Compilation

William Hendrix

Lecture 23

Today

- Machine code
- Programming languages
- Compilation in C/C++
- Preprocessor directives
- `goto`

Machine code

- Binary instructions that are actually executed by the computer
  - Instruction set depends on architecture
- **Example:** `0x401046: 03 45 08`
  - Adds 8 to the value in register `%ebp` and stores the result in register `%eax`
- Instructions encode for operations like arithmetic, branching (control flow/loops), store/load (from main memory), and jumps (e.g., function return)
- When an executable is running, instructions are loaded into memory
  - Also loads program data, like constant strings
- Program counter (instruction pointer) register keeps track of where we are in execution of program
  - Advances by one instruction each time, except when branching, looping, function return, etc.
- Machine code can be interpreted quickly by CPU, but essentially incomprehensible to humans

Assembly

- Lowest level programming language
- Defines a set of mnemonics for machine instructions
  - May vary by architecture
- **Example**

```
addl 8 (%ebp), %eax
```

- Improves readability for people
- Straightforward (one-to-one) conversion into machine code
- Assembler: program that translates assembly into machine code
- Disassembler: program that translates machine code back into assembly
- Assembly can be directly added to C/C++ programs with `__asm__` keyword
  - **Example**

```
    __asm__ volatile {"rdtsc": "=a"(low) "=d"(high)}
```

Higher-level programming languages

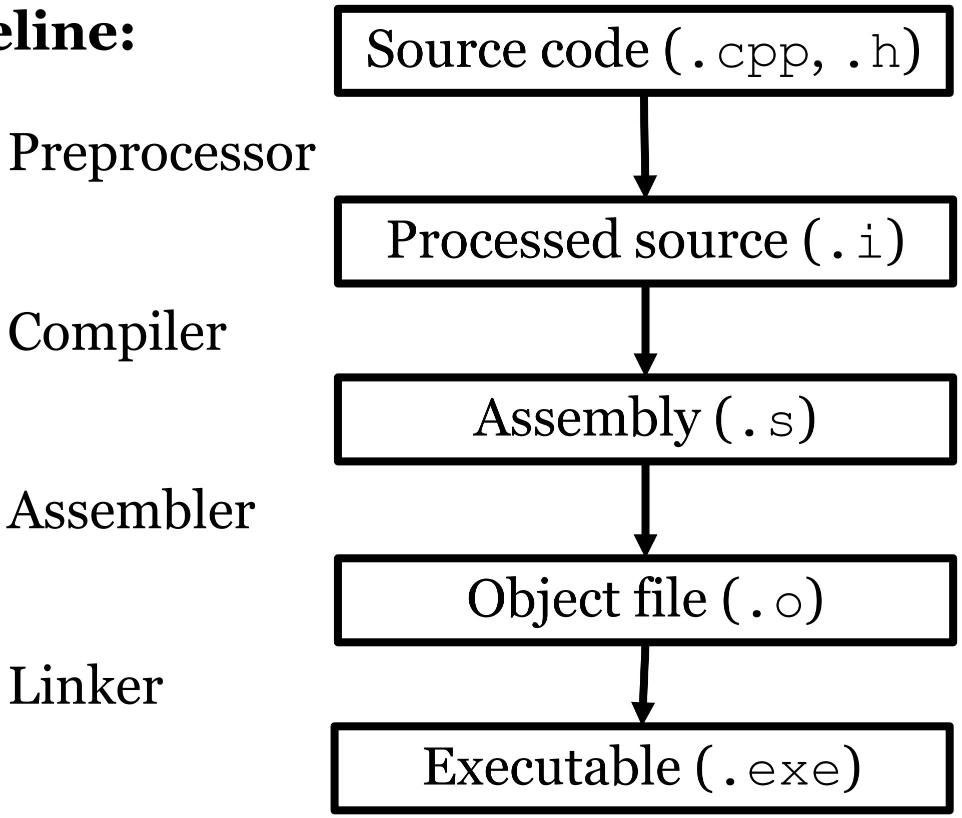
- Languages that can be converted into assembly and machine code
  - Each language defines its own syntax and semantics
  - **Examples:** C++, Java, Python, FORTRAN, Lisp, Ruby, Lua, …
  - Different languages may have different strengths/weaknesses
- Higher-level languages hide the details of assembly from the programmer
  - Simplify tasks
  - Improve productivity
- Compiler
  - Program that transforms code in a higher-level programming language into assembly
  - Checks code to ensure that it follows language syntax
  - May print errors or warnings

Linking

- Program that “binds” function calls to the function implementation
- Each source file (`.cpp`) is compiled to a separate *object file* (`.o`)
  - Object files are machine code with information about functions they define
- Function calls in object files are “linked” with the functions they call
  - Includes calls to built-in functions
  - Linker produces the final executable or library
  - Throws an error if it cannot find a function
    - “Unresolved symbol”
- Static linking (`.a`)
  - Code for library functions inserted directly into library/executable
- Dynamic linking (`.so`, `.dll`)
  - Stores offsets from library file
  - Library file must be present for the code to run
  - Executable size is much smaller (avoids replication)
  - Execution is slightly slower
  - Potential issues with different library versions

Compilation in C/C++

- C/C++ have an additional step before compilation
- Preprocessor
  - Modifies code before compilation (and syntax checking)
  - Controlled by # directives
    - E.g., `#include “menagerie.h”`
- **Compilation pipeline:**



Preprocessor directives

- `#include`: pastes contents of header file at this location

- `#define`: defines symbols or macro functions that are replaced before compilation

```
#define SECONDS_PER_MINUTE 60
```

- `#undef`: "undefines" a symbol

```
#undef SECONDS_PER_MINUTE
```

- `#line`: replaced by line number

- `#if, #ifdef, #ifndef, #elif, #else, #endif`
  - Act like `if, else if, else`
  - Code is removed if condition fails
  - Commonly used with `#define DEBUG`

Macro functions

- Function-like symbols that can insert a parameterized block of code
- **Warning:** can make your code more difficult to read and use
- **Example**

```
#define print_array(arr, len) \
    for (int i = 0; i < (len); i++) \
    cout << (arr)[i] << ' ';
```

- "Implementation" appears on the rest of the line
- Preprocessor replaces all "calls" to macro functions with function body
  - Inserts passed "parameters" in code
    - Expressions (e.g., `arr + 10`) just get pasted directly into code
    - Use parentheses around parameters
- Does not invoke a true function call
- Can cause syntax problems due to pasting behavior
  - E.g., calling macro inside `if` condition

Goto statement

- Statement that should never appear in your code
- “Jumps” to a label and continues execution of the program
- Label
  - Denoted by a name followed by :
- **`goto` example:**

```
int i = 0;
loop_begin:
total += arr[i];
i++;
if (i < len)
  goto loop_begin;
```

- “Go to statement considered harmful” (Dijkstra, 1968)
  - Made the case against `goto`
    - Replace with loops, functions, “structured” programming
  - Overuse of `goto` results in “spaghetti code” that is hard to follow
  - Makes testing/debugging difficult because you can jump to a label at any point in the code

Tonight

- **Lab 5** due Monday
- **Recommended reading:** Sections 12.1-3