Real-Time and Embedded Systems @ SIT

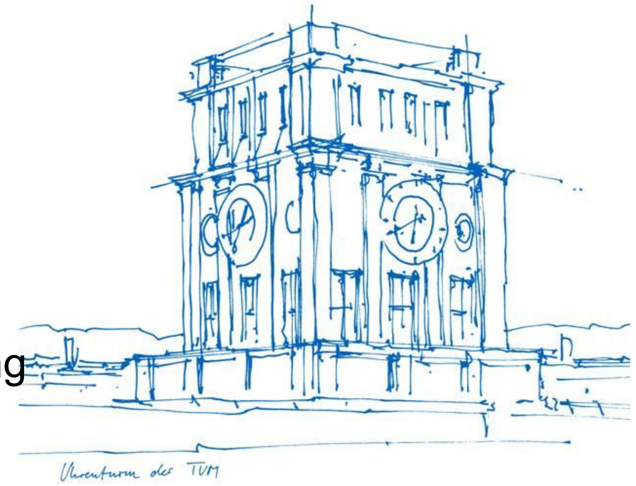
Compiling Code

Alexander Hoffman

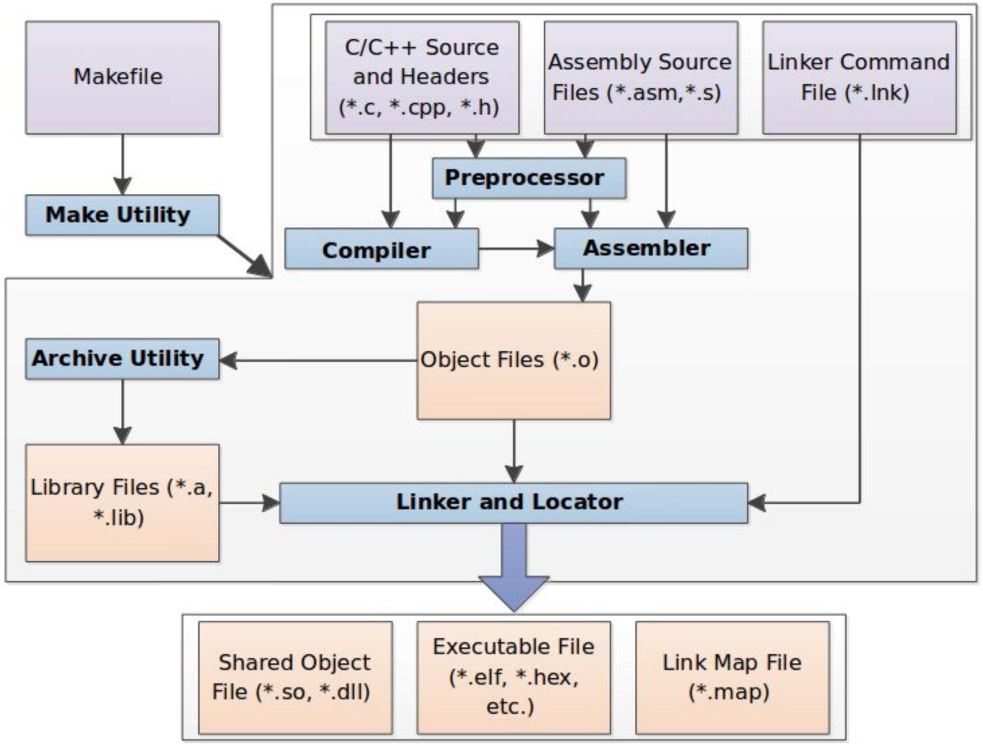
Technical University of Munich

Department of Electrical and Computer Engineering

Chair of Real-Time Computer Systems

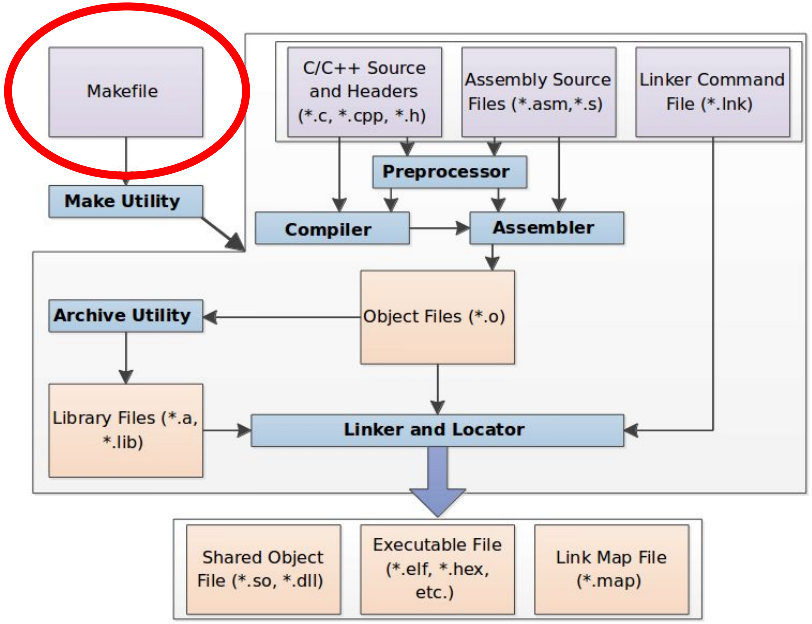


How is your code compiled?!?!


Makefile
Make Utility
C/C++ Source and Headers (*.c, *.cpp, *.h)
Assembly Source Files (*.asm,*.s)
Linker Command File (*.lnk)
Preprocessor
Compiler
Assembler
Object Files (*.o)
Archive Utility
Library Files (*.a, *.lib)
Linker and Locator
Shared Object File (*.so, *.dll)
Executable File (*.elf, *.hex, etc.)
