CSC501 Fall 2018

PA 0: Getting Acquainted with XINU

Due: September 10 2018, 4:00 AM

ChangeLog

09/02: adding the necessary changes to the Makefile while using a local QEMU installation, making the task description consistent with the sample output file, and changing Task 2 description to "segments of the current process".

08/30: adding a link to instructions for local QEMU installation

08/29: for zfunction, I meant "counting from right" but not "counting from left"

08/28: first version online

1. Objective

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

2. Lab Setup Guide

XINU is a small Unix-like operating system originally developed by Douglas Comer for instructional purposes at Purdue University. It is small enough so that we can understand it entirely within one semester. As part of lab assignment, we will re-implement or improve some apsects of XINU.

Step 0: Installing QEMU

Following the instructions at the xv6 site to install QEMU.

I have tested this on Ubuntu 14.04, and it works. However, the QEMU from "apt-get install qemu" has problems. I have also tried Ubuntu 18.04, but it does not work well with the main problem that the XINU tarball fails to compile properly.

After you install QEMU, you need to change your PATH environment accordingly so that the later "make run" and "make debug" commands can find the binary.

If this is not enough for you to setup a local QEMU environment, I would suggest you just use the VCL environment. Post your questions to Moodle if you need help regarding local QEMU setup.

Step 1: Setting environment variables for lab assignments:

Get access to a customized VCL image -- *XINU+QEMU (CSC501)* -- through the VCL facility

Step 2: Untar the XINU source files as follows:

1. Change to a working directory you would like to use for this project, preferable using /afs/unity.ncsu.edu/users/m/myid/this/is/my/project/dir

   ```
   cd /afs/unity.ncsu.edu/users/m/myid/this/is/my/project/dir
   ```

2. Untar the XINU source by typing the following:

   ```
   wget -U firefox https://people.engr.ncsu.edu/gjin2/Classes/501/Fall2018/assignments/PA0/csc501-lab0.tgz
   tar xzvf csc501-lab0.tgz
   ```

   In your working directory, you will now have a directory named csc501-lab0. The subdirectories under this directory contain source code, header files, etc, for XINU.

   NOTE: the tar file name may be different from the above depending on the project you are working on. Please refer to the project handouts for the location of the tar file for the current project.

If you have a way to get rid of "-U firefox", let the instructor know.

LD = /usr/bin/gcc

Step 3: Building XINU

To compile the XINU kernel which will be downloaded and run on the backend machines, run "make" in the compile directory as follows:

```
cd csc501-lab0/compile
make depend
make
```

This creates an OS image called 'xinu.elf'.

If you use the local QEMU environment, you may need to change two lines in the Makefile:

```
LD      =       /usr/bin/ld
```

to

```
LD      =       /usr/bin/gcc
```

and

```
        $(LD) -m elf_i386 -dn -Ttext 0x10000 -e start ${XOBJ} ${OBJ} ${LIB}/libxc.a \
```

to

```
        $(LD) -m32 -dn -Ttext 0x10000 -e start ${XOBJ} ${OBJ} ${LIB}/libxc.a \
```

Step 4: Running and debugging XINU

The XINU image runs on the QEMU emulator machines. To boot up the image, type:

```
make run
```

XINU should start running and print a message "Hello World, Xinu lives."

Typing "Ctrl-a" then "c" will always bring you to "(qemu)" prompt. From there, you can quit by typing **q**.

To debug XINU, run the image in the debug mode by:

```
make debug
```

Then execute GDB in another ssh session:

```
gdb xinu.elf
```

In the (gdb) console, connect to the remote server by:

```
target remote localhost:1234
```

You can use many debugging features provided by GDB, e.g., adding a break point at the main function:

```
b main
```

To run to the next breakpoint, type:

```
c
```

The detailed document for GDB can be found here.

3. Readings

1. AT&T assembly information specific to the gnu assembler is available here as a wikibook.
2. Any man pages/manuals you discover that you need.

4. Tasks

You will be using the `csc501-lab0.tgz` you have downloaded and compiled by following the lab setup guide. And you are asked to write several XINU functions that perform the following tasks:

1. **`long zfunction(long param)`**

   Clear the 9th to 17th bits, counting from right and starting with 0, shift the parameter param by 8 bits to the right, and then fill the left most bits with 1. For example, the input parameter `0xaabbccdd` should generate a return value of `0xffaab800`. You can assume that the size of long is 4 bytes. The code for this function should be entirely written in x86 assembly. You should not use inline assembly, (i.e., do not use `asm(???)`). To investigate the 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. You can also see some of the *.S files in the `/sys` directory for reference.

2. **`void printsegaddress()`**

   Print the addresses and their contents indicating the end of the text, data, and BSS segments of the current process. You can refer to the manual page for "etext". Also print the 4-byte contents (in hexadecimal) preceding and after those addresses. This function can be written in C.

