Purdue University
Purdue e-Pubs

Department of Computer Science Technical Reports | Department of Computer Science

1996

A Laboratory Environment for Experimenting with Xinu

Douglas E. Comer
Purdue University, comer@cs.purdue.edu

John C. Lin

Report Number:
96-047

Comer, Douglas E. and Lin, John C., "A Laboratory Environment for Experimenting with Xinu" (1996). *Department of Computer Science Technical Reports.* Paper 1301.
https://docs.lib.purdue.edu/cstech/1301

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

A LABORATORY ENVIRONMENT FOR EXPERIMENTING WITH XINU

Douglas E. Comer
John C. Lin

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD-TR 96-047
August 1996

A Laboratory Environment For Experimenting With Xinu

Purdue Technical Report CSD-TR 96-047

Douglas E. Comer and John C. Lin
Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907, U.S.A.

August 23, 1996

Abstract

Twelve years ago, the Computer Science Department established the Xinu laboratory, which is currently used for education as well as operating systems research. During the past year, facilities in the lab were replaced by Intel Pentium computers, and the support system was redesigned to accommodate the new hardware. This paper describes the new hardware and software used in the lab. It shows how computers are connected to networks, and explains how the support software permits users to cooperate in using the new hardware.

1 Introduction

Xinu [2, 3] is a small, multi-threaded operating system that follows a hierarchical structure. Twelve years ago, the Computer Science Department established the Xinu laboratory, which is currently used for education as well as operating systems research. The laboratory includes two sets of computers. Computers in one set, called *front-ends*, are conventional UNIX workstations that students use to create, compile, and link operating system images. Computers in the other set, called *back-ends* are bare machines that students use to test the operating systems they create.

We have designed support software that allows students to cooperate in using the back-end machines. Designed to be convenient, the support system allows a student to obtain exclusive use of a back-end computer, download a memory image into the computer, start execution, and then monitor the system while it executes. Once a student finishes using a back-end, the student releases the back-end, which allows other students to allocate the machine.

During the past year, facilities in the lab were replaced by Intel Pentium computers, and the support system was redesigned to accommodate the new hardware. This paper describes the new hardware and software used in the lab. It shows how computers are connected to networks, and explains how the support software permits users to cooperate in using back-end. Finally, the paper discusses the difficult problem of regaining access to a back-end that has been left disabled.

2 Hardware Environment

There are three groups of computers in the Xinu laboratory: front-end computers, back-end computers, and server computers. The three groups of computers are attached to a local area network (LAN) (e.g., an Ethernet). A front-end computer (or a *frontend*) is a general purpose computer that runs a conventional multitasking operating system such as UNIX. A user uses software tools on a frontend to compile Xinu and create a memory image for loading to a backend.

A back-end computer (or a *backend*) is a bare machine that users use to test the memory images they create. A backend has a processor for code execution, a non-volatile memory device (e.g., a ROM or a floppy disk device) for storing bootstrapping code, a LAN interface for supporting network communication with other computers, and a serial interface for console I/O. A backend has neither a keyboard nor a video display. A user interacts with a backend from a frontend using the backend's serial interface.

A server computer (or a *server*) is a computer that runs server programs. Various kinds of servers are used to support the lab environment. For example, the Xinu image to be loaded to a backend and the information needed to initialize Xinu are stored on server computers.

2.1 An Example Arrangement

Figure 1 illustrates an example arrangement of the three groups of computers.

