

**RUNNING LINUX ON BACK-END
COMPUTERS IN THE XINU LAB**

**Douglas E. Comer
Xuxian Jiang**

**Department of Computer Sciences
Purdue University
West Lafayette, IN 47907**

**CSD TR #03-012
May 2003**

Running Linux On Back-end Computers In The Xinu Lab

Douglas E. Comer and Xuxian Jiang

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA

Abstract

The Xinu Lab at Purdue University contains equipment used for instruction and research in operating systems and networks. In addition to conventional workstations, the lab contains a set of back-end computers and facilities that automatically allocate and download the back-ends as requested by users. The Xinu lab has been enhanced by integrating Embedded Linux into the set of available facilities. A user can download Linux on one or more back-end computers, and can run an application on each machine that acts as a packet source, a packet sink, or another type of packet processor. This report describes the technical details and challenges involved in porting Linux to the Xinu lab environment.

1. Introduction

The Xinu Lab at Purdue University was one of the earliest laboratories for instruction and research in operating systems and networks. Computers in the lab are divided into two types: *front-ends* and *back-ends*; all computers are interconnected by networks. The front-end computers consist of conventional Unix workstations (currently x86 architecture) that are used to create and compile software. Each front-end system runs a standard Unix operating system (currently Linux), and contains the usual facilities that allow users to create, compile, and link programs. Most important, software exists that allows a user to create a complete memory image, including all parts of an operating system. Each back-end system consists of a bare computer on which a user can run an arbitrary memory image, including code for an operating system. A back-end does not contain any code other than the image that a user supplies, which means the image must include all low-level system details.

In addition to hardware, the Xinu lab includes software facilities that automate the allocation, downloading, and control of back-end computers. When a user needs a back-end, the user runs a program on a front-end that makes a request. A server in the lab consults a database of back-ends, selects one that satisfies the request, and grants exclusive use of the back-end to the user. A window on the user's screen provides a connection to the back-end console. Additional software allows the user to download an arbitrary image into the back-end, and control execution. Once a user finishes using a back-end, the user informs the control software, which allows another user to allocate the machine. Details about the lab architecture can be found in [1].

The name *Xinu* is taken from an operating system created by Comer. Xinu is a small, elegant, multi-threaded system that works well in an embedded environment. Xinu is used in courses — students in the graduate operating systems or network courses are each given a copy of Xinu, which they modify or extend (e.g., replace TCP/IP protocol software with their own

code). Similarly, researchers often use Xinu as the basis for their work. As a consequence, the lab facilities are designed to ensure that users can download a Xinu image into a back-end, interact with the running system, and recover from catastrophic errors. In particular, control facilities in the lab can reclaim a back-end even if the downloaded image disables interrupts and executes a tight loop.

Although it works well as a research platform, Xinu does not offer emulation facilities to support commercial application software. Thus, users cannot easily use Xinu to run off-the-shelf packet generators, traffic analyzers, servers, or client applications such as a web browser. To accommodate such applications, we integrated a version of Linux into the Xinu lab facilities. That is, we created a version of Embedded Linux that can be downloaded onto one or more back-end computers using the standard lab facilities.

This paper describes the Embedded Linux system used in the Xinu Lab, presents the configuration, and discusses difficulties encountered. The paper is organized as follows: Section 2 describes the problem, states the constraints imposed by the Xinu lab architecture, and describes related work. Section 3 gives the design goals. Section 4 contains an overview of the steps used to run Linux on a back-end, and Section 5 describes the memory layout used with a Linux kernel image. Section 6 discusses the BIOS services that Linux requires, and explains how they are provided. Section 7 considers how space constraints impact the design of a Linux root file system, and Section 8 concludes the paper.

2. Lab Environment And Related Work

This section examines details of the Xinu lab environment that are relevant to Linux, and outlines the overall design. The focus is on issues that pose challenges and difficulties. The section also compares our design to related work.

The Xinu lab [1,2] provides facilities for research and instruction in the areas of operating systems and networking. Salient features of the lab include:

- *Automated Back-end Allocation.* A user in the lab is able to request one or more back-end computers quickly and automatically.
- *Fast Download.* An OS image can be downloaded from a front-end computer into a back-end computer and started in a few seconds. To facilitate high-speed download, an idle back-end runs a monitor program that handles communication with the downloading software.
- *Communication With Back-End Console.* There exists a mechanism, including relevant hardware and software facilities, that connects a window on a user's front-end computer to the console of a back-end computer. The window allows the console to be used for debugging output from a back-end.
- *Back-End Recovery.* Because the Xinu lab is used primarily for experimental system software, the software running on a back-end can inadvertently stop responding to console input. The lab includes a facility to restart a back-end that has become jammed; upon restart, the back-end loads the monitor image from a boot floppy, which allows the lab system to regain control.

