Linker and Loader

Samit Bhattacharya

Comp Sc & Engg

Indian Institute of Technology Guwahati

Linking

- Process of combining various pieces of code, module or data together to form a single executable unit that can be loaded in memory

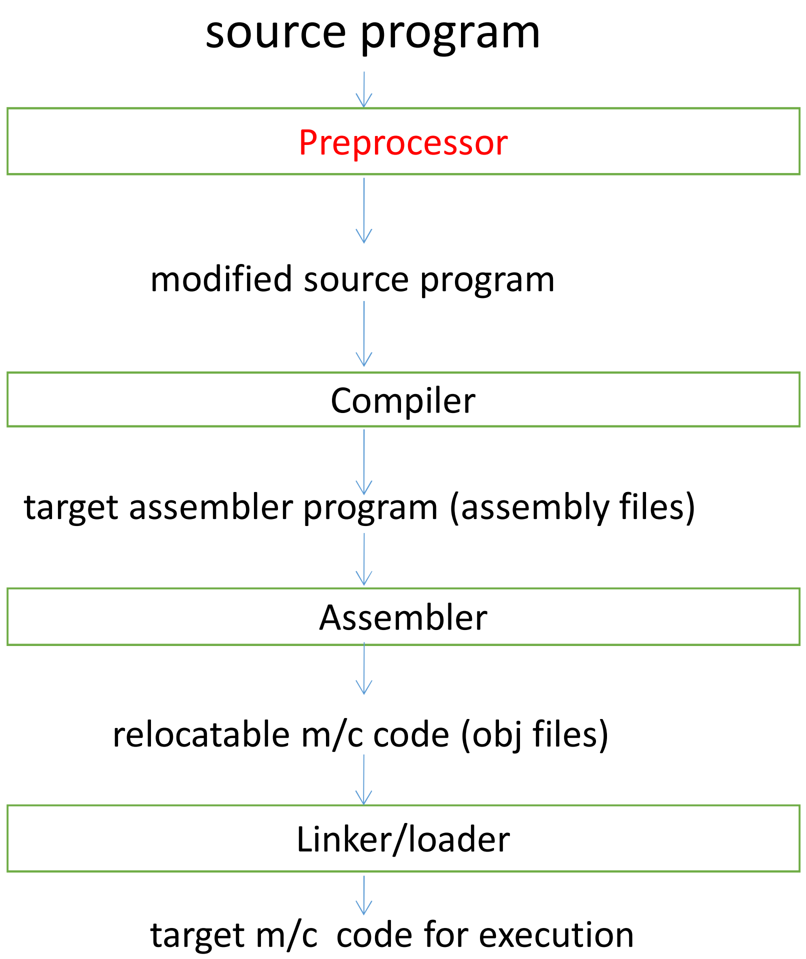
- Can be done
  - at compile time
  - at load time (by loaders)
  - at run time (by application programs)

