Lab #3: C, Processes, and Memory

Introduction

This lab is a review of C programming and how processes work with memory at a lower level.

Learning Objectives

1. Review C programming language: pointers and structs
2. Understand how C program use stack space and heap memory.
3. Understand how to use the C standard library

Part I C programming

C programs have different sections like the code, global variables, heap, shared library code, and the stack. We can figure out where something is by getting the "address of" - the & operator – a variable, function, or other things in the program itself. For example the following C program:

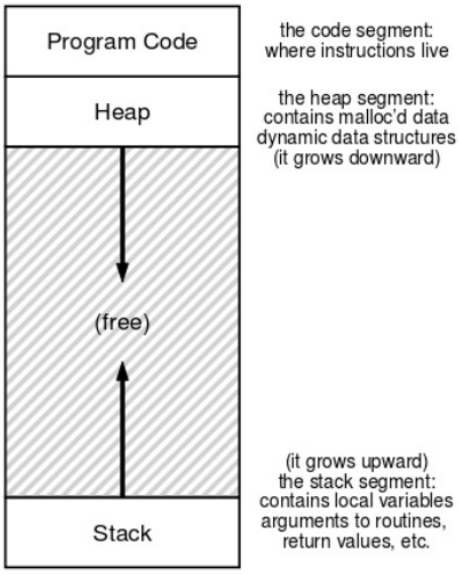
```c
/*-----------------------------------------------------------------------
 *  map.c - prints out addresses so we can make a map of memory
 *
 *-----------------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>

char x   = 'a';
int  y   = 10000;
double z = -10.0;

int main(void) {
   printf(" %p - main code\n\n", &main);
   printf(" %p - x:    0x%02x\n", &x, x);
   printf(" %p - y:    0x%08x\n", &y, y);
   printf(" %p - z:    0x%016lx\n\n", &z, (unsigned long)z);

   int* a = malloc(10);
   a[0] = 92; a[9] = 16;
   printf(" %p - a[0]: 0x%08x\n",    a,    a[0]);
   printf(" %p - a[9]: 0x%08x\n\n", &a[9], a[9]);

   printf(" %p - printf code\n\n", &printf);
   printf(" %p - a:    %p\n", &a, a );
}
```

will print out something like this when it is run on a 64-bit Ubuntu OS running on an x86_64 CPU:

```
cs429 $ ./map
 0x562a35ffa6ea - main code

 0x562a361fb010 - x:    0x61
 0x562a361fb014 - y:    0x00002710
 0x562a361fb018 - z:    0xfffffffffffffff6

 0x562a36ead670 - a[0]: 0x0000005c
 0x562a36ead694 - a[9]: 0x00000010

 0x7f1c1be2ce80 - printf code

 0x7ffc357a65c0 - a:    0x562a36ead670
```

The addresses – the locations of variables and other things – for your run may not appear as shown above and may even change between runs of this program because most operating systems use something called “address space layout randomization” to deter certain kinds of malware. These addresses are virtual addresses and the above output tell us roughly where things are in a C program.

We know that a C program has a code segment, a heap segment, and a stack segment as shown to the right:



By looking at addresses of variables when we run a program, we can make a **memory map** of virtual memory addresses any C program:

1. Run the **`map.c`** program and show the outputs of 3 runs (on your VM or computer) in your report. Use Word, LibreOffice Text/Draw, or Adobe Illustrator (no hand-drawn pictures or screen shots) to show the memory map of your last run of the **`map`** program.

2. Compile and run the C program, **`lab3n1.c`**. In Lab #2, we saw how to use **`objdump`** to get the assembly language equivalent of a C program. Use `objdump` to disassemble the `lab3n1` executable.

   In your report, show the disassembled main function and make sure each C statement is displayed in the disassembled code. Numbers are represented using hexadecimal notation in assembly language. We can spot hex numbers by looking for a leading "**`0x`**" before the number.
   Highlight the code that implements the for loop. Explain how the assembly language instructions that make up the for loop work:
   - which assembly language instructions implement the i=i+10; and j=j+17; statements?
   - do you see a 10 or a 17 in the assembly code? Why or why not?
   - where are the assembly language "jump" instructions jumping to?
   - which assembly language instruction(s) implement the ++ operation?

