Process Laboratory

CS 105, Fall 2019

Due on Tuesday, September 24, 2019, at 11:59 PM

The exercises in this laboratory are designed to introduce you to the idea of processes—what they are, how they are expressed in the C language, how they use memory, how they are executed on a computer, and how you can watch the execution in a debugger.

As with the previous lab, work in teams of two. You may choose your own partner—anyone except your partner for the Data Lab.

To get started, ssh into the pom-itb-cs2 server, create a directory in which to work on this laboratory, and make it your working directory. Then use

```
% tar xvf /data/processlab.tar
```

to unpack the tar file into your working directory. You may also want to re-mount your cs105 folder to the server, as we did in the Data Lab.

You will now have a directory `processlab` which contains this writeup, four short C language programs, and a template for you to fill complete and submit. Begin by opening the file `processlab.txt` in an editor and put the names of all the team members at the top. Fill in your results as you work on the lab. When you have completed the lab, submit the file `processlab.txt` on the course submission page. Only one team member should submit the result. You may, of course, submit updates to your work; just be sure that everything is submitted by the same team member.

The five parts of this laboratory are weighted equally. Your score will be based on a total of 50 points.

1 Re-interpreting Data Values

We begin with a short exercise in the C language. In many ways, the C language is like Java. The syntax for variable and function declarations, assignment statements, `for` and `while` loops, and `if`-statements are the same in both languages. The big difference is that in C we have a different view of data, one that is closer to the actual hardware. We must be aware of where in memory values are located and how much space they occupy. This part of this laboratory assignment will give you practice in thinking about variables, pointers, and arrays—and how they relate to addresses and values in memory. Later parts will extend that understanding to machine instructions and how they are stored in memory.

The program `plab1.c` reads six integers into an array. It then interprets the first four integers as a string and

Credits: Many of the exercises here are adapted from the Debugger Lab written by Professor Geoff Kuenning of Harvey Mudd College. We are grateful to him.

the last two as a `double` and prints the results. Your job is to find integers that will cause the program to print some specified values.

Begin by reading the program and understanding what it does. Become familiar with the notation used in the program, particularly the parts using pointers and casts. Then compile it by tying

```
% gcc -o plab1 plab1.c
```

on the command line. Do not change the program's source. Next, create a text file named `plab1.soln` with six integers, one to a line. Begin by making them all zero. Run the `plab1` program with the input redirected from the `plab1.soln` file.

```
% ./plab1 <plab1.soln

0.0000000000000000
```

The blank line in the output shows that the string is empty. The `double` is zero. As a further warm-up, change the first of the six integers to 14132. The string now has two characters—which happen to be digits. (It is a string of characters, not a number!) The `double` is still zero.

```
% ./plab1 <plab1.soln
47
0.0000000000000000
```

1. Fill `plab1.soln` with six integers to produce this result.

   ```
   % ./plab1 <plab1.soln
   Cecil Sagehen
   3.1415926535897931
   ```

   When you have the solution, copy the six integers, one to a line, under the heading Part 1 in the file `processlab.txt`.

   Suggestion: It is possible, but long and tedious, to compute by hand the four integers corresponding to "Cecil Sagehen." But it is not practical (or a good use of your lifespan!) to compute the two integers corresponding to the decimal expansion of π. Think about writing a short program, separate from `plab1.c`, that will calculate the integers for you.

2. The six-integer sequence you produced is not unique. Other sequences will produce the same result. How many different solutions are there?

Before proceeding to the next part, take a few minutes to reflect on what you did. The actual values of the integers are not important. If they were all you cared about, you could ask someone in the lab. Be sure that you understand what is happening with the bytes in memory. Also, take some time to understand the pointer arithmetic and type casts in the source file `plab1.c`.

2 Looking at Data in the Debugger

The second part of the lab continues our tour of data representation. The debugger `gdb` lets you look at data at the the bit- and byte-level.