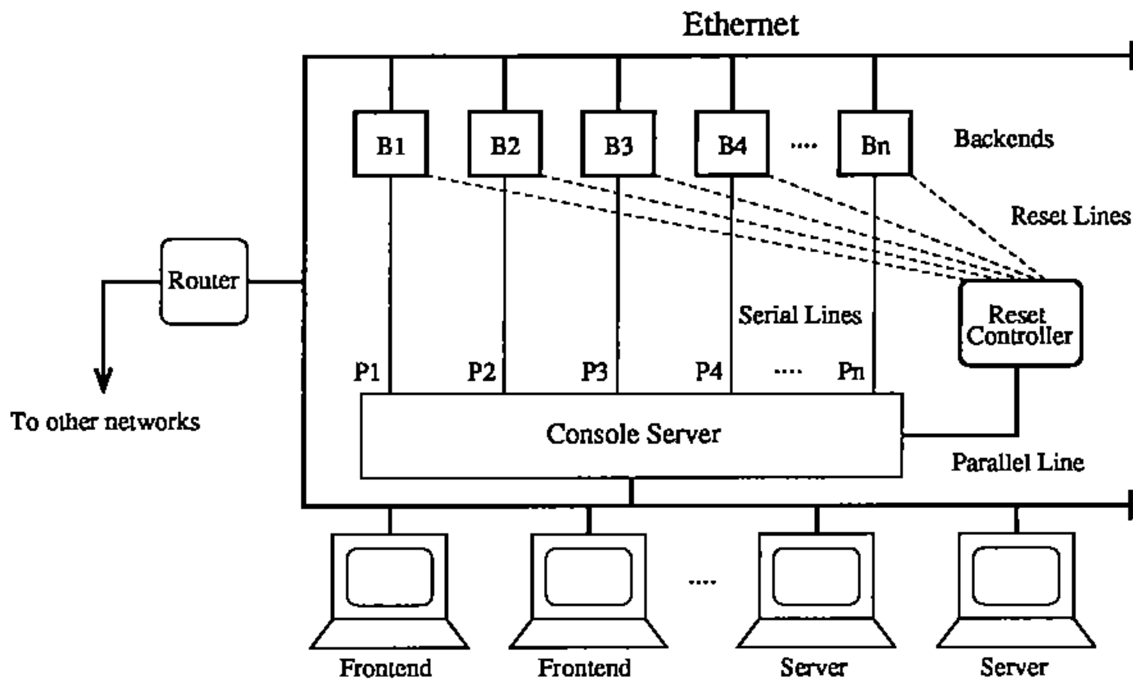

Figure 1: An example arrangement of the hardware in a Xinu lab

In the figure, a single Ethernet cable connects all the frontends, backends, and servers. In addition, a router is added to allow communication with computers outside the lab. The Ethernet supports direct communication among the three groups of computers. A backend can use an additional path to communicate with a frontend. As Figure 1 shows, a serial line connects the console port of each backend to a unique serial port on the console server. To send data to a backend's console port, a frontend sends the data to the console server, which in turn relays the data over the serial link that leads to the backend[1]. Similarly, data from the backend's console port travels the reverse path to reach the frontend.

The console server also connects to a *reset controller* that allows a frontend to remotely reset a backend[2]. As Figure 1 illustrates, the reset controller maintains a reset line to each backend and communicates with the console server using the parallel interface. For each reset line, one end connects to a backend's reset circuitry, and the other end attaches to a unique port on the reset controller. To reset a backend, a frontend sends a request across the Ethernet to the console server, which uses the request to form a command sequence and sends the sequence over the parallel interface. The reset controller receives the sequence and strobes the reset line that leads to the target backend, causing the backend to perform a warm reboot.

[1]The console server uses a separate process to manage each serial port and the parallel port.

[2]The Appendix section describes the design of the reset controller.

Note that the arrangement described above can accommodate heterogeneous processing architectures: computers in different groups as well as in the same group need not install the same type of processor. For example, a subset of the backends can use the Pentium processors, while the rest uses the SPARC processors. Utility programs allow a user to specify which type of backend to use.

3 Software Environment

A number of utility programs have been implemented to allow convenient access to the hardware environment. The utility programs consist of a set of *client* programs and a *server* program called *Connection Server* (or `cserver`). A user uses the client programs running on a frontend to perform operations that are related to bootstrapping and testing Xinu on a backend. For example, a user can use a client program to transfer a Xinu image to a file directory for bootstrapping. Each client program relies on a `cserver` to carry out the supported operations. A `cserver` runs on a server host and manages a subset of the backends (see Figure 2). One or more `cservers` can be configured to manage all the backends. To perform an operation on a given backend, a client program sends a request across the network to the `cserver` that manages the backend; the `cserver` processes the request and answers with a response.

