CS 354 Fall 2025

Lab 1: Getting Acquainted with Xinu and Controlling Access to Shared Variables [20 pts]

Due: 9/8/2025 (Monday), 11:59 PM

1. Purpose

The objective of this lab assignment is to familiarize you with the steps involved in compiling and running XINU in our lab, errors that can occur when processes share variables, and the use of semaphores to guarantee safe and correct access of shared variables.

---

2. Setting up Xinu

This section provides background material to help you get started with Xinu. If you are already familiar with the Xinu environment and basic setup, you may proceed directly to Section 3. However, you are encouraged to return to this section whenever you need clarification or a refresher on configuration, compilation, or running 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 and setting environment variables that specify the architecture you are using, the download file name, and the name of a server in the lab that manages the back-end computers:

```
# path of cs-console, cs-status
export PATH=${PATH}:/p/xinu/bin
# architecture
export CS_CLASS="cortex"
# download filename
export CS_FILENAME="xinu"
# name of server csconsole contacts
export CS_SERVERS="xinuserver"
```

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 or a sub-directory inside your home directory.
2. Unpack:

```
tar zxvf /homes/cs354/xinu-fall2025.tar.gz
```

In your home directory, you will now have a directory called `cs354_bbb_xinu`. The subdirectories under this directory contain all the Xinu source code. There is an `include` directory that contains header files, a `system` directory that contains the source files for the operating system kernel, a `device` directory that contains device drivers, and a `config` directory that contains configuration programs. We will explore many of these later. For this lab, you need to know (1) file `main.c` resides in the `system` directory, and (2) the compile directory contains object files as well as a `Makefile` to rebuild 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 cs354_bbb_xinu/compile
% make clean
% make
```

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

If, upon loading xinu 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 50 dedicated backends: **beagle117.cs.purdue.edu**, ..., **beagle183.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 ARM Cortex-M microcontrollers which run the cortex 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
```

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 backend by issuing the command:

```
% cs-console
```

This will connect you to the first available backend (skipping any that have been removed from the rotation).

Optionally, you can specify a particular backend name, for example:

```
% cs-console beagle117
```

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) g    // download and power-cycle
```

After some time Xinu should boot with a "Welcome to Xinu!" message that looks something like this:

```
Xinu for bbb -- version #93  (comer)  Thu May  8 09:08:25 AM EDT 2025
Ethernet Link is Up. Speed is 100Mbps
Link is Full Duplex
MAC Address is: 6C:EC:EB:AC:51:11
 519766112 bytes of free memory.  Free list:
           [0x8104DFA0 to 0x9FFFDFFF]
     93452 bytes of Xinu code.
           [0x81000000 to 0x81016D0B]
    148416 bytes of data.
           [0x81017000 to 0x8103B3BF]

Obtained IP address  10.168.15.117   (0x0aa80f75)



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


Welcome to Xinu!


xsh $
```

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.

2.4 Disconnecting Xinu

To disconnect and free up 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 may prevent others from using that backend.

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 Troubleshooting

1. If you are using tcsh, edit **.cshrc**, add a line: **setenv PATH ${PATH}:/p/xinu/bin**, and then run **tcsh**. If you are using bash, edit **.bashrc**,

add a line: **export PATH=${PATH}:/p/xinu/bin**, and then run **source .bashrc**. If you use other shell types, please contact the TAs.

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.

---

3. Guarding Access to Shared Global Variables [20 pts]

3.1 Log into a front-end machine

Use your **career account** to log into one of the front-end computers in the Xinu lab, and perform the settings following Section 2.1 of the handout. For remote login via SSH, please follow instructions in Section 2.5.

> **Note:** No matter which computer you choose, your home directory will appear the same. Later in the course, we will study how this shared home directory functionality is achieved.

3.2 Unpack, build, and run the starter code

Follow the instructions in Section 2.2 and Section 2.3, **compile, download, and run** the provided version of Xinu.

Try running the command:

```
date
```

to confirm that the system is working correctly.

3.3 Modify system/main.c (simple printing)

Now we will experiment with performing some minor modification to the Xinu source file, by

- Write a function `pr10` that uses `kprintf` to print the numbers **1 through 10** to the console.
- Modify `main` by removing the call to the Xinu shell and calls `pr10` and then enters an infinite loop:

  ```
  while (1) ;
  ```

- Recompile Xinu and run the new version to verify that it prints the integers 1 through 10.

3.4 Run pr10 concurrently

Modify `main` again:

- Before calling `pr10`, create a new process that runs `pr10`
- This means that both the new process and the `main` process will run `pr10` concurrently.
- Recompile Xinu and run the new version to verify that it prints the integers 1 through 10.

3.5 Global variable test

1. Modify the `Makefile`
   - In `Makefile` located in the `compile/` directory, modify this line:

     ```
     CFLAGS  = -mcpu=cortex-a8 -mno-unaligned-access
     -marm -fno-builtin -fno-stack-protector
     -nostdlib -c -Wall -O ${DEFS} ${INCLUDE}
     ```

     to