Loading a back-end is a two-step process. When the first phase monitor code is loaded from the floppy into memory, the back-end CPU begins executing in *real mode* (i.e., *16-bit mode*). The first phase of the bootstrap requests the second phase from a server. The first phase downloads the second phase into high-end memory (address 0x10000:0x0000). Once it has downloaded an image, the second phase bootstrap code waits for further instructions from the console. If a user requests that the image be run, the bootstrap code configures the CPU to

execute in *protected mode*, and transfers control to the image. The image is responsible for moving itself to physical address 0x0000:0000 and branching to the appropriate entry point.

Whenever a new image is downloaded to a back-end, the monitor follows the same steps: the image is placed into high memory, and the image must move itself into low memory. Although it is efficient, the downloading scheme imposes unintentional side effects for Linux:

- *Absence of BIOS services.* The *Interrupt Vector Table (IVT)* is overwritten when an image moves itself to address zero. Thus, BIOS services are not available following the move. Unfortunately, Linux bootstrapping code relies on BIOS services to obtain information needed for configuration. For example, Linux uses the size of available memory to configure the kernel memory address space and a ramdisk (provided *ramdisk* and *initrd* are enabled[†]).
- *Switch from protected mode to real mode.* A downloaded Xinu image always runs in protected mode. However, Linux is designed so the initial bootstrap code runs in *16-bit mode* (i.e., *real mode*). Thus, before a downloaded Linux image runs, the processor must be switched from protected mode to real mode.
- *Absence of a file system.* Xinu can run as an embedded system in which the entire downloaded image is self-contained without a file system. However, Linux usually requires a root file system. For Embedded Linux, the downloaded image typically has *initrd* enabled, which means that Linux must be notified of the starting address and size of the root file system in a ramdisk.
- *Small size and functionally complete file system.* Although back-end computers in the Xinu lab are reasonably large, a version of Embedded Linux must be crafted that is small enough to be booted quickly and sufficiently powerful to be useful in a variety of applications.

A variety of projects have created diskless thin clients [3, 4, 5, 6]. Unfortunately, none of the approaches suffice for the Xinu environment because each relies on BIOS support. As an alternative, several systems use BIOS replacement. That is, the normal BIOS is replaced by a BIOS that can be booted from a cold start [7, 8, 9]. We cannot use such an approach because we cannot change the hardware BIOS.

3. Design Goals

We had three goals for the project:

- *Compatible with current facilities.* Because the lab is heavily used, it was imperative that our solution preserve the current lab software and hardware facilities; we had to adapt Embedded Linux to work in the existing environment.
- *Uniform interface.* A single interface should exist for all lab facilities. In particular, a user should have the ability to download and boot a Xinu image or a Linux image through the same interface.
- *Minimal overhead.* The overhead, including downloading and startup cost should be minimized. The size of the Linux image is of concern because the requisite download time is proportional to the image size.

[†]A ramdisk is especially useful for an embedded system that does not have secondary storage.

4. Downloading Sequence

This section explains the sequence of steps required to run a Linux image on a back-end. There are three basic steps:

1. *Download the complete Linux image.* A Linux image contains the Linux boot code, the kernel, and a root file system. The existing lab facilities download an entire image in one step.
2. *Execute the Linux boot code.* Once a Linux image has been downloaded and moved into a low memory address, control passes to the image, and the Linux boot code takes the responsibility for bootstrapping the kernel. To supply the BIOS services that Linux needs, the BIOS is temporarily provisioned as discussed below.
3. *Execute the Linux root code.* Upon completion of the bootstrapping process, the Linux kernel obtains control of the entire system. The kernel handles initialization and sets up the table of system calls. After initializing the system, the kernel must locate and mount the root file system, which provides a traditional UNIX environment to users.

5. Memory Layout

The downloader used in the lab places an image at memory address *BOOTPLOC* (i.e., 0x15000:0x0000), and the image must relocate itself to absolute address 0x0000. As Table 1 shows, a typical Linux image contains several *segments* of memory: a bootstrapping segment, a setup segment, the kernel, and a file system segment. To accommodate future releases of the Linux kernel, we chose to avoid making changes to the kernel unless absolutely necessary. As a consequence, to adapt Linux to the Xinu lab downloading environment, we added a *Linux Setup Layer (LSL)* that prepares memory as required by the Linux image. In essence, LSL is a modified and extended Xinu image that acts as an intermediary between the original Xinu bootstrap code and the Linux kernel. From the Xinu bootstrap point of view, LSL forms the entry point that must be invoked. LSL can recognize both Xinu code and a Linux image. When it finds a Linux image, LSL arranges memory exactly as the Linux bootstrapping segment expects (i.e., as if the Linux image had been read from disk during a normal computer boot). Thus, the conceptual bootstrapping steps are:

- Use the lab downloader software to load and run the LSL image.
- Download the full Linux image, which is recognized and arranged by the LSL.
- Run the Linux image to create a ramdisk.

