CS 33: Introduction to Computer Organization

Week 4

Agenda

- Midterm 1 Review
- Homework 3
  - struct practice problem
- Lab 2
  - gdb demonstration

Admin

- Homework 3 “due” Monday, October 26th
- Lab 2 “due” Wednesday, October 28th
  - What? Is that right?

Midterm 1 Stats

- Out of 99 points:
- Median: 50
- Mean: 50.9
- Std Dev: 13
- If you're not feeling optimistic, recall that the midterm is only worth 12.5% of your grade. Compare that against 10%, which is how much the labs are worth.

Midterm Solutions

- Question 1: In 1973, James [redacted]

REDACTED

[redacted] to find the left kidney.

Homework 3

- At this point, you will probably be well equipped with handling homework 3.
- But here's a practice problem. You know. For “fun”.

Problem 3.69 (3rd ed.)

```
typedef struct {
  int first;
  a_struct a[CNT];
  int last;
} b_struct;

void test(long i, b_struct *bp)
{
  int n = bp->first + bp->last;
  a_struct *ap = &bp->a[i];
  ap->x[ap->idx] = n;
}
```

```
<test>:
mov    0x120(%rsi), %ecx
add    (%rsi), %ecx
lea    (%rdi,%rdi,4), %rax
lea    (%rsi,%rax,8), %rax
mov    0x8(%rax), %rdx
movslq %ecx, %rcx
mov    %rcx, 0x10(%rax,%rdx,8)
ret
```

- **What is CNT?**
- **What is the declaration of a_struct?**

Problem 3.69 (3rd ed.)

```
<test>:
[1] mov    0x120(%rsi), %ecx
[2] add    (%rsi), %ecx
[3] lea    (%rdi,%rdi,4), %rax
[4] lea    (%rsi,%rax,8), %rax
[5] mov    0x8(%rax), %rdx
[6] movslq %ecx, %rcx
[7] mov    %rcx, 0x10(%rax,%rdx,8)
[8] ret
```

- %rsi is b_struct *bp.
- We use bp for “int n = bp->first + bp->last;”
- [1+2] perform *(bp+0x120) + *(bp). That looks like a match.
- So bp->last is at bp + 256 + 32 or bp + 288.
- How much does bp->a take up?

Problem 3.69 (3^{rd} ed.)

- How much does bp->a take up?
  - 284 bytes (the 4 bytes of bp->first (int) + the 284 bytes of bp->a make an offset of 284). Right?
- This means sizeof(a_struct) * CNT = 284 right?

Problem 3.69 (3rd ed.)

- How much does bp->a take up?
  - 284 bytes (the 4 bytes of bp->first (int) + the 284 bytes of bp->a make an offset of 284). Right?.
- This means sizeof(a_struct) * CNT = 284 right?
  - Not necessarily. Due to alignment, bp->last must be 4-aligned. We know that bp->a requires 284 bytes but bp->a could actually be anywhere between 281 to 284 bytes and still have bp->last be at offset 288.
  - Heck, maybe there's padding between bp->first and bp->a. If this were true, what would this say about the data types within an instance of a_struct? If you're not sure, stay tuned...

Problem 3.69 (3rd ed.)

```
<test>:
[1] mov    0x120(%rsi), %ecx
[2] add    (%rsi), %ecx
[3] lea    (%rdi,%rdi,4), %rax
[4] lea    (%rsi,%rax,8), %rax
[5] mov    0x8(%rax), %rdx
[6] movslq %ecx, %rcx
[7] mov    %rcx, 0x10(%rax,%rdx,8)
[8] ret
```

- [3] %rax = i + 4*i = 5i
- [4] %rax = bp + 40*i
- [5] %rdx = *(bp + 8 + 40*i)
- What's going on here?

Problem 3.69 (3^{rd} ed.)

[3] %rax = i + 4*i = 5i

- [4] %rax = bp + 40*i
- [5] %rdx = *(bp + 8 + 40*i)
- What's going on here?
  - The 'i' is the index used to access &bp->a[i]. That's probably what this expression is doing.
- If we use i to access elements of the a_struct, what does this say about the size of a_struct?

