Memory Layout, File I/O

Bryce Boe

2013/06/27

CS24, Summer 2013 C

Outline

- Review HW1 (+command line arguments)
- Memory Layout
- File I/O

HW1 REVIEW

HW1 Common Problems

- Taking input from stdin (via scanf)
- Performing *failure* testing too late
- Not handling the 0 case
- Whitespace issues
- Others?

HW1 Solution

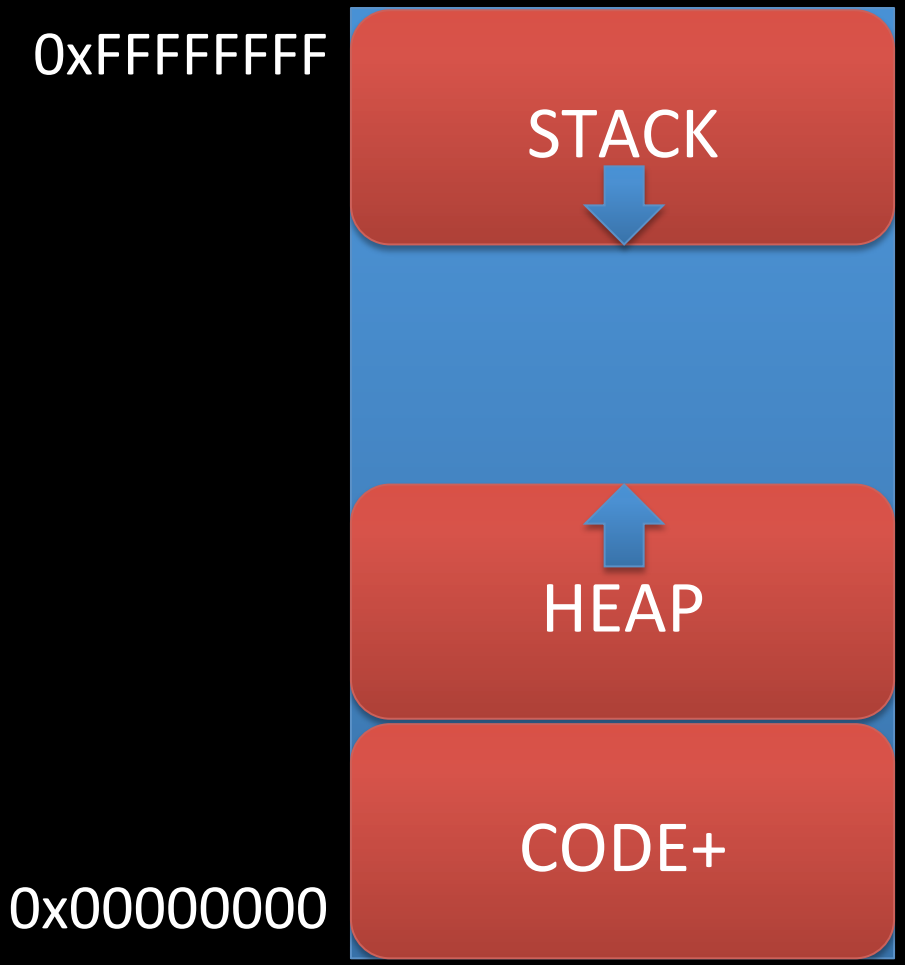
- <In-class review of hw1 solution source code>

MEMORY LAYOUT

View from three Levels

- The address space of a process
- Function activation records on the stack
- Data within an activation record

Simplified process's address space



Data Segments

- Code(+)
  - Program code
  - Initialization data, global and static variables
- Heap
  - Memory chunks allocated via malloc
- Stack
  - Function activation records

Creating and destroying activation records

- Program starts with main
- main calls strsafeconcat
- strsafeconcat calls strlength
- strlength returns
- strsafeconcat calls strlength
- strlength returns
- strsafeconcat returns

