CS 354 Spring 2024

Lab 1 Part-1: Getting Acquainted with XINU [50 pts]

Due: 1/22/2024 (Monday), 11:59 PM

1. Objectives

The objectives of this lab assignment is to familiarize you with the steps involved in compiling and running XINU in our lab and creating a few XINU processes.

---

2. Setting up XINU

2.1 XINU Configuration

To use XINU, several environment variables must be set. First log onto one of the frontends **xinu01.cs.purdue.edu**, **xinu02.cs.purdue.edu,** ..., **xinu21.cs.purdue.edu** which are Linux PCs. For remote login see Section 2.5. To login you should be able to use your career account user and password. Please note that CS has recently moved to two-factor authentication when using ssh. If you have any trouble with your account, please contact ScienceHelp@purdue.edu.

Note: If you are accessing a frontend machine remotely from a Windows machine, it is strongly recommended that you install and use the freeware PuTTY. Using Windows PowerShell or cmd terminal may not correctly map keyboard input CTRL-space correctly which is needed to connect to a backend machine from a frontend machine. The default configuration of PuTTY on Windows 10 and 11 (and likely earlier versions) will work correctly. Another stable working environment is to install WSL2 (Windows Subsystem for Linux version 2) on Windows running Ubuntu and open an Ubuntu terminal instead of PowerShell/cmd. If you choose not to use PuTTY or WSL2 on Windows, you are responsible for handling issues that may arise. There should be no issues when accessing the frontends remotely from a Linux or MacOS machine via ssh.

The following assumes that your shell is bash. The syntax may vary slightly if you use a different shell. (Run "echo $0" to determine your current shell.)

Setting environment variables for XINU:

1. Edit **.bashrc** in your home directory by adding /p/xinu/bin to your path:

export PATH=${PATH}:/p/xinu/bin

2. Run **source .bashrc** (or its equivalent) to make the change take effect.

Accessing and untarring XINU tarball source files:

1. Change to your home directory, if you are not already in it.
2. Unpack:

```
tar zxvf /homes/cs354/xinu-spring2024.tar.gz
```

In your home directory, you will now have a directory called xinu-spring2024. The subdirectories under this directory contain source code, header files, configuration files, and Makefile for compiling XINU.

2.2 Building XINU

To compile the XINU kernel which will be loaded and executed on a backend machine, run "make" in the **compile/** directory:

```
% cd xinu-spring2024/compile
% make clean
% make
```

This creates a loadable executable file named **xinu.xbin**. This file is loaded on a backend machine in Section 2.3 ('Running XINU').

If, upon loading xinu.xbin and power cycling to run XINU the system hangs, run 'make rebuild' then 'make clean' followed by make to rebuild a XINU image that removes dependencies from the current build. In cases where a change is incremental and straightforward, just running 'make' may suffice to generate a correct build without slowing down gcc by recompiling the entire source code. If XINU hangs or crashes after following the above steps, the cause is likely to be bugs in your code

modifications. To avoid starting from scratch, maintain working versions that you can return to.

2.3 Running XINU

The executable XINU binary runs on a selected backend machine. We have 96 dedicated backends: **galileo101.cs.purdue.edu**, ..., **galileo196.cs.purdue.edu**. As these are real physical machines, some may be down due to hardware and software issues. If they do not function as specified below, select another machine and try again.

The backends are all Intel Galileo development boards which run the quark system on chip processor.

The backend machines are shared resources. When a backend machine is grabbed by a student, it is dedicated for use by that student to run his/her version of XINU. To see which backends are available for booting XINU, type:

```
% cs-status -c quark
```

This will show you who is using each backend and how long they have been using it. As with all hardware, sometimes they fail and may become unavailable until repaired by our technical staff.

To boot your copy of XINU on a backend, connect to a back-end, say, galileo115, by issuing the command:

```
% cs-console galileo115
```

Similarly for the other backends by specifying a different backend name. With no arguments cs-console will connect you to the first available backend (including broken ones). Occasionally, backend hardware malfunctions. If you suspect this to be the case, please inform the TAs and select a different backend.

To load your copy of XINU onto a selected backend perform:

```
(control-@) OR (control-spacebar)      //esc to local command-mode
```

(command-mode) d   //download command

file: xinu.xbin   //tell it to download 'xinu.xbin' (this example assumes that you are in the xinu-spring2024/compile directory)