Loader

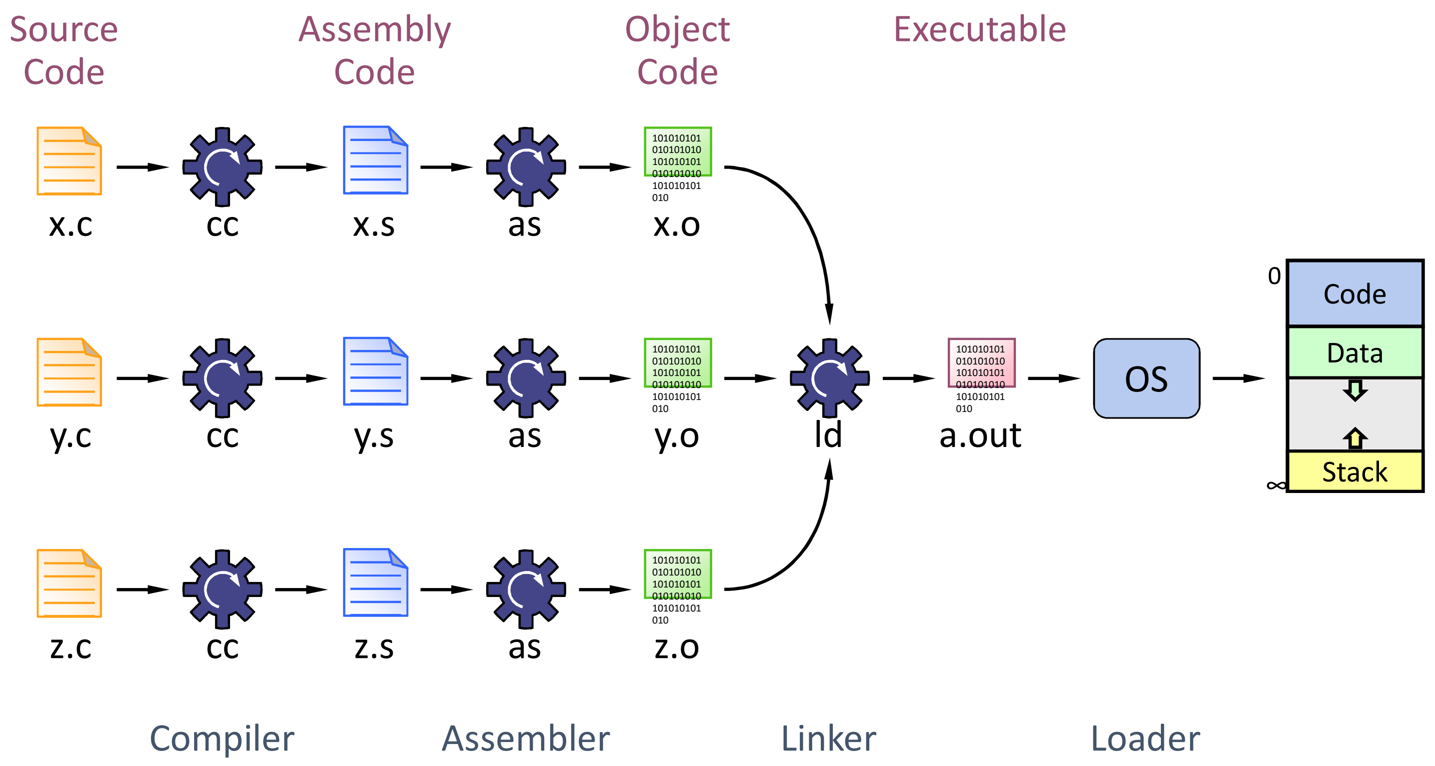
- Program that takes object program and *prepares* it for execution
  - Once executable file has been generated, the actual object module generated by the linker is deleted

Review - Calling Sequence Convention

- Compiler output is assembly file
- Assembler output is object file
- Linker joins object files into one executables
- Loader brings it into memory and starts execution


source program
Preprocessor
modified source program
Compiler
target assembler program (assembly files)
Assembler
relocatable m/c code (obj files)
Linker/loader
target m/c code for execution

