CS 503 Fall 2013

Lab 1: Getting Acquainted with XINU (230 pts)

Due: 09/09/13 (Mon), 11:59 PM

1. Objectives

The objectives of this introductory lab are to familiarize you with the process of compiling and running XINU, the tools involved, and the run-time environment and layout of XINU processes.

---

2. Readings

1. XINU Setup handout
2. Chapters 1-4 from the XINU textbook.
3. Intel x86 manuals available on the course web documentation page. Manuals are meant to be references that are selectively utilized on a need-to-know basis. Chapters 1 and 2 of volume 1, chapter 2 of volume 3 provide an overview of the x86 architecture which is useful to know as part of background.
4. AT&T assembly information linked through the course web documentation page.

Note: The Intel document is hundreds of pages long. Please do not print it. Relevant information should be selectively examined on-line.

---

3. Inspecting XINU's Run-time Environment [100 pts]

Write XINU functions and code that perform the following:

1. ***long host2netl_asm (long param)***
   Convert the parameter *param* from host byte order to network byte order (always Big Endian). The code for this function should be entirely written in x86 assembly. You should ***not use*** in-line assembly (i.e., do not use asm("...")) or use compiler/assembler generated output. You can assume that the size of long is 4 bytes and the byte order of the host machine is Little Endian. *Note: Coding done in CS 503 tends to be short, but, oftentimes, subtle and run-time sensitive.*
   In general, to investigate assembly code generated by the compiler, you can use the tool ***objdump -d <___.o>*** to disassemble an object file. The object files reside in the compile/ directory within the main Xinu directory (xinu-fall2013-x86). You can also see some of the *.S files in the **system/** directory for reference.

2. ***void printsegaddress()***
   Print the address of the end of the text, data, and bss segments of the Xinu OS. Also print the 8 bytes

(by default, hexadecimal) preceding the end of the three segment boundaries, and similarly for the 4 bytes following the segment boundaries. This function can be written in C.

3. ***Examining the run-time stack***

Using the create() system call, create a XINU process that runs a simple program that you have written in C. Call this program "myprogA()" which returns a value of type char. The number of arguments and their type is up to you to decide. Your program myprogA() should contain one function, call it myfuncA(), which has two arguments of your choice and returns an integer. When calling create(), set the stack size limit to 1024 and process priority to 20.

Note that programs in XINU are represented as functions (not files containing main() as in UNIX/Linux, Windows) that are spawned as independent processes by calling create() with the function pointer (i.e., name of program) from within XINU's main(). Hence although both myprogA() and myfuncA() are coded as C functions, myprogA() becomes a process because it is activated through the create() system call, whereas myfuncA() remains a regular function (not a separate process). This implies that myprogA() is allocated a private run-time stack as discussed in class, but myfuncA(), when invoked by myprogA(), is managed as a regular function call by pushing a stack frame in the run-time stack of process myprogA(). Thus although from a programming language perspective myprogA() and myfuncA() appear similar, they are night-and-day when viewed from an operating systems perspective.

Since XINU's create() system call, after creating a process, immediately suspends it (the process exists but it is "hybernating" in a frozen state), the resume() system call must be invoked to get XINU's process scheduler to allocate CPU cycles to the process at some point in the future.

Print the address and content of the top of the run-time stack before myprogA() is created. Print the address and content of the top of the run-time stack after myprogA() is created but before myfuncA() is called by myprogA(). Compare their values. Print the address of the top of the run-time stack just after myfuncA() is called but before it returns. Also print the contents of up to ten stack locations at and "below" the top of the stack (i.e., higher addresses since the stack grows from high-to-low). **The code to print stack addresses and contents should not be written as functions but inserted directly into main(), myprogA(), and myfuncA().** Remember that stack elements are 32 bits wide in our XINU implementation on the x86 XINU backends, and be careful to perform pointer arithmetic accurately. Note that there are local variables, stored register values, arguments, etc. as discussed in class.