To allow LSL to arrange the segments of a Linux kernel and a ramdisk root file system, the set of segments must be combined into a single downloadable image that contains a description of each segment. We added an additional segment named *LOADMAP* (see Table 3) to the beginning of the Linux image file. The LSL uses the *LOADMAP* segment to determine the composition of the image (i.e., the contents and size of each segment). The resulting Linux image is composed of six segments as follows:

- *BOOTSEG.* The *BOOTSEG* segment contains the Linux kernel bootstrapping code.
- *SETUPSEG.* The *SETUPSEG* segment contains code that establishes the Linux kernel (e.g., it queries BIOS services to obtain the size of memory and the video mode, and possibly relocates the Linux kernel, depending on the sizes of the kernel and memory).

- *PARAMSEG*. The PARAMSEG segment contains parameters needed to tune the Linux kernel.
- *LOADMAP*. The LOADMAP segment describes remaining segments by giving the starting offset address from the beginning of the Linux image, the size, and the expected runtime memory address.
- *KERNEL*. The KERNEL segment contains the actual Linux kernel.
- *RAMDISK*. The RAMDISK segment contains an initial file system, which will be mounted as the Linux root file system.

The Linux bootstrap and startup code assumes that the system is always placed in the same memory locations. Thus, to ensure that Linux starts correctly, LSL must arrange memory as Linux expects. As Tables 1 and 2 show, there are two arrangements: one used for a small Linux kernel and one used for a large Linux kernel.

Segment	Type	Starting Address	Description
BOOTSEG	zImage	0x9000	Linux boot code (always 512Bytes)
SETUPSEG	zImage	0x9020	Linux setup code (typical 2K bytes)
KERNEL	zImage	0x1000	Small Linux kernel image
RAMDISK	ramdisk	N/A	Ramdisk contents (alignment to 4096 boundary)

Table 1. Memory layout Linux uses for a small kernel.

Segment	Type	Starting Address	Description
BOOTSEG	bzImage	0x9000	Linux boot code kernel (always 512Bytes)
SETUPSEG	bzImage	0x9020	Linux setup code (typical 2K bytes)
KERNEL	bzImage	0x10000	Large Linux kernel image
RAMDISK	ramdisk	N/A	Ramdisk contents (alignment to 4096 boundary)

Table 2. Memory layout Linux uses for a large kernel.

LSL places the parameter segment and load map in memory directly following *SETUPSEG*. Table 3 lists the actual contents of memory after LSL finishes.

Segment	Type	Starting Address	Description
BOOTSEG	bzImage	0x9000	Linux boot code kernel (always 512Bytes)
SETUPSEG	bzImage	0x9020	Linux setup code (typical 2K bytes)
PARAMSEG	N/A	0x9240	Parameters passed to Linux kernel
LOADMAP	N/A	0x9280	Segment address map
KERNEL	bzImage	0x10000	Large Linux kernel image
RAMDISK	ramdisk	N/A	Ramdisk contents (alignment to 4096 boundary)

Table 3. Actual memory contents when a large kernel begins.

Interestingly, the CPU mode needed to execute code varies among segments. In particular, code in the BOOTSEG segment must be executed in real mode, and code in other segments executes in protected mode. Recall that Xinu code always runs in protected mode. Thus, LSL

needs to switch to real mode before running the BOOTSEG code; BOOTSEG code will switch to protected mode before running the kernel.

6. BIOS Provisioning

When it boots, the Linux kernel uses BIOS services to obtain hardware configuration information (i.e., available memory size). Because the BIOS is overwritten when the image is copied into low memory, LSL must re-provision the BIOS before allowing Linux to start. Table 4 shows the set of BIOS calls that Linux issues.

Interrupt Number	Description Of Service
0x10	VGA services
0x11	Equipment determination
0x13	Disk services
0x15	Miscellaneous services (including memory services)
0x16	Keyboard services

Table 4. BIOS services that Linux uses during startup.