Link Map File (*.map)

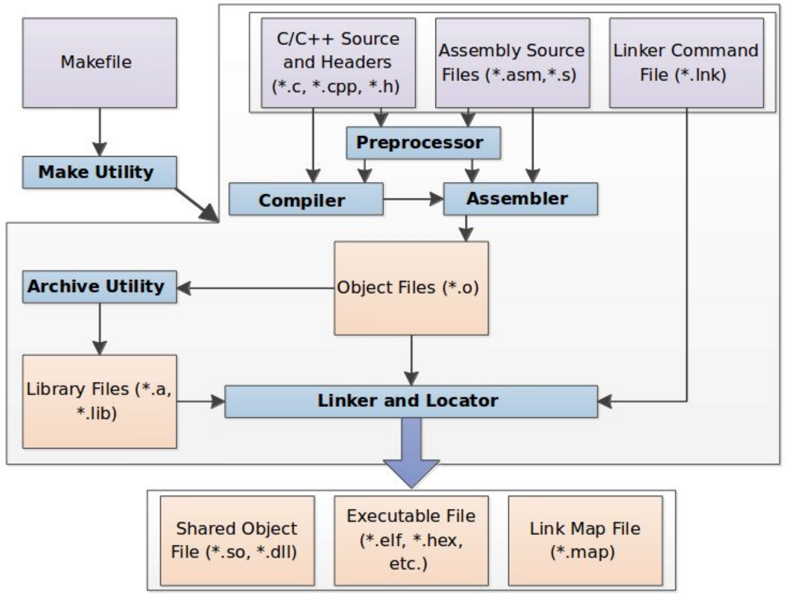
The build “helper”

- Makefiles (Ie. GNU Make scripts) automate the building process
- Easier to build large and complex projects
- Outdated*
    - More common to use programs nowadays such as CMake to generate Makefiles instead of directly writing them


Makefile
C/C++ Source and Headers (*.c, *.cpp, *.h)
Assembly Source Files (*.asm,*.s)
Linker Command File (*.lnk)
Preprocessor
Make Utility
Compiler
Assembler
Archive Utility
Object Files (*.o)
Library Files (*.a, *.lib)
Linker and Locator
Shared Object File (*.so, *.dll)
Executable File (*.elf, *.hex, etc.)
Link Map File (*.map)

