CSC2/452 Computer Organization
Introduction

Sreepathi Pai

URCS

August 28, 2019
Outline

Introduction

The Long Journey to an Executable

Running a Program

What about Python, Java and JavaScript?

Administrivia
Outline

Introduction

The Long Journey to an Executable

Running a Program

What about Python, Java and JavaScript?

Administrivia
\#include <stdio.h>

int main(void) {
    float PI = 3.1415926535897932384626433832;

    printf("Hello, the value of pi is \%f\n", PI);

    return 0;
}

Behold, a Program
To compile the code:

$ gcc -g hellopi.c

(The $ indicates the Unix shell prompt and is not part of the command.)

The output is an executable binary. You can run it as:

$ ./a.out
   Hello, the value of pi is 3.141593
Inside a.out

$ hexdump -C a.out

...  

00000690  20 00 55 48 8d 2d 26 07  20 00 53 41 89 fd 49 89  | .UH.-&. .SA..I.|
000006a0  f6 4c 29 e5 48 83 ec 08  48 c1 fd 03 e8 3f fe ff  | .L).H...H.....?..|
000006b0  ff 48 85 ed 74 20 31 db  0f 1f 84 00 00 00 00 00  | .H..t 1..........|
000006c0  4c 89 fa 48 89 f6 44 89  ef 41 ff 14 dc 48 83 c3  | L..L..D..A...H..|
000006d0  01 48 39 dd 75 ea 48 83  c4 08 5b 5d 41 5c 41 5d  | .H9.u.H...[]A\A|
000006e0  41 5e 41 5f c3 90 66 0f 1f  84 00 00 00 00 00 00 00  | A^A_...A..f......|
000006f0  f3 c3 00 00 48 83 ec 08  48 83 c4 08 c3 00 00 00  | ....H...H.........|
00000700  01 00 02 00 48 65 6c 6f  ef 41 ff 14 dc 48 83 c3  | ....Hello, the v|
00000710  61 6c 75 65 20 70 69 20 69 73 20 25 66 2e 0f 1f  | alue of pi is %f|

...  

This is a *hexdump*, a convenient way of viewing binary data. Four columns on each line:

- Offset (in file, 0: beginning)
- 8 bytes of data in hexadecimal
- 8 bytes of data in hexadecimal
- 16 bytes of data, printable as-is, while non-printable is ‘.’
Outline

Introduction

The Long Journey to an Executable

Running a Program

What about Python, Java and JavaScript?

Administrivia
gcc is a *compiler driver*

It orchestrates the execution of many programs:

- the C preprocessor (cpp)
- the C compiler proper (cc)
- the assembler (as)
- the (static) linker (ld)

You can invoke these programs individually

But it's more convenient to let gcc drive them

```
input.c
  ▼
  cpp
  ▼
  cc
  ▼
  as
  ▼
  ld
  ▼
a.out
```
The Preprocessor (for C programs)

$ cpp hellopi.c > hellopi.i
... some 700+ lines not shown ...

int main(void) {
    float PI = 3.1415926535897932384626433832;

    printf("Hello, the value of pi is \%f\n", PI);

    return 0;
}

- The preprocessor handles lines starting with # (e.g. #include).
- Output is a C file without #include
  - and other preprocessor directives
- The 700 plus lines not shown above come from #include <stdio.h>
$ cc -S hellopi.c

(The -S forces cc to produce assembly code instead of a binary)

.file "hellopi.c"
.text
.section .rodata
.LC1:
.string "Hello, the value of pi is %f\n"
.text
.globl main
.type main, @function
main:
.LFB0:
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movss .LC0(%rip), %xmm0
movss %xmm0, -4(%rbp)
cvtss2sd -4(%rbp), %xmm0
leaq .LC1(%rip), %rdi
movl $1, %eax
call printf@PLT
movl $0, %eax
leave
ret
.section .rodata
.align 4
.LC0:
.long 1078530011
How to Read Assembly Language Programs

- Assembly language programs are line-based.
- Lines beginning with a ‘.’ are assembler *directives*.
  