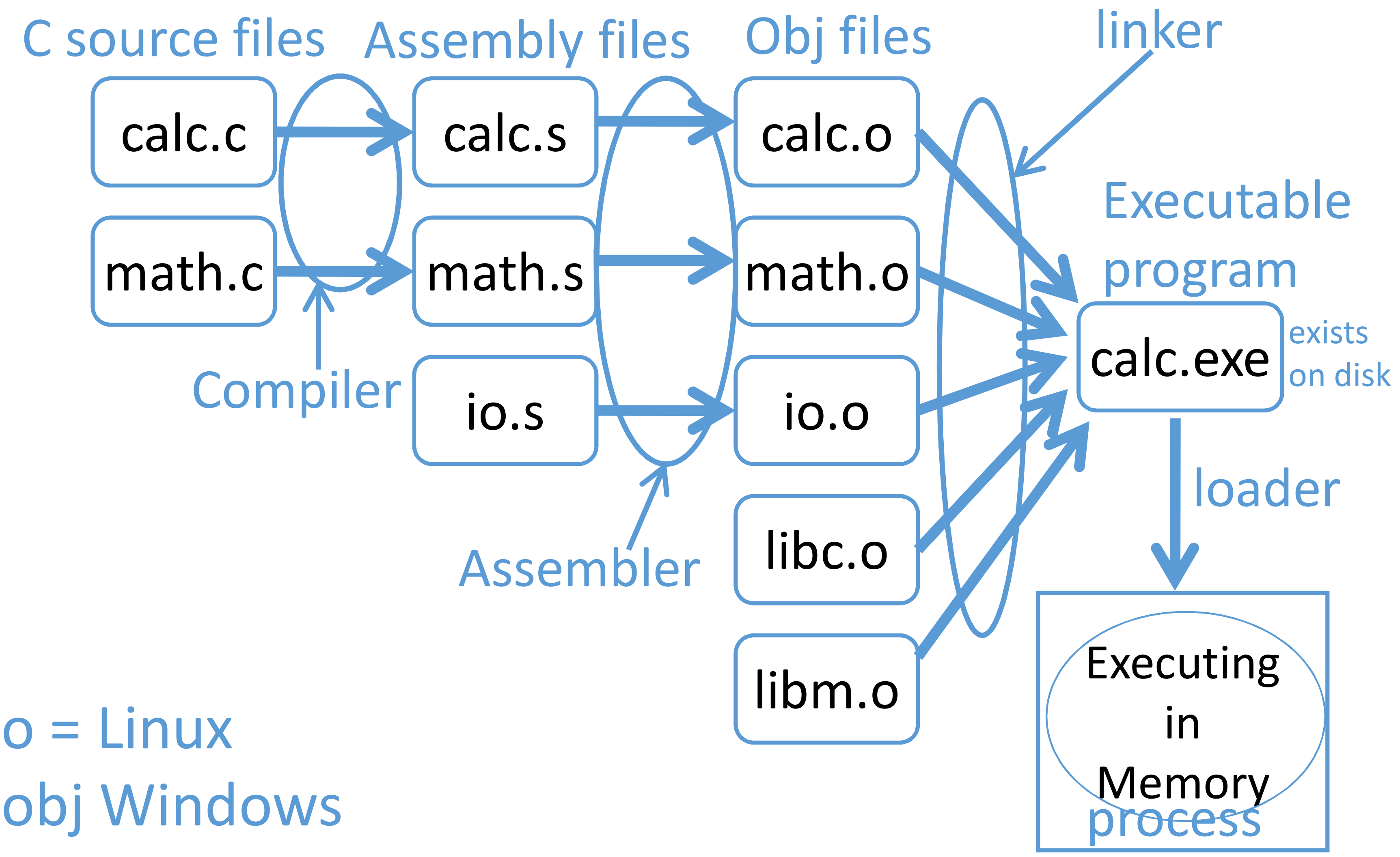
Creating a Process



Functions of Linkers and Loaders

- Linkers
  - Resolves problems with external references (EXTREF) and external definitions (EXTDEF)

- Loaders
  - Brings object program into memory and starts its execution
  - Initializes registers, stack, arguments to first function
  - Jumps to entry-point



.o = Linux

.obj Windows

Note

- Three tasks
  - Loading (into memory)
  - Relocation (program level & memory level)
  - Symbol resolution (when combining multiple files)

- Linkers – symbol resolution & program level relocation (typically)
- Loader – memory level reloacation & loading (typically)

Object File

- Compilers and assemblers create object files containing the generated binary code and data for a source file

- Three forms of object file:
    - Relocatable
    - Executable
    - Shared

Note: Compilers and assemblers generate relocatable and shared form of object files. Linkers combine these object files together to generate executable form of object files.

