CS 503 - Fall 2014

Lab 0 - Getting to know XINU

Objectives:

By the end of this lab students will be able to:

- Log onto a XINU Lab workstation, download, extract, and compile a copy of the XINU source
- View the status of existing XINU backends
- Load a built XINU image onto a XINU backend and successfully boot the image
- Modify the source for XINU to start several processes that print messages to the console

Lab 0 Slides

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 5). To login you should be able to use your career account user and password. If you have any trouble with your account and its setup, please see the department help pages at http://www.cs.purdue.edu/help. If these pages do not address your particular problem, please use the trouble command from the command line to report the problem.

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 /u/u3/cs503/xinu/xinu-fall2014.tar.gz
   ```

   In your home directory, you will now have a directory called xinu-fall2014. The subdirectories under this directory contain source code, header files, configuration files, and Makefile for compiling XINU.

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-fall2014/compile
make
```

This creates an executable file called **xinu.boot**.

3. Running XINU

The executable XINU binary runs on a selected backend machine. We have 120 dedicated backends: **xinu101.cs.purdue.edu**, **xinu102.cs.purdue.edu**, ..., **xinu220.cs.purdue.edu**.

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:

NOTE: For this lab we will be using the Linksys (MIPS) backends. For all other labs we will be using the Galileo (x86) backends

```
cs-status -c mips
```

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 by issuing the command:

```
cs-console [linksys__]
```

With no arguments cs-console will connect you to the first available backend (including broken ones). For best results, you should specify a backend by name (linksys101 through linksys146).

To load your copy of XINU onto the backend:

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

(command-mode) d    //download command

file: xinu.boot      //tell it to download 'xinu.boot' (this example assumes that you are in the xinu-fall2014/compile directory)

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

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

After several seconds, you will see a message that looks like this:

```
        eth0 up
        eth0
        ### main_loop entered: bootdelay=1

        Hit any key to stop autoboot:  1
```

You will have 1 second to press any key to stop the boot process which will allow you to boot XINU.

If you hit a key fast enough you should see a prompt that begins with arXXXX>

```
        Hit any key to stop autoboot:  0
        ar7100>
```

From this prompt enter the following command:

```
        ar7100> bootp 0x81000000
```

You will see some output consisting of a Dynamic Host Configuration Protocol (DHCP) and a BOOTP message. Eventually you will see a prompt again. When this happens enter the following command:

```
        ar7100> bootm
```

This should boot XINU and you should see a welcome message which will look similar to this:

```
Starting kernel ...


(MIPS Xinu) #30 (lembkej@xinu20.cs.purdue.edu) Mon Aug 25 11:00:43 EDT 2014

  33554432 bytes physical memory.
           [0x80000000 to 0x81FFFFFF]
     65536 bytes reserved system area.
           [0x80000000 to 0x8000FFFF]
    265568 bytes Xinu code.
           [0x80010000 to 0x80050D5F]
      8192 bytes stack space.
           [0x80050D60 to 0x80052D5F]
  33215136 bytes heap space.
           [0x80052D60 to 0x81FFFFFF]

Creating shell...


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


Welcome to Xinu!


xsh $
```

If you do not see this message notify your teaching assistant. The most common problem is not hitting a key fast enough to bring up the "arXXXX>"

NOTE: You will have to follow these steps every time you power cycle the backend.

To disconnect and free up the backend:

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

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

(control-@) OR (control-spacebar)

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

NOTE:

Please do not leave a running copy of your Xinu on a backend. This will prevent others from using that backend.

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.

5. Remote Login

Use of the backends is limited to implementing, testing, and evaluating lab assignments of CS 503. To access the backends, you don't have to be physically present in the XINU lab but may remote login using ssh to one of the lab's frontend machines.

