Course: Operating Systems - Semester 1 of 5786
Assignment 2

Directions

A. Due Date: 19 November 2025 at 11:55pm

B. Groups of up to two (2) students may submit this assignment.

C. Code for this assignment (Ass2) must be submitted via Github using the per-assignment private repository opened for you in the OSCourse organization. More details on the repository are found below.

D. The free response questions for this assignment must be submitted via the Moodle assignment.

E. There are 100 points total on this assignment.

F. GitHub has autograding tests included that run after each push. The tests and points are not complete, but do cover the basic input/output behavior of the programs you must write. Some test files are found in the `tests/` subdirectory in the repository. Do not remove or modify those files.

G. What to turn in via GitHub:

   (a) All source code and library files necessary to compile and run your programs

   (b) README.md file with contents specified in Section 4

General Requirements

1. All of the code below must be written in C and compilable and executable in a standard Linux Ubuntu (14+) or Linux Mint (20+) environment.
2. All code must have comments - each function must have an introductory comment detailing its purpose, input parameters, and return value.

Basic OS attributes[1]

The goal of this assignment is to understand what is really inside of a running program and what the operating system needs to deal with.

1 map (44 points)

Code: (5 points) We will write a program that reveals its own executing structure. The files `map.c` and `recur.c` provide a rather complete skeleton, but you will need to modify the files a bit to get the addresses we are looking for. The first task is to modify the files so that the program output looks like the following (the addresses may be different).

```
=== main() entry - Return address: 0x7fe683b921ca ===
_main @ 0x5573d6168189
recur @ 0x5573d6168305
_main stack: 0x7ffe2daa6a2c
static data: stuff 0x5573d616b010
static data: bar   0x5573d616b018
static data: foo   0x5573d616b020
Heap: malloc 1: 0x5573d77b46b0
Heap: malloc 2: 0x5573d77b4770

=== recur() call 2 - Return address: 0x5573d61682ea ===
recursive call 2: parameter int 2 @ 0x7ffe2daa49cc
recursive call 2: parameter float 4.50 @ 0x7ffe2daa49c8
recursive call 2: parameter char a @ 0x7ffe2daa49c4
recursive call 2: stack int 2 @ 0x7ffe2daa49e0
recursive call 2: stack float 4.50 @ 0x7ffe2daa49e4
recursive call 2: stack char a @ 0x7ffe2daa49df
recursive call 2: stack double[] 0.00 @ 0x7ffe2daa49f0
static data: stuff 7.00 @ 0x5573d616b010

=== recur() call 1 - Return address: 0x5573d6168569 ===
recursive call 1: parameter int 1 @ 0x7ffe2daa297c
recursive call 1: parameter float 3.00 @ 0x7ffe2daa2978
recursive call 1: parameter char b @ 0x7ffe2daa2974
recursive call 1: stack int 1 @ 0x7ffe2daa2990
recursive call 1: stack float 3.00 @ 0x7ffe2daa2994
recursive call 1: stack char b @ 0x7ffe2daa298f
recursive call 1: stack double[] 0.00 @ 0x7ffe2daa29a0
static data: stuff 7.00 @ 0x5573d616b010

=== recur() call 0 - Return address: 0x5573d6168569 ===
recursive call 0: parameter int 0 @ 0x7ffe2daa092c
recursive call 0: parameter float 1.50 @ 0x7ffe2daa0928
recursive call 0: parameter char c @ 0x7ffe2daa0924
recursive call 0: stack int 0 @ 0x7ffe2daa0940
recursive call 0: stack float 1.50 @ 0x7ffe2daa0944
recursive call 0: stack char c @ 0x7ffe2daa093f
recursive call 0: stack double[] 0.00 @ 0x7ffe2daa0950
static data: stuff 7.00 @ 0x5573d616b010
```

1.1 Code

Modify the code in map.c and recur.c so that they compile and run properly.

1.2 Steps and Questions

Now perform the following steps and answer the following questions about the results. Write your responses in the Moodle questions.

[1]This assignment is adapted from HW0 of CS162 at UC Berkeley by Kubiatowicz from this year and previous years. The original can be found at: `https://cs162.org/static/homeworks/homework0.pdf`

Compiling

Run the following commands:

```
gcc -c map.c
gcc -c recur.c
```

The output should be two files - map.o and recur.o.

(map) 1. (3 points) Run `objdump -D map.o` and then `objdump -D recur.o`. The output includes the sections of the object files. Below are 3 important sections. For each section, copy 1 sample line of information from the section and write a 1 sentence summary the section's role in the program.