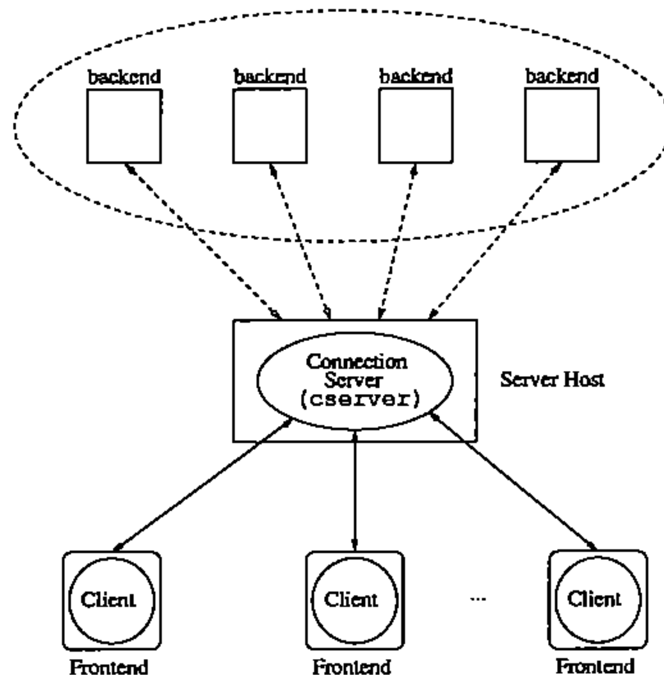

Figure 2: Illustration of multiple clients communicating with a `cserver`. Each `cserver` manages a unique subset of the backends.

An essential operation that the `cservers` support is to allow a client to establish a reliable network connection to the console port of a backend. The connection allows the user to interact with the Xinu OS

running on the backend.

The following subsections describe the cserver design, the interaction between a client and a cserver, a set of commonly used operations, and a useful client program called cs-console.

3.1 Connection Server (cserver) Design

The design of cserver is simple but powerful. Each cserver supports a set of built-in operations and a set of *external* operations. The code for a built-in operation is compiled into each cserver; the code for an external operation resides on a file system. To execute an external operation, a cserver creates a separate process. Thus, a cserver can support concurrent invocation of various external operations from multiple clients. In contrast, a cserver executes built-in operations sequentially, without using a separate a process.

Each cserver manages a subset of the backends. During initialization, a cserver reads a configuration file to learn the set of external operations that it supports for each backend that it manages. That is, the set of external operations that a cserver supports for each backend is configurable. Thus, a cserver can support an arbitrary set of external operations for each backend. Furthermore, addition and modification to the set requires no modification or recompilation of the server code. The configuration file also indicates the processor type of each backend, allowing a client to perform operations on a backend with a specific processor type.

3.2 Client-Server Interaction

A client uses both UDP [6] and TCP [7] to communicate with a cserver. To invoke an operation on a cserver, a client uses a UDP packet to carry the request to the cserver. The cserver processes the request and uses another UDP packet to carry the results back to the client. If the request invokes a built-in operation, the cserver executes the built-in operation immediately and sends the results back to the client. If the request invokes an external operation, the cserver does not create a process immediately to execute the operation; instead, it creates a passive TCP endpoint [1, 4] and uses the reply message to carry the port number of the TCP endpoint back to the client. When it receives the port number, the client immediately establishes a TCP connection to the cserver. The cserver accepts the connection, and then creates a process to execute the external operation. The new process inherits the TCP connection as its input and output device. Thus, the client can use the TCP connection to send additional data to the new process for processing. Furthermore, the new process can use the TCP connection to transport output back to the client.