Make files

C programmers use the **`make`** utility and set up `Makefile`s to make the process of compiling C programs simpler. for example, the following Makefile creates two object files (`one.o`, `two.o`) and an executable (`one`) file:

```
all: one

one.o: one.c
	gcc -c one.c

two.o: two.c
	gcc -c two.c

one: one.o two.o
	gcc -o one one.o two.o
```

Read this wikipedia page on the make program and make sure you understand:
- which files the **`make`** program looks for when it is run
- what a target of a Makefile is
- what **`make`** does when is run without any targets
- how **`make`** figures out from a Makefile when to recompile source code

C `structs`

We use a C struct to put things of different types into one object. An example of a simple struct:

```
struct MyStructure{
	int  id;
	char group;
	double balance;
};
```

Note: a semicolon is needed at the end of a `struct` declaration.

While arrays are collections of elements of the same type, **`structs`** are collections of elements of different type. In the above example, we are declaring a new type called "`MyStructure`" with 3 elements or data members: `id`, `group`, and `balance`. To create an instance of this new type, we use:

```
struct MyStructure x;
```

This declaration creates an object named x but does not initialize the data members. To set a "field" or a data member, we use the "dot" operator (similar to the way we access a data member in a Java class):

```
x.id = 7;
x.group = 'A';
```

to access the data members for reading or writing to.

It is possible to have arrays of structs:

```
struct MyStructure club[ 100 ];
```

structs within structs:

```
struct BigStructure{
      int district;
      MyStructure account;
};
```

and pointers to structs:

```
struct MyStructure* px = &x;
```

Pointers to structs can be dereferenced in two ways to access the individual data members of the struct object they are pointing to. One is the way we have already used earlier, using the asterisk dereferencing operator:

```
(*px).id
```

and the other uses the "arrow" operator:

```
px->id
```

The latter (the arrow) is often preferred when using structs as it is a little more readable but they are equivalent.

Note, however, that there is a big difference between:

```
(*px).id
```

which is saying "in the thing that px is pointing to, get the data member id", versus:

```
*px.id
```

which is actually equivalent to `*(px.id)` and says: "the thing that id is pointing to, with id being a data member of the object px". In the first case (`(*px).id`) the asterisk dereferencing operator is applied to px and in the latter case (`*px.id`), the asterisk is applied to id, not px.

To summarize the syntax for accessing data members in a struct:

`o.d`	Data member `d` in object `o`
`p->d`	Data member `d` in object pointed to by pointer `p`
(*p). d	Data member `d` in object pointed to by pointer `p`
*o.p	Memory pointed to by the data member `p` in object `o`
*(o.p)	Memory pointed to by the data member `p` in object `o`

Java programmers may recognize some similarities to Java classes: `struct`s are similar to classes. There are two ways of using `struct`s: one is the **C style**, in which structs have only public data members and no methods (because that is the only kind available in C) and the other way of using `struct`s is the **C++** style, in which structs can have both data members and method members with different levels of visibility (private, public, etc) and inheritance. In fact, there is not much difference between C++ `struct`s and classes. In this course we will write C programs and so we will use C structs.

We allocate memory for a struct using the malloc system call to get space from the heap:

```
struct MyStructure* qx = (struct MyStructure*)
                          malloc( sizeof( struct MyStructure ));
```

Note that the above statement is a bit long-winded and repetitive.
To make statements like this more succinct, we can use a **typedef**:

```
typedef struct MyStructure Account;
```

and with this typedef, the earlier statement becomes:

```
Account* qx = (Account*) malloc( sizeof( Account ));
```

For more details on structs and their use see this Wikipedia article.

2. Write a C program, **lab3n2.c**, that will print the sizes of "MyStructure", the **x** and **px** variables, and the **club** array (all shown above) and submit the lab3n2.c file. In your report document, note how big MyStructure, **x**, **px**, and **club** are and explain these sizes from knowing the sizes of their components.