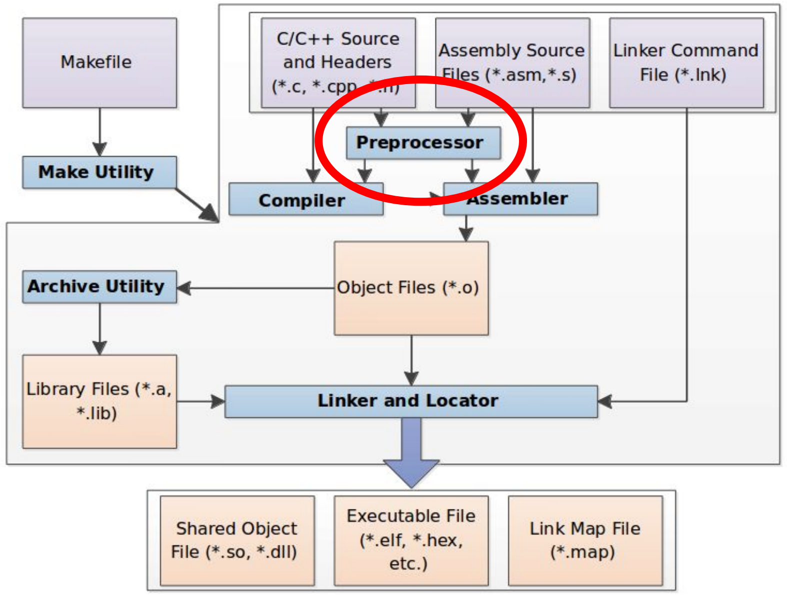
“Compilation” Steps

- The entire build process is often called “compiling” although compilation is only one step
- Steps:
    - Preprocessor
    - Compiler
    - Assembler
    - Linker


Makefile
C/C++ Source and Headers (*.c, *.cpp, *.h)
Assembly Source Files (*.asm,*.s)
Linker Command File (*.lnk)
Make Utility
Preprocessor
Compiler
Assembler
Object Files (*.o)
Archive Utility
Library Files (*.a, *.lib)
Linker and Locator
Shared Object File (*.so, *.dll)
Executable File (*.elf, *.hex, etc.)
Link Map File (*.map)

“Compilation” Steps

- The entire build process is often called “compiling” although compilation is only one step
- Steps:
  - Preprocessor
  - Compiler
  - Assembler
  - Linker

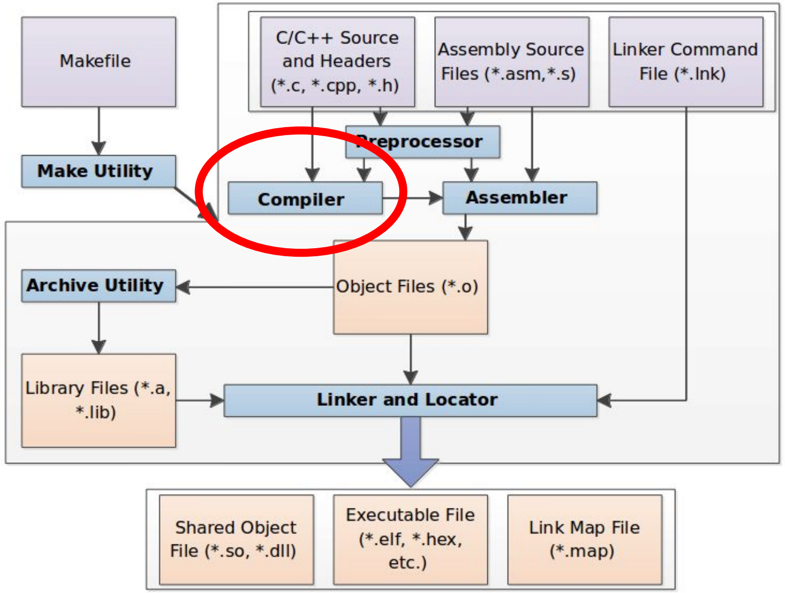

Makefile
Make Utility
C/C++ Source and Headers (*.c, *.cpp, *.h)
Assembly Source Files (*.asm,*.s)
Linker Command File (*.lnk)
Preprocessor
Compiler
Assembler
Object Files (*.o)
Archive Utility
Library Files (*.a, *.lib)
Linker and Locator
Shared Object File (*.so, *.dll)
Executable File (*.elf, *.hex, etc.)
Link Map File (*.map)

Preprocessor

- Processes "#" directives to translate source files before compilation
  - `#include`
  - `#define`
  - `#ifdef`
- Can be thought of as text processing/translation of source files
- `if-else` directives used for conditional statements, usually to control what does and does not get compiled
- Includes allow for the "copying and pasting" of the specified file into the source file, providing functions prototypes etc from library files
- Macros and defines, allow for human readable code, code generation and easily modifiable code
- Generates ASCII intermediate files, Eg. `main.i`, that is still C code

“Compilation” Steps

- The entire build process is often called “compiling” although compilation is only one step
- Steps:
    - Preprocessor
    - Compiler
    - Assembler
    - Linker

