<u>EECS 338 Homework 2</u>

General requirements:

- See posted due date.
- Create a typed report for discussion questions.
- Upload a single, compressed file (e.g. zip) to Canvas that contains all required files (programs + report).
- All work should be your own, as explained in the Academic Integrity policy from the syllabus.
- ***NEW: Include either a single makefile that compiles, all programs or a separate makefile for each.***

Special requirements:

- If the Linux commands specified in the instructions don't work as needed on your system (e.g. Mac), you should use the EECS virtual machines (see Tutorial #1).
- For screen output, copy/paste actual text from the terminal window. Do <u>not</u> use screenshots.

Problems:

1. Run each of the following programs using the "time" command to analyze the real time, user mode time, and kernel (system) mode time. For each program, provide the output in your report and briefly <u>explain</u> why the "user" and "sys" times are different or the nearly the same.

    a. Use "for.c" from HW #1. If desired, you may reduce the value of "N" for a more convenient (shorter) run time. In addition to report requirements, provide your program (either original or revised).
    b. Revise "for.c" such that "user" time <u>and</u> "sys" time are each less than 2% of the "real" time. Hint: try adding "sleep". In addition to report requirements, provide your new program.
    c. Revise "for.c" such that "user" time is less than 50% of the "sys" time. Hint: try adding "printf" in a strategic location. You can adjust the value of "N" to be convenient. You can adjust the value of "N" to be convenient. In addition to report requirements, provide your new program.
    d. Compute and compare (explain) the %CPU for the program in part (c) above using the following:
        - Using "top". Note that the program should run long enough to get a stable %CPU value.
        - Using the three values reported by "time" and the formula (*user* + *sys*) / *real*.
        - OPTIONAL: Display the %CPU using "time". See the lecture notes for information links.

2. Run any C program you wish and use "pmap" to see the virtual memory addresses while it is running. **Include the "pmap" output in your report.** Then run the same program again and **report** the addresses again, using "pmap". Compare the addresses for the two instances as follows:
    a. Which memory segments have the same starting addresses?
    b. Which memory segments have different starting addresses?
    c. Which memory segments have the same sizes?

3. Create any C program you wish that prints the addresses of two variables: an integer and a *constant* integer. Then run "pmap" to visualize the memory usage, and answer the questions that follow. **Include program output in your report.** The following example uses the %p format specifier to print a pointer value (address) and the "address of" operator (&) to obtain the pointer:

```
int a = 0; // Regular integer
const int b = 0; // Constant integer
// Print the pointer values (addresses) in hexadecimal
printf("%p, %p\n", &a, &b);
```

Hypothetical output:

```
0x7ffd90ed9038, 0x7ffd90ed903c
```

    a. Run "top" and find your process in the list. How well does the total virtual memory (VIRT) reported by "top" compare to the amount displayed by "pmap"? Are they the same or different? **In your report, include the relevant output from "pmap" and "top", and describe the similarity or difference.** Note: some Linux computers may show the same values, and some may show different values. You are not required to explain any differences.

b. In your report, include the output that shows the addresses of two variables. What is the name of the memory segment(s) (far right column) in which those variables are located? What are the permissions for that segment? Do the permissions agree with the concept of a "constant" (read-only)?

c. Suppose "pmap" produced the output line shown below for a memory segment. What is the maximum hexadecimal address for that memory segment? What is the maximum possible hexadecimal starting address for a 4-byte integer in that memory segment?

```
00007f7c01248000      8K rw---   [ anon ]
```

4. Create a program that forks a child process, where the parent prints the child's process ID. Use "top" to show the PID and PPID (parent process ID). Note that you will need to have each process run long enough to allow you to capture the "top" output. To add the PPID column: (a) use SHIFT-F to open the Fields Management screen, (b) cursor down to the PPID field, press the SPACE bar to select it, and press ESC to return. You may modify any of the sample programs provided with the lecture notes. Provide (i) your program, (ii) program output, and (iii) the "top" output, including the PPID as explained above. **In your report, describe what you see for your processes' PIDs and PPIDs**.

5. The program "problem5.c" is provided with these instructions. Answer the following questions:

   a. How many child processes are created? Briefly explain your answer.
   b. Is it possible for two child processes to be executing concurrently? Briefly explain your answer.
   c. Would the program behave the same way if the if-else statements were replaced with the following? Briefly explain your answer.

```
if (pid == 0) { /* child process */
    printf("Child\n");
    return 0;
}
wait(NULL);
printf("Parent\n");
```