Problem 3.69 (3rd ed.)

[3] %rax = i + 4*i = 5i

- [4] %rax = bp + 40*i
- [5] %rdx = *(bp + 8 + 40*i)
- What's going on here?
  - The 'i' is the index used to access &bp->a[i]. That's probably what this expression is doing.
- If we use i to access elements of the a_struct, what does this say about the size of a_struct?
  - sizeof(a_struct) = 40.
- What is the 8 in the expression?

Problem 3.69 (3^{rd} ed.)

[5] %rdx = *(bp + 8 + 40*i)

- What's going on here?
  - The 'i' is the index used to access &bp->a[i]. That's probably what this expression is doing.
- If we use i to access elements of the a_struct, what does this say about the size of a_struct?
  - sizeof(a_struct) = 40.
- What is the 8 in the expression?
  - In order to get to bp->a, we must offset by 8 (even though bp->first is an int). Thus, a_struct has a long in it that forces it to be 8 aligned.
- Thus, bp + 8 + 40*i is &bp->a[i]. Thus, we can rewrite this.

Problem 3.69 (3rd ed.)

[5] %rdx = *(&bp->a[i])

- We're dereferencing a pointer to a struct though. What significance does that have?
  - %rdx is set to the first element in bp->a[i].
- What is the size of the very first element?

Problem 3.69 (3^{rd} ed.)

[5] %rdx = *(&bp->a[i])

- We're dereferencing a pointer to a struct though. What significance does that have?
  - %rdx is set to the first element in bp->a[i].
- What is the size of the very first element?
  - It should be 8 bytes.
- Since &bp->a[i] is assigned to ap:
- [5] %rdx = *(ap);

Problem 3.69 (3rd ed.)

```
<test>:
[1] mov    0x120(%rsi), %ecx
[2] add    (%rsi), %ecx
[3] lea    (%rdi,%rdi,4), %rax
[4] lea    (%rsi,%rax,8), %rax
[5] mov    0x8(%rax), %rdx
[6] movslq %ecx, %rcx
[7] mov    %rcx, 0x10(%rax,%rdx,8)
[8] ret
```

- [5] %rdx = *(ap)
- [6] %rcx = n
- [7] *(16 + bp + 40i + *(ap)*8) = n

Problem 3.69 (3^{rd} ed.)

[7] *(16 + bp + 40i + *(ap)*8) = n
=> *(bp + 8 + 40i + 8 + (*ap) * 8) = n
=> *(ap + 8 + (*ap)*8) = n

- This line matches up with:
  - ap->x[ap->idx] = n;
- Thus, *ap is ap->idx. This implies the first element is the ap->idx.
- If this were true, then in order to access ap->x, we would need to offset ap by an additional 8 bytes to skip over ap->idx.
- Oh look: *(ap ***+ 8*** + (*ap)*8) = n

Problem 3.69 (3rd ed.)

- *(ap **+** ***8*** + (*ap)*8) = n

  ```
  struct a_struct {
    long idx;
    <?> x[?];
  }
  ```

- What is the data type of array a_struct->x?

Problem 3.69 (3^{rd} ed.)

- *(ap ***+ 8*** + (*ap)*8) = n

  ```
  struct a_struct {
    long idx;
    <?> x[?];
  }
  ```

- What is the data type of array a_struct->x?
  - It's 8 bytes. *ap is idx and in order to use it to index into x, we multiply idx by 8. Thus:
- struct a_struct {

  ```
    long idx;
    long x[?];
  ```
- }

Problem 3.69 (3^{rd} ed.)

```
struct a_struct {
  long idx;
  long x[?];
}
```

- What is the length of x?

- Recall sizeof(a_struct) is 40 bytes. Thus:

```
struct a_struct {
  long idx;
  long x[4];
}
```

Problem 3.69 (3^{rd} ed.)

```
struct a_struct {
  long idx;
  long x[4];
}
```

- Let's do a final consistency check and determine CNT. Recall that in b_struct:
  - Offset 0: int first
  - Offset ?: a_struct a[CNT];
  - Offset 288: int last
- If this a_struct is correct, the offset would need to be 8. Thus, the array of a would take 280.
- CNT = 280 bytes / 40 bytes per a_struct = 7

Pexex Lab: Getting started

- The objective:
  - Examine the execution of an actual program with a debugger.
  - Learn more about more complicated ways of handling arithmetic.
  - Examine the effect of compiling code with -fwrapv, -fsanitize=undefined, and default.
- Essentially, we're forcing you to learn gdb and using debuggers.