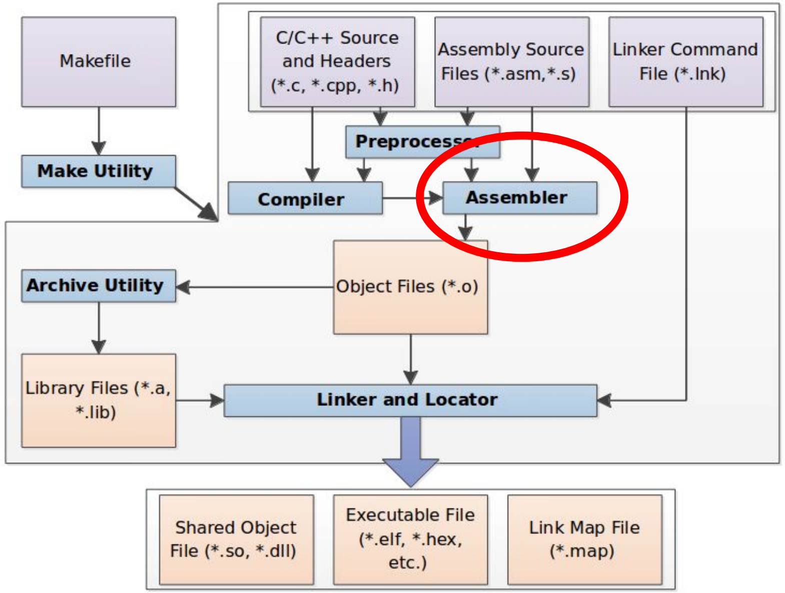

Makefile
C/C++ Source and Headers (*.c, *.cpp, *.h)
Assembly Source Files (*.asm,*.s)
Linker Command File (*.lnk)
Preprocessor
Make Utility
Compiler
Assembler
Archive Utility
Object Files (*.o)
Library Files (*.a, *.lib)
Linker and Locator
Shared Object File (*.so, *.dll)
Executable File (*.elf, *.hex, etc.)
Link Map File (*.map)

Compiler

- Turns the ASCII C code into ASCII assembly file, Eg. main.s
- Performs:
    - Lexical analysis: breaking up of code into **tokens**, creates “derivations”
    - Semantic analysis: derivations are checked for errors
    - Intermediate code generation: creates a representation that is more easily compiled into assembly, Eg. `a + b * c` becomes `b * c` followed by `a + result`
    - Intermediate code optimization: Makes code more efficient
    - Object code generation: Memory locations chosen for variables and instructions are chosen for each operation
    - Object code optimization: Makes assembly code more efficient
    - **Symbol table** stores a “map” that linker can use to know where what is

“Compilation” Steps

- The entire build process is often called “compiling” although compilation is only one step
- Steps:
  - Preprocessor
  - Compiler
  - Assembler
  - Linker

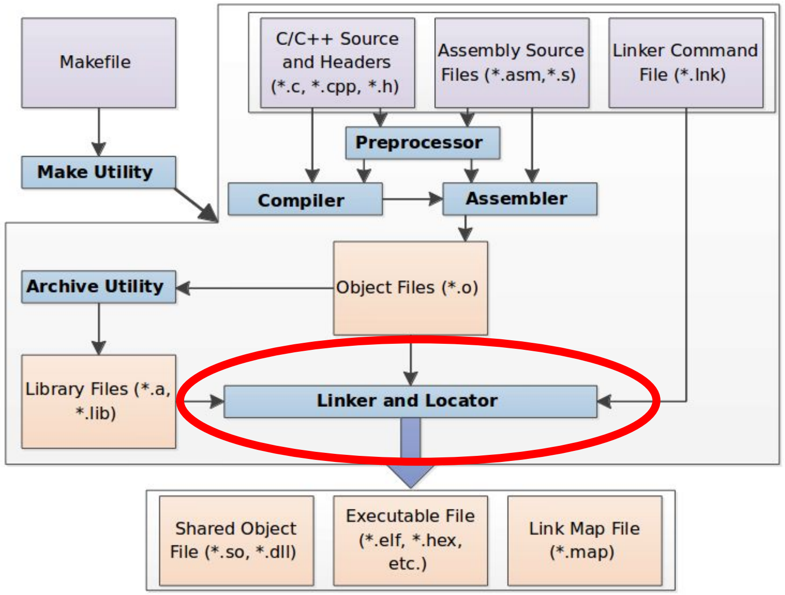

Makefile
C/C++ Source and Headers (*.c, *.cpp, *.h)
Assembly Source Files (*.asm,*.s)
Linker Command File (*.lnk)
Make Utility
Preprocessor
Compiler
Assembler
Archive Utility
Object Files (*.o)
Library Files (*.a, *.lib)
Linker and Locator
Shared Object File (*.so, *.dll)
Executable File (*.elf, *.hex, etc.)
Link Map File (*.map)

