

# RECITATION 5

## INFO

- Jon Rutkauskas
- Recitation: Tue 12-12:50
- Office Hours: Tue 11-11:50  
Thur 11-12:50  
**SENSQ 5806**  
(additional hours by appointment if needed)
- On discord: @jrutkauskas
- By email: [jsr68@pitt.edu](mailto:jsr68@pitt.edu)
- Github:  
<https://github.com/jrutkauskas/spring2019-449-rec>
- Ask me any questions you have!!!

## WARMUP POLLS

# DEBUGGING AND GDB

- Isn't it annoying when your program doesn't do what you want it to do?
- We have to **debug** it to figure out what's going wrong.
- Currently, you've probably been doing a lot of debugging with print statements
  - E.g., `printf("Made it to function B!, x is %d", x)`
- We can do better!

# DEBUGGING AND GDB

- A **debugger** is a tool we can use to “watch your program fail in slow motion” (Jarrett’s words)
- We can see where it’s breaking, how it’s breaking, and other information about our program as it breaks
- You may have used an IDE’s built-in debugger before, such as in Visual Studio or Eclipse
  - Have you set breakpoints, stepped through lines of code, viewed local variables live, or viewed the current stack? -- If so, you’ve used a debugger!
- We’ll use gdb in this class for debugging

# SUPER SIMPLE PROGRAM

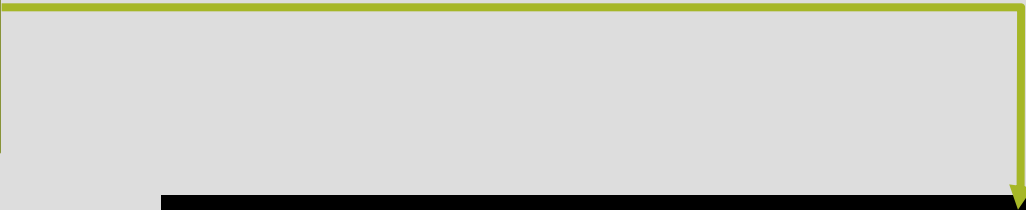
```
#include <stdio.h>
#include <stdlib.h>

int add(int* x, int* y);
int main()
{
    int *x = malloc(sizeof(int));
    *x = 4;
    int* y = *x + 10;
    int sum = add(x,y);
    printf("Sum is: %d", sum);
    return 0;
}

int add(int* x, int* y)
{
    return *x + *y;
}
```

## WHEN WE RUN IT...

Use `-g` flag to  
help debug later



```
[thoth ~/private/449/lab4]: gcc -g -o simple simple.c
simple.c: In function 'main':
simple.c:11: warning: initialization makes pointer from integer without a cast
[thoth ~/private/449/lab4]: ./simple
Segmentation fault (core dumped)
```

## LET'S USE GDB!

```
[thoth ~/private/uta-449/lab4]: gdb ./simple
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-64.el6_5.2)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /afs/pitt.edu/home/j/s/jsr68/private/uta-449/lab4/simple...
done.
(gdb) █
```



TYPE 'run' TO RUN

```
(gdb) run
```

```
Starting program: /afs/pitt.edu/home/j/s/jsr68/private/uta-449/lab4/simple
```

```
Program received signal SIGSEGV, Segmentation fault.
```

```
0x00000000040057d in add (x=0x601010, y=0xe) at simple.c:22
```

```
22         return *x + *y;
```

```
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.132.el6_5.3.x86_64
```

```
(gdb) █
```

## LET'S ADD A BREAKPOINT!

```
(gdb) break add
Breakpoint 1 at 0x400573: file simple.c, line 22.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /afs/pitt.edu/home/j/s/jsr68/private/uta-449/lab4/simple

Breakpoint 1, add (x=0x601010, y=0xe) at simple.c:22
22          return *x + *y;
(gdb) █
```

```
(gdb) stepi  
0x0000000000400577      22      return *x + *y;
```

Use 'stepi' to step to the next instruction.  
GDB prints the line number and code

```
(gdb) stepi  
0x00000000000400577      22      return *x + *y;  
(gdb) print x
```

Let's see what x is

```
(gdb) stepi
0x0000000000400577      22      return *x + *y;
(gdb) print x
$1 = (int *) 0x601010
```

