CSCI 8530: Computer Science
Advanced Operating Systems
Spring 2020
Program Assignment 1

Due on February 27, 2020

Introduction

There are actually two small functions to be written in this assignment. Neither of them requires a large amount of code, but rather requires that you understand the relevant components and structure of the system. It is recommended that you do the two parts in the order given here, as the first part is likely simpler to understand and implement than the second.

Part 1

Write a function

```
void printsegaddress()
```

that prints the address of the start and end of the **text**, **data**, and **bss** segments of the XINU OS. Print the 4 bytes (in hexadecimal) at the start of each segment. Don't print these as if they form a 4-byte integer, but print them as four separate bytes, in the order they appear in memory (remember "endian-ness"). Calculate and print the lengths of the **text**, **data**, and **bss** segments. Save the code in a file **printsegaddress.c** under **system/**. Test that the function works correctly by calling it from the XINU process running **main().** Note how XINU makes use of linker generated external variables for the segment boundaries (**text**, **data**, **bss**) by inspecting code in **system/** and **include/** (using the **grep** command can be helpful).

To amplify your understanding of the correct results from this function, try executing the **memstat** program. Then try to resolve the results of your **printsegaddress** function with the first result (Xinu code size) from the **memstat** program. You should look at the code for **memstat**; it's in **shell/xsh_ memstat.c**. Also examine the file **xinu.map** that is produced by the **ld** program when the Xinu system is linked; it's in the **compile** directory. In the section of that file with the heading "Linker script and memory map" you can see the address associated with each external symbol; the symbol names appear in the rightmost column, and the hexadecimal numbers to their left are their definitions. The leftmost column identifies the part of an object code file being linked. The name **.text** identifies executable code, **.rodata** identifies read-only data (constants), **.eh_frame** identifies exception-handling information, **.data** identifies static initialized read-write data, **.bss** identifies static uninitialized data, and **COMMON** is used to identify uninitialized

data objects (usually aggregates, like arrays and structures) that have not already been allocated space in memory. There may also be a few other parts of object files that are unimportant to us (for the most part). ***fill*** is used to mark unused regions of memory that are skipped to force expected or required alignment of objects in memory.

Part 2

Write a function

```
int stackdepth()
```

that determines how many nested function calls (each with its own stack frame) have been made by the current process, including the call to **stackdepth()**, and returns that value to the caller. This is not an accumulated count of all function calls ever made by the process, but only the number of functions that have been called but not yet returned, starting from the function with which the process began execution, including a count of one for the initial function's stack frame. Note that **stackdepth()** and any function calls made within **stackdepth()** produce additional stack frames in the current process's run-time stack. To access the last stack frame pushed onto the run-time stack, use the in-line assembly function **asm()** to read the values of **%esp** and **%ebp** and save them into local variables

```
unsigned long *top_esp, *top_ebp;
```

of **stackdepth()**. Making use of the relationship between the stack frame of a calling function and the stack frame of the function it calls, iteratively follow the frame pointers starting at **top_ebp** and inspect the stack frames that were pushed in the run-time stack. The bottom of the run-time stack is recorded in the process table

```
procent proctab[NPROC];
```

in the field/member **prstkbase** of the appropriate entry. You can find the definition of **procent** in **include/**. The process ID of the current process is available in the global variable **currpid**. While iterating (or backtracking) through the frames of the run-time stack, count the number of stack frames traversed (the very top given by **top_ebp**, **top_esp** counts as 1), print the count value, **ebp** and **esp** values of each stack frame as it is traversed. Also print the difference of **ebp** and **esp** which gives the size of each stack frame. Addresses/pointers should be printed in hexadecimal format (always with eight hexadecimal digits). After the bottom of the run-time stack is reached, **stackdepth**() should print the address of the bottom and return the count of the frame stacks it has traversed (i.e., the depth).

Note that the very act of inspecting the run-time stack with the help of **stackdepth()** (and its in-line function **asm()**) disturbs the run-time stack by pushing additional stack frames. The system call **stacktrace()** in **system/** performs a more comprehensive backtrace

of a process's run-time stack which you can examine. However, write your own code that focuses on efficiently accomplishing the task at hand from scratch, not by pruning the code of **stacktrace()**.

Put the code of **stackdepth()** in **stackdepth.c** under **system/**. Make nested function calls from XINU's process running **main()** and test that your **stackdepth()** function works correctly.

To accommodate this assignment, the **Makefile** in **compile/** includes the option **-fno -omit-frame-pointer** in the definition of **CFLAGS** to force the gcc compiler to use the frame pointer. By default, gcc stopped using **ebp** to keep track of stack frame boundaries some time ago.

Demonstrating Execution

To demonstrate the execution of your solutions to part 1 and part 2, it is appropriate to add a few small functions, perhaps all in a single file, to the **system/** directory, and to invoke them and **printsegaddress** function from the main function in the **system/** directory. The **main** function is found in the file **main.c**. Immediately after the statement "**nsaddr = 0x...;**" add two lines similar to these:

```
printsegaddress();
func1();
```

In another file named **func.c** include code similar to this:

```
/* func.c - func1, func2, func3 */
#include <xinu.h>
int stackdepth(void);

void func3(void) {
    kprintf("In func3, stack depth = %d\n", stackdepth());
}

void func2(void) {
    kprintf("In func2, stack depth = %d\n",
    stackdepth()); func3();
}

void func1(void) {
    kprintf("In func1, stack depth = %d\n",
    stackdepth()); func2();
}
```

The purpose of these functions is to invoke the **stackdepth** function several times, from different stack depths. It should be clear that if **func1** displays a stack depth of N that **func2** should display N+1 and **func3** should display N+2. You may also wish to include the declaration of some local variables in one or more of those functions (and perhaps some code to initialize and manipulate them so the compiler doesn't warn about unused variables). The purpose of doing this is to explicitly guarantee that the sizes of the stack frames for the various functions will be different.
If you included declaration of the **stackdepth** function in the **prototypes.h** file in the **include/** subdirectory, then you would not need the declaration of **stackdepth** given in the **func.c** file. This declaration is provided just to silence the compiler.

You do not need to provide your **func.c** file or your modified **main.c** file as part of your submission, since suitable tests of your functions will be provided during evaluation.

Submission

To submit your work, create a directory on Loki named **/home/myusername/csci8530-prog1-s20**, replacing **myusername** with your username. For example, if your username was **phuang**, then the directory you create should be named **/home/phuang/csci8530-prog1-s20**. Put the source code for **printsegaddress.c** and **stackdepth.c** in that directory, **and nothing else**.
Include comments in your code that explain what's happening. That is, an individual reading your code should be able to understand its purpose and the procedure used to achieve that purpose. Turn off, disable, or remove any debugging code you may have used before submitting your work.
Make certain you include – in each file – a comment near the top of the file that includes the name or names of each author of the work and also **snapshot the results** from the console after porting the codes to the board. If there are multiple authors, submit the joint work, only once, through **a single submission directory**. For example, if there are two members in your group, only ONE of these individuals should have a submission directory.