6. Using the 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         exit         led          ping         udpecho
arp          devdump      help         memdump      ps           udpeserver
cat          echo         ipaddr       memstat      sleep        uptime
clear        ethstat      kill         nvram        udpdump      ?
```

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 0x80052D60 0x80052B78       8192
  1 rdsproc          wait   200    0 0x81FFBFFC 0x81FFBA70       8192
  2 Main process     recv    20    0 0x81FF9FFC 0x81FF9F30      65536
  3 shell            recv     1    2 0x81FE9FFC 0x81FE9C38       4096
  5 ps               curr    20    3 0x81FE8FFC 0x81FE8DF0       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. 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. It behaves similar to the "System Idle Process" in the Windows Operating System.

7. Creating processes in XINU

The code for the first user process that is created by XINU is located in system/main.c. The code by default creates a single process to execute the XINU shell. The code to create she shell process looks like this:

```
kprintf("Creating shell...\n\r");
resume(create(shell, 4096, 1, "shell", 1, CONSOLE));
retval = recvclr();
while (TRUE) {
        retval = receive();
        kprintf("\n\n\rMain process recreating shell\n\n\r");
        resume(create(shell, 4096, 1, "shell", 1, CONSOLE));
}
```

Processes in XINU are created using the create system call which does the following:

1. Creates all necessary control structures for the new process
2. Initializes the process stack for the new process

The create system call returns the process identifier (PID) for the new processes. The code for the create system call is located in create.c. Open system/create.c and familiarize yourself with what it does. Here is a summary of the create call along with its parameters:

```
/*------------------------------------------------------------------------
 *  create, newpid  -  Create a process to start running a procedure
 *------------------------------------------------------------------------
 */
pid32   create(
          void          *funcaddr,      /* address of function to run   */
          uint32        ssize,          /* stack size in bytes          */
          pri16         priority,       /* process priority             */
          char          *name,          /* process name (for debugging) */
          uint32        nargs,          /* number of args that follow   */
          ...
        )
```

The call to create in main.c creates a new process to execute the code located in the "shell" function.

1. The stack size is set to 4096 bytes
2. The priority is set to 1 (the larger the number the higher the priority)
3. The name of the process is set to "shell"
4. There is 1 argument sent to the shell function: the value of CONSOLE

NOTE: when a process is created, it is not automatically executed. The status of the process must be changed to "ready". This is done by using the resume system call which takes as input the processed identifier. This tells XINU that the processes is ready to execute on the CPU.

Create the following function at the top of main.c:

```
void myprocess(int a) {
        kprintf("Hello world %d\n", a);
}
```

Remove the code in main.c that creates a shell process and replace it with the following:

```
resume(create(myprocess, 4096, 30, "helloworld", 1, 1));
while(TRUE) {
        // Do nothing
}
```

Compile a new xinu image, download it to your xinu backend, and power cycle the backend.

Once XINU boots up, you should see the message "Hello world" printed to the console.

Add some additional calls to create and resume to create several processes that call the myprocess function.

```
resume(create(myprocess, 4096, 50, "helloworld1", 1, 1));
resume(create(myprocess, 4096, 50, "helloworld2", 1, 2));
resume(create(myprocess, 4096, 50, "helloworld3", 1, 3));
resume(create(myprocess, 4096, 50, "helloworld4", 1, 4));
resume(create(myprocess, 4096, 50, "helloworld5", 1, 5));
while(TRUE) {
        // Do nothing
}
```

Compile, load, and run this new code. You should see multiple messages printed to the console (one for each process).

8. Changing process priority

The process scheduler in XINU uses a very simple rule for deciding which process to run. Each process gets a set time to run on the CPU before it gets context switched for another process. XINU always chooses to execute the highest priority process that is ready to run.

Create another function in the main.c file that contains the following code:

```
void myhungryprocess(void) {
        kprintf("I'm a looping process\n");
        while(TRUE) {
                // Do nothing forever
        }
}
```

Now modify your existing code in the main function to call the new process, but give it a high priority:

```
resume(create(myhungryprocess, 4096, 1000, "hungryprocess", 0));
resume(create(myprocess, 4096, 50, "helloworld1", 1, 1));
resume(create(myprocess, 4096, 50, "helloworld2", 1, 2));
resume(create(myprocess, 4096, 50, "helloworld3", 1, 3));
resume(create(myprocess, 4096, 50, "helloworld4", 1, 4));
```

```
while(TRUE) {
        // Do nothing
}
```

What do you observe when you run this on a XINU backend?

Since XINU always chooses the highest priority processes to run and all 5 of the processes created are ready, the hungry process with priority 1000 gets to execute on the CPU. From that point on, every time the XINU scheduler runs, it chooses the hungry process to run and no other processes get to execute.

Familiarize yourself with setting priorities of various processes. Take a look at the output from the ps command to see the priorities for the processes created by XINU at startup.

See your teaching assistant for additional activities and challenges.