## CS 354 Spring 2018

## Lab 2: Dynamic Behavior of XINU Processes (160 pts)

## Due: 02/13/18 (Tue.), 11:59 PM

## 1. Objectives

We will inspect basic features of XINU processes and their dynamic behavior.

## 2. Readings

Read Chapters 3-4 of the XINU textbook.

## 3. Adding assembly functions to XINU [45 pts]

Write code to convert an argument x from host byte order (little endian) to network byte order (big endian). Implement three versions. In the first, long host2net(long x), use a special assembly x86 instruction to write very compact assembly code for the function. Follow the AT&T assembly language syntax and put the function in file host2net.S under system/. [15 pts]

In the second version, write long host2netca(long x) in C and let gcc do the work of producing assembly code. Save the C code in host2netca.c and the assembly code generated by gcc in host2netca.s under system/. [15 pts]

In the third version, use in-line assembly, i.e., function asm(), to achieve the same goal. Save the function long host2neta(long x) in host2neta.c under system/. [15 pts]

For all three versions, test that they work correctly by calling the functions from the XINU process running main(). During evaluation of your code, the TAs will replace your main() with their own main() function for testing.

## 4. Inspecting XINU processes and their run-time stack [55 pts]

### Problem 4.1 [15 pts]

Write a function, void prnsegadd(), that prints the address in hexadecimal of the following segments: start and end of text, data, bss segments and the end of XINU. Refer to memory.h to find references to these boundaries and consult intialize.c to see examples of how they are used. Also print lengths of the text, data, and bss segments. Save the code in a file prnsegadd.c under system/. Replace the four kprintf() calls in nulluser() contained in initialize.c that output related data with a call to your prnsegadd() function. Test the function by rebooting XINU.

### Problem 4.2 [40 pts]

Using the create() system call, create a XINU process that runs a simple app program that you wrote in C. Note that in XINU apps are not written as programs with separate main() but as functions that are compiled/linked with XINU's main(). Call this app, i.e., function, appl1() which returns a value of type int. The number of arguments and their types is up to you. Your function appl1() should call a function fun1(), which has two arguments of your choice and returns a value of type char. When calling create(), set the stack size limit to 2048 bytes and process priority to INITPRIO. Find the system header file where INITPRIO is defined and change its value to 30.

Programs in XINU are represented as functions (as opposed to files containing main() as is common in UNIX/Linux and Windows) that are spawned as independent processes by calling create() with a function pointer. Hence, even though both appl1() and fun1() are coded as C functions, appl1() becomes a process because it is constructed through the create() system call, whereas fun1() remains a regular function (not a separate process) when it is called by the process running the code of function appl1(). Make sure that this distinction is clear. As a consequence appl1() is allocated a private run-time stack as discussed in class but fun1(), when invoked by appl1(), is managed as a regular function call by pushing a stack frame onto the run-time stack of the process running appl1(). From a programming language perspective, appl1() and fun1() are both just C functions. However, at run-time when they are executed, one runs as a new process whereas the other is just a plain old function call from within an existing process. As such, they are night-and-day when viewed from an operating system perspective.

Since XINU's create() system call, after creating a process, immediately suspends it, the resume() system call must be invoked to induce XINU's process scheduler to allocate CPU cycles to the process at some point in the future. XINU and most operating systems including UNIX/Linux and Windows always pick a highest priority ready process to run next. If there are two or more ready processes with the same highest priority, round-robin is used to allocate CPU cycles among these processes. Hence, depending on a process's priority relative to other processes, there is no guarantee that a process will ever receive CPU cycles. XINU implements static priority scheduling, meaning that process priorities never change after they are initialized through create(). Lower priority processes will starve unless higher priority processes terminate or do not become ready. For example, a process that is suspended (but not terminated) is not ready. More commonly, a process that blocks becomes not ready. To block means to relinquish the CPU voluntarily by making a blocking system call such as sleepms() which puts the calling process to sleep for a specified time period (in msec). As discussed in class, a process may also block because it is waiting on disk I/O to complete, or it needs to wait for a packet from a web server to arrive.