Assembler

- Creates machine code object files (`*.o`) from the ASCII assembly file (`*.s`)
- Gives memory offsets to each memory location
- Makes a list of all unresolved references
  - Functions calls from other source files
  - Presumably defined in other object file, Eg. `printf`
- Also compiled debugging info if required
  - Allows for ASCII labels on memory offsets

“Compilation” Steps

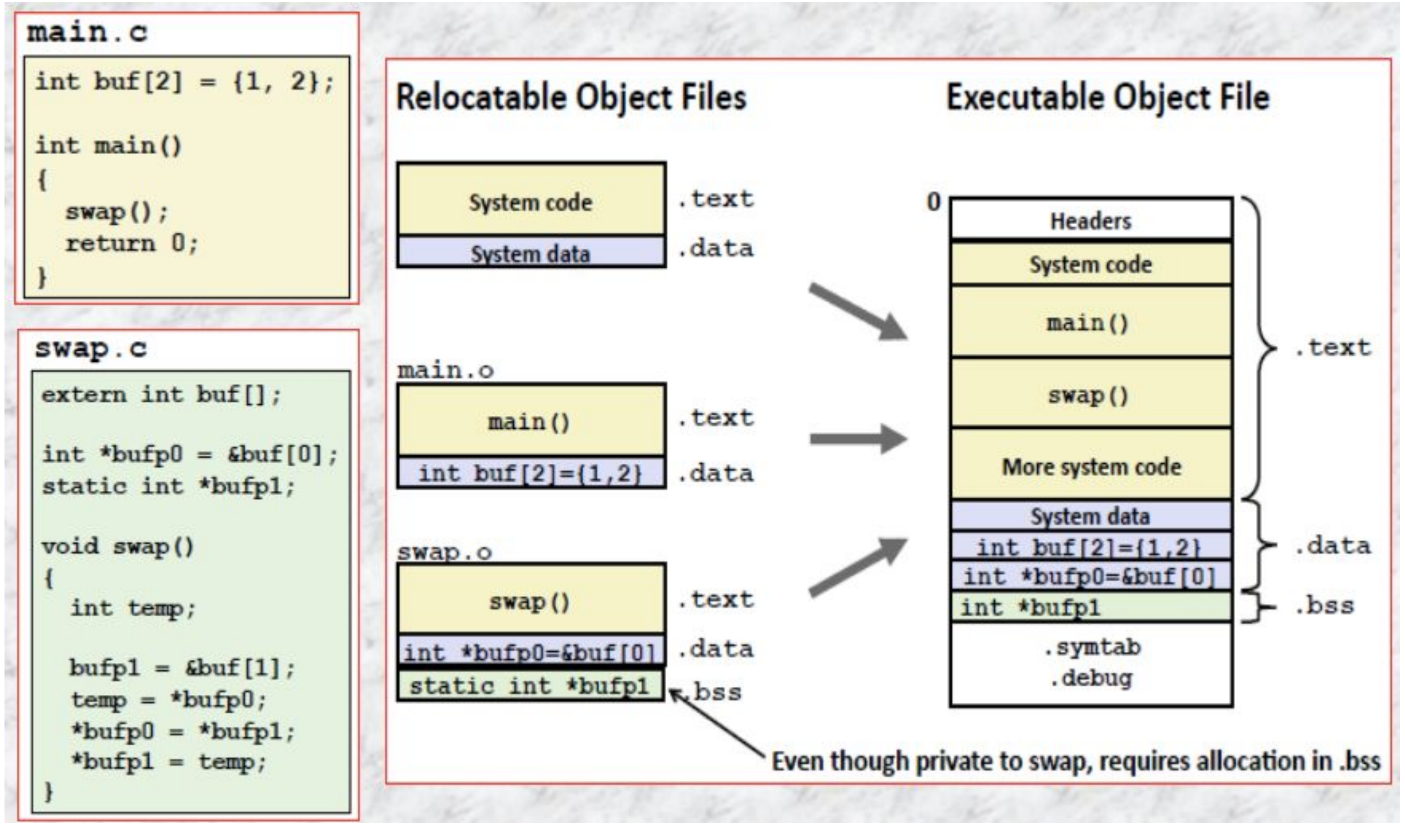
- The entire build process is often called “compiling” although compilation is only one step
- Steps:
  - Preprocessor
  - Compiler
  - Assembler
  - Linker