3.3 Handling Multiple Connection Servers

The client-server interaction described above assumes that a client already knows a correct cserver to send each request. Because it is possible to use multiple cservers to manage the backends, a client needs a mechanism to deliver each request to a cserver that will handle the request. One approach is to deliver each request sequentially to all the cservers contained in a list (i.e., simulating a multicast using unicast)[3]. A receiving cserver uses an identifier contained in a request to determine whether it should handle the request. This approach requires each client to access the manually configured server list and incurs processing overhead on servers that are not interested in the request.

Another approach does not require clients to access the server list, but assumes that all the clients and cservers reside in the same LAN segment. A client uses broadcast to deliver each request to all the cservers on the local LAN. Like the previous approach, a cserver uses an identifier to determine whether it should handle a request. Using local broadcast is efficient in bandwidth usage and requires no prior knowledge of the address of each cserver. However, broadcasting incurs processing overhead on every host attached to the LAN.

The third approach takes user behavior into consideration. Observe that a user normally does not invoke operations on random backends. Instead, a user acquires a backend, uses the backend for a while, and then releases the backend. The observation suggests that a client program that allows a user to acquire a backend and invoke multiple operations on the backend is feasible and useful. Furthermore, such a client program can avoid the overhead of delivering each request using broadcast or simulated multicast as described previously. The client program caches the cserver information after a successful acquisition and delivers subsequent requests to the cserver using unicast. Broadcast or simulated multicast is used only for the initial acquisition.

3.4 An Example Client: cs-console

Client program cs-console is designed exactly using the third approach described above. Cs-console provides a convenient interface for a user to acquire a backend and invoke operations on the backend. Normally, a user invokes cs-console with a command line argument that specifies which backend to acquire. The argument specifies either the backend's unique host identifier (e.g., host name) or a *class identifier*, which identifies the processor type of the backend. If the host identifier is used, the user can acquire the backend only if the backend is not currently in use by another user. If the backend with the host identifier is available, the cserver that manages the backend will answer with a positive reply, permitting the cs-console client to acquire the backend. If the class identifier is used, any cserver that has an available backend with the same class identifier will reply with a positive answer; the cs-console uses

[3]The current version of the client-server software does not use link-level multicast.

the first positive reply to acquire the backend.

To acquire a backend, a `cs-console` client sends a request to the `cserver` that manages the backend. The `cserver` creates a TCP endpoint, answers the request, and waits for the `cs-console` to initiate a TCP connection. Once it has established a TCP connection with the `cs-console`, the `cserver` spawns a *telnet* [8] client process to handle the connection. The `telnet` client immediately establishes another TCP connection to the `telnet` server on the console server that handles the serial link leading to the backend. After the two TCP connections are in place, the `telnet` client relays data between the `cs-console` and the `telnet` server. The connection between the `cs-console` and the backend is called the *console* connection. Figure 3 illustrates the entities that form the console connection.

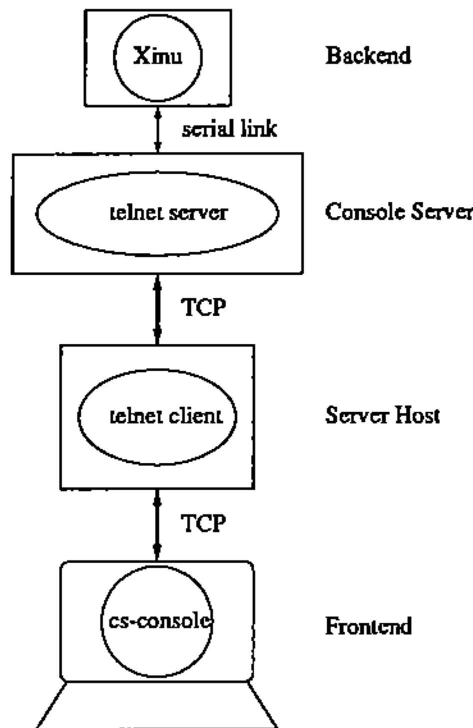

Figure 3: The entities that form the console connection between a frontend and a backend

Once the console connection is in place, the user obtains exclusive access to the console port of the backend. The user can use `cs-console` to interact with any program running on the backend that uses the console port for input and output. In addition, *cs-console* provides the user with access to the following operations on the `cserver`.