Pexex Lab: Getting started

- You will be examining the execution of Emacs, the text editor.
- Emacs also provides an interpreter for handling “Elisp”, a functional programming language that allows for simple computation.
- Professor Eggert seems to like Emacs so much. It's almost as if Professor Eggert has some hand in making Emacs...
  - For fun, in Linux, type in the command “diff -v”

Pexex Lab: Getting started

- You will concern yourselves with the following files which are located on the SEASnet server.
- Yup, for this one, you've really got no choice but to use SEASnet, preferably lnxsrv09.
- Emacs executable located in:
  - ~eggert/bin64/bin/emacs-24.5
- Emacs source code located in:
  - ~eggert/src/emacs-24.5/

Pexex Lab: Getting started

- In particular, you will examine the execution of the following invocation of Emacs:
  - ~eggert/bin64/bin/emacs-24.5 -batch -eval '(print (* 6997 -4398042316799 179))'
- You will run this command with gdb via the following:
  - gdb --args ~eggert/bin64/bin/emacs-24.5 -batch -eval '(print (* 6997 -4398042316799 179))'
- Or...
  - gdb ~eggert/bin64/bin/emacs-24.5
  - (gdb) r -batch -eval '(print (* 6997 -4398042316799 179))'

Pexex Lab: gdb quick start

- After opening gdb:
- Run the program
  - run (or simply 'r')
- Run the program with arguments
  - r arg1 arg2
- This will run the executable to completion.

Pexex Lab: gdb quick start

- Set break point at function foo:
  - break foo
- Set break point at instruction address 0x100
  - break *0x100
  - Note the asterisk. Without it, it will look for a function called 0x100. Please don't write a function called 0x100.
- When program is run or “continued”, it will run until it hits a break point in which it will stop.
- Resume a program that is stopped
  - continue (or just 'c')

Pexex Lab: gdb quick start

- When a program is break-ed (breaked? broken?) you can step through each instruction either for each assembly instruction or each high level C instruction.
- Execute the next instruction, stepping INTO functions:
  - stepi (or just 'si')
- Execute the next line of source code, stepping INTO functions:
  - step (or just 's')

Pexex Lab: gdb quick start

- Execute the next instruction, stepping OVER functions:
  - nexti (or just 'ni')
- Execute the next line of source code, stepping OVER functions:
  - next (or just 'n')
- Stepping INTO functions means any time you call another function, you will descend into the function. Stepping OVER functions meaning stepping over the function as if it were a single instruction.

Pexex Lab: gdb quick start

- Examining the assembly instructions of function foo:
  - disassemble foo (or just “disas foo”)
- If you are stopped at some instruction in “foo”, the disassembled assembly will also show you where execution is paused at:

```
   0x54352e <arith_driver+46>: 49 89 d7 mov  %rdx,%r15
=> 0x543531 <arith_driver+49>: 49 89 f5 mov  %rsi,%r13
   0x543534 <arith_driver+52>: 41 89 fe mov  %edi,%r14d
```

- This means arith_driver+46 has been executed but arith_driver+49 has not yet been executed.

Pexex Lab: gdb quick start

- disas /m <function name>
  - Display assembly for <function name> prefaced with the corresponding lines of C.
- Using next(i) or step(i), you can also set the debugger to print each instruction as it's executed:
  - set disassemble-next-line on

Pexex Lab: gdb quick start

- Examining registers:
- Print the current state of all registers:
  - info registers
- Print the state of a particular register:
  - info registers $rdi
- Don't you mean “**%**rdi”?
- No. No I don't. And I cry about it every night.

Pexex Lab: gdb quick start

- Print out contents at some memory address with the “x” command.
  - x [addr]
- This was kind of a waste of a whole slide.
- Here is some more text to make it seem like I'm not wasting the slide.
- So, how was your day?
- Please don't print out this slide.