Open the file `plab2.c`. It file contains three `static` constants and and a short `main` function. The function is there only so that the program will compile. Right now, we are only concerned with the data.

Compile the program without optimization (but with the `-g` flag!) and bring up the debugger on it.

```
% gcc -g -o plab2 plab2.c
% gdb plab2
```

Put your answers to the following questions under the heading Part 2 in `processlab.txt`.

1. `gdb` provides you lots of ways to look at memory. For example, type "`print puzzle1`." What is printed?

2. Gee, that was not very useful. Sometimes it is worth trying different ways of exploring things. How about "`p/x` puzzle1"? What does that print? Is it more edifying?

3. You've just looked at puzzle1 in decimal and hex. There is also a way to treat it as a string, although the notation is a bit inconvenient. The "`x`" (examine) command lets you look at arbitrary memory in a variety of formats and notations. For example, "`x/bx`" examines bytes in hexadecimal. Try it by typing "`x/4bx &puzzle1`." (The "`&`" symbol means "address of"; it is necessary because the `x` command requires addresses rather than variable names.) How does the output you see relate to the result of "`p/x puzzle1`"? (Incidentally, you can look at any arbitrary memory location with `x`, as in "`x/wx 0x8048500`".)

4. OK, that was interesting and a bit weird (and we will be covering it in class soon). But we still do not know what is in `puzzle1`. We need help! And fortunately `gdb` has help built in. So type "`help x`". Then experiment on `puzzle1` with various forms of the `x` command. For example, you might try "`x/16i &puzzle1`". (`x/16i` is one of our favorite `gdb` commands—but since here we suspect that `puzzle1` is data, not instructions, the results might be interesting but probably not correct.) Keep experimenting until you find a sensible value for `puzzle1`. What is the human-friendly value of `puzzle1`? Do not accept an answer that is partially garbage!

   Hint: Although `puzzle1` is declared as an `int`, it is not. On our machine an `int` is 4 bytes, 2 halfwords, or one—in `gdb` terms—word.

5. Having solved `puzzle1`, look at the value carefully. Is it correct? (You might wish to check it online.) If it is wrong, why is it wrong?

6. Now we can move on to `puzzle2`. It pretends to be an *array* of `ints`, but you might suspect that it is something else. Using your newfound skills, figure out what it is.

   Hint: since there are two `ints`, the entire value occupies 8 bytes. What is the human-friendly value? It iss not "105."

7. Are you surprised?

8. Is it correct?

9. We have one puzzle left. By this point you may have already stumbled across its value. If not, figure it out; it is often the case that in a debugger you need to make sense of apparently random data. What is stored in `puzzle3`?

3 Debugging Optimized Code

Let us now move on to exploring instructions and their execution. The file `plab3.c` contains a function that has a small `while` loop, and a simple `main` that calls it. Briefly study the `loop_while` function to understand how it works. It is not necessary to fully decode it right now; just get a clue about what i going on.

There are a few details that are not obvious.

- Find out what the `atoi` function does by typing "`man atoi`" in a terminal window. (The function name is pronounced "ay-2-eye".)
- The `printf` function is quite complicated, for now we will just say that it prints answers, and `"%d"` means "print in decimal." We encourage you to read more about `printf` in Kernighan & Ritchie or online. The advantage of reading in K&R is that the description there is less complex.
- Finally, `argv` is an array containing the strings that were passed to the program on the command line (or from `gdb`'s `run` command); `argc` is the number of arguments that were passed. By convention, `argv[0]` is the name of the program, so `argc` is always at least 1.

Compile the program with the `-g` switch and with *no* optimization and then run it in the debugger.

```
% gcc -g -o plab3 plab3.c
% gdb plab3
```

Set a breakpoint in `main`, tell `gdb` not to debug the `atoi` function, and then run the program.

```
(gdb) break main
(gdb) skip atoi
(gdb) run
```

(Often commands in `gdb` may be abbreviated by single letters, like `b` or `r`.) The program will stop at the beginning of `main`.