4. ***void printprocstks()***

Spawn two processes, one running myprogA(), and the other running myfuncA(). **Note that the same myfuncA() as in 3.3 is now used as the code of a new process.** For each process, print the *stack base*, *stack size*, *stack limit*, *stack pointer* and the *process id*. An example output might look something like:

```
Proc [prnull]. Pid = 0.
   Stack: Base  = 4095996
          Len   = 0
          Limit = 4091904
          StkPtr  = 4095820

Proc [main]. Pid = 49.
   Stack: Base  = 4091896
```

```
Len    = 4096
Limit = 4087804
StkPtr  = 4091872
```

To aid with this task, please look into **process.h** in the **include/** directory. The *proctab[]* array holds key information about processes and the *prstkptr* member of the *procent* structure holds the *saved* stack pointer. In a multiprogramming OS where a CPU is shared among multiple processes, the CPU is dynamically allocated to processes before they are completed (in contrast to batch mode). Thus when a CPU is reassigned from a process A to another process B then process A's run-time state must be saved so that it can be restored when A is allocated the CPU again in the future. We call this bookkeeping operation context switching between processes. Hence although the stack pointer need not be saved when performing a function call within a process that pushes a stack frame in the process's run-time stack, when context switching between processes occurs the stack pointer value must be saved. The saved stack pointer field in the process table entry of a process serves this purpose in XINU.

A consequence of this is that the *currently* executing process (note that in a uniprocessor system only one process runs at any time with all other processes context switched out and in a "state of limbo"; XINU, even on multi-core systems, utilizes only one core) has a stack pointer whose value is different from the saved stack pointer value (which contains the stack pointer value when it was last context switched out). In order to help you access the stack pointer of the currently executing process, study **stacktrace.c** in the **system/** directory. The register *%esp* (AT&T assembly syntax) denotes the 32-bit stack pointer. You may use in-line assembly (i.e., asm("...")) to carry out this part.

With (at least) two concurrent processes running in XINU, it is useful to know how XINU's default process scheduler, **resched()**, works. When invoked, resched() always picks the highest priority "ready" process to run. "Ready" means that a process is not blocking (i.e., waiting on an event, such as message arrival on a network interface) but can immediately make use of the CPU if allocated to execute. When, as in lab1, all processes are created with equal priority 20, a round-robin policy is implemented to pick the next process to run.

What prevents a process from hogging the CPU forever is the notion of a time budget, called time slice or quantum, that the operating system keeps track of. This time budget is configured in the kernel constant QUANTUM which is defined in the header file **include/kernel.h**. When a process is created, it is allocated this initial time budget. The kernel keeps track of how much CPU time a process has consumed by performing bookkeeping in the clock interrupt handler **system/clkint.S** written in assembly. When a process that's running expends all of its time slice -- this event is detected by clkint.S -- it calls XINU's process scheduler, resched(). Since all processes in lab1 are assigned equal priority, resched() picks the next process in line and runs this process until it also exhausts its budget. The process that exhausted its time slice and was context switched out is made to wait at the end of the line (hence round robin). When its turn comes up again because all other processes have spent their time budgets and have been put behind in the line, the process's time slice is replenished back to QUANTUM and the dynamics repeats again.

When a process relinquishes the CPU because its time slice has depleted, we say that relinquishing the CPU was mandatory. When a process gives up the CPU because it has to wait on an event (such as message arrival) before it can proceed (a process may also just decide to sleep for a while by making a sleep() system call), then we say that a process relinquished the CPU voluntarily. The above is but a conceptual description of how the default XINU process scheduler works. In CS 503, reading the kernel source code resched() and clkint.S is required to understand the workings of the operating system. In lab2, you will be asked to modify XINU's scheduler to enhance its capabilities similar to those of UNIX/Linux and Windows.

As usable software output, the three functions described in Problems 3.1, 3.2 and 3.4 should be implemented as library code that are included (in source code format) as separate files in the **system/** directory. Update Makefile accordingly to reflect this change. Name the files after the functions they implement with C files having the **.c** extension and assembly files having the **.S** extension. For example, the file that holds ***void printsegaddress()*** should be named ***printsegaddress.c***, and the file that holds ***long host2netl_asm(long param)*** should be named ***host2netl_asm.S***. If you require a header file, please name it **lab1.h**. The code of Problem 3.3 is embedded in the main(), myprogA() and myfuncA() test code.
When evaluating your lab assignment, the TAs will (1) examine the results from your own test cases including the source code, and (2) add new test cases to further gauge correctness and performance of your code. Please follow any additional instructional posted in the TA notes.

4. Process Memory Layout [30 pts]

Write your answers to the following questions in a file named **Lab1Answers.pdf**. Please place this file in the **system/** directory and turn it in, along with the above programming assignment.

