ECE492/592 – Operating Systems Design: Project #1

Due date: September 12, 2019

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

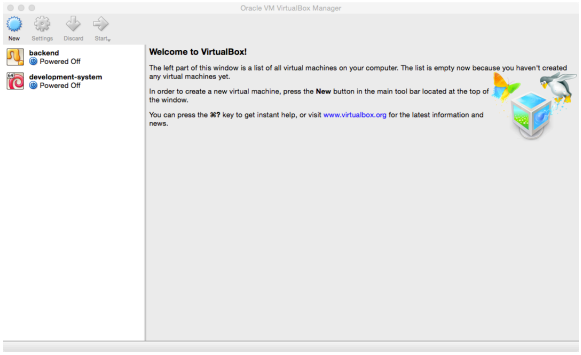
Xinu (http://xinu.cs.purdue.edu) is an operating system developed by Prof. Douglas Comer's group at Purdue University. It is a small operating system suitable for embedded environments that supports dynamic process creation, dynamic memory allocation, network communication, local and remote file systems, 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

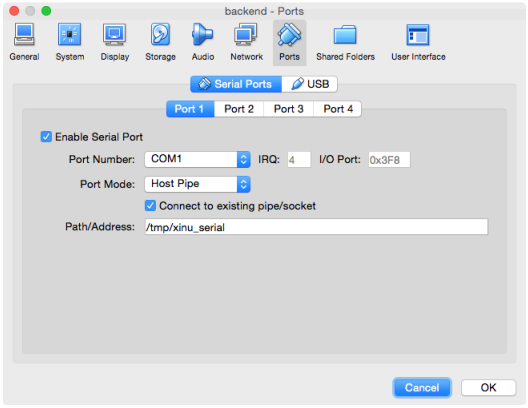
There are different versions of Xinu. In this course, we will use the one described in the second edition of the textbook above. More specifically, we will use the version of Xinu that runs in a Virtual Machine environment called Virtual Box.

Step 1: Installing Xinu on VirtualBox

- 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. Xinu has been tested on VirtualBox 5.2.18, so you might want to install that version to be on the safe side.
- Download the 2015 VirtualBox version of Xinu from: ftp://ftp.cs.purdue.edu/pub/comer/private/Xinu/xinu-vbox-appliances.tar.gz
- 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.



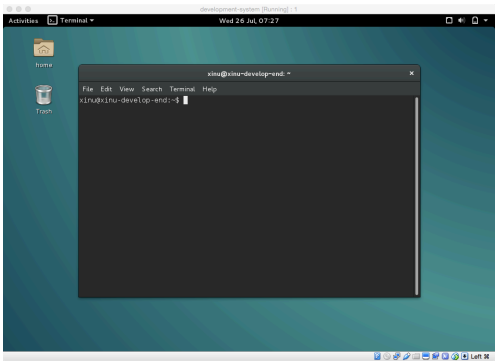
- Highlight xinu-backend in the left-hand menu, and click on Settings. From this menu, navigate to Ports (or "Serial Ports" in some distributions). Make sure that Enable Serial Port and Connect to Existing Pipe/Socket are both checked. Under Path/Address:
  - If you're on a Linux or Mac: type in /tmp/xinu_serial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



- 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:



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.

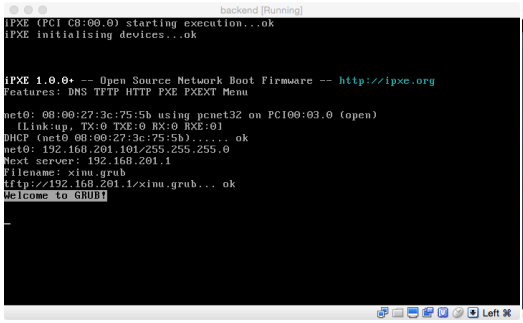
   Most of your time in development will be spent in the include/ and system/ directories.

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.

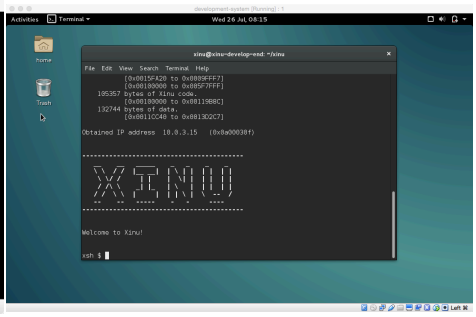
4. Now type sudo minicom (or sudo minicom --color=on if you want to see colors in the terminal window). You'll be prompted for the password. This turns the terminal window into a serial console that is connected to the back-end VM, effectively emulating a terminal for the back-end VM. All output from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



Development system:



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 (note: ECE492 students are required to implement only P1 and P2)

P1: Timing

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 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* (in ms). 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. See the `project1_template.txt` file for the format of the output of the modified `ps` command.