Part II Stack frames, Heap memory

3. Compile and run the **lab3n3.c** program. In your report, show the output of the program and note the locations of the different C variables. Draw a "map" of the memory locations showing memory addresses (in increasing order) and the byte locations of the variables, the locations on the heap that variables point to, the variables on the stack, and the locations of functions of this C program.

4. Compile and run the **lab3n4.c** program. This program builds a linked list of nodes on the stack. Make a copy of this program named **lab3n5.c** and have lab3n5 ask the user how many nodes the linked list should have. Then set up a loop in which **malloc** is called to allocate memory from the **heap** for each node in the linked list. Assign a random value for the int data field of each node. As each node is created, it should be added to some location in the linked list so that we end up with a linked list that has as many nodes as the user requested. When it is done creating the linked list, the program should print the linked list using the printList function.
Submit the **lab3n5.c** program in your .tar.gz file.

Part III Using libc functions to access the file system

libc is the "standard library" for the C language. It is a collection of functions you would want to use from a C program to access lower level operating system features no matter which instruction set your CPU has. Examples of functions are:

- printf – to output text to a terminal or file
- open, read, write – for low-level file operations
- malloc – to allocate memory from the heap space
- inet, getaddrinfo – for network addresses

The idea is to make C a portable programming language that we could use on any computer from Raspberry Pis and smartphones to desktops, servers, and supercomputers, we will need some "core" functions that will work the same way on these various hardware platforms (32- and 64-bit Intel, ARM, PowerPC, etc) and operating systems (Linux, macOS, Windows). In addition to the code for these functions, libc also has many data structures, often defined as C structs in include files like stdlib.h file that is usually in the /usr/include directory in Linux and macOS systems.

Turns out it's not a perfect "standard" mainly because many organizations and vendors supply slightly different versions of libc. We will use one of the more widely used ones, **glibc**, the GNU C library on a 64-bit Linux system on an Intel/AMD core in this course.

5. Take a look at the "`lab3n6.c`" C program and notice that it uses a number of C functions you may not have seen before along with the usual `printf` function. Rread the man pages for the `opendir`, `readdir`, and `closedir` functions so you know what they do and how they might use the **`dirent`** struct. Compile and run this program.

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>

char* dir = "/";

int main(){
   DIR* d;
   struct dirent* e;

   printf("3 entries in %s:\n", dir);
   d = opendir( dir );
   e = readdir( d );  printf("  %s\n", e->d_name );
   e = readdir( d );  printf("  %s\n", e->d_name );
   e = readdir( d );  printf("  %s\n", e->d_name );

   closedir( d );
   return 0;
}
```

What does this program do? Make sure to read the man pages for opendir, readdir, and closedir before you explain what the above program does. Modify this program to list **all** entries in the specified directory and submit the revised source code with your report. Highlight your changes to the code in your report.

6. The next program (see the file "`lab3n7.c`") can detect errors in glibc calls:

```
#include <stdio.h>
#include <sys/stat.h>
#include <errno.h>
#include <string.h>

int main(int ac, char** av){
   int i=0;
   struct stat statbuffer;

   errno = 0;
   if (ac < 3) {
      printf(" %s: Usage: %s <file1> <file2>\n", av[0], av[0] );
      return -1;
   }

   lstat( av[1], &statbuffer );
   if (errno != 0) perror( av[1] );
   else
     if ( S_ISDIR( statbuffer.st_mode ))
       printf("    %s is a directory \n", av[1] );
     else
       printf("    %s is not a directory \n", av[1] );

   lstat( av[2], &statbuffer );
   if (errno != 0)
    fprintf( stderr, " %s: Error detected -- %s\n",
                          av[2], strerror( errno ) );
   else
      if ( S_ISDIR( statbuffer.st_mode ))
         printf("    %s is a directory \n", av[2] );
      else
         printf("    %s is not a directory \n", av[2] );
```

