ECE592 – Operating Systems Design: Homework #1

Due date: September 5, 2017

Objectives

- To become familiar with the Xinu development and runtime environment.
- To become familiar with the general structure of Xinu codebase.
- To become familiar with the following parts of Xinu: initialization, process management, shell.

Overview

Xinu (http://xinu.cs.purdue.edu) is an operating system developed by Prof. Douglas Comer's group at Purdue University. It is a small operating system suitable for embedded environments that supports dynamic process creation, dynamic memory allocation, network communication, local and remote file systems, a shell, and device-independent I/O functions. Xinu's internals and operation are fully described in the following textbook:

D. Comer, Operating System Design - The Xinu Approach, Second Edition CRC Press, 2015. ISBN 9781498712439

There are different versions of Xinu. In this course, we will use the one described in the second edition of the textbook above. More specifically, we will use the version of Xinu that runs in a Virtual Machine environment called Virtual Box.

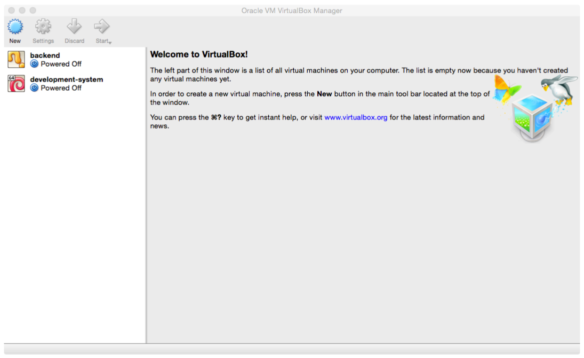
Step 1: Installing Xinu on VirtualBox

1. VirtualBox (https://www.virtualbox.org/wiki/VirtualBox) is a general-purpose virtualization application for x86 hardware that allows you to run multiple operating systems (each within a separate virtual machine) on your computer. Install the corresponding VirtualBox on your platform. You can download the software from https://www.virtualbox.org/wiki/Downloads

2. Download the 2015 VirtualBox version of Xinu from: ftp://ftp.cs.purdue.edu/pub/comer/private/Xinu/xinu-vbox-appliances.tar.gz

3. Once you have installed VirtualBox, import the virtual machines from the *xinu-vbox-appliances.tar.gz* tarball. There are two virtual machines. One (*development-system.ova*) acts as a development platform running Linux, and you can use this virtual machine to modify and compile Xinu. The other (*backend.ova*) acts as a bare machine running Xinu. The two machines will have a virtual serial connection between them that allows you to communicate with the Xinu machine while Xinu runs. In order to install Xinu in VirtualBox, follow the following instructions.

   - Open VirtualBox. In VirtualBox main window, select File > Import Appliance. Browse and select the *development-system.ova* file, then click Continue. *Do not* select the "Reinitialize the MAC address of all network cards" checkbox. Click Import to import the development system virtual machine. Use the same procedure to import *backend.ova*. You should now see the virtual machine images on the left-hand panel of the VM Manager window.



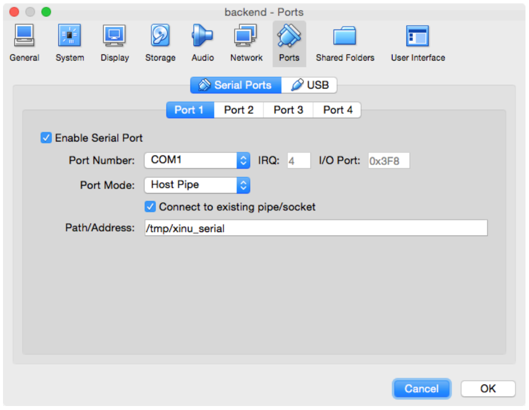
Welcome to VirtualBox!

The left part of this window is a list of all virtual machines on your computer. The list is empty now because you haven't created any virtual machines yet.

In order to create a new virtual machine, press the **New** button in the main tool bar located at the top of the window.

You can press the ⌘? key to get instant help, or visit www.virtualbox.org for the latest information and news.

