National University of Singapore
School of Computing

CS2106 **Laboratory 7** Semester 1 11/12

Deadline

30 October, 2011, Sunday, 10:00pm.

Platform

This exercise MUST be done using Linux installed in the OS lab.

Submission

Download the template for your solution:

```
wget http://www.comp.nus.edu.sg/~cs2106/lab07-A000000X.txt
```

Rename the file by replacing the string A000000X with your matriculation number.

Enter your answer into the text file and submit the file into IVLE Workbin (a folder named Lab 7) before the deadline. Please pay attention to the length limit of your answer.

Marking Scheme

This lab exercise is worth 15 marks, to be completed over two weeks.

0.1 marks will be deducted per minute after the deadline for late submission, and 3 marks will be deducted for file naming violation or format violation.

It might be good to review your Lab 1 before you begin this lab exercise.

This lab exercise requires you to examine the `proc` file system, which are a set of files located under `/proc` containing information about the current kernel states.

You can `man proc` to learn about the file system, but for this exercise, we are going to focus on the file `/proc/<process id>/maps`.

Suppose we have a process with process ID 1234. The file `/proc/1234/maps` contains information about the mapped memory regions. Go ahead and open up one such file belonging to a process you own.

You will see something like this:

```
00985000-00986000 r--p 0001c000 fd:00 49303      /lib/ld-2.8.so
```

The first column (`00985000-00986000`) is the address space occupied. The second column (`r--p`) is the permission that indicates how pages in that address space can be accessed. The first three characters here can be either `r`, w, x, or -, which corresponds to read permission, write permission, execute permission, and no permission respectively. The last character can be either `p` or `s`, corresponding to either private page or shared page.

The last column may show a filename (if the content of the memory is taken from a file) or special names, such as `[stack]` (if the region corresponds to the stack segment of the process).

We are not interested in the rest of the columns for now.

1. (4 points) **Examining Stack**

   Download, compile, and run the following program using the debugger[1]: `http://www.comp.nus.edu.sg/~cs2106/lab07-stack.c`.

   At the gdb prompt:

   ```
   (gdb) break main
   (gdb) display $pc
   (gdb) run
   ```

   The command `break main`, if you recall from your Lab 1, sets a breakpoint at the function `main()`. The command `display $pc` tells gdb to display the content of the program counter register every time the debugger stops. Finally, the command `run` runs the program within the debugger.

   After issuing the three commands above, you should see

   ```
   Breakpoint 1, main (argc=1, argv=0xbffff734 "o\370\377\277") at lab07-stack.c:12
   12                the_total = 5;
   1: $pc = (void (*)()) 0x80483df <main+9>
   Missing separate debuginfos, use: debuginfo-install glibc-2.13-2.i686
   ```

   Our program now stops just before executing Line 12. The value of the program counter is being displayed. In this case, the program counter is pointing to the address 0x80483df (your address might be different).

   (a) (0 points) Use `ps` to find out the process ID of the process you are debugging (note: not the process ID of gdb). Let p be the process ID. Open up the file `/proc/<p>/maps` and examine the content. You may want to refer to the content of this file when explaining some of your observations below.

[1] remember to compile with -g flag

(b) (1 point) Type `n` or `next` at the (gdb) prompt *five* times. This command asks gdb to execute the current line of code and move to the next line. Note down the different values of program counter you see after every `next` command.

Why is the value of last program counter displayed vastly different from the previous ones?

(c) (1 point) After the last set of commands, you should have exited from the program (but still remains in the debugger). Set a new breakpoint at the beginning of function `foo` and run your program again.

Since we still have a breakpoint at main(), the debugger will first stop there. Type `c` or `continue` at the prompt of gdb to continue. The debugger should stop at the beginning of function `foo` now.

Examine the addresses of the variables `the_total` and `x`? Why is there a significant difference in the addresses of these two variables?

(d) (0 points) We will now examine the content of the stack. You can print out the content of the stack point (SP) register with:

```
(gdb) x $sp
```

You will see two numbers for each `x` command. The first is the content of the SP register (which is an address), and the second is the content of that address.

(e) (2 points) Now, examine the output of the following commands.

```
(gdb) x $sp+4
(gdb) x $sp+8
(gdb) x $sp+12
(gdb) x $sp+16
```

You have looked at the content of the call frame on top of the stack that foo() has created. Identify where (i) the return address, and (ii) the parameter passed to foo is stored. Write your answer in the form of `$sp+x$`.