```
    return 0;
}
```

Read the man page for **`lstat`**, run this program with various command line parameters and find out what it does. Modify it to use the appropriate macro so that the program can detect if each command line argument is an ordinary file, a directory or something else and print an appropriate message. Submit the modified source file and in your report, highlight the changes you made.

7. Read the man page for the `getcwd` function. Take a look at man page for the C function called `basename` using the command:
   `man 3 basename`.
   This man page is a bit different from others you may have seen and you may want to read all of it.
   Compile and run the file "`lab3n8.c`":

```
#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char* argv[]){
   char* path = strdup( argv[0] );
   char* program_name = (char*) basename( path );
   char* here;
   const int size_here = 300;
   fprintf(stdout, "%s: starting ... \n", program_name );

   here = (char*) malloc( size_here );
   if ( getcwd(here, size_here) )
     printf("%s: executing in %s\n", program_name, here);
   else {
      fprintf(stderr, " %s: Unable to get wd.\n", argv[0] );
      return 1;
   }
}
```

   In your lab report, explain what this program does. Show 3 sets of inputs you can use to test the program and explain how each input set tests this program.

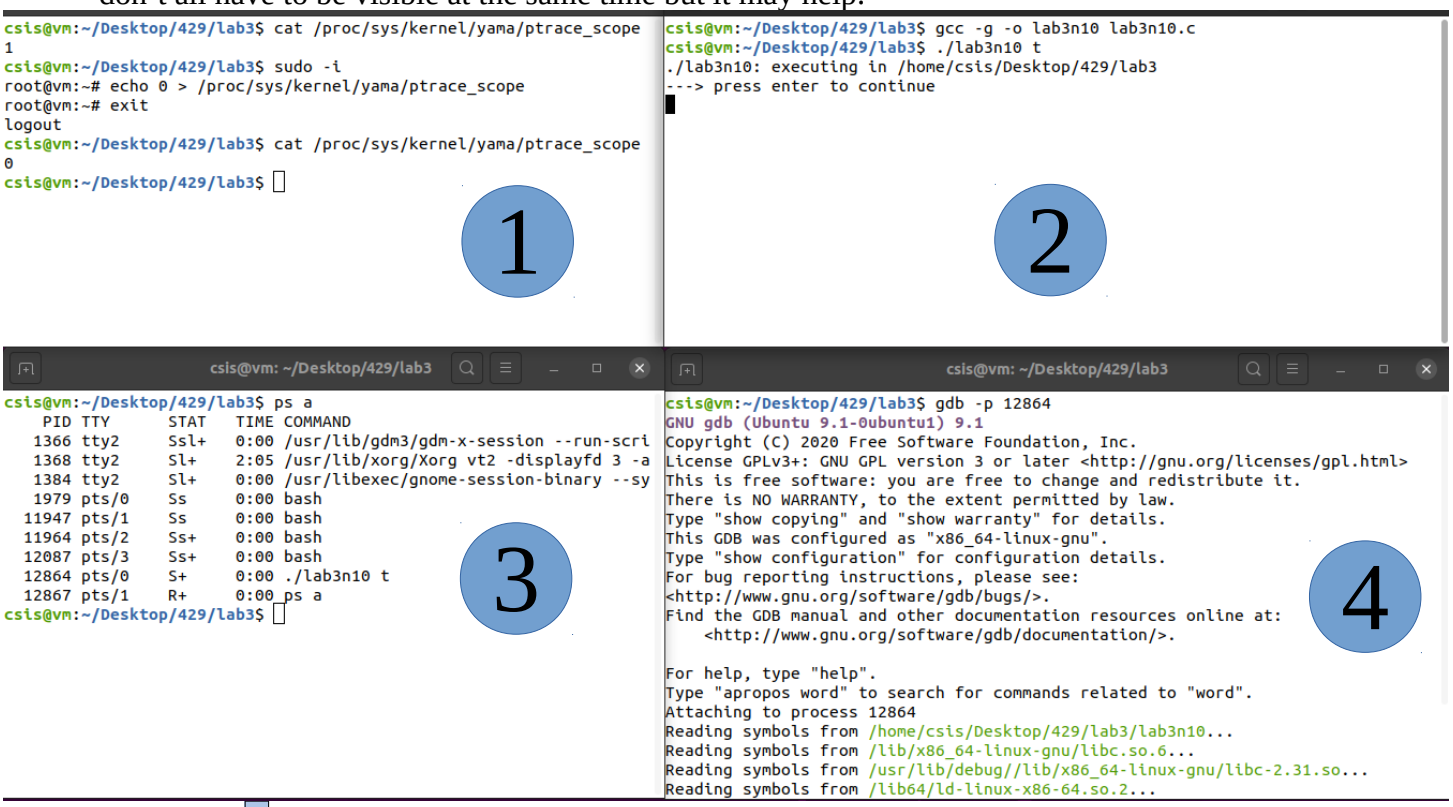
8. Read the man page for the `chdir` C function. Compile and run the file "`lab3n9.c`":

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char* argv[]){
   char* here;
   const int size_here = 100;
   int status = 0;
   char* sub = "test.dir";

   here = (char*) malloc( size_here );
   if ( getcwd(here, size_here) ) {
     printf("%s: executing in %s\n", argv[0], here);

     status = chdir( sub );
     if (status == 0){
        char* here_now;
        int size_here_now = 0;

        size_here_now = strlen(here) + strlen(sub) + 1;
        here_now = (char*) malloc( size_here_now );
        strncpy( here_now, here, size_here_now );
        strncat( here_now, "/", 1 );
        strncat( here_now, sub, strlen(sub) );
```

```
            printf(" %s: now executing in %s\n", argv[0], here_now );
        }
    } else {
        fprintf(stderr, " %s: Unable to get wd.\n", argv[0] );
        return 1;
    }
}
```