- Highlight xinu-backend in the left-hand menu, and click on Settings. From this menu, navigate to Ports (or "Serial Ports" in some distributions). Make sure that Enable Serial Port and Connect to Existing Pipe/Socket are both checked. Under Path/Address:
    - If you're on a Linux or Mac: type in /tmp/xinu_serial
    - If you're on Windows: type in \\.\pipe\xinu_com1



☑ Enable Serial Port

Port Number: COM1 IRQ: 4 I/O Port: 0x3F8

Port Mode: Host Pipe

☑ Connect to existing pipe/socket

Path/Address: /tmp/xinu_serial

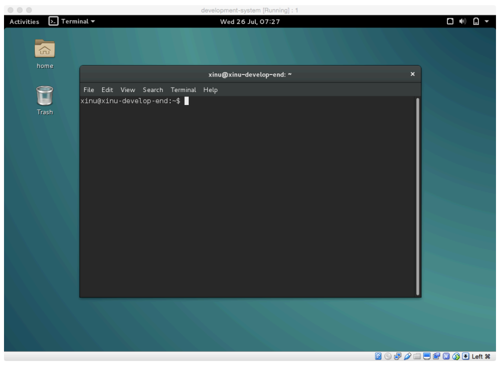
Click OK to exit settings.

For the development-system, make sure that Enable Serial Port is checked and the Path/Address is set as above, but leave Connect to Existing Pipe/Socket **unchecked**.

Now the two machines can communicate using a virtual serial connection and they are ready to use.

Step 2: Compiling and Running Xinu

1. Run the development-system virtual machine by double-clicking on it. The default user name is xinu and the default password is xinurocks. After logging in, your terminal should look like:



```
xinu@xinu-develop-end:~$ 
```

2. Navigate into the xinu directory, and you'll see the following subdirectories:

   - compile - contains the Makefile and scripts to upload the kernel to the back-end.
   - config - contains device configurations (do not touch files in this directory).
   - device - contains device files (do not touch files in this directory).
   - include - contains header files, which define constants and function prototypes.
   - lib - contains a small library of standard C functions. The UNIX system libraries are not available.
   - net - contains C functions for networking tasks.
   - shell - contains the implementation of the Xinu shell.
   - system - contains the source code for the Xinu kernel.

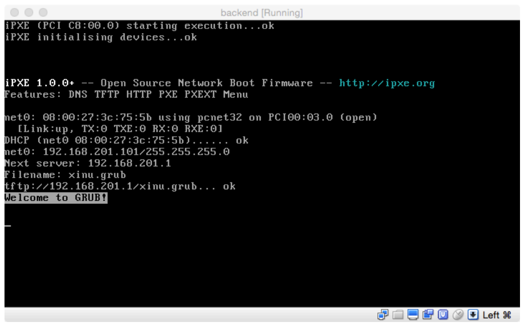
   Most of your time in development will be spent in the include/ and system/ directories.

3. Navigate into the compile directory. Type make clean to clean up. Type make to compile the kernel. If the compilation is successful (and it should be successful at this point), this will create a binary file called xinu.elf in this directory, and prepare it for upload onto the back-end VM.

4. Now type sudo minicom (or sudo minicom --color=on if you want to see colors in the terminal window). You'll be prompted for the password. This turns the terminal window into a serial console that is connected to the back-end VM, effectively emulating a terminal for the back-end VM. All output from Xinu will now appear in the minicom window, and input typed to the minicom window will be sent to Xinu.

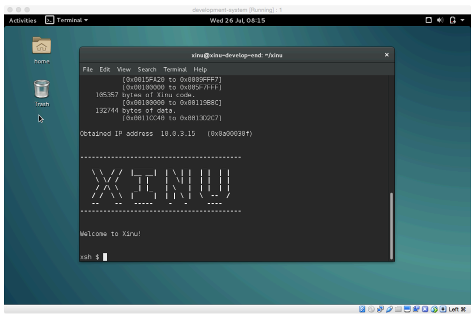
5. At this point, start the backend virtual machine from VirtualBox by double-clicking on it. It should take a few seconds for it to automatically retrieve the kernel binary from the development-end and boot it. Because minicom turned the development system VM into the screen that is "attached" to the backend machine, the minicom terminal will show the Xinu bootup information and Xinu shell. If everything went smoothly, you should get the following output.

Backend:



```
iPXE (PCI C8:00.0) starting execution...ok
iPXE initialising devices...ok



iPXE 1.0.0+ -- Open Source Network Boot Firmware -- http://ipxe.org
Features: DNS TFTP HTTP PXE PXEXT Menu

net0: 08:00:27:3c:75:5b using pcnet32 on PCI00:03.0 (open)
  [Link:up, TX:0 TXE:0 RX:0 RXE:0]
DHCP (net0 08:00:27:3c:75:5b)...... ok
net0: 192.168.201.101/255.255.255.0
Next server: 192.168.201.1
Filename: xinu.grub
tftp://192.168.201.1/xinu.grub... ok
Welcome to GRUB!

_
```