- Download: for transferring a Xinu image to a file directory for loading to the acquired backend
- Break: for interrupting the running kernel on the backend
- Reset: for resetting the backend

3.4.1 The Download Operation

Once the use has created a memory image of Xinu ready for testing on the acquired backend, she can use the *download* command of `cs-console` to transfer the image from her home directory to a *bootstrapping* directory. The download command prompts the user for a file name. Once the user enters the file name, the `cs-console` client invokes the download operation on the `cserver`, opens the file, and delivers the contents of the file to the process spawned by the `cserver` to handle the download operation. The process stores the received data on the bootstrapping directory.

3.4.2 The Monitor Program

Once the Xinu image is stored on the bootstrapping directory, the user uses a *monitor* program to handle the details of bootstrapping Xinu. The monitor program resides in a non-volatile storage device (e.g., a boot ROM or the boot tracks of a floppy disk) of each backend[4]. When a backend reboots, the backend automatically executes the monitor program. If the monitor program is not already running on the backend when the user establishes the console connection, `cs-console` provides two mechanisms for the user to invoke the monitor program. The two mechanisms will be described in subsections 3.4.3 and 3.4.4, respectively. For now, assume that the monitor program is already running on the backend and is ready to bootstrap Xinu.

The monitor program relies on two protocols to bootstrap Xinu: BOOTP [5] and TFTP [9]. The two protocols each require a server to handle the requests from the monitor program. The BOOTP server provides the monitor program with the initial bootstrapping information, such as the IP address of the backend, the file name of the memory image to be loaded and executed on the backend, and the address of the TFTP server from which the monitor can retrieve the memory image. The TFTP server implements a simple file transfer protocol that allows the monitor program to obtain the Xinu image across the network block by block.

The bootstrapping procedure consists of two phases, as illustrated in Figure 4. In first phase, the monitor program becomes a client of the BOOTP server, retrieving bootstrapping information from the server. In the second phase, the monitor program communicates with the TFTP server to load the Xinu image across the network into the main memory of the backend.

3.4.3 The Break Operation

Once bootstrapping completes, the monitor program transfers the control of CPU to Xinu. The user uses the console connection to observe and interact with the running kernel. If the user is not satisfied with the

[4]The Pentium version of Xinu also includes the monitor program.

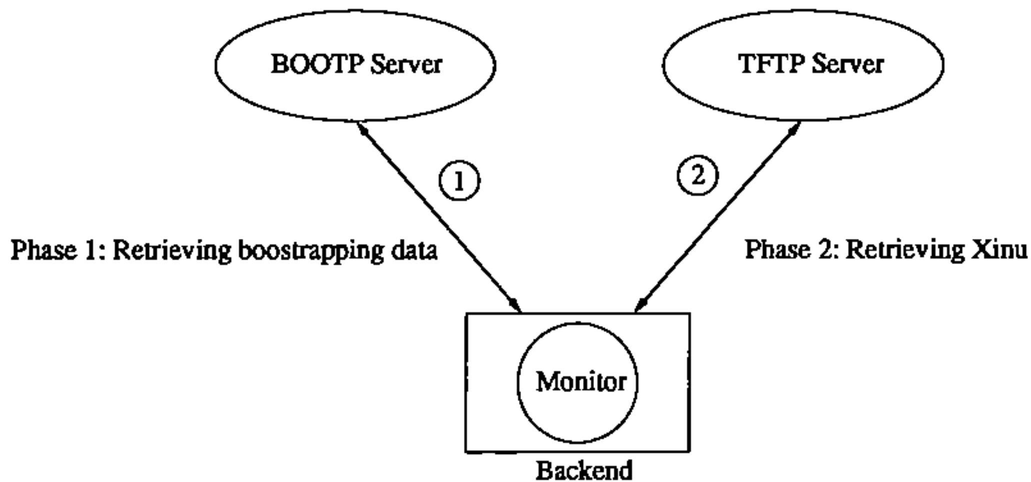

Figure 4: Illustration of the two phases of bootstrapping

version of Xinu, she can modify the source code and create another version of Xinu image. To allow the user to use the same backend for testing various versions of Xinu, `cs-console` supports a *break* command that allows the user to interrupt the running kernel and invoke the monitor program.