What does this program do? What are the inputs to the program? Note that there could be inputs on the command line and also input files and other things that the program expects. Explain in your lab report. Show 3 sets of input you can use to test the program and in your lab report how each input set tests this program.

9. Debugging a running process: For this step, you will need 4 Terminal windows as shown below. They don't all have to be visible at the same time but it may help:



In window 1 use the `cat` command as shown above to take a look at the contents of the kernel file `ptrace_scope` in the `/proc/sys/kernel/yama` directory. Ordinarily it should be a 1 to indicate that the kernel should prevent all processes from accessing the memory of unrelated processes. Start a root shell using an interactive `sudo` session and the `echo` command to set the contents of `ptrace_scope` to 0 to allow us to start a process that can be accessed from another process.

Note: We are purposely turning off a kernel security mechanism in order to accomplish a particular debugging task. This should really only be done in a VM since forgetting to reset this security mechanism can leave your entire system vulnerable to attacks and damage the OS! We will use this lowering of security to run the `lab3n10.c` program in one window (2) and debug it from another window (4)

In window 2, compile the `lab3n10.c` program with the debug mode on and run it to see what it does – make sure to take a look at the code so that you can understand how it works. If the program crashes with a Segmentation fault when you run it, use `gdb` to see why it crashes and make sure you know how

to prevent it from crashing. Once you have figured this program out, run it as shown in the above screenshot.

Once you have a debug version of the `lab3n10.c` program running in window 2 and it's waiting for user input, in window 3, use the command "**`ps a`**" as shown above to get a list of your current processes and one of them (only 1) should be `lab3n10`. This will tell us the process id (**`PID`**) of the `lab3n10` process – in the example above, it was 12864 but yours will be different.

Finally, we are ready to debug this running `lab3n10` process whose PID we now know. We can do this by using the **`gdb`** command in window 4 with the **`-p`** flag and the PID we saw in window 3. This is why we needed to turn off that kernel security mechanism – so that our `gdb` process can access the internals of the `lab3n10` process.

Note: if `lab3n10` crashes or completes before we are done debugging it, we will have to restart it, find the new PID and then run **`gdb`** again to command to attach that running process and debug it.