Put your answers to the following questions under the heading Part 3 in `processlab.txt`.

Suggestion: In the instructions below, the italicized comments show the breakpoint where the debugger should have stopped for that step. Use them to keep track of where you are in the program. Do not be afraid to start over if you lose track or become confused.

1. *Existing breakpoint at `main`.*
   Type "`c`" (or "`continue`") to continue past the breakpoint. What happens?

2. *Existing breakpoint at `main`.*
   Type "`bt`" (or "`backtrace`") to get a trace of the call stack and find out how you got where you are. Take note of the numbers in the left column. Type "`up` *n*", where *n* is one of those numbers, to get to `main`'s *stack frame* so that you can look at main's variables. What file and line number are you on?

3. *Existing breakpoint at `main`.*
   Usually when bad things happen in the library it is your fault, not the library's. In this case, the problem is that `main` passed a bad argument to `atoi`. There are two ways to find out what the bad argument is: look at `atoi`'s stack frame, or print the argument. Figure out how to look at `atoi`'s stack frame. Can you see the value that was passed?

4. *Existing breakpoint at `main`.*
   The lack of information is sometimes caused by compiler optimizations, other times by minor debugger issues. In either case it is a nuisance. Rerun the program by typing "`r`" and let it stop at the breakpoint. Note that in step 2, `atoi` was called with the argument "`argv[1]`". You can find out the value that was passed to `atoi` with the command "`print argv[1]`". What is printed?

5. *Existing breakpoint at `main`.*
   The number you see is the value of a NULL pointer. Like many library functions, `atoi` does not like NULL pointers. Rerun the program with an argument of 5 by typing "`r 5`". Continue from the the breakpoint. What does the program print?

6. *Existing breakpoint at `main`.*
   Without restarting `gdb`, type "`r`" (without any further parameters) to run the program yet again. (If you restarted `gdb`, you must first repeat Step 5.) When you get to the breakpoint, examine the variables `argc` and `argv` by using the `print` command. For example, type "`print argv[0]`." Also try "`print argv[0]@argc`", which is `gdb` is notation for saying "print elements of the `argv` array starting at element 0 and continuing for `argc` elements." What is the value of `argc`? What are the elements of the `argv` array? Where did they come from, given that you did not add anything to the `run` command?

7. *Existing breakpoint at `main`.*
   The `step` or `s` command is a useful way to follow a program's execution one line at a time. Type "`s`". Where do you wind up?

8. *Existing breakpoint at `main`.*
   `gdb` always shows you the line that is about to be executed. Sometimes it is useful to see some context. Type "`list`" What lines do you see? Hit the return key. What do you see now?

9. *Existing breakpoint at `main`.*
   Enter "`s`" to step to the next line. Then hit the return key three times. What do you think the return key does?

10. *Existing breakpoint at `main`.*
    What are the values of `result`, `a`, and `b`?

11. Type "`quit`" to exit `gdb`. (You will have to tell it to kill the "inferior process", which is the program you are debugging. Insulting!) Recompile the program, this time optimizing it with `-O1` (and including `-g` for debugging). Debug it, set a breakpoint at `loop_while` (not at `main`!), and run it with an argument of 10. Step three times. What four lines of code from `program1.c` are shown to you? Why do you think the debugger is showing you those lines in that order?

12. Quit `gdb` again and recompile with -O2. Debug the program. Disassemble the `main` function by typing "`disassem main`". What is the address of the instruction that calls `atoi`? What is the address of the instruction that calls `printf`? You will have to do some deduction here, because `gcc` mangles the names a bit.

13. What is the address of the instruction that calls `loop_while`? *Hint: don't spend too much time on this one*

14. A handy feature of `print` is that you can use it to convert between bases. For example, what happens when you type "`print/x 42`"? How about "`p 0x2f`"?