P2: Process creation and stack handling

Implement a *fork* system call similar to Unix's *fork*. The implementation should be in a separate `fork.c` file within the `system` folder. The function declaration must be as follows:

```
pid32 fork()
```

The fork primitive creates a new process (the child) by *almost* duplicating the parent process. Note that Xinu's processes are essentially threads: they share the memory address space but have *private stacks*.

Initialization: The child process should be initialized to have the same name, priority and stack length as the parent process. The *prsem*, *prhasmg* and *prdesc* fields of its process control block (PCB) should be initialized as during standard process creation (to `-1`, `FALSE` and `CONSOLE`, respectively). The child process should be set in READY state (and inserted in the ready list) upon creation. The ready list contains the processes that are eligible for execution. Have a look at `ready.c` (line 25) to see how to add a process to the ready list.

Return value: On success, *fork* must return the PID of the child process to the parent, and value `NPROC` to the child (note: differently from Unix's *fork*, the function you are coding does not return value 0 to the child because that's a PID in use). On failure, *fork* returns SYSERR.

Execution: After creation, *the child process should resume execution starting from the first instruction following the fork system call*.

Testing: Test cases for the *fork* system call are provided in `main.fork`. To run these test cases, rename `main.fork` into `main.c`, and compile the code. Note that you can selectively disable the test cases by commenting out the respective "`#define TESTCASEx`" macro definition at the beginning of the file. The expected output is contained in the provided `fork.output` file.

Hints: This problem requires a good understanding of the stack and its handling by Xinu. To this end:

- 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.
- Have a look at `create.c` and `ctxsw.S` to see how the stack is initialized and how it is handled upon context switch.
- The `stacktrace` system call (in `stacktrace.c`) prints the stack backtrace for a given process, and can help you understand how to traverse the stack.
- Have a look at the three test cases in `main.fork` and make sure that you fully understand their expected output (in `fork.output`).
- If you want to the see assembly file corresponding to any Xinu file, you can proceed as follow. First, modify the `Makefile` (in the `compile` folder) and add "`-S`" to the `CFLAGS` compiler flag. Second, compile using "`make clean; make`;". After compilation, the `.o` files in the `binaries` subfolder will contain assembly (rather than object) code.

Question Q7: what is the effect of the "`receive()`" call in the test cases provided? What would happen if that function call was not present? *Hint*: see `receive.c` and `send.c` files.

P3: Tracking of system calls (only 592 students)

Consider the following system calls, which cause process state changes: `create`, `kill`, `ready`, `sleepms`, `suspend`, `wait`, `wakeup`, `yield`. In this coding problem, you need to implement a function that prints out a summary of all the invocations to these system calls occurred since the start of Xinu execution. The function declaration must be as follows:

```
void pr_status_syscall_summary())
```

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 in clock cycles* (i.e., how long it takes to execute each system call on average). In order to do this, you will need to modify the implementation of these system calls to trace their invocations.

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. See the `project1_template.txt` file for the format of the output of the `pr_status_syscall_summary` function.

Important note: Unfortunately, you cannot rely on the interrupt-based timer of P2 to measure system calls, because interrupts get disabled inside most of the system calls you are attempting to measure. So, you need to use a hardware-based mechanism that does not rely on interrupts reaching the OS. The hardware timestamp counter in x86

architecture provides this functionality. It measures the number of CPU clock cycles since the last reset of the CPU. You can implement a time measurement (in cycles) based on the values read from this counter. More details about this counter are in this whitepaper from file://localhost/Intel https/::www.intel.com:content:dam:www:public:us:en:documents:white-papers:ia-32-ia-64-benchmark-code-execution-paper.pdf

Submission instructions

1. Make sure that your output follows the provided output template.
2. Important: You can write code in `main.c` to test your functions, 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 (the `tmp` folder should have the same directory structure as the `xinu` folder). For example, if you have modified `Makefile`, `system/create.c` and `open.c`, your `xinu` directory 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 the TA to quickly identify what files have been modified. Please **do not delete** the files that you have created/modified from the original folders (i.e., the `xinu/include`, `xinu/system`, etc., you have been working on).
5. The project report should contain:
   - The answer to all questions posed in this assignment (Q1-Q7)
   - 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?)
6. Go to the parent folder of the `xinu` folder. Compress the whole `xinu` directory into a *tgz* file.

```
tar czf xinu_project1.tgz xinu
```

7. Submit your code and report (in pdf format) through Moodle. Please upload only one tgz file.