...it's some memory address, which we expect to point to the integer  
we want to sum with \*y

```
(gdb) stepi
0x0000000000400577      22      return *x + *y;
(gdb) print x
$1 = (int *) 0x601010
(gdb) print y
```

Okay, so what's y

```
(gdb) stepi
0x0000000000400577      22      return *x + *y;
(gdb) print x
$1 = (int *) 0x601010
(gdb) print y
$2 = (int *) 0xe
```

...hmm that seems like a pretty odd memory address?

```
(gdb) stepi
0x00000000000400577      22      return *x + *y;
(gdb) print x
$1 = (int *) 0x601010
(gdb) print y
$2 = (int *) 0xe
(gdb) print *y
```

Let's see what it points to...



```
(gdb) stepi
0x0000000000400577      22      return *x + *y;
(gdb) print x
$1 = (int *) 0x601010
(gdb) print y
$2 = (int *) 0xe
(gdb) print *y
Cannot access memory at address 0xe
```


Yikes! There's our segfault. We can't dereference that!

```
(gdb) stepi
0x0000000000400577      22      return *x + *y;
(gdb) print x
$1 = (int *) 0x601010
(gdb) print y
$2 = (int *) 0xe
(gdb) print *y
Cannot access memory at address 0xe
(gdb) print/d y
$3 = 14
```

Printing that out as a decimal tells us it's 14.

## LET'S LOOK BACK AT OUR CODE

Looks like we  
should have done  
this differently



```
#include <stdio.h>
#include <stdlib.h>

int add(int* x, int* y);
int main()
{
    int *x = malloc(sizeof(int));
    *x = 4;
    int* y = *x + 10;
    int sum = add(x,y);
    printf("Sum is: %d", sum);
    return 0;
}

int add(int* x, int* y)
{
    return *x + *y;
}
```

## HOW CAN WE FIX THIS CODE?

```
#include <stdio.h>
#include <stdlib.h>

int add(int* x, int* y);

int main()
{
    int* x = malloc(sizeof(int));
    *x = 4;
    int* y = *x + 10;

    int sum = add(x,y);

    printf("Sum is: %d", sum);

    return 0;
}

int add(int* x, int* y)
{
    return *x + *y;
}
```

## HOW CAN WE FIX THIS CODE?

```
#include <stdio.h>
#include <stdlib.h>


int add(int* x, int* y);

int main()
{
    int* x = malloc(sizeof(int));
    *x = 4;
    int* y = *x + 10;
    int sum = add(x, y);

    printf("Sum is: %d", sum);

    return 0;
}

int add(int* x, int* y)
{
    return *x + *y;
}
```



```
int y = *x + 10;
int sum = add(x, &y);
```

# QUITTING

- After we're done, to quit, just type 'quit' into gdb and answer 'y' that we want to end the currently running process

## LAB 4

- Lab 4 gives you a much bigger look into gdb.
- It can do a ton more!
- You'll need this a lot for your projects, both current, and future :-)
- Try to get a lot of practice with this, especially setting breakpoints, stepping through instructions (step, next, stepi, nexti), using 'where' for a stack trace, and printing out variables (or any other C expression)

OTHER QUESTIONS



WHICH OF THESE IS **NOT** TRUE OF  
BITMAPS (MEMORY ALLOCATION)

Bitmaps are prone to  
internal fragmentation

Bitmaps generally take up  
a lot of space to store

Bitmaps break up memory  
into variable-size chunks

Bitmaps are slow to find  
free memory

## WHICH OF THESE IS **NOT** TRUE OF BITMAPS (MEMORY ALLOCATION)

1. Bitmaps, by breaking up memory into FIXED size chunks; the allocator will always use AT LEAST as much memory as the user requires. So, if the memory is broken up into 32 byte chunks (for example) and a user wants 60 bytes of memory, we would allocate 2 32 byte chunks = 64 bytes, thereby wasting 4 bytes – Internal Fragmentation
  - The Linked Chain memory allocation also has internal fragmentation, but not like this
2. Bitmaps CAN take up a lot of space, 1 bit per chunk. If we have a bunch of tiny allocations, bitmaps might be better space-wise, but 32 byte chunks when we need to allocate, say, a 3MB WAV file...
  - $3\text{MB}/32\text{ Bytes} = 93750\text{ bits in bitmap} = \text{over } 11\text{k!}$  Versus a Linked chain with 1 single header (2 pointers and 2 ints)
3. Bitmaps must use fixed-size chunks, otherwise we wouldn't know how large a single chunk is since we denote a chunk with a single bit (not some int or something)
4. Bitmaps ARE slow. Lots of shifting and searching down the bitmap, lots of bitwise operations. Computers aren't really built to be fast for all that.