(control-@) OR **(control-spacebar)**   //esc to local command-mode

(command-mode) p   //power cycle the backend

After several seconds XINU should boot with a "Welcome to Xinu!" message that looks something like this:

```
Xinu for galileo -- version #18  (hkalluri)  Sat Jan 13 12:31:04 AM EST 2024

Ethernet Link is Up
MAC address is 98:4f:ee:00:08:0e
 250088992 bytes of free memory.  Free list:
           [0x001591E0 to 0x0EFD8FFF]
           [0x0FDEF000 to 0x0FDEFFFF]
    103629 bytes of Xinu code.
           [0x00100000 to 0x001194CC]
    132840 bytes of data.
           [0x0011CBC0 to 0x0013D2A7]


Hello World!

I'm the first XINU app and running function main() in system/main.c.

I was created by nulluser() in system/initialize.c using create().

My creator will turn itself into the do-nothing null process.

I will create a second XINU app that runs shell() in shell/shell.c as an
example.

You can do something else, or do nothing; it's completely up to you.

...creating a shell


----------------------------------------
   __    __   _____    _   _    _    _
   \ \  / /  |__ __|  | \ | |  | |  | |
    \ \/ /     | |    |  \| |  | |  | |
    / /\ \    _| |_   | \   |  | |  | |
   / /  \ \  |     |  | | \ |  \  --  /
```

```
   --    --   -----    -   -     ----
----------------------------------------


Welcome to Xinu!


xsh $
```

To disconnect and free up the backend:

```
(control-@) OR (control-spacebar)

(command-mode) q     //quit
```

If there are any issues disconnecting please power cycle before quitting:

```
(command-mode) p     //power cycle the backend; wait until you see
"Press [Enter] ..."

(control-@) OR (control-spacebar)

(command-mode) q     //quit
```

****NOTE****:
Please do not leave a running copy of your XINU on a backend. This may prevent others from using that backend.

2.4 Troubleshooting

1. We only show the mapping from bash to tcsh. If you use other shell types, please contact the TAs. The mapping is as follows: Edit **.cshrc** instead of **.bashrc**. Add a line **setenv PATH ${PATH}:/p/xinu/bin** instead of **export PATH=${PATH}:/p/xinu/bin**. Run **tcsh** instead of **source .bashrc**.
2. Try to figure out what's going on by yourself. Oftentimes the steps described above were not precisely followed.
3. When your XINU executable misbehaves or crashes (i.e., does not do what you intended when programming the kernel) then debug the problem(s) and try again. Since your version of the XINU kernel runs over dedicated hardware, you are in full control. That is, there are no hidden side effects introduced by

other software layers that you are not privy to. By the same token, everything rests on your shoulders.

4. If you get repeatedly stuck with "Booting XINU on ... " please contact the TAs.
5. If you are not able to get a free backend, please contact the TAs.

2.5 Remote Login

You can remote access the frontend lab PCs using TLS/SSL applications such as ssh on Linux/MacOS and ssh.exe using Command/Power shell (or PuTTY, OpenSSH, etc.) on Windows. Please note that CS has recently moved to two-factor authentication when using ssh. Use of the backends is limited to implementing, testing, and evaluating lab assignments of CS 354. You access the backends through one of the frontend machines in the XINU Lab.

2.6 An Example App: XINU Shell

By default, when XINU boots up, it runs a shell (xsh). To view all the usable shell commands, run the help command:

```
xsh $ help

shell commands are:

argecho     date        help        netinfo     udp         ?
arp         devdump     kill        ping        udpecho
cat         echo        memdump     ps          udpeserver
clear       exit        memstat     sleep       uptime
```

Take a moment to run some shell commands and view their output. Specifically make sure you run the ps command which lists information about running processes.

```
xsh $ ps
Pid Name           State Prio Ppid Stack Base Stack Ptr  Stack Size
--- ---------------- ----- ---- ---- ---------- ---------- ----------
  0 prnull         ready 0     0 0x0EFDEFFC 0x0EFDEEC0  8192
  1 rdsproc        wait   200  0 0x0EFDCFFC 0x0EFDCAAC  16384
  2 Main process   recv  20    2 0x0EFD8FFC 0x0EFD8F64  65536
  3 shell          recv  50    3 0x0EFC8FFC 0x0EFC8C6C  8192
  4 ps             curr  20    4 0x0EFC6FFC 0x0EFC6E18  8192
```

Here you can see the several pieces of information for all processes currently running. The name gives some detail about what the process is intended for. XINU always contains a special process with process identifier (pid) 0 called the null process (or

prnull). This process is the first process created by XINU and is always ready to execute. Unlike other processes, prnull is handcrafted without using create(). The inclusion of this process ensures that there is always something for XINU to execute even if all other processes have finished or are waiting for something. By default, UNIX/Linux and Windows have similar null or idle processes.

---

3. Basic system initialization and process creation [50 pts]

(a) *XINU initialization.* Inside the system/ subdirectory in xinu-spring2024/, you will find the bulk of relevant XINU source code. The file start.S contains assembly code following AT&T syntax that is executed after XINU bootstraps on a backend using the Xboot bootloader. Some system initialization is performed by Xboot, but most OS-related hardware and software initialization is carried out by start.S and other XINU code in system/ after Xboot jumps to label *start* in start.S. Eventually, start.S calls nulluser() in initialize.c which continues kernel initialization. nulluser() calls sysinit() (also in initialize.c) where updates to the process table data structure proctab[] are made. In XINU, as well as in Linux/UNIX and Windows, a process table is a key data structure where bookkeeping of all created but not terminated processes is performed.