The break command uses a serial line *BREAK* signal to interrupt the running kernel[5]. To generate the BREAK signal, the user invokes the *break* command to pass a command sequence to the `telnet` client over the console connection. Upon receiving the sequence, the `telnet` client uses the `telnet` BRK control sequence [8] to inform the `telnet` server. The `telnet` server processes the sequence and generates a BREAK signal on the serial line. The signal causes the backend's serial interface to issue a processor interrupt that interrupts Xinu and invokes the monitor program.

Note that `cs-console` uses the console connection to send the BREAK request. Thus, `cs-console` must mark the request to allow the `telnet` client to distinguish the request from user data. Also, `cs-console` must *escape* user data that forms the request. `Cs-console` use the two-octet sequence "/b" for the BREAK request. To escape user data with the same sequence, `cs-console` converts each occurrence of "/b" to "//b" before sending to the `telnet` client. The `telnet` client converts the escaped data back to the original form (i.e., "/b").

3.4.4 The Reset Operation

When the processor of the backend is in a state that ignores the interrupt raised by a serial line BREAK signal, the BREAK command cannot invoke the monitor program. When such circumstances occur, the user can use a *reset* command to reboot the backend. Because each backend is configured to execute the monitor program at start-up time, rebooting the backend ensures that the monitor program will regain the

[5]A BREAK signal is a framing error indication supported by the serial interface hardware.

control of CPU.

The reset command invokes an external operation on the `cserver`. The process that the `cserver` spawned to handle the operation transmits a request to the server process on the console server that manages the parallel interface. The server process communicates with the reset controller over the parallel interface to reset the target backed.

3.4.5 Reclaiming a Backend

After finishing using the backend, the user can voluntarily release the backend by exiting `cs-console`, thus allowing other users to use the backend. If the user forgets to release the backend, the `cs-console` client and the `cserver` use a timeout mechanism to reclaim the backend. When the console connection is established, the `cserver` maintains a timestamp for the connection, and the `cs-console` client checks the activities on the connection at fixed intervals. If the user has used the connection, the `cs-console` client will send a message to the `cserver` to refresh the timestamp. The `cserver` permits other users to acquire the backend if the `cs-console` client does not refresh the timestamp within a predefined timeout interval.

3.5 Other Clients

Other than `cs-console`, there are other clients available for both administrative use and general use. For example, an administrator can use a special client program to break a user's console connection. Administrative clients are available only for a user with special privilege (e.g., a *super* user). Among the non-administrative clients, we mention one, *cs-status*, that is used most often. A user uses `cs-status` to query the status of the backends managed by a given `cserver` or by all the `cservers` on the local LAN. Figure 5 illustrates an example output that results from using `cs-status` to query all the `cservers` on the local Ethernet.

```
michelangelo.cs
    kiwi      Sun4c    user= muckel    time= 00:35:38
    mango     Sun4c    user=           time=
balan.cs:
    cosimo    Sun4m    user=           time=
    lorenzo   Sun4m    user=           time=
excalibur.cs:
    emu       i486     user= lin       time= 00:00:06
```

Figure 5: An example output of the `cs-status` program

The output shows the hosts (e.g., excalibur.cs) that the `cservers` execute, the set of the backends (e.g.,

emu) managed by each `cserver`, the processor type (e.g., i486) of each backend, and the user ID (e.g., lin) and idle time of each established console connection. A user can use the output to determine which backend is available, and which backend is in use by whom for how long.