Object File Contd…

- Relocatable
  - contains binary code and data in a form that can be combined with other relocatable object files at compile time to create an executable object file
- Executable
  - contains binary code and data in a form that can be directly loaded into memory and executed
- Shared
  - a special type of relocatable object file that can be loaded into memory and linked dynamically, either at load time or at run time

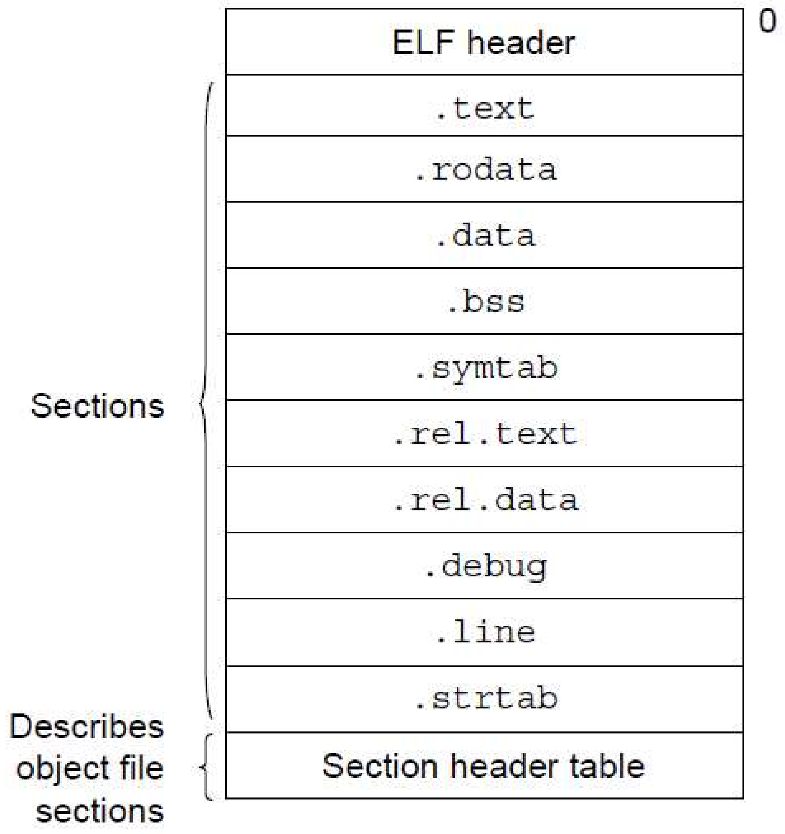
Object File Format

- An object file contains five basic types of information
  - Header: size and position of segments of file
  - Object code: instructions
  - Relocation information: final location (symbol, function)
  - Symbols: external (exported) references, unresolved (imported) references
  - Debugging information: line number, code address map, etc.

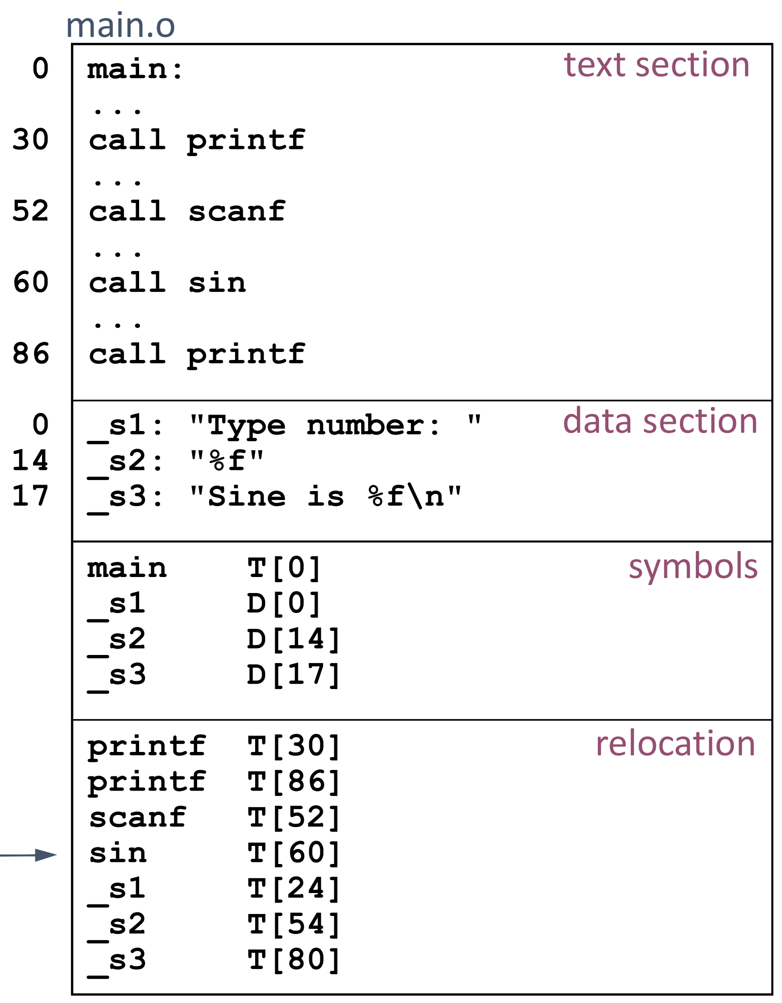
Note: Some object files may contain some more information apart from these five types.

