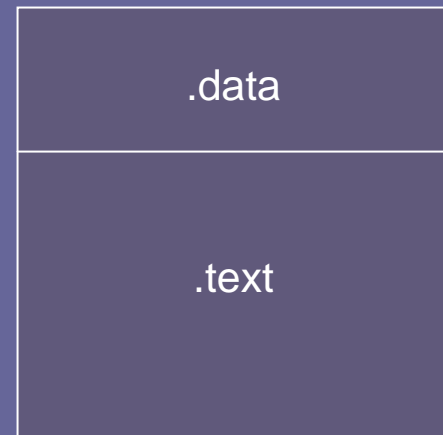
# What's in a binary file?

- Relocatable object file:
  - .o in Unix/Cygwin; .obj in Windows-native tools.
  - Contains one C file's contributions to executable image.
  - “Relocatable”: doesn't care where it's placed wrt other files in final program.
- Linker “collates” all files' .text sections, .data sections, *etc.*

file1.o

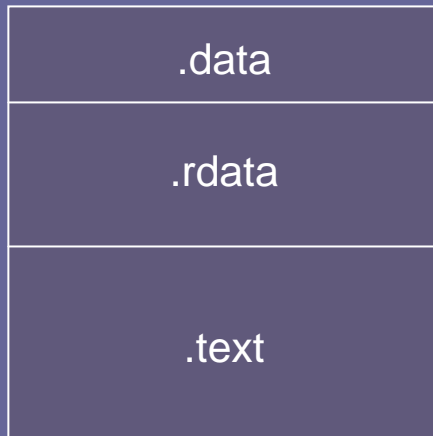


file2.o

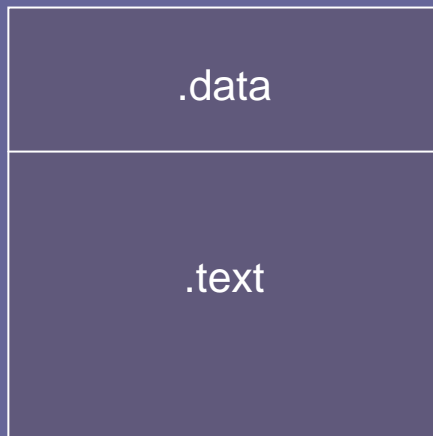


# Linking

file1.o

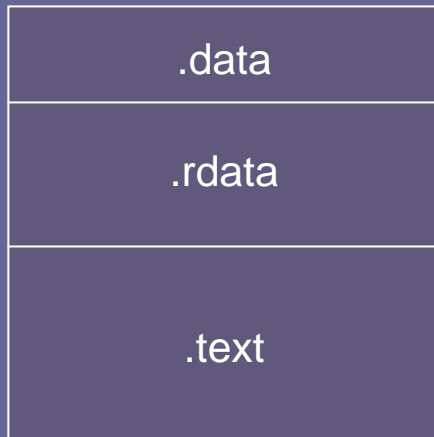


file2.o

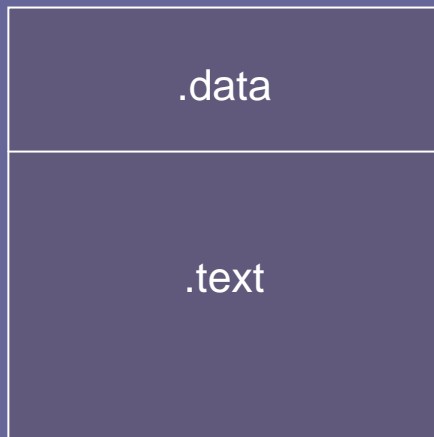


# Linking

file1.o

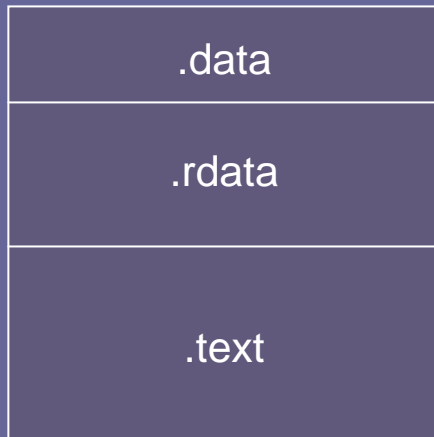


file2.o

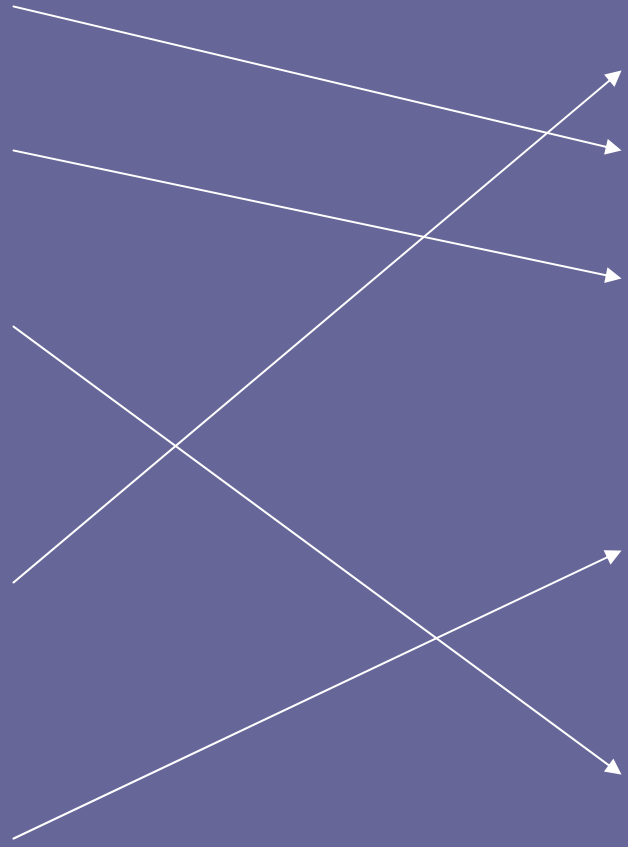
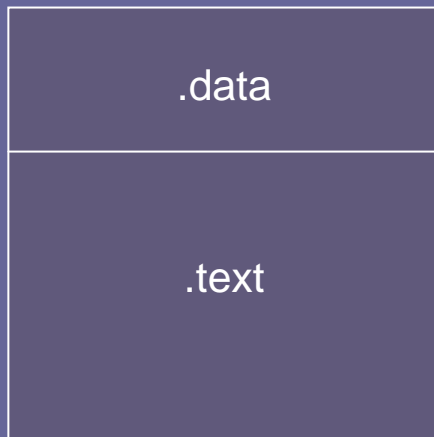


# Linking

file1.o



file2.o



# File Formats

- ELF: Executable & Linking Format
  - Used by most Unix systems
- COFF: Common Object File Format
  - Used in MS-DOS
- PE: Portable Executable
  - Extension of COFF (also called PE/COFF)
  - Used in MS Windows (including Cygwin).
- Historical note:
  - a.out (“assembler output”), default output file name for Unix C compilers, is the name of the file format they used to use.
  - Nowadays, the file named “a.out” is in ELF or PE/COFF format, as appropriate.

# Symbols

- Assembler source defines and uses **symbols**.
- Most important are **labels**: addresses of code or data.
- By the time code is executed, symbols must be replaced by their definitions.

```
loop:  ...  
      ...  
      ...  
      jecxz  loop
```

# Symbols

- Problem 1: Assembler doesn't know the address where this code will be.
  - How can it output a binary encoding for the `jecxz` instruction?

```
loop: ...  
...  
...  
jecxz  loop
```