15. We may not have covered it yet, but functions return results in `%rax` (also known, for this problem, as `%eax`), so the result of `atoi` will be in `%rax`. After the call to `atoi` there is a `lea`, a `mov`, a `nopl`, then an `add` and a `sub`. Where does the constant in the `sub` come from?

16. Now you (kind of) understand the optimized `main`. What happened to the call to `loop_while`?

17. If you compile with `-O3` and disassemble the main program, you will discover that all traces of the loop have disappeared. We will not try to analyze it now—in particular, some of the instructions may not be covered in this course—but it is useful to know that the compiler has figured out the underlying math of `loop_while` and replaced it with a straight-line calculation. Wow!

There are other ways of looking at the code generated by the compiler. You may want to experiment with them and verify that they give the same information—perhaps in a slightly different form—as `gdb`.

- You can use the compiler directly, with the `-S` flag.
```
% gcc -g -S plab3.c
```
Insert whatever optimization flag you like. The resulting assembly language listing will appear in a file named `pleb.s`.
- Or, you can use the utility program `objdump` to "de-compile" or "disassemble" an object file.
```
% gcc -g -c plab3.c
% objdump -d plab3.o
```
Again, include an optimization flag on the `gcc` line if you wish. The assembly language listing will appear in the terminal.

A process is a complex abstraction. There are instructions, memory addresses, data values, and interpretations of data values—all expressed as sequences of bits. We will spend time in the semester filling in the details of what you see here. For now, it is important to distinguish among the various kinds of entities (instructions, addresses, values, . . .) that you are seeing.

4 Stepping through Compiled code

Turn now to the file `plab4.c`. It contains three functions. Read the functions and figure out what they do.

Recall that `argc` is the number of arguments on the command line, including the name of the program. The `malloc` line allocates a variable-sized array big enough to hold `argc` integers (which is slightly wasteful, since we only store `argc` − 1 integers there, but what the heck).

Compile the program without optimization but with `-g`, and bring up the debugger on it.

```
% gcc -g -o plab4 plab4.c
% gdb plab4
```

Put your answers to the questions under the heading Part 4 in `processlab.txt`.

1. Set a breakpoint in `fix_array`. Run the program with the arguments `1 1 2 3 5 8 13 21 44 65`. When it stops, print `a_size` and verify that it is 10. Did you really need to use a `print` command to find the value of `a_size`? *Hint:* look carefully at the output produced by `gdb`.

2. *Existing breakpoint at `fix_array`.*
   What is the value of `a`?

3. *Existing breakpoint at `fix_array`.*
   Type "`display a`" to tell `gdb` that it should display `a` every time you stop. Step six times. You will note that one of the lines executed is a right curly brace; this is common when you're in `gdb` and often indicates the end of a loop or the return from a function. After returning, what is the value of `a`?

4. *Existing breakpoint at `fix_array`.*
   Step again (a seventh time). What is the value of `a` now? What is `i`?

5. *Existing breakpoint at `fix_array`.*
   At this point you should (again) be at the call to `hmc_pomona_fix`. You already know what that function does, and stepping through it is a bit of a pain. The authors of debuggers are aware of that fact, and they always provide two ways to step line-by-line through a program. The one we've been using (`step`) is traditionally referred to as "step into"—if you are at the point of a function call, you move stepwise *into* the function being called. The alternative is "step over"—if you are at a normal line it operates just like `step`, but if you are at a function call it does the whole function just as if it were a single line. Try that now. In `gdb`, it is called `next` or just `n`. What line do we wind up at? (Incidentally, in `gdb` as in most debuggers, the line shown is the *next* line to be executed.)

6. *Existing breakpoint at `fix_array`.*
   Use `n` to step past that line, verifying that it works just like `s` when you are not at a function call. What is `a` now?