We are now ready to debug a running program. In window 2 enter one or more letters like "x" or numbers followed by the Enter key (Return on a Mac) and in the debug window 4 we can step through the code using the "next" or "n" gdb command a few times until we get to a line in the `lab3n10.c` program as shown here:

```
Reading symbols from /usr/lib/debug//lib/x86_64-linux-gnu/libc-2.31.so...
Reading symbols from /lib64/ld-linux-x86-64.so.2...
--Type <RET> for more, q to quit, c to continue without paging--c
(No debugging symbols found in /lib64/ld-linux-x86-64.so.2)
0x00007feadf348fb2 in __GI___libc_read (fd=0, buf=0x556022d88720, nbytes=1024) a
t ../sysdeps/unix/sysv/linux/read.c:26
26      ../sysdeps/unix/sysv/linux/read.c: No such file or directory.
(gdb) n
_IO_new_file_underflow (fp=0x7feadf423980 <_IO_2_1_stdin_>) at fileops.c:519
519     fileops.c: No such file or directory.
(gdb) n
526     in fileops.c
(gdb) n
535     in fileops.c
(gdb) n
537     in fileops.c
(gdb) n
__GI__IO_default_uflow (fp=0x7feadf423980 <_IO_2_1_stdin_>) at genops.c:363
363     genops.c: No such file or directory.
(gdb) n
365     in genops.c
(gdb) n
pauser () at lab3n10.c:20
20              c = getchar();
(gdb)
```

In your report, show the output of the the `gdb` command **`info frame`** while you are in the `pauser` function. It should show us low-level details of the stack in the particular function call we are currently in – this will be important in the next lab and so make sure you can figure out as much as possible about stack frames now.

You may have to try this whole process a few times to get it just right but it should work eventually and you will be able to figure out how to debug one process from another - a very important skill to have and one that is used by IDE debuggers!

Step through the code until you can **`step`** into the `check` method and successfully change the working directory of the process to the command line argument you used. While still in the `check` method, show the output of the `gdb` command **`info frame`**.

Make a copy of `lab3n10.c` called `lab3n11.c` and modify `lab3n11.c` so that after after successfully executing the `chdir` function, it prints a list of all the files and directories in the current working directory. In effect, your `lab3n11` program should work like a combination of the `cd` and `ls` commands without actually using these commands!

When you are done with this step, be sure to set the contents of `ptrace_scope` back to 1 so that our system is back to its original security settings.

10. The textbook, "Advanced Programming in the UNIX Environment", 3rd edition (APUE3) is an excellent overview of the kinds of things you can do with low-level operating system features from C programs.
    Even though the title of the book has "UNIX", most of the programs in the book are POSIX compliant.
    POSIX stands for "Portable Operating System Interface" and is a set of standards based on Unix that would allow developers/programmers to write code that could work on any OS that supported the various versions of the POSIX standard. POSIX defines an API for C programs and this API is like `libc` but is "larger" than `libc` and uses many of the functions and data structures in `libc`.
    The related Single Unix Specification (SUS) standard is now built around POSIX.
    As a result of standards like these being widely adopted, the programs in the book will also work on Linux, macOS and properly configured Windows systems.

    Take a look at the first C program in the APUE3 book on page 5 (Figure 1.3). Notice that it uses an include file named "**`apue.h`**" which is listed on pages 895-898. This header file is used in just about all the C programs listed in the book and as a result it has more code that we use in most programs.

    Take a look at which header files in `/usr/include` are used to accomplish some basic tasks in C. Figure out how to compile and run the 12 programs found on these pages:
    p. 5 – Figure 1.3
    p. 9 – Figure 1.4
    p. 10 – Figure 1.5
    p. 11 – Figure 1.6
    p. 12 – Figure 1.7
    p. 15 – Figure 1.8
    p. 17 – Figure 1.9
    p. 19 – Figure 1.10
    pp. 50-51 – Figure 2.16
    p. 52 – Figure 2.17
    pp. 68-69 – Figure 3.2
    p. 72 – Figure 3.5

    Rewrite all of these programs to include only those standard C header/include files needed by each program. That is, rather than using the `apue.h` file or any similar include file you define, find out which standard C include files in `/usr/include` are needed by each program and **only** `#include` those files needed for each program to compile correctly.
    Include all of these programs in the `tar.gz` file you submit for this lab. Make sure your `Makefile` compiles all of these without errors.

Submit a single tar.gz file containing
- your report document
- C programs as separate .c files
- A Makefile that can compile all C programs with one make command