Compilation, Disassembly, and Profiling
(in Linux)

CS 485: Systems Programming
Fall 2015

Instructor:
James Griffioen
Recall the compilation process/steps
Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -O1 p1.c p2.c -o p`
  - Use basic optimizations (`-O1`)
  - Put resulting binary in file `p`

```
text
  C program (p1.c p2.c)
  Compiler (gcc -S)
  Asm program (p1.s p2.s)
  Assembler (gcc or as)
  Object program (p1.o p2.o)
  Linker (gcc or ld)
  Executable program (p)
  Static libraries (.a)
```
Turning C into Object Code

- Code in files `p1.c` `p2.c`
- Compile with command: `gcc -O1 p1.c p2.c -o p`
  - Use basic optimizations (`-O1`)
  - Put resulting binary in file `p`

Can stop the compilation process at any stage.

<table>
<thead>
<tr>
<th>Text</th>
<th>C program (<code>p1.c</code> <code>p2.c</code>)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compiler (<code>gcc -S</code>)</td>
</tr>
<tr>
<td>Text</td>
<td>Asm program (<code>p1.s</code> <code>p2.s</code>)</td>
</tr>
<tr>
<td></td>
<td>Assembler (<code>gcc or as</code>)</td>
</tr>
<tr>
<td>Binary</td>
<td>Object program (<code>p1.o</code> <code>p2.o</code>)</td>
</tr>
<tr>
<td></td>
<td>Linker (<code>gcc or ld</code>)</td>
</tr>
<tr>
<td>Binary</td>
<td>Executable program (<code>p</code>)</td>
</tr>
</tbody>
</table>
Invoking the compiler

- **We will use the Gnu command line compilers**
  - gcc – compiles C programs
  - g++ - compiles C++ programs (and C programs)

- **Useful command line options**
  - **-o** filename
    - Defines the output filename
    - Example: gcc -o hello hello.c
    - will create an executable file named “hello”
  - **-E**
    - Preprocess only – (the same as running the cpp program)
    - Example: gcc -E hello.c > hello.i
    - Will run the preprocessor and process header files to create “hello.i”
  - **-c**
    - Create an object file (.o) – i.e. Compile/Assemble, but do not link
    - Example: gcc -c hello.c
    - Will create an object file called hello.o
  - **-S**
    - Create an assembly language file (.s) – i.e., Compile, but do not assemble
    - Example: gcc -S hello.c
    - Will create an assembly language file called “hello.s”
Turning C into Object Code

- Code in files `p1.c` `p2.c`
- Compile with command: `gcc -O1 p1.c p2.c -o p`
  - Use basic optimizations (`-O1`)
  - Put resulting binary in file `p`

![Diagram showing the compilation process from C program to executable program.](image)

- **Information is lost at each step of the compilation Process.**

- **Static libraries (.a)**

- **C program (p1.c p2.c)**
  - **Compiler (gcc -S)**

- **Asm program (p1.s p2.s)**
  - **Assembler (gcc or as)**

- **Object program (p1.o p2.o)**
  - **Linker (gcc or ld)**

- **Executable program (p)**
Compiling Into Assembly

C Code

```c
int sum(int x, int y)
{
    int t = x+y;
    return t;
}
```

Generated IA32 Assembly

```assembly
sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    popl %ebp
    ret
```

For example:
Variable names are lost
Parameters are lost

Some compilers use instruction “leave”

Obtain with command

```
/usr/local/bin/gcc -O1 -S code.c
```

Produces file `code.s`
Object Code

Code for sum

0x401040 <sum>:
  0x55
  0x89
  0xe5
  0x8b
  0x45
  0x0c
  0x03
  0x45
  0x45
  0x08
  0x5d
  0xc3

- Total of 11 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address 0x401040

Just binary bytes. Most assembly language is lost

- **Assembler**
  - Translates .s into .o
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files

- **Linker**
  - Resolves references between files
  - Combines with static run-time libraries
    - E.g., code for malloc, printf
  - Some libraries are dynamically linked
    - Linking occurs when program begins execution
Invoking the compiler

- **We will use the Gnu command line compilers**
  - gcc – compiles C programs
  - g++ - compiles C++ programs (and C programs)