3. **`void printtos()`**

   Print the address and their contents of the top of the run-time stack for whichever process you are currently in, right before and right after you get into the `printos()` function call. In addition, print the address and their contents of upto four stack locations below the top of the stack (the four or fewer items that have been the most recently pushed, if any). Remember that stack elements are 32 bits wide, and be careful to perform pointer arithmetic correctly. Also note that there are local variables and arguments on the stack, among other things. See the hints given for #4 below, especially on stacktrace.c and proc.h. Your function can be written entirely in C, or you can use in-line assembly if you prefer. This wiki page can be helpful.

4. **`void printprocstks(int priority)`**

   For each existing process with larger priority than the parameter, print the stack base, stack size, stacklimit, and stack pointer. Also, for each process, include the process name, the process id and the process priority.

   To help you do this, please look into `proc.h` in the `h/` directory. Note the `proctab[]` array that holds all processes. Also, note that the pesp member of the pentry structure holds the saved stack pointer. Therefore, the currently executing process has a stack pointer that is different from the value of this variable. In order to help you get the stack pointer of the currently executing process, carefully study the `stacktrace.c` file in the `sys/` directory. The register `%esp` holds the current stack pointer. You can use in-line assembly(i.e., `asm("...")`) to do this part.

5. **`void printsyscallsummary()`**

   Print the summary of the system calls which have been invoked for each process. This task is loosely based on the functionality of LTTng. There are 43 system calls declared. Please look into `kernel.h` in the `h/` directory to see all declared system calls. However, only 27 system calls are implemented in this XINU version. The implementation of these 27 system calls are in the `sys/` directory. You are asked to print the frequency (how many times each system call type is invoked) and the average execution time (how long it takes to execute each system call type in average) of these 27 system calls for each process. In order to do this, you will need to modify the implementation of these 27 types of system calls to trace them whenever they are invoked. To measure the time, XINU provides a global variable named `ctr1000` to track the time (in milliseconds) passed by since the system starts. Please look into `sys/clkinit.c` and `sys/clkint.S` to see the details.
   You will also need to implement two other functions:
   `void syscallsummary_start()`: to start tracing the system calls. All the system calls are invoked after calling this function (and before calling `syscallsummary_stop()`) will be presented in the system call summary.
   `void syscallsummary_stop()`: to stop tracing the system calls.
   In other words, these two functions determine the duration in which the system calls are traced.
   To help you complete this task, we provide two files, syscalls.txt lists all the system calls you will need to trace, and test.c demonstrates the usage of the functions you will implement (note that this is only the test file and will not be used for grading).

Implement this lab as a set of functions that can be called from `main()`. Each function should reside in a separate file in the sys directory, and should be incorporated into the Makefile. The files should be named after the functions they are implementing with C files having the .c extension and the assembly files having the .S extension. For example, the file that will hold `void printsegaddress()` should be named `printsegaddress.c`; and the file that will hold `long zfunction(long param)` should be named `zfunction.S`. You should put `syscallsummary_start, syscallsummary_stop` functions in the same file as `printsyscallsummary` function and name it as `printsyscallsummary.c` . If you require a header file, please name it `lab0.h`. Note: as you create new files, you may need to update the `Makefile` (located in the `compile/`directory) to configure it to compile your files correctly. Just look at what is done for the existing files (e.g., `main.c`) to see what you have to do.

5. Additional Questions

Write your answers to the following questions in a file named Lab0Answers.txt(in simple text).

Please place this file in the sys/ directory and turn it in, along with the above programming assignment.

1. Assuming the XINU text begins at address `0x0`, draw a rough 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. What is the difference in stack top address before and after calling `printtos()`? Draw a diagram to illustrate what are the contents of the items pushed into the stack between these two time points.
3. In a stack frame, local variables are stored below the top of the stack. In task 3, does your result show all the local variables declared in your **printtos** function? If not, can you explain that? (hint: try to disable the compiler optimization by specifing -O0 in your Makefile)

Turn-in Instructions

Electronic turn-in instructions:

1. make sure your output follows the output template, as mush as possible
2. go to the `csc501-lab0/compile` directory and do `make clean`.
3. go to the directory of which your `csc501-lab0` directory is a subdirectory (NOTE: please do not rename `csc501-lab0`, or any of its subdirectories.)
4. create a subdirectory TMP (under the directory `csc501-lab0`) and copy all the files you have modified/written, both .c files and .h files into the directory.
5. compress the `csc501-lab0` directory into a tgz file and submit on Moodle. Please only upload one tgz file.

```
tar czf csc501-lab0.tgz csc501-lab0
```

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.

Also, ALL debugging output **MUST** be turned off before you submit your code.

Grading Policy

- (10%) Source code can be compiled and the generated image is bootable. Please note that you will also get 0 point for the second part if your source code can not be compiled or can not generate a bootable image.
- (75%) Each task rewards 15 points (losing points on minor problems)
- (15%) Each additional question earns 5 points

Back to the CSC501 web page