7. *Existing breakpoint at `fix_array`.*
   It is often useful to be able to follow pointers. `gdb` is unusually smart in this respect; you can type complicated expressions like `p *a.b->c[i].d->e`. Here, we have kind of lost track of `a`, and we just want to know what it is pointing at. Type "`p *a`". What do you get?

8. *Existing breakpoint at `fix_array`.*
   Often when debugging, you know that you do not care about what happens in the next three or twelve lines. You could type "`s`" or "`n`" that many times, but we're computer scientists, and CS types sneer at work that computers could do for them—especially mentally taxing tasks like counting to twelve. So on a guess, type "`next 12`". What line are you at?

9. *Existing breakpoint at `fix_array`.*
   What is the value of `a` now?

10. *Existing breakpoint at `fix_array`.*
    What is the value of `*a`?

Finally, a small side comment: if you've set up a lot of `display` commands and want to get rid of some of them, investigate `info display` and `undisplay`.

5 Assembly-Level Debugging

So far, we've been taking advantage of the fact that `gdb` understands your program at the source level: it knows about strings, source lines, call chains, and even complicated data structures. But sometimes we have to get down and dirty with the assembly code.

We will use `plab4` again. To be sure we are all on the same page, quit `gdb` and bring it up on `plab4` again. Run the program with arguments of `1 42 2 47 3`. As you have been doing, put your answers under the heading Part 5 in `processlab.txt`.

1. What is the output?
2. Set a breakpoint in `main`. Run the program again. Where does it stop?
3. *Existing breakpoint at `main`.*
   Type "list" to see what is nearby, then type "`b 41`" and "`c`". Where does it stop now?
4. *Existing breakpoints at `main` lines 33 and 35.*
   So since that is the start of the loop, typing "`c`" will take you to the next iteration, right?
5. *Existing breakpoints at `main` lines 33 and 35.*
   Oops. Good thing we can start over by just typing "`r`". Continue past that first breakpoint to the second one, which is what we care about. But why, if we are in the `for` statement, did it not stop the second time? Type "`info b`" (or "`info breakpoints`" for the terminally verbose). Lots of good stuff there. The important thing is in the "address" column. Take note of the address given for breakpoint 2, and then type "`disassem main`". You will note that there is a helpful little arrow right at breakpoint 2's address, since that is the instruction we are about to execute. Looking back at the corresponding source code, what part of the `for` statement does this assembly code correspond to?
6. *Existing breakpoints at `main` lines 33 and 35.*
   The code at `+44` jumps to `main+104`, which has three instructions that jump back to `main+46`. This is all part of the `for` loop pattern we covered in class. We have successfully breaked ("broken?" "Set a breakpoint?") at the initialization of the loop. But we'd like to have a breakpoint *inside* the for loop, so we could stop on every iteration. The jump to `main+46` tells us that we want to stop there. But that is not a source line; it is in the middle clause of the `for` statement. No worries, though, because `gdb` will let us set a breakpoint on *any* instruction even if it is in the middle of a statement. Just type "`b *(main+46)`" or "`b *0x40068f`" (assuming that is the address of `main+46`, as it was when we wrote these instructions). The asterisk tells `gdb` to interpret the rest of the command as an address in memory, as opposed to a line number in the source code. What does "`info b`" tell you about the line number you chose? (Fine, we could have just set a breakpoint at that line. But there are more complicated situations where there is not a simple line number, so it is still useful to know about the asterisk.)
7. *Existing breakpoints at `main` lines 33 and 35, and instruction `main+46`.*
   We can look at the current value of the array by typing "`p array[0]@argc`". But the current value is not interesting. Let us continue a few times and see what it looks like then. Typing "`c`" over and over is tedious (especially if you need to do it 10,000 times!) so continue to breakpoint 3 and then try "`c 4`". What are the full contents of `array`?