(f) (0 points) To verify the explanation you gave in Part (c), you can type `fin` or `finish` to step out of the function `foo`, and examine the content of the stack again.

```
(gdb) x $sp
(gdb) x $sp+4
(gdb) x $sp+8
(gdb) x $sp+12
(gdb) x $sp+16
```

2. (5 points) **Virtual Memory**

For this question, you need at least three terminal windows. On one terminal window, run

```
top -u userX
```

where `userX` is your user id. You should see a list of your processes listed and updated periodically. We are interested in the columns labeled `VIRT`, `RES`, `SHR`. Read the man page for `top` to find out what they mean. The list of processes displayed are, by default, sorted by the share of CPU time a process takes (since the last update), which is shown in the column `%CPU`.

Keep this window opened and visible for the rest of this question.

Do the following with a second terminal window. Download, read, and understand the code: `http://www.comp.nus.edu.sg/~cs2106/lab07-malloc.c` Now, compile and run the program above.

(a) (2 points) While the program is running, observe the output of `top` in the other window, paying attention to the columns `VIRT`, `RES`, `SHR`, and `%CPU` and how they change over time.

Write down your observations and explain what happen.

(b) (1 point) Wait until your program terminates. How many bytes can be allocated by a process in your system?

(c) (1 point) Now, run the same program again on the second terminal window. Observe the process ID of the corresponding process. Let say it is p.

Repeat the following command a few times with a few seconds of interval in between. You have to do this quickly before the program terminates.

```
cp /proc/<p>/maps maps.<n>
```

where $< p >$ is the process ID of your process, and $< n >$ is 1, 2, 3, etc.

You should now have several copies of the `maps` file of the process, each is a snapshot at different time.

Compare the content of the maps files. Do you see any difference in the `maps` files? What are the differences (if any)? Explain.

Hint:

- You can use the command `diff` to compare two files.
- To verify your answer, you may want to change the program to print out the pointer returned by `malloc`.

(d) (1 point) Now, copy `lab07-malloc.c` to a new file called `lab07-memset.c`. Uncomment the following line:

```
memset(x, i, MEGABYTE);
```

(The line marked with `Line A`.)

Read the man page to find out what this line does.

Now, compile and run the program. Observe the output of `top` in the other window, paying attention to the columns `VIRT`, `RES`, `SHR`, and `%CPU` and how they change over time.

Write down your observations and explain what happen, comparing your observations this time with what you observed in Part (a).

3. **(6 marks)** The next exercise helps us estimate the overhead of page faults in our system. Download and read two slightly different programs from the URLs below:

   - `http://www.comp.nus.edu.sg/~cs2106/lab07-time1.c`
   - `http://www.comp.nus.edu.sg/~cs2106/lab07-time2.c`

   The given programs take in a single parameter, which is the number of pages to `malloc()`. Each page is 4KB[2].

   `lab07-time1.c` goes through each page and touches each page once by modifying the content of one memory location in each page. `lab07-time2.c` goes through 1/4 of the pages and touches each page four times, by modifying the content of four memory locations within a page each time. Both programs make the same number of modifications.

   The average time (in μs) to modify the content of a memory location is printed as the output.

   (a) (2 points) Estimate the number of page faults that would occur for both programs if they are executed with command line argument N, where N is a large number.

   (b) (1 point) Now run both programs with some large values of N as input. Pick several values with different order of magnitude (1000, 10000 etc.). Note down your output.

   Note that you should pick an N that is large enough so that the timing and looping overhead is negligible. One way to verify this is to make a copy of `lab07-time1.c` and comment out the line inside the for loop (but not the for loop itself). Now you have a program that measures the timing and looping overhead.

   If N is too big, `malloc()` will fail.

   (c) (2 points) Let m be the overhead of a page fault and h be the overhead of page access. Form two linear equations relating m, h, and the timings in (b) above.

   (d) (1 point) Using the two linear equations above, estimate the overhead of a page fault in your system. Show your workings.

4. (0 points) **(0 marks)** Did you notice that running the same program with the same argument would result in different regions of memory allocated almost every time? Why? (Hint: ASLR)

THE END

[2] you can verify this by running the command `getconf PAGESIZE`