Makefile
C/C++ Source and Headers (*.c, *.cpp, *.h)
Assembly Source Files (*.asm,*.s)
Linker Command File (*.lnk)
Preprocessor
Make Utility
Compiler
Assembler
Archive Utility
Object Files (*.o)
Library Files (*.a, *.lib)
Linker and Locator
Shared Object File (*.so, *.dll)
Executable File (*.elf, *.hex, etc.)
Link Map File (*.map)

Linker

- Our unresolved references need to be resolved!
- Takes an input of object files and generates a “fully linked” executable object file (Eg. `*.elf`) that can be loaded and run
- This achieves:
    - Symbols are resolved, Ie. symbol references are associated with exactly one symbol definition
        - Common error is having multiple definitions of a function, thus creating two of the same symbol definitions
    - Relocation of memory offsets such that the generated code and data is all in the same memory space, starting at address 0
        - These are the run-time address offsets

Linker



main.c

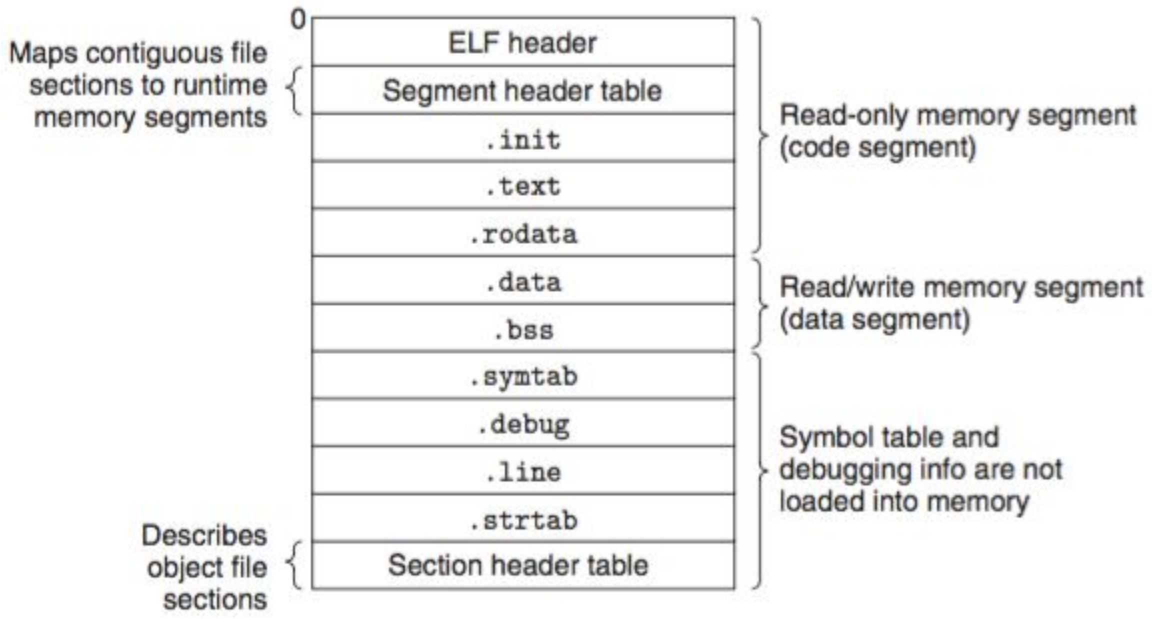
```
int buf[2] = {1, 2};

int main()
{
  swap();
  return 0;
}
```

swap.c

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

[https://people.cs.pitt.edu/~xianeizhang/notes/Linking.html]

Executable Object File


0
Maps contiguous file sections to runtime memory segments
Describes object file sections
ELF header
Segment header table
.init
.text
.rodata
.data
.bss
.symtab
.debug
.line
.strtab
Section header table
Read-only memory segment (code segment)
Read/write memory segment (data segment)
Symbol table and debugging info are not loaded into memory

[https://people.cs.pitt.edu/~xianeizhang/notes/Linking.html]