Example of a Relocatable Object file

- Typical Unix Executable and Linkable Format (ELF) relocatable object file
  - .text : machine code of the complied program
  - .rodata : read-only data (strings in printf)
  - .data : Initialize global variables
  - .bss : Uninitialized global variables
  - .sysmtab : entry for symbols (variable name, function name)
  - .rel.text : locations in .text section need to modified when linker combine it in other obj
  - .rel.data : relocation information for any global variables that are referenced or defined by the module
  - .debug : entries for local variables and typedefs defined in the source
  - .line : mapping between line numbers in the original source
  - .strtab : Null terminated character string for .sysmtab, .debug, and header section



Example Object File with Basic Information



main.c

```
extern float sin();
extern printf(), scanf();

main() {
  double x, result;
  printf("Type number: ");
  scanf("%f", &x);
  result = sin(x);
  printf("Sine is %f\n",
         result);
}
```

"Store the final location of sin at offset 60 in the text section"

main.o

text section

```
 0  main:
    ...
30  call printf
    ...
52  call scanf
    ...
60  call sin
    ...
86  call printf
```

data section

```
 0  _s1: "Type number: "
14  _s2: "%f"
17  _s3: "Sine is %f\n"
```

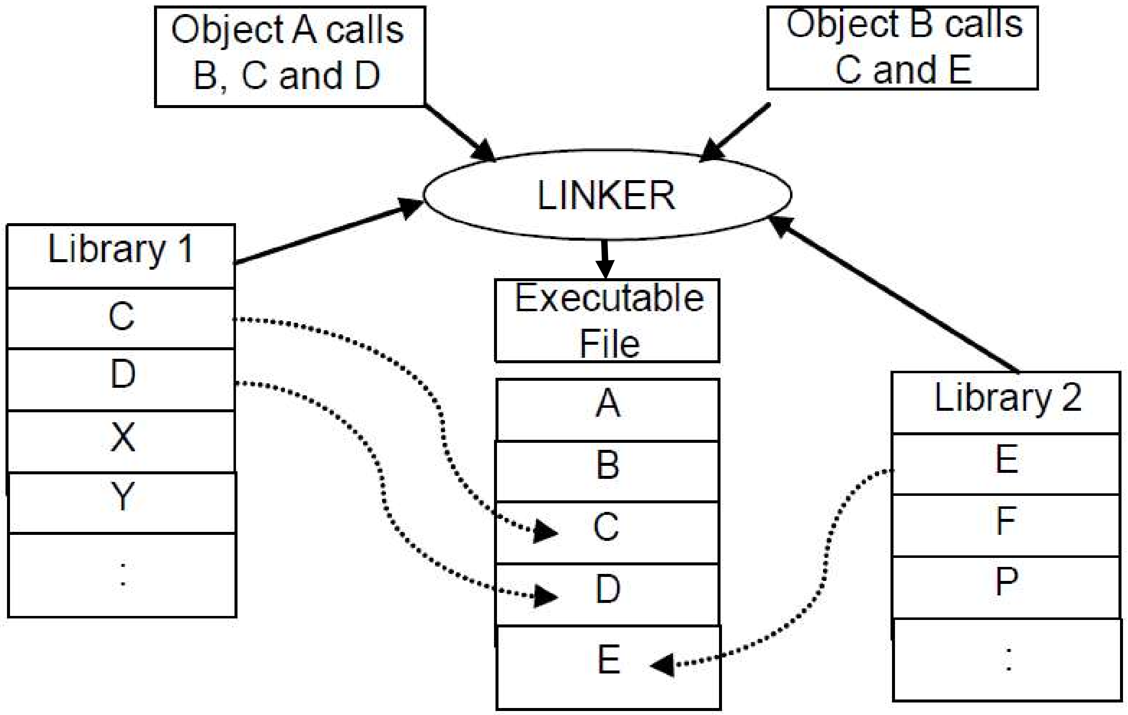
symbols

```
main     T[0]
_s1      D[0]
_s2      D[14]
_s3      D[17]
```