- **Useful command line options**
  - `-o filename`
    - Defines the output filename
    - Example: `gcc -o hello hello.c`
      - will create an executable file named “hello”
  - `-E`
    - Preprocess only – (the same as running the cpp program)
    - Example: `gcc -E hello.c > hello.i`
      - Will run the preprocessor and process header files to create “hello.i”
  - `-c`
    - Create an object file (.o) – i.e. Compile/Assemble, but do not link
    - Example: `gcc -c hello.c`
      - Will create an object file called hello.o
  - `-S`
    - Create an assembly language file (.s) – i.e., Compile, but do not assemble
    - Example: `gcc -S hello.c`
      - Will create an assembly language file called “hello.s”
  - `-g`
    - Add symbol information to the file
    - Useful when debugging and disassembling programs
  - `-p`
    - Add profiling information to the file
    - Useful when profiling performance
  - `-Olevel`
    - Optimize the code using the specified level’s optimizations
    - *Level 0* is the fewest optimizations, *Level 3* is the most optimizations
    - Example: `gcc -O3 hello.c`
Disassembling Code

- There are a variety of tools that can be used to look at compiled code.
- Some are useful for seeing the assembly language code/instructions
  - objdump -d
  - gdb – using the disassemble command
- Some provide information about the data/variables
  - nm
- Some are basics tools that can give hints about what is in the file
  - strings
  - od
- Some are graphical front ends
  - dissy
Disassembling Object Code

Disassembled

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>080483c4</td>
<td>55</td>
<td>push %ebp</td>
</tr>
<tr>
<td>080483c5</td>
<td>89 e5</td>
<td>mov (%ebp),%esp,%ebp</td>
</tr>
<tr>
<td>080483c7</td>
<td>8b 45 0c</td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>080483ca</td>
<td>03 45 08</td>
<td>add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td>080483cd</td>
<td>5d</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>080483ce</td>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>

- **Disassembler**
  - `objdump -d filename`
  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either a.out (complete executable) or .o file
Alternate Disassembly

- **Within gdb Debugger**
  - First run "gdb filename"
  - Then inside gdb type "disassemble sum"
    - Disassemble procedure
  - x/11xb sum
    - Examine the 11 bytes starting at sum
Invoking the compiler

- **We will use the Gnu command line compilers**
  - gcc – compiles C programs
  - g++ - compiles C++ programs (and C programs)

- **Useful command line options**
  - `-o` filename
    - Defines the output filename
    - Example: `gcc –o hello hello.c`
    - will create an executable file named “hello”
  - `-E`
    - Preprocess only – (the same as running the cpp program)
    - Example: `gcc –E hello.c > hello.i`
    - Will run the preprocessor and process header files to create “hello.i”
  - `-c`
    - Create an object file (.o) – i.e. Compile/Assemble, but do not link
    - Example: `gcc –c hello.c`
    - Will create an object file called hello.o
  - `-S`
    - Create an assembly language file (.s) – i.e., Compile, but do not assemble
    - Example: `gcc –S hello.c`
    - Will create an assembly language file called “hello.s”
  - `-g`
    - Add symbol information to the file
    - Useful when debugging and disassembling programs
  - `-p`
    - Add profiling information to the file
    - Useful when profiling performance
  - `-O level`
    - Optimize the code using the specified level’s optimizations
    - *Level 0* is the fewest optimizations, *Level 3* is the most optimizations
    - Example: `gcc –O3 hello.c`
Profiling Code

- Compile with the –pg option to gcc
- gprof – commonly installed and used profiling tool for unix-based systems
- Valgrind – more advanced tool that also comes with graphical user interfaces to visualize a program’s performance and call graph
Gprof concepts

- **Step 1: Add profiling information to the program**
  - gcc –pg –o myprog myprog.c

- **Step 2: Run the program to create *gmon.out* (profile info)**
  - ./myprog

- **Step 3: Analyze the performance information**
  - View time spent in each procedure
    - gprof –p ./myprog
  - View call graph
    - gprof –q ./myprog
Invoking the compiler

- **We will use the Gnu command line compilers**
  - gcc – compiles C programs
  - g++ - compiles C++ programs (and C programs)

- **Useful command line options**
  - `-o` filename
    - Defines the output filename
    - Example: `gcc -o hello hello.c`
      - will create an executable file named “hello”
  - `-E`
    - Preprocess only – (the same as running the cpp program)
    - Example: `gcc -E hello.c > hello.i`
      - Will run the preprocessor and process header files to create “hello.i”
  - `-c`
    - Create an object file (.o) – i.e. Compile/Assemble, but do not link
    - Example: `gcc -c hello.c`
      - Will create an object file called hello.o
  - `-S`
    - Create an assembly language file (.s) – i.e., Compile, but do not assemble
    - Example: `gcc -S hello.c`
      - Will create an assembly language file called “hello.s”
  - `-g`
    - Add symbol information to the file
    - Useful when debugging and disassembling programs
  - `-p`
    - Add profiling information to the file
    - Useful when profiling performance
  - `-Olevel`
    - Optimize the code using the specified level’s optimizations
    - *Level 0* is the fewest optimizations, *Level 3* is the most optimizations
    - Example: `gcc -O3 hello.c`