  ```
  .file "hellopi.c"
  .text
  .section .rodata
  .LC1:
      .string "Hello, the value of pi is %f
"
  ```

- Lines ending with a ‘:’ are convenience labels for addresses (e.g. `.LC1:` above)
  
  - I.e. the assembler converts them to addresses so you don’t have to
  - Note that directives DO NOT end with a ‘:’, nor do labels have to start with a ‘.’ (e.g. `main`)

- All other lines specify instructions.
Instructions in x86 assembly

- Different assemblers have different syntax for the same CPU instruction
- as uses AT&T syntax
  - Source operands are at beginning
  - Destination operands are at end

```
subq $16, %rsp
movss .LC0(%rip), %xmm0
```

- Operands follow
  - Constant operands: $16
  - Register operands: %rsp, %xmm0, %rip
  - Address operands: .LC0(%rip)

- Different processors will have different instruction sets
  - x86, ARM, PowerPC, RISC-V, etc.
  - Assembly program for one processor will not run on another processor
The Assembler

- The assembler converts the output of the compiler to an object file (machine code).
  - Object files are not directly executable
- Not very friendly to view, so let’s look at a listing file
  - Obtained using `as -adhln hellopi.s`
An assembly listing for hellopi.c

```
1 .file "hellopi.c"
2 .text
3 .section .rodata
4 .LC1:
5 0000 48656C6C .string "Hello, the value of pi is %f\n"
5 6F2C2074
5 68652076
5 616C7565
5 206F6620
6 .text
7 .globl main
9 main:
10 .LFB0:
11 0000 55 pushq %rbp
12 0001 4889E5 movq %rsp, %rbp
13 0004 4883EC10 subq $16, %rsp
14 0008 F30F1005 movss .LC0(%rip), %xmm0
14 00000000
15 0010 F30F1145 movvss %xmm0, -4(%rbp)
15 FC
16 0015 F30F5A45 cvtss2sd -4(%rbp), %xmm0
16 FC
17 001a 488D3D00 leaq .LC1(%rip), %rdi
17 00000000
...```
Object files

- Usually an object file correspond to a single C file
  - Multiple C files lead to multiple object files
- Object files are incomplete
- One: some data addresses are unknown
  - The 000000 indicate space left for an address to be filled in later
  - Final executable binary decides placement of multiple object files

14 0008 F30F1005 movss .LC0(%rip), %xmm0
14 00000000

- Two: Some functions may live in different C files and so their addresses are unknown

23 0026 E8000000 call printf@PLT
23 00
The (Static) Linker

$ ld hellopi.o ...

- The static linker combines all the object files to form a single binary
  - static means compile-time
  - there is also a dynamic linker which is used at run-time
- It fills in nearly all addresses
  - the ones it leaves unfilled are filled by the dynamic linker and loader just before the program runs
  - this happens if a function is in a shared object (similar to Windows DLLs)
- Shared objects are objects that are common to all programs on the system
  - you can avoid shared objects, but your binary size will increase
Disassembler

$ objdump -S a.out

...  
0000000000000064a <main>:
  64a: 55 push %rbp  
  64b: 48 89 e5 mov %rsp,%rbp  
  64e: 48 83 ec 10 sub $0x10,%rsp  
  652: f3 0f 10 05 ca 00 00 movss 0xca(%rip),%xmm0  
  659: 00  
  65a: f3 0f 11 45 fc movss %xmm0,-0x4(%rbp)  
  65f: f3 0f 5a 45 fc cvtss2sd -0x4(%rbp),%xmm0  
  664: 48 8d 3d 99 00 00 00 lea 0x99(%rip),%rdi  
  66b: b8 01 00 00 00 mov $0x1,%eax  
  670: e8 ab fe ff ff callq 520 <printf@plt>  
  675: b8 00 00 00 00 mov $0x0,%eax  
  67a: c9 leaveq  
  67b: c3 retq  
...
Outline

Introduction

The Long Journey to an Executable

Running a Program

What about Python, Java and JavaScript?

Administrivia
$ ./a.out

- After you type ./a.out but before it starts running:
  - it must be loaded into memory
  - all shared objects must be linked in
  - the processor must be instructed where to start executing

- This is the job of the dynamic linker and loader
  - Usually one program on Linux, ld.so
Memory Layout of a Program

- Kernel virtual memory
- User stack (created at runtime)
- Memory mapped region for shared libraries
- Run-time heap (created by malloc)
- Read/write data
- Read-only code and data

Memory invisible to user code

Program start

printf function

Loaded from the hello executable file
The Processor

- For now, we’ll treat the processor as a box.
- It reads instructions and data from memory (also a box).
- Performs operations on data specified by the instructions.
- Stores the data back into memory.
- This is the “von Neumann” model of computation.
  - After John Von Neumann who wrote a report about it in 1945.
  - We will read this report!
The Operating System

- There are many programs running on your computer
  - Each “believes” it is running alone
  - Illusion of having CPU and RAM to itself
- This illusion is created by the CPU and managed by the *operating system*
  - Loosely speaking, Linux, Windows, macOS X are all operating systems
- The operating system is responsible for:
  - Mediating access to the hardware (through drivers)
  - Protecting programs from each other
  - Protecting users from each other
  - Managing resources such as disks, memory, etc.