Run XINU where main() creates a process running appl1(). The process running appl1() calls function fun1(). (a) Print the address and 4-byte content (in hexadecimal) of the base (i.e., beginning) of the run-time stack of the main() process before the appl1() process is created. Consult the process.h which defines the fields of the XINU process table to find a variable that stores the base of a process's stack. Do the same for the top (i.e., end) of the stack which is available in the ESP register. Use in-line assembly to print the address of the top of the stack and its 4-byte content. (b) Print the address and content of the base and top of the run-time stack after appl1() is created and resumes (i.e., runs) but before it calls fun1(). (c) Print the address and content of the base and top of the run-time stack after the process running appl1() process calls fun1() but before fun1() returns. (d) Print the values just after fun1() returns. Discuss your results in Lab2Answers.pdf and place it in system/.

## 5. Stack smashing [60 pts]

The layout of XINU's image where the stacks belonging to different processes are adjacent makes processes vulnerable to overflow -- i.e., overwriting of a neighboring stack -- by other processes. This is a form of stack overflow (also called stack smashing) that has plagued security and reliability of computing systems for several decades. Your aim, in the context of XINU, is to create an attacker process that through nested function calls overwrites the stack area of another process causing it to malfunction.

The basic set-up is as follows. The attacker process runs the function void stackoverflowA(void) which performs sufficiently many nested function calls to overwrite the stack area of the process whose stack is just below its own in the RAM address space. The victim process runs the function void stackoverflowB(void) which prints the character 'B' stored in a local variable using putc() and calls sleepms() to sleep for 1.5 seconds. After returning from sleepms(), it prints 'B' again. The aim of the attacker process running stackoverflowA() is to prevent the second 'B' from being printed by corrupting the victim process's stack. This can be done in multiple ways. For example, stackoverflowA() could overwrite the content of stackoverflowB()'s local variable where 'B' is stored so that a different value is printed. A more disruptive attack might overwrite the return address of the function call to sleepms() so that when sleepms() tries to return it jumps to an address that is invalid which causes it to crash. Describe your attack strategy and what impact you expect it to have in Lab2Answers.pdf. Your code should confirm that it works.

During testing, the process running main() should first spawn the attacker process using create() with stack size 2048 bytes and process priority 10 (INITPRIO remains 30), immediately followed by resume(). This will allocate the stack area for the attacker process and make it ready to run. However, the main() process will continue executing since its priority is higher. The main() process then spawns the victim process using create() with stack size 2 KB and priority 15, followed by resume() to ready the process for execution. Thereafter the main() process sleeps for 3 seconds. When main() sleeps, the victim process will execute before the attacker process since it has higher priority. Only when it sleeps after the first printing of 'B' does the attacker process get the CPU and tries to wreak havoc.

Put the attacker and victim codes in stackoverflowA.c and stackoverflowB.c, respectively, under system/. Compile XINU with the compilation option -fno-omit-frame-pointer which forces gcc to use the base pointer EBP when managing caller/callee function calls.

## Bonus problem [15 pts]

In XINU, create() must be followed (at some point in the future) with resume() to enable a child process to receive CPU after it has been created. In Linux/UNIX and Windows, a child process is put in a ready-to-run state after creation (e.g., fork() in Linux) without needing to make a follow-up system call such as resume(). We have touched upon this in lab1. Inspect the code of create() and resume() in system/ and determine if create() can be modified so that newly created processes become ready to run from the get go. Implement a new system call, gocreate(), that implements this new semantic. Test that it works correctly and put gocreate() in gocreate.c under system/.

## Turn-in instructions

Before submitting your work, make sure to double-check the [TA Notes](http://www.cs.purdue.edu/homes/ogg/cs354_spring2018.html) to ensure that additional requirements and instructions have been followed.

*Electronic turn-in instructions:*

        i) Go to the xinu-spring2018/compile directory and do "make clean".

        ii) Go to the directory of which your xinu-spring2018 directory is a subdirectory. (NOTE: please do not rename xinu-spring2018, or any of its subdirectories.)

                e.g., if /homes/joe/xinu-spring2018 is your directory structure, go to /homes/joe

        iii) Type the following command

                turnin -c cs354 -p lab2 xinu-spring2018

You can check/list the submitted files using   
  
turnin -c cs354 -p lab2 -v

***Important: Please provide comments inside your code so that its function and flow can be conveyed to the reader. Turn off all debugging output before you submit your code.***