```
CFLAGS  = -mcpu=cortex-a8 -mno-unaligned-access
-marm -fno-builtin -fno-stack-protector
-nostdlib -c -Wall -O0 ${DEFS} ${INCLUDE}
```

Note: This modification turns off compiler optimization so we can simulate concurrent modifications to the same global variables.
Note: The added character is the numeric '0'(zero), not the letter 'o'.
Note: This modification is only relevant to lab 1, unless stated otherwise. If you start from a fresh copy of Xinu source code for any subsequent labs, you do **not** need to manually reset the flags.

2. Add global state (in `main.c`)
   - Create a global integer `x` initialized to `0`.
   - Define constants:
     - `#define NPR 10` (number of processes),
     - `#define ITER 1000` (number of iterations), and
     - `#define DTIME 6` (delay time).
   - Add a global array `done[NPR]` (initialized later) to track when each worker finishes.

3. Write the worker function
   - Implement `void domany(int id)` that:
     a) Loops `ITER` times, doing `x++`.
     b) Loops `ITER` times, doing `x--`.
     c) Performs one final `x++`.
     d) Marks process completion with `done[id] = 1`.

     Note: In essence, each worker's intended net effect is `+1` to `x`, but without proper synchronization, races can corrupt intermediate updates.)

4. Initialize global states in `main`
   - In `process main(void)`, set all `done[i] = 0` for `i = 0 .. NPR-1`.

- You can optionally print a banner like `Starting process creation` using `kprintf` to indicate global states have been set.

5. Create multiple low-priority processes

   For each `i = 0 .. NPR-1`,

   - Build a process name `pname` that looks like `do_num_0, do_num_1, …, do_num_{NPR-1}` (you can use `sprintf` to convert the integer `i` into a string containing it.)
   - create a new process with the following parameters
     - stack size of `8192` bytes,
     - priority `10`,
     - process name `pname`, and
     - single int argument `i`.

6. (Optional) Early-finish peek
   - Immediately after creating the processes, scan `done[i]`.
   - If any `done[i] != 0`, print `"%d is done early\n", i` to illustrate non-deterministic timing.

7. Report immediate state and pause
   - Print `All processes created, and x is %d` to show `x` right after creation.
   - Call `sleep(DTIME);` (sleep for 6 seconds) to give workers time to run.

8. Wait for completion without synchronization
   - Loop until all `done[i] == 1`.
     - If a `done[i] == 0`, call `sleepms(1)` and keep waiting. (This busy-wait is intentionally simple; the key point is that we are **not** using semaphores or locks.)

9. Compare expected vs. actual
   - After all workers are done, print the expected and actual values, e.g.,

    - Expected: `NPR` (since each worker intends to contribute `+1`).
    - Actual: the final `x`.
- Example: `Expected x to be %d and found %d\n\n`, with `%d` filled by `NPR` and `x`.

Boot the system **three times** and record the results.

3.6 Compare printf vs. kprintf

Change the code to use `printf` instead of `kprintf`. Observe any differences in behavior.

Note: The change is only for this step (3.6) unless noted otherwise. Please revert to using `kprintf` for all subsequent steps of the lab.

3.7 Add synchronization with semaphores

- In `main`, Create a global semaphore id `sid32 mutex`.
- Modify `main` so that **before** it creates the processes, it initializes the semaphore with an initial count of 1 using `semcreate`.
- Modify `domany` so that it:
    - Calls `wait(mutex)` before modifying `x`.
    - Calls `signal(mutex)` after modifying `x`.

Boot the system **three times** and record the results.

---

4. Questions

Place **short answers** to the following questions in a file named `questions.txt`. You will later convert this file to a PDF for submission.

1. In Step 3.3, what was the output, and what happened after the output appeared?
2. In Step 3.4, what was the output?
3. In Step 3.5, what was the output of the three runs? Was the final x equal to NPR? Explain why.

4. In Step 3.6, did the output differ between `printf` and `kprintf`?
5. In Step 3.7, were results from the three runs the same? Was the final x equal to NPR? Explain why.

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 binary.

---

5. Submission

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.

You must submit a directory named `lab1` that contains both required files:

- Your final version of `main.c` from Xinu
- `questions.pdf`, containing your answers

Go to the directory where `lab1` is a subdirectory.

For example, if `/homes/alice/cs354/lab1` is your directory structure, go to `/homes/alice/cs354`

Type the following command to submit the directory with `turnin`:

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

Be sure the files inside the directory are named exactly `main.c` and `questions.pdf`.

You can check/list the submitted files using

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