Pexex Lab: gdb quick start

- For more options, use x/nfu [addr] where
  - n specifies how much memory to display (in terms of the unit specified by u), default 1
  - f specifies the format (ie decimal, hexadecimal, etc.), default hex
  - u specifies the unit size for each address (words, bytes, etc.), default word
  - Check out the following link for a more precise listing of the parameters:
  - https://sourceware.org/gdb/onlinedocs/gdb/Memory.html

Pexex Lab: gdb quick start

- Also, print out memory based on an address stored in an register:
- Ex: x/20xw $rsp
  - Print out 20 words(w) in hex(x) starting from the address stored in %rsp.

Pexex Lab: TODO

- Set a break point at Ftimes.
- For each instruction until Ftimes completes, examine what each instruction is doing, checking the status of the registers and memory as necessary.
- Answer the questions.

Pexex Lab: TODO

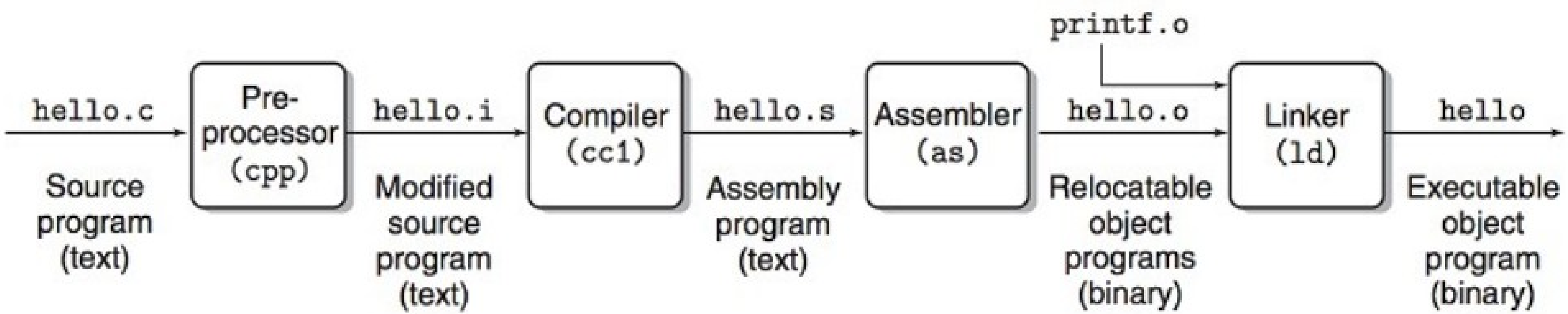
- For part 2, there are three ways:
- gcc -S -fno-asynchronous-unwind-tables <ADDITIONAL FLAGS> testovf.c
  - This produces testovf.s which you can read with any text editor.
- gcc -c <ADDITIONAL FLAGS> testovf.c
  - This produces testovf.o, which you will have to read using “objdump -d testovf.o”. To save the output of object dump, use “objdump -d testovf.o > testovf.txt”

Pexex Lab: TODO

- gcc <ADDITIONAL FLAGS> testovf.c
  - This produces a.out, which you can examine with “gdb a.out”.
- If you use the -S or -c options, you don't need to include a “main” function. However, if you compile to completion (no -S or -c), you will need to include a main function.
- What is all this nonsense?