Development system:


development-system [Running] : 1
Activities
Terminal
Wed 26 Jul, 08:15
home
Trash
xinu@xinu-develop-end: ~/xinu
File Edit View Search Terminal Help
Left ⌘

```
           [0x0015FA20 to 0x0009FFF7]
           [0x00100000 to 0x005F7FFF]
    105357 bytes of Xinu code.
           [0x00100000 to 0x00119B8C]
    132744 bytes of data.
           [0x0011CC40 to 0x0013D2C7]

Obtained IP address  10.0.3.15   (0x0a00030f)


------------------------------------------

------------------------------------------


Welcome to Xinu!


xsh $ 
```

6. Now Xinu is still "running" over on the backend. Shutdown the backend VM to terminate Xinu. In the development-system VM, to quit from the minicom serial console: first press "`ctrl+a`" (hold together), then press "`q`". Now your terminal should be detached from the serial console and be back in the compile directory.

7. From here on, remember this workflow as you proceed with development:

   - Write your code on the development system VM
   - Navigate into the compile subdirectory
   - Run make clean and make to compile the kernel and prepare it for upload to the backend VM
   - Run sudo minicom
   - Start up the backend VM
   - Once you are finished, power down the backend VM. In the development-system VM, quit the minicom serial console.

Step 3: Getting Oriented in Xinu

Start having a look into Xinu code to get a high-level picture of its structure.

Types and Constants

All headers files are in the include directory. You may want to start from the following header files:

- include/xinu.h: unifies the inclusion of all necessary header files.
- include/prototypes.h: declares most system-call prototypes.
- include/kernel.h: contains definition of some important constants, types, and function prototypes.

For Debugging

See `kprintf()` system call in system/kprintf.c

Process-related code

- Process definition and PCB table: include/process.h
- Process creation/termination/suspension/resumption: system/create.c, system/kill.c, system/exit.c, system/suspend.c, system/resume.c
- Process scheduling: include/resched.h, system/resched.c
- Context switch and state change: system/ctxsw.S, system/ready.c, system/sleep.c, system/unsleep.c, system/wait.c, system/yield.c, system/wakeup.c
- Process queue management: include/queue.h, system/queue.c, system/newqueue.c, system/getitem.c, system/insert.c

Bootstrap procedure

The startup code (system/start.S) invokes `nulluser()` in system/initialize.c to initialize the system. Analyze the initialization code in initialize.c.

Step 4: Questions (include the answers to these questions in the report):

Q1. What is the maximum number of processes accepted by Xinu? Where is it defined?
Q2. What does Xinu define as an "illegal PID"?
Q3. What is the default stack size Xinu assigns each process? Where is it defined?
Q4. Draw Xinu's process state diagram.
Q5. When is the shell process created?
Q6. Draw Xinu's process tree (including the name and identifier of each process) when the initialization is complete.

Step 5: Coding problems

P1: Calling convention and run-time stack

Note: You can find information on the x86 architecture and assembly in the Intel Architecture Software Developer's Manual (available on the course website). In particular, refer to Volume 1, Chapters 4.1-4.3 for information on the handling of the stack. You can find also a brief introduction on the runtime stack in the Xinu's book, Chapter 3.9.1.

Download the file `main_stackframe.c` and put it in the `system` folder. Rename the original `main.c` in the system folder to `main.orig` (so that it won't be compiled). Invoke `make clean` twice, then invoke `make` to re-compile Xinu.

In `main_stackframe.c` there are four lines labeled with the `CASE#` keyword. These lines are located either right before or right after a function call. Insert your own code at each labeled line to print out the following run-time stack information:

- the address of the top of the runtime stack
- the content of the last 12 elements that have been pushed onto the stack. If the stack has fewer than 12 elements, print out the whole stack.

Remember that stack elements are 32-bit wide, and the stack grows in the direction of decreasing memory addresses (that is, from high to low memory addresses). The register `esp` holds the current stack pointer. You may use inline assembly (i.e., `asm(…)`) to get the value of the stack pointer (`esp`) of the running process (refer to `system/stacktrace.c` to study how to use inline assembly in C code). Compile and run the modified Xinu. Explain the content of the stack at the 4 labeled positions. Specifically:

- In all cases, draw a diagram illustrating the content of the stack and indicating what kind of information is stored in each stack element (e.g. return address? function parameter?)
- What is the difference in stack content before and after a function call?
- During a function call, which items are pushed onto the stack by the caller function and which items are pushed by the callee function?