# Symbols

- Problem 1: Assembler doesn't know the address where this code will be.
  - How can it output a binary encoding for the `jecxz` instruction?
- Solution: IA-32 direct jump operands can be **PC-relative**.
  - `jecxz` is encoded in 2 bytes:  
11100011 (displacement)
  - If jumping, **adds** displacement to EIP.  
(EIP = address of next instruction.)
- Objdump demo #1.

```
loop:  ...  
      ...  
      ...  
      jecxz  loop
```

- PC-relative addressing takes care of **jump targets** defined in the **same section** in the **same file**.

# Symbols

- Problem 2: **Non-jump instructions** can't use PC-relative addressing.
  - Not in IA-32, anyway.
  - Instruction encoding needs the **absolute address**.
  - Also, at right, **x** is defined in a different section.
  - Assembler doesn't know where linker will place this file's `.rodata` wrt other files'.

```
x:      .data
        .asciz "Hello!"
        .text
loop:   pushl   $x
        ...
        ...
        jecxz  loop
```

# Symbols

- Problem 2: **Non-jump instructions** can't use PC-relative addressing.
  - Not in IA-32, anyway.
  - Instruction encoding needs the **absolute address**.
  - Also, at right, **x** is defined in a different section.
  - Assembler doesn't know where linker will place this file's `.rodata` wrt other files'.
- Solution: Relocation.
  - Assembler outputs instruction with placeholder.
  - Puts "relocation entry" in object file.
  - Tells linker to insert symbol's value in proper location.
- Objdump demo #2.

```
x:      .data
        .asciz "Hello!"
        .text
loop:   pushl   $0
        ...
        ...
        jecxz  loop
```

## **.reloc.text:**

At <location of \$0 placeholder>, insert 32-bit value of **x**.