4 A Typical Use of the Lab Environment

Figure 6 illustrates a typical use of the lab environment.

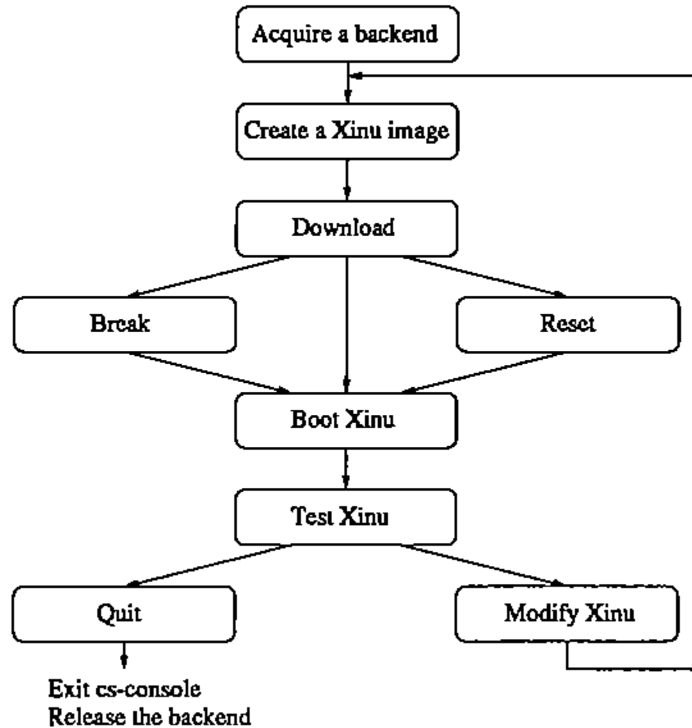

Figure 6: A typical usage of the lab environment to experiment with Xinu

A user first uses `cs-console` to acquire a backend. Then, the user creates a memory image of Xinu ready for testing on the backend. The user uses the download operation to transfer the image to the bootstrapping directory. If the monitor program is already running on the backend, the user bootstraps Xinu directly; otherwise, the user invokes either the break operation to interrupt the running program or the reset operation to reboot the backend, depending on whether the break operation can invoke the monitor program. In all cases, the user uses the monitor program to bootstrap Xinu across the local LAN. Once Xinu is started on the backend, the user observes and interacts with the running kernel using `cs-console` via the console connection. Finally, the user either exits `cs-console` or decides to modify Xinu. In the later case, the user creates another Xinu image and repeats the steps.

5 Acknowledgment

Vincent F. Russo and Patrick A. Muckelbauer designed and implemented the software described in section 3.

References

[1] D. E. Comer and D. L. Stevens. *Internetworking with TCP/IP Vol. III: Client-Server Programming and Applications*. Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

[2] Douglas E. Comer. *Operating System Design: The Xinu Approach*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[3] Douglas E. Comer. *Operating System Design Vol. II: Internetworking With Xinu*. Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[4] Douglas E. Comer. *Internetworking with TCP/IP: Principles, Protocols, and Architecture, Vol. I*. Prentice-Hall, Englewood Cliffs, New Jersey, third edition, 1995.

[5] Bill Croft and John Gilmore. RFC-951: Bootstrap Protocol (BOOTP). *Request For Comments*, Sept. 1985. Internet Network Information Center.

[6] J. Postel. RFC-768: User Datagram Protocol. *Request For Comments*, Aug. 1980. Internet Network Information Center.

[7] J. Postel. RFC-793: Transmission Control Protocol. *Request For Comments*, September 1981. Internet Network Information Center.

[8] J. Postel and J. Reynolds. RFC-854: Telnet Protocol specification. *Request For Comments*, May 1983. Internet Network Information Center.

[9] K. Sollins. RFC-1350: The TFTP Protocol (Revision 2). *Request For Comments*, Oct. 1992. Internet Network Information Center.