1. Assuming the XINU text begins at address 0x0, draw a diagram of XINU's memory layout with addresses derived from your experimental measurements. Include the information you uncovered from running your version of ***printsegaddress()*** and ***printprocstks().***
2. Draw a layout of the stack for a user process based on your measurements. Show the relative locations of local variables and arguments. If you found other information about the stack, include them in the diagram.

5. Impact of Intentional Stack Overflow [70 pts]

Write a program, someprogA(), that prints the character 'A' using putc(), then calls a function somefuncA() with a single argument of type char. Suppose the value passed is 'a'. When invoked, somefuncA() first makes a system call to sleepms() which requests that the kernel put the calling process to sleep for the time duration specified in the argument (in unit of millisec). Make the value sufficiently large compared to the scheduler's time slice indicated in Problem 3, say, at least 10 times larger. (Look up the kernel's time slice constant in the system header files.) When sleepms() is called, the kernel context switches out the calling process (putting it in a frozen state until the specified sleep period has elapsed) and switches in a process that is ready to run following the round robin policy described in Problem 3. After returning from sleepms(), somefuncA() prints the value passed in its function argument and then returns with integer value 1. someprogA(), when the call to somefuncA() returns, prints 'A' and exits.
Write a program someprogB() that is identical to someprogA() except that it prints the characters 'B' and 'b' in place of 'A' and 'a'. create()/resume() someprogB() before someprogA() in main() and observe the

sequence of character output which should be 'B', 'A', 'b', 'B', 'a', 'A'. Use the same priority 20 for all processes as in Problem 3 but you may choose a different value for the stack size limit.

Now modify someprogB() such that it triggers a stack overflow by performing a large number of nested function calls. Name the modified someprogB(), rogueB(). Before it engages in nested function calls, make rogueB() print 'B' followed by a sleepms() system call with the same sleep period as before. This gives someprogA() a chance to run and call somefuncA() before it also goes to sleep. At the moment when the process executing someprogA() calls sleepms() and is context switched out by the kernel, its run-time stack will contain the stack frame for somefuncA() but the character 'a' will not have been printed yet. The code of the nested function calls in rogueB() should have arguments and local variables to fill up rogueB's run-time stack. You may also use a smaller stack size limit (not "too small" though) when calling create() to help trigger stack overflow sooner than later. The goal is for the nested function calls of rogueB() to cause stack overflow, i.e., overwrite onto the run-time stack belonging to someprogA(), which exploits how XINU arranges process stacks in memory: process stacks are located adjacent to each other. After rogueB() has caused stack overflow, make it sleep again so that the kernel context switches in someprogA() and resumes where things were left off.

For this exercise, you will have succeeded if rogueB(), by overflowing into the run-time stack of the process executing someprogA(), prevents someprogA() from printing the remaining characters 'a' and 'A'. What happens to the system after rogueB() succeeds in preventing someprogA() from printing the remaining characters? Discuss your observations in Lab1Answers.pdf.

---

6. Hijacking a Process through Stack Overflow [30 pts]

How can rogueB() be modified to achieve the same stack overwrite/corruption without engaging in nested function calls? Can rogueB(), by carefully choosing what it writes onto someprogA()'s run-time stack, "hijack" someprogA() when it is scheduled to run again so that the process running someprogA() ends up running other code, say, roguefuncB()? That is, function roguefuncB() is not called by either process -- victim process someprogA() or attacker process rogueB() -- but when someprogA() resumes running again it branches to roguefuncB() which prints 'Q' and returns. Discuss your solution in Lab1Answers.pdf. If you implement your solution and demonstrate that it works correctly, you will receive 70 bonus points. To avoid ambiguity, please call this version of rogueB(), rogueB2().

Turn-in Instructions

Electronic turn-in instructions:

i) Go to the xinu-fall2013-x86/compile directory and do "make clean".

ii) Go to the directory of which your xinu-fall2013-x86 directory is a subdirectory. (NOTE: please do not rename xinu-fall2013-x86, or any of its subdirectories.)

e.g., if /homes/jsr/xinu-fall2013-x86 is your directory structure, go to /homes/jsr

iii) Type the following command

```
turnin -c cs503 -p lab1 xinu-fall2013-x86
```

Important: You can write code in main.c to test your procedures, but please note that when we test your programs we will replace the main.c file! Therefore, do not put any functionality in the main.c file. ALL debugging output should be turned off before you submit your code.

Back to the CS 503 web page