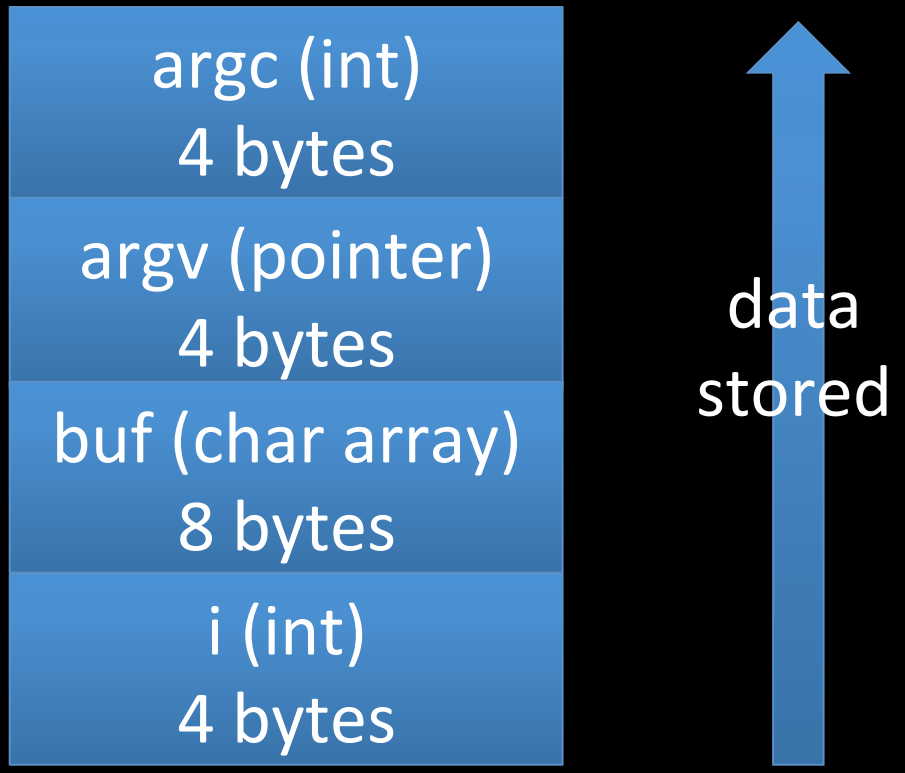
Simplified function activation records

- Stores values passed into the function as parameters
- Stores the function's local variables

How many bytes is the simplified activation record?

```
int main(int argc, char *argv[]) {
        char buf[8];
        for (int i = 0; i < argc; ++i)
                buf[i] = argv[i][0];
}
```

main's simplified activation record



Total: 20 bytes

Think about it

```
int main() {
        char msg[] = “hello”;
        char buf[] = “1234567”;
        char msg2[] = “world”;
}
```

- What value does buf[8] hold?
- What about msg[7]?

Similar but different

```
int main() {
        int x = 0xDEADBEEF;
        char buf[] = “1234567”;
}
```

- What value does buf[8] hold?

Inspecting addresses in a program

- <In class review of variables_in_memory.c>

FILE I/O

File I/O

- I/O stands for input/output
- Provided by the stdio library (stdio.h)
- Allows reading and writing to/from streams (type: FILE *)
  - stdin (read only)
  - stdout / stderr (write only)
  - named files

stdio.h

- Explore via opengroup: http://pubs.opengroup.org/onlinepubs/009695399/basedefs/stdio.h.html
- Explore via `man stdio.h` (same info as opengroup, but not easily searchable or linkable)

Opening named files and closing streams

- FILE* fopen(char *filename, char *mode)
  - open a file specifying whether to open for reading, writing, appending, and create or truncate if desired
- int fclose(FILE *stream) – close an open FILE*
  - Flush and close the stream
  - Return 0 on success

Reading from streams

- **int** fgetc(FILE *stream)
  - Returns the next character in the stream or EOF (this is why it returns an int)
- char *fgets(char *buf, int n, FILE *stream)
  - Read at most n-1 bytes from stream into buf
  - Also stops when ‘\n’ or EOF is reached
  - Terminates buf with the null character ‘\0’

Other read functions

- fread – very useful (especially when input is unbounded), we won't use in this class
- fscanf – useful, but tricky, don't use in this class
- Functions with similar names less the 'f'
  - Uses the **stdin** stream thus doesn't require the *stream* argument

Writing to streams

- int fputc(**int** c, FILE *stream)
  - Write a single character **c** to **stream**
- int fputs(char *buf, FILE *stream)
  - Write the null-terminated string in **buf** to **stream**
- int fprintf(FILE *stream, char *format, …)
  - Write formatted string to stream making the specified replacements

SECURITY WARNING

- **NEVER** do the following:

```
fprintf(stdout, buf);  // buf is some c-string
```

- Could allow an attacker to inspect and change your program (format string exploit)
- Use either *fputs* or *fprintf(stdout, "%s", buf)*

Other write functions

- fwrite – generally very useful, we won't use in this class
- Functions with similar names less the 'f'
  - Uses the **stdout** stream thus doesn't require the *stream* argument

Other useful stream functions

- int feof(FILE *stream)
  - Return non-zero if the stream has reached the end of the file
- int fflush(FILE *stream)
  - Force writing any buffered data to the stream
  - Flushing *typically* occurs when a newline is encountered, thus fflush is often needed when newlines aren't used

I/O Questions

- Why does fgetc/fputc return/get an integer?
- If a file with only a single newline at the end is 32 bytes long, how many bytes does the buffer for fgets require to read the entire file?

More I/O Questions

- When using **fgets**, how can you determine if the string is longer than the value of ‘n’ (the number of bytes to read)
- What will happen if the ‘n’ parameter to **fgets** is larger than the buffer?

Real-time cat program writing

- <In class creation of simple_copy.c>

For Tuesday

- Finish reading chapter 1 in the text book