A Appendix

Brian Board (`boardbd@cs.purdue.edu`) designed the reset controller used in the Xinu lab. Figure 7 shows the schematic diagram of the controller. The design uses a single `74LS154` chip to control 16 relays and allows additional `74LS154` chips to be added as needed. Each relay controls the reset circuitry of one backend. The input control signals to the chip come from the parallel (printer) port of the console server (named *Annex*). Pins 2 to 5 of the parallel port select which relay to activate; pins 6 and 7 are used to enable or disable the chip.

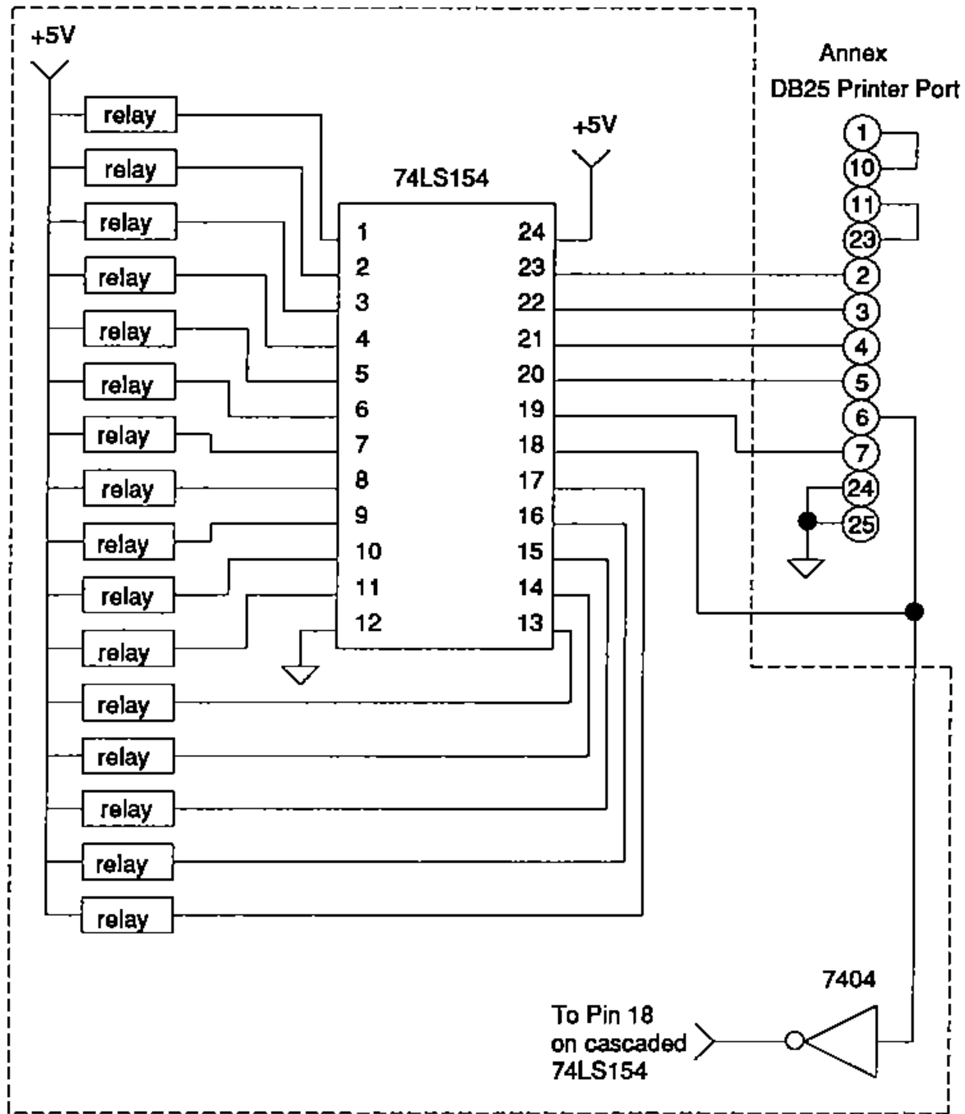

Figure 7: The schematic diagram of the reset controller