P2: Extending the Xinu shell

The shell is a user interface provided by most operating systems to allow users to easily interact with the OS. Refer to Chapters 26.8 and 26.11 of Xinu's textbook for information on its shell. In this coding problem, you will extend the Xinu shell by implementing and adding the following command to it:

```
printprocstk <prio>
```

where `printprocstk` is the command name, and `<prio>` is the input parameter taken by it. This command operates as follows. For each existing process with priority larger than `<prio>`, `printprocstk` prints the following information: (i) process id, (ii) process name, (iii) process priority, (iv) process state, (v) stack base, (vi) stack size, (vii) stack limit, and (viii) stack pointer.

In order to accomplish this goal, you need to follow two steps.

First, implement a function "`void printprocstk(int prio)`" that prints the required information. The Xinu shell is implemented in the files within the `shell` folder: to maintain coherence with the current implementation, you should add your C file to this folder. Note that the `proctab` array in `include/process.h` holds all process-related information. Also, note that the `prstkptr` member of the `procent` structure holds the *saved stack pointer*. Therefore, the currently executing process has a stack pointer that is *different* from the value of its `prstkptr` attribute. You can again use in-line assembly to get the value of the stack pointer of the running process.

Second, make the `printprocstk` function available as a shell command. Carefully study the Xinu shell implementation to understand how a function is registered as a shell command and how function parameters are passed from shell command line to the underlying function. *Hint*: pay special attention to the `cmdtab` data structure.

P3: Timing

In this coding problem, you need to extend the functionality of an existing shell command: `ps`. For each existing process, the modified `ps` command should display the following additional information: *amount of time elapsed since the process was created*. To implement such extension, you need to record the system time at process creation and at invocation of the `ps` command. Xinu provides two global variables to track the system time: `clktime` (in seconds) and `ctr1000` (in millisecond). Note that `ctr1000` is currently never updated. Refer to `include/clock.h` and `system/clkhandler.c` for the use of these global variables, and modify these files to make sure that `ctr1000` is properly updated. Refer to `system/create.c` for process creation. *Hint*: The NULL process is an exception. Revisit the bootstrap procedure to understand how to record the creation time of the NULL process.

P4: Tracking of system calls

Analyze the `include/prototypes.h` header and the system calls defined in the `system` folder, and identify the system calls that cause changes in the state of a process. In this coding problem, you need to implement and invoke a function (called `void pr_status_syscall_summary()`) that prints out a summary of all the invocations to system calls causing process state changes that occurred since the start of Xinu execution.

Specifically, *for each* ***valid*** *process and system call*, `pr_status_syscall_summary` must print the following information: *frequency* (how many times each system call is invoked on that process) and *average execution time* (how long it takes to execute each system call in average). In order to do this, you will need to modify the implementation of these system calls to trace their invocations. To measure the time, you can again use the `clktime` and `ctr1000` variables.

Implement the `pr_status_syscall_summary` function as a separate file in the `system` folder. You can test this function by invoking it from the `main.c`, but the correctness of your implementation should be independent of the `main.c`. In other words, the results of `pr_status_syscall_summary` should be the same when it is called by any user function.

Submissions instructions

1. Make sure that your output follows the provided output template.
2. **Important:** You can write code in `main.c` to test your procedures, but please note that when we test your programs we may replace the `main.c` file. Therefore, do not implement any essential functionality in the `main.c` file. Also, turn off debugging output before submitting your code.
3. Go to the `xinu/compile` directory and invoke `make clean`
4. Create a `xinu/tmp` directory and copy all the files you have modified/created (both .h files and .c files) into it (please maintain the tmp directory structure as the `xinu` folder). For example, if you have modified `Makefile`, `system/create.c` and `open.c`, your directory `xinu_tmp` will look like:

```
-xinu
   -[existing folders...]
   -tmp [this is the folder you created]
      -compile
           Makefile
      -system
            create.c open.c
```

Note that the use of the `tmp` folder aims to help TA to quickly identify what files have been modified. Please make sure these files still have their original versions in the corresponding folders so that the Xinu can be successfully compiled.

5. The homework report should contain:
    - The answer to all questions posed in this homework assignments
    - For each coding problem, a **brief** description of your implementation (which files have you added/modified? What are the main data structures used by your implementation?)

    Put your homework report in the `xinu/tmp/report` folder and name it `homework1.pdf.`
6. Go to the parent folder of the xinu folder. Compress the whole `xinu` directory into a *tgz* file.

```
tar czf xinu_homework1.tgz xinu
```

7. Submit your assignment through Moodle. Please upload only one tgz file.