8. *Existing breakpoints at `main` lines 33 and 35, and instruction `main+46`.*
   Perhaps we wish we had done "`c 3`" instead of "`c 4`". We can rerun the program, but we really do not need all the breakpoints; we're only working with breakpoint 3. Type "`info b`" to find out what is going on right now. Then use "`d 1`" or "`delete 1`" to completely get rid of breakpoint 1. But maybe breakpoint 2 will be useful in the future, so type "`disable 2`". Use "`info b`" to verify that it is no longer enabled ("Enb"). Continue past breakpoint 1, where we're stopped. Where do we stop next? (We hope it is not a surprise!)

9. Sometimes, instead of stepping through a program line by line, we want to see what the individual instructions do. Of course, instructions manipulate registers. Quit `gdb` and restart it, setting a breakpoint in `fix_array`. Run the program with arguments of `1 42 2 47 3`. Type "`info registers`" to see all the processor registers in both hex and decimal. Which of the registers have *not* been covered in class?

10. *Existing breakpoint at `fix_array`.*
    Well, that is because they're weird and not terribly important. (Except `eflags`, which holds the condition codes among other things. Note that instead of being given in decimal, it is given symbolically—things like `CF`, `ZF`, etc.) What flags are set right now?

11. *Existing breakpoint at `fix_array`.*
    Often, looking at *all* the registers is excessive. Perhaps we only care about one. Type "`p $rax`". What is the value? Is "`p/x $rax`" more meaningful?

12. *Existing breakpoint at `fix_array`.*
    We mentioned a fondness for "`x/16i`". Actually, what we really like is "`x/16i $rip`". What do you see? Compare that to the result of "`disassem fix_array`".

13. *Existing breakpoint at `fix_array`.*
    Finally, we mentioned stepping by instructions. That is done with "`stepi`" ("step one instruction"). Type that now, and note that `gdb` gives a new instruction address but still says that you're in the for loop. Hit return to `stepi` again, and keep hitting return until the displayed line `does not` contain a hexadecimal instruction address. Where are you?

14. *Existing breakpoint at `fix_array`.*
    It is useful to use "`x/16i $rip`" here to make sure we understand what is about to happen. You should see three `mov` instructions followed by a `call`. Use `stepi 3` to get past the `movs`. What instruction address will be executed next?

15. *Existing breakpoint at `fix_array`.*
    As with source-level debugging, at the assembly level it is often useful to skip over function calls. At this point you have a choice of typing "`stepi`" or "`nexti`". If you type "`stepi`", what do you expect the next instruction to be (hexadecimal address)? What about "`nexti`"? (By now, your debugging skills should be strong enough that you can try one, restart the program, and try the other, so there is little excuse for getting this one wrong!)

16. *Existing breakpoint at `fix_array`.*
    Almost there! Stepping one instruction at a time can be tedious. You can always use "`stepi` *n*" to zip past a bunch, but when you're dealing with loops and conditionals it can be hard to decide whether it

is going to be 1,042 or 47,093 instructions before you reach the next interesting point in your program. Sure, you could set a breakpoint at the next suspect line. But sometimes the definition of "interesting" in *inside* a line. Let us say, just for the sake of argument, that you are interested in how the `leavq` instruction works. You can set a breakpoint there by typing "`b *0x40065f`" (assuming that 0x40066f is its address, as it was when we wrote these instructions). Do so, and then continue. What source line is listed?

17. *Existing breakpoints at `fix_array` and `*0x40066f`.*
    The `leaveq` instruction manipulates registers in some fashion. Start by looking at what `%rsp` points to. You can find out the address with "`p $rsp`" and then use the `x` command, or you could just try "`p/x $rsp`". What are the values of `rsp` and `rbp`?

18. *Existing breakpoints at `fix_array` and `*0x40066f`.*
    Use "`info reg`" to find out what all the registers. Then stepi until you execute the `leave` instruction, and look at all the registers again. Have the values in the `rsp` or `rbp` registers changed, and what are their old and new values?