Bitmaps are prone to internal fragmentation

Bitmaps generally take up a lot of space to store

~~Bitmaps break up memory into variable-size chunks~~

Bitmaps are slow to find free memory

# FIRST FIT



Note: X is  
most recently  
allocated block

?

- Where would this memory go if we tried to allocate it using **first fit**?
- We need 54 Bytes.
  - A is 60 Bytes
  - B is 58 Bytes
  - C is 28 Bytes
  - D is 80 Bytes
  - E is 128 Bytes

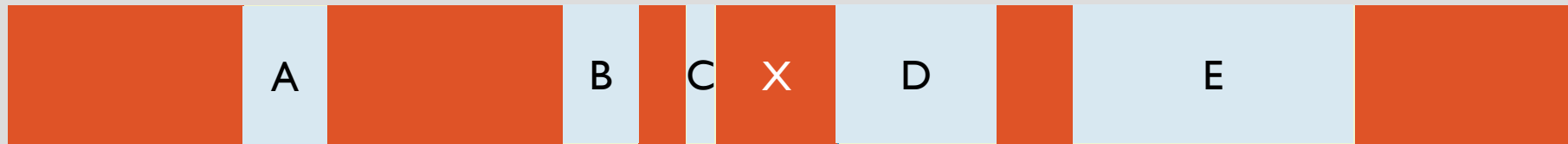
# FIRST FIT



Note: X is  
most recently  
allocated block

- Where would this memory go if we tried to allocate it using **first fit**?
- We need 54 Bytes.
  - A is 60 Bytes
  - B is 58 Bytes
  - C is 28 Bytes
  - D is 80 Bytes
  - E is 128 Bytes

# NEXT FIT



Note: X is  
most recently  
allocated block

?

- Where would this memory go if we tried to allocate it using **Next-fit**?
- We need 54 Bytes.
  - A is 60 Bytes
  - B is 58 Bytes
  - C is 28 Bytes
  - D is 80 Bytes
  - E is 128 Bytes

# NEXT FIT



Note: X is  
most recently  
allocated block

- Where would this memory go if we tried to allocate it using **Next-fit**?
- We need 54 Bytes.
  - A is 60 Bytes
  - B is 58 Bytes
  - C is 28 Bytes
  - D is 80 Bytes
  - E is 128 Bytes

# BEST FIT



Note: X is  
most recently  
allocated block



- Where would this memory go if we tried to allocate it using **Best-fit**?
- We need 54 Bytes.
  - A is 60 Bytes
  - B is 58 Bytes
  - C is 28 Bytes
  - D is 80 Bytes
  - E is 128 Bytes

# BEST FIT



Note: X is  
most recently  
allocated block

- Where would this memory go if we tried to allocate it using **Best-fit**?
- We need 54 Bytes.
  - A is 60 Bytes
  - B is 58 Bytes
  - C is 28 Bytes
  - D is 80 Bytes
  - E is 128 Bytes



# WORST FIT



Note: X is  
most recently  
allocated block

?

- Where would this memory go if we tried to allocate it using **Worst-fit**?
- We need 54 Bytes.
  - A is 60 Bytes
  - B is 58 Bytes
  - C is 28 Bytes
  - D is 80 Bytes
  - E is 128 Bytes

# WORST FIT



Note: X is  
most recently  
allocated block

- Where would this memory go if we tried to allocate it using **Worst-fit**?
- We need 54 Bytes.
  - A is 60 Bytes
  - B is 58 Bytes
  - C is 28 Bytes
  - D is 80 Bytes
  - E is 128 Bytes

## PROJECT 2

- Project 2 will give you some wonderful experience with how linked-chain memory allocators work.

ANY OTHER QUESTIONS FOR THE  
GOOD OF THE GROUP?