# Symbols

- Problem 3: What about jumps to code in other files?
  - Encoding of call needs displacement from PC.
  - But assembler has no idea where that function will be.
  - Can't even be sure symbol means anything!

```
x:      .data
        .asciz "Hello!"
        .text
loop:   pushl   $x
        call   puts
        ...
        jecxz  loop
```

# Symbols

- Problem 3: What about jumps to code in other files?
  - Encoding of call needs displacement from PC.
  - But assembler has no idea where that function will be.
  - Can't even be sure symbol means anything!
- Solution: Different kind of relocation entry.
  - Assembler outputs 4 bytes where displacement will go.
  - Adds relocation entry asking linker for *difference* between symbol and placeholder address.

```
x:      .data
        .asciz "Hello!"
        .text
loop:   pushl   $0
        call   -4
        ...
        jecxz  -disp
```

**.relo.text:**

At <location of \$0 placeholder>,  
insert 32-bit value of x.

At <location of -4 placeholder>,  
insert 32-bit value:  
(puts - address of placeholder +  
placeholder)

# Libraries

- Linker is not intended to be very smart.
  - Doesn't understand fine structure of contents of a .o file.
    - Where functions (as opposed to intra-function blocks) start and end.
    - Which variables are referenced from which code.
- This info is not stored in the files, and undecidable for raw machine code.
- Consequence: Passing a .o file to ld is all-or-nothing.
  - Dilemma: What about big libraries?
    - E.g., C standard library; C math library; Boehm-Demers-Weiser GC; ...
    - Should a library be one big .o file? Executables would all be very large.
    - Should it be a .o file for each function (say)? Linker arguments would get out of hand.

# Libraries are Archives

- A (static) library is an archive file containing several `.o` files and an index of the symbols they export.
- Create using `ar` utility, *e.g.*:  
`ar rcs mylib.a file1.o file2.o ...`  
creates library called `mylib.a`.
- Passing an archive to `ld` (via `gcc`):  
`gcc -o myprog myprog.c mylib.a`  
Linker includes only `.o` files from `mylib.a` whose exported symbols are referenced in `myprog.c`
- Library demo.

# Common Libraries

- Most Unix systems have tons of libraries.
- Stored in `/lib`, `/usr/lib`, `/usr/local/lib`, `/usr/share`, *etc.*
- Linker knows to look for libraries in these places.
  - Like C compiler knows to look for header files in `/usr/include`, *etc.*
- (Static) library files have names of form `libsomething.a`
  - Tell gcc (or ld) you need this library with `-lsomething`.  
(that's a lowercase L)
  - If `libsomething` is in a nonstandard place, tell gcc the directory to search with `-L`.  
`gcc -o myprog -L/path/to/library -lmystuff`  
(Can find `libmystuff.a` in the given directory.)
- gcc implicitly specifies (among others) `-lc` for `libc`, the C library.

# Dynamic Linking and Sharing

- Most programs use a lot of the same library routines (e.g., malloc).
- As described so far, linker embeds a copy of each routine's code in every executable that uses it.
- Idea to keep executables small:
  - Wait until *load time* to link common big libraries.
  - A copy of the library lives somewhere in the filesystem (e.g., /usr/lib).
  - Executable object files don't need to have library code embedded.
  - (This has another big benefit -- later...)
- These **shared libraries** (with **dynamic linking**) are actually the default.
  - To get a statically-linked executable, use gcc **-static**.

# Shared Libraries

## Benefits:

- Smaller executables.
- Less memory: By virtual memory tricks, get away with only *loading* one copy of a shared library at a time.
- Maintainability: Application can always run with latest version of library, no matter when it was compiled.
  - Bug fixes and security patches

## Drawbacks:

- Application can only run on computer with all the libraries it needs installed.
- Application written for one library version may not work with another version.
  - May need more than one version of a library installed.
  - “DLL Hell”

# Implementing Shared Libraries

- Main concerns:
  - Shared libraries don't know where in memory they will be loaded.
    - Code must be **position-independent** (PIC). Essentially, no absolute addressing, ever. Must use various tricks (see B&O'H book).
    - Generating a shared library:

```
gcc -fPIC -c file1.c  
gcc -shared -o libname.so file1.o ...
```
  - Linker generating executable doesn't know where shared library code/data will be.
    - References to shared libraries are resolved at load time or run time by the *dynamic linker*.