  - Lots of other responsibilities (sign up for CSC256)
What did the CPU do?

$ perf stat -e instructions ./a.out
Hello, the value of pi is 3.141593

Performance counter stats for './a.out':

662,172 instructions

0.001168841 seconds time elapsed

- The Linux `perf` command gives you lots of statistics about CPUs and programs (called performance counters)
- `main` was about 10 instructions, where did more than half-a-million instructions come from?
- How fast is this processor (instructions/second)?
Why build separate programs (cc, as, ld)?
  - Think about which programs need to change if the processor changes

Are there other designs other than von Neumann computers?
  - Yes, I research them

Why do we have different processors?
  - Intel/AMD for desktops/laptops
  - ARM for mobile phones

Why was Unix such a success?
  - Turing Award in 1983!
  - Still being used nearly 50 years later!
The Big Picture for this Course

Part I

Processor

System Software

Part II

Program

Operating System
What next?

CSC251 Advanced Computer Architecture
ECE112 Logic Design
Outline

Introduction

The Long Journey to an Executable

Running a Program

What about Python, Java and JavaScript?

Administrivia
Interpreted Languages

- Some languages do not compile to a binary
  - No assembler
  - Language-specific linkers and loader (but these are usually absent)
- Interpreters
  - “CPUs in software” (sometimes called virtual machines)
  - It’s very easy to write an interpreter...
- Three notable languages that do this:
  - Python (compiles to a stack machine bytecode)
  - Java (also compiles to bytecode)
  - JavaScript (originally not compiled, but nearly all browsers do just-in-time (JIT) compilation)
- Not our focus in this course
  - But may talk about Just-in-Time compiling if we have time
Why focus on C?

- Old systems language
  - Invented in 1972 at AT&T Bell Labs
  - Used to write the Unix *kernel* (the core of the operating system)
- Low-level language
  - But not as low-level as assembler
- “Portable” assembler (not specific to a processor)
  - Usually straightforward mapping to assembler
  - (Mostly) easy to understand how the translation from C to assembler is done
- Tremendous impact
  - Both positive and negative
  - Should you write new programs in C?
Outline

Introduction

The Long Journey to an Executable

Running a Program

What about Python, Java and JavaScript?

Administrivia
People

- **Instructor:** Dr. Sreepathi Pai
  - E-mail: sree@cs.rochester.edu
  - Office: Wegmans 3409
  - Office Hours: Mondays and Wednesdays 16:40 to 17:45 (i.e. after class)

- **TAs:**
  - Yu Feng
  - Yiming Gan
  - Daniel Busaba
  - Benned Hedegaard
  - Olivia Morton
  - Max Kimmelman
  - Daniel Ro
Places

- **Class:** Wegmans 1400
  - M,W 1525–1640

- **Course Website**
  - [https://cs.rochester.edu/~sree/courses/csc-252-452/fall-2019/](https://cs.rochester.edu/~sree/courses/csc-252-452/fall-2019/)

- **Blackboard**
  - Announcements, Assignments, etc.

- **Piazza**
  - Link accessible through Blackboard

- **Sign up for a CSUG Account if you don’t have one:**
  - [https://accounts.csug.rochester.edu/](https://accounts.csug.rochester.edu/)
  - Required for assignments!
References

- One textbook
  - *Computer Systems: A Programmers Perspective, 3/ed, Bryant and O’Hallaron*
- This course requires a lot of reading!
  - Books have been placed on reserve
  - Online materials will be linked throughout course
- See Blackboard for information on accessing Reserves
Defined Readings

- Required by The College for 2xx courses
- An hour per week of independent reading
- I will assign material that are not textbooks
  - Manuals
  - Papers
  - Articles
- These will make you a better systems programmer and computer scientist
- I will assume you have read these – you will need them for your assignments
Grading

- Participation: 5%
- Homeworks: 5%
- Assignments: 55% (4–6)
- Exam: 15% (midterm) + 20% (final)
- Graduate students should expect to read a lot more, and work on harder problems.

There is no fixed grading curve. Assume absolute grading. Course website has details.
See course website and syllabus for other details.
 Academic Honesty

- Unless explicitly allowed (e.g. teams), you may not show your code to other students
- You may discuss, brainstorm, etc. with your fellow students but all submitted work must be your own
- All help received must be acknowledged in writing when submitting your assignments and homeworks
- All external code you use must be clearly marked as such in your submission
  - Use a comment and provide URL if appropriate
- If in doubt, ask the instructor

All violations of academic honesty will be dealt with strictly as per UR’s Academic Honesty Policy.
References and Next Week

- Read chapter 1 of the textbook for today’s lecture
- Read chapter 2 of the textbook for next week’s lecture
- Acknowledgements:
  - Program memory layout figures from the textbook