relocation

```
printf   T[30]
printf   T[86]
scanf    T[52]
sin      T[60]
_s1      T[24]
_s2      T[54]
_s3      T[80]
```

Use of Object Code Library



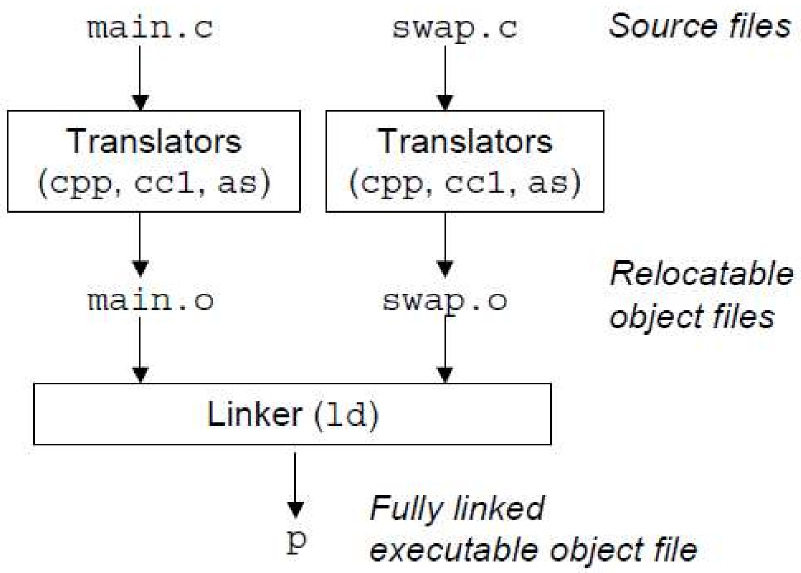
Static and Dynamic Linking

- Static linking
  - Big executable files (all/most of needed libraries inside)
  - Don’t benefit from updates to library
  - No load-time linking

- Dynamic linking
  - Small executable files (just point to shared library)
  - Library update benefits all programs that use it
  - Load-time cost to do final linking
    - But dll code is probably already in memory
    - And can do the linking incrementally, on-demand