In a Galileo x86 backend which has a single CPU (or core), only one process can occupy the CPU at a time. In a x86 machine with 4 CPUs (i.e., quad-core) up to 4 processes may be running concurrently. Most of XINU's kernel code can be found in system/ (.c and .S files) and include/ (.h header files). At this time, we will use the terms "process" and "thread" interchangeably. Their technical differences will be discussed under process management.

When a backend machine bootstraps and performs initialization, there is as yet no notion of a process. That is, the hardware just executes a sequence of machine instructions compiled by gcc and statically linked on a frontend Linux PC for our target backend x86 CPU, i.e., the executable binary xinu.xbin. nulluser() in initialize.c calls sysinit() which sets up the process table data structure proctab[] which keeps track of relevant information about created processes that have not terminated. When a process terminates, its entry is removed from proctab[]. proctab[] is configured to

hold up to NPROC processes, a system parameter defined in config/ as 100. Change NPROC to 10.

(b) *XINU idle process and removing xsh.* sysinit() sets up the first process, called the idle or NULL process. This idle process exists not only in XINU but also in Linux/UNIX and Windows. It is the ancestor of all other processes in the system in the sense that all other processes are created by this special NULL process and its descendants through system calls, e.g., fork() in UNIX, CreateProcess() in Windows, and create() in XINU. The NULL process is the only process that is not created by system call create() but instead custom crafted during system initialization.

After nulluser() sets up the NULL/idle process, it spawns a child process using system call create() which runs the C code startup() in intialize.c. Upon inspecting startup(), you will find that all it does is create a child process to run function main() in main.c. We will use the child process that executes main() as the test app for evaluating the impact of kernel modifications on application behavior as noted above. In the code of main() in main.c, a "Hello World" message is printed after which a child process that runs another app, xsh, is spawned which outputs a prompt "xsh $ " and waits for user command.

We will not be using xsh since it is but an app not relevant for understanding operating systems. Remove the "Hello World" message from main() and put it in a separate function, void greetings(void), in system/greetings.c. Call greetings() from nulluser() before it executes an infinite while-loop as an idle process. Customize the "Hello World" message so that it contains your name, username, year and semester. Don't forget to insert the function prototype of greetings() in include/prototypes.h. Modify main() so that it is completely empty but for a message that says "Test process executing main():", followed by the PID of the process. You may determine the PID of the current process by calling getpid(). Carry over these modifications to subsequent lab assignments unless specified otherwise.

(c) *Process creation*

Implement void sndch(char ch), as shown in the module 1 slides covered in lecture, in system/sndch.c. In main(), create and resume new processes that execute sndch() to test that the implementation of sndch() works correctly (Note that a parent process executing main() spawns child processes by calling create() to execute sndch).

Furthermore, you can create many such new processes to test that your change to NPROC in part (a) had the intended effect.

Important: When new functions including system calls are added to XINU, make sure to add its function prototype to include/prototypes.h. The header file prototypes.h is included in the aggregate header file xinu.h. Every time you make a change to XINU, be it operating system code (e.g., system call or internal kernel function) or app code, you need to recompile XINU on a frontend Linux machine and load a backend with the new xinu.xbin binary.

Turn-in instructions

General instructions:

When implementing code in the labs, please maintain separate versions/copies of code so that mistakes such as unintentional overwriting or deletion of code is prevented. This is in addition to the efficiency that such organization provides. You may use any number of version control systems such as GIT and RCS. Please make sure that your code is protected from public access. For example, when using GIT, use git that manages code locally instead of its on-line counterpart github. If you prefer not to use version control tools, you may just use manual copy to keep track of different versions required for development and testing. More vigilance and discipline may be required when doing so.

Submission format for Lab 1 Part-1:

```
lab1-part1

|--- xinu-spring2024
```

Specific instructions:

1. Format for submitting kprintf() added for testing and debugging purposes in kernel code:

- For problems where you are asked to print values using kprintf(), use conditional compilation (C preprocessor directives #define combined with

#ifdef and #endif) with macro XINUTEST (in include/process.h) to effect print/no print depending on if XINUTEST is defined or not. For your debug statements, do the same with macro XINUDEBUG.

2. Please make sure to follow the file and function naming convention as mentioned in the lab handout.

3. Electronic turn-in instructions:

i) Go to the xinu-spring2024/compile directory and run "make clean".

ii) Go to the directory where lab1-part1 (lab directory containing xinu-spring2024/) is a subdirectory.

For example, if /homes/alice/cs354/lab1-part1/xinu-spring2024 is your directory structure, go to /homes/alice/cs354

iii) Type the following command

```
turnin -c cs354 -p lab1-part1 lab1-part1
```

You can check/list the submitted files using

```
turnin -c cs354 -p lab1-part1 -v
```

Please make sure to disable all debugging output before submitting your code.