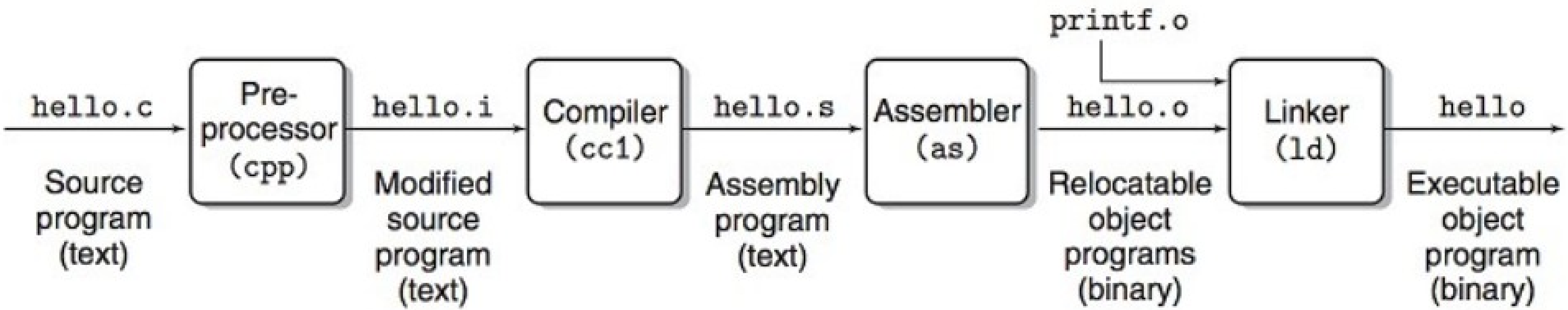
Pexex Lab: TODO

- Recall from Chapter 1 that compilation occurs over several different steps.



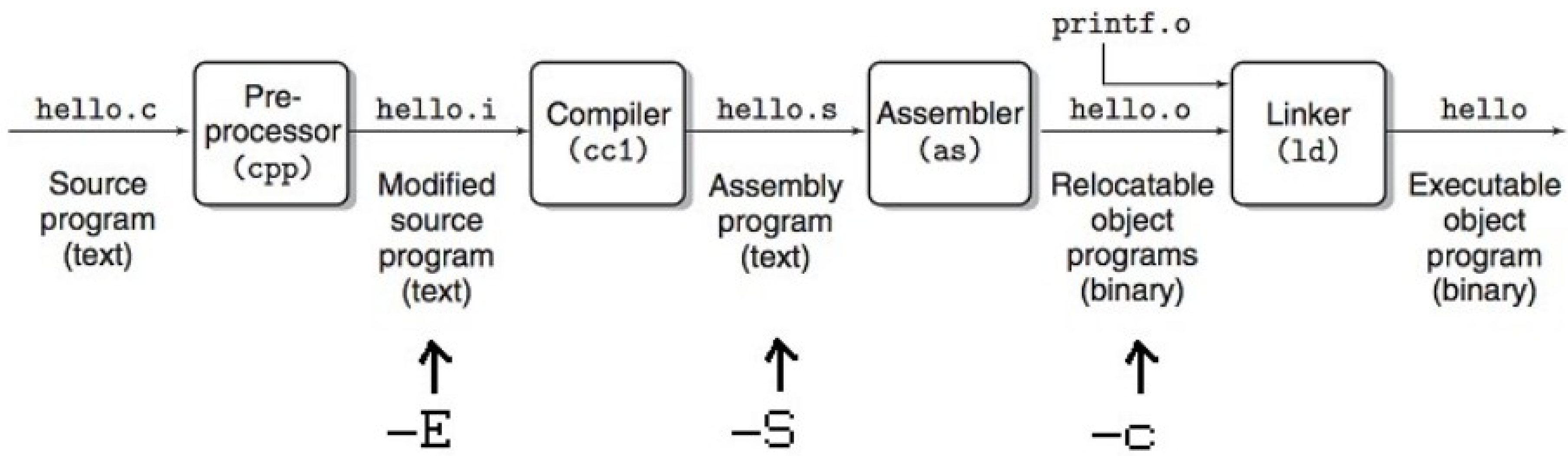
- The result of the Pre-processor step is a modified source with the preprocessor directives(#define, #include) replaced

Pexex Lab: TODO



- The result of the Compiler step is compiled code, which is readable assembly
- The result of the Assembler step is the assembled code which is a binary file.
- Finally, the result of the linker is a fully executable file.
- gcc allows you to compile up to certain steps

Pexex Lab: TODO



- Ex: gcc -E [filename] will get you the modified source file
- Note: Using -E and -S will get you files that you can read with a text editor. To read the output of -c, use objdump.
- To read/disassemble the final executable, use gdb

End of

The Fourth Week

-Six Weeks Remain-