Static Linking

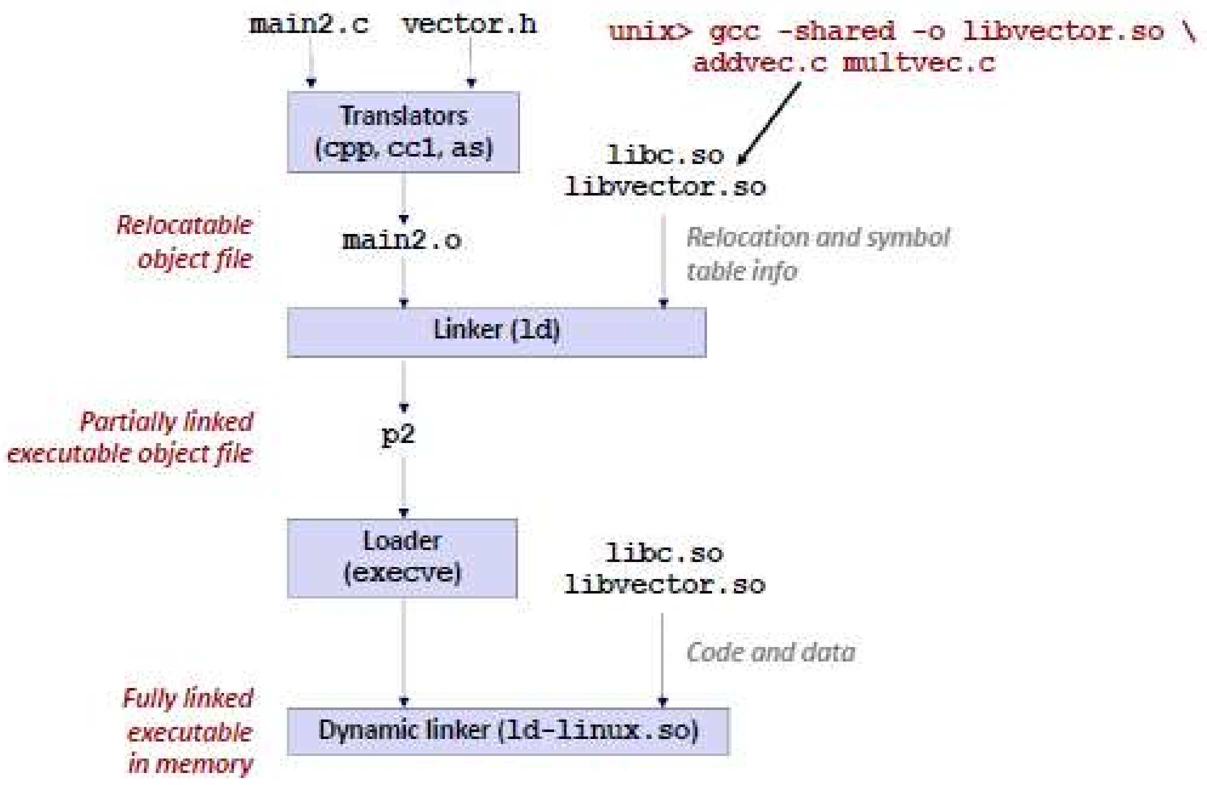
- The linker combines relocatable object files to form an executable object file.


main.c
swap.c
Source files
Translators (cpp, cc1, as)
Translators (cpp, cc1, as)
main.o
swap.o
Relocatable object files
Linker (ld)
p
Fully linked executable object file

Dynamic Linking

- A subroutine is loaded and linked to the rest of the program when it is first called
- Often used to allow several executing programs to share one copy of a subroutine or library
- Provides the ability to load the routines only when they are needed
- When dynamic linking is used, the binding of the name to an actual address is delayed from load time until execution time

Dynamic Linking



main2.c
vector.h
unix> gcc -shared -o libvector.so \
addvec.c multvec.c
Translators
(cpp, cc1, as)
libc.so
libvector.so
Relocatable object file
main2.o
Relocation and symbol table info
Linker (ld)
Partially linked executable object file
p2
Loader
(execve)
libc.so
libvector.so
Code and data
Fully linked executable in memory
Dynamic linker (ld-linux.so)

Linking Algorithm (read yourself!)

- Fundamentally a two stage process (like assembler)
  - Should be two pass
  - One pass is also possible (more complex conceptually and implementation-wise)

Question

- Who loads the first loader???

Type of Loaders (read yourself!)

- Bootstrap loader
- Compile and Go loader
- Absolute loader
- Dynamic linking and loading

Reference

- ➢ Book: J R Levine, Linkers & Loaders.
- ➢ Book: J J Donovan, Systems Programming (Chapter 5)
- ➢ Book: L L Beck & D Manjula, Systems Software (Chapter 3).