(a) .rodata

(b) .text

(c) .data

The object symbol table

Use the `man objdump` command to find out what flags you need to add to print the object file *symbol table*. Print the symbol tables for both map.o and recur.o.

Here's a sample output for the map.o symbol table:

```
00000000 g O .data 00000004 stuff
00000000 g F .text 00000060 main
...
00000000 *UND* 00000000 malloc
00000000 *UND* 00000000 recur
```

(mp) 2. (3 points) Below are four flags that you see in the symbol table. For each flag, write what the flag means:

(a) `g`

(b) `O`

(c) `F`

(d) `*UND*`

Linking and Executable Symbol Table

Run the following command to link the object files into a single executable:

```
gcc -o map map.o recur.o
```

(mp) 3. (3 points) Use `objdump` to examine the symbol table of the executable `map`.

What changed in the symbol table between the table for `map` and the tables for `map.o` and `recur.o`?

(mp) 4. (2 points) The symbol table for `map` still contains a few *UND* symbols, including `printf@@GLIBC_2.2.5` and `malloc@@GLIBC_2.2.5` (at least that's the version in my output).

Given the meaning of a *UND* symbol, why do those two symbols remain *UND* even after the executable has been prepared?

Running `map`

Run the resulting executable - `./map`. Look at the output.

Executable Symbol Disassembly

Run the following command:

```
objdump -x -d map
```

The output shows a disassembly of the program.

(mp) 5. (5 points) Look for the following symbols from the symbol table. For each symbol: write (1) the value output by `map` for the symbol, (2) the address for the symbol in the objdump output, and (3) the section that the symbol is defined in.

(a) main

(b) recur

(c) stuff

(d) foo

(e) bar

1.3 Output Analysis

(mp) 6. (3 points) What is the meaning of the return address line in main?

(mp) 7. (3 points) Find the 3 return addresses for the recur function. Why are two the same and one different?

(mp) 8. (3 points) Why does the address of `stuff` not change from one recursive call to the next?

(mp) 9. (2 points) Integers in C are 4 bytes. Prove that from the output of the program.

(mp) 10. (3 points) Analyze the output from `map`. What direction is the stack growing in? How do you know?

(mp) 11. (3 points) How large is the stack frame for each recursive call?

(mp) 12. (3 points) Where is the heap? What direction is it growing in?

(mp) 13. (3 points) Are the two `malloc()`ed memory areas contiguous? (*i.e.* is there any extra space between their addresses?)

2 Using user limits (8 points)

The operating system needs to deal with the size of the dynamically allocated segments: the stack and heap. How large should these be? To answer those questions, I provided you with the almost complete code for a program that gets the limits on Linux. **Modify** `limits.c` so that it prints out the maximum stack size, the maximum number of processes, and maximum number of file descriptors. Currently, when you compile and run `limits.c` you will see it print out a bunch of system resource limits with value 0. Change the program to make it print the actual limits (use the soft limits, not the hard limits). (Hint: run `man getrlimit`)

You should expect output similar to this:

```
Maximum Stack Size: 8388608
Maximum Number of Processes: 3842
Maximum Number of File Descriptors: 1024
```

3 Testing the limits (15 points)

I provided three testing programs to test the limits you found in the user limits program you wrote. Compile and run them on your computer, then answer the following questions:

(tl) 1. (5 points) What maximum stack size did you actually find? Show a screen shot of the tool output. Is it the same as the value from the limits program?

(tl) 2. (5 points) What maximum process count did you actually find? Show a screen shot of the tool output. How can you tell when you reach the limit?

(tl) 3. (5 points) What maximum file descriptor size did you find? Show a screen shot of the tool output. Is it the same as what the user limits program said?

4 Using Git and GitHub (8 points)

Submit all of your code for the homework using the private repository for your work on GitHub. You must perform the following actions on the repository:

1. Each team member must make at least 1 substantive commit of code or text to the repository.
2. The team must make **at least** two (2) commits to the repository and add a comment to each. Write good commit messages, following the instructions at this tutorial: `https://www.theodinproject.com/lessons/foundations-commit-messages`
3. The last commit must include the comment "Submitted for grading"
4. The repository must include a README file with the following details:
   - Student names and IDs
   - Assignment name (Assignment 1)
   - Course name
   - Semester and Year

Repositories missing any of the above actions will be penalized.