The most critical BIOS call consists of the service that computes the physical memory size. Implementations of Linux try a list of memory services that are each specified as an interrupt number and specific service within the interrupt. The startup code tries each method on the list until one succeeds. If none succeeds, the startup code aborts without starting Linux. Current versions of Linux make the following three calls: *int 0x15, ax=0xe820*; *int 0x15, ax=0xe801*; and *int 0x15, ah=0x88*. To satisfy Linux, our code constructs a memory table named *e820map*, which corresponds to the value returned by the *ax=0xe820* BIOS call. Table *e820map* provides a map of the memory that is available to contain the kernel, root file system, and Linux startup code. We pass the address of the table to Linux as the result of the call. Linux continues to execute startup code; other BIOS services are provided similarly.

7. Root File System

Upon finishing the boot sequence, the Linux kernel attempts to locate and mount a root file system. As noted above, the downloaded image contains a ramdisk, which serves as the root. Once the root has been mounted, additional remote file systems can be mounted using the *Network File System (NFS)*.

The limited memory space (32 MBytes) poses an interesting and challenging problem for the root file system: what should be included and how should it be organized? On one hand, the system should be sufficiently general for many purposes (e.g., a user can choose to use a back-end for packet generation, packet analysis, a firewall, a proxy server, or as a NAT box). On the other hand, adding data files and applications to the root file system results in an image that exceeds the available memory. It may seem that the best solution consists of keeping the root file system small, and mounting a remote file system that contains auxiliary applications and data. However, if the back-end is used for networking research, a production TCP/IP may not be available or the added packet traffic may interfere with the experiment being conducted.

To permit flexibility without depending on NFS, we chose a modular approach. We constructed a set of modules that can be combined with the basic files to produce a root file system. Before a user downloads Linux, the user must specify which of the modules to select. Figure 1 illustrates the set of six optional modules, which are each associated with a general domain or application.

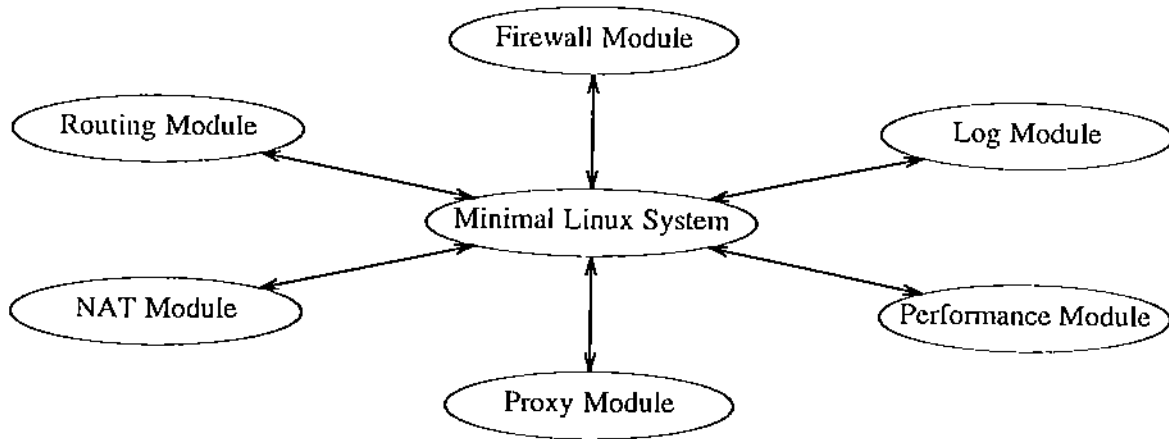


Figure 1. The six optional file system modules available to a user.

The *Firewall Module* contains code that implements an Internet firewall subsystem. It provides stateless or stateful packet filtering. The *NAT Module* handles packet header translation needed for *Network Address Translation*. NAT can be integrated with a firewall or used separately. The *Routing Module* contains code for routing protocols, including RIP, OSPF, and BGP; the code is taken from the zebra routing package. The *Proxy Module* provides the functionality of the SOCKS proxy services. The *Log Module* handles maintenance of a system log that allows postmortem analysis. The *Performance Module* includes the *tcp* application used to measure end-to-end throughput. All of the modules depend on the underlying *Minimal Linux System*, which is required for operation of Linux.

8. Conclusions

We have found Embedded Linux to be a useful tool in the lab. Applications that run under Linux, such as *tcp*, allow programmers to test network protocols without spending time creating tools. More important, Linux can be used as a source or sink for interoperability testing.

The approach we selected to integrate Embedded Linux into the Xinu lab environment has proven to be both practical and efficient. The LSL software allows Linux to be downloaded without changes to either the downloading code or the Linux startup code. Allowing a user to select modules for a root file system provides flexibility, while keeping the resulting image small enough to fit into memory.

Acknowledgments

We thank Gustavo Rodriguez-Rivera and Dongyan Xu for suggestions and insightful discussion.

9. References

- [1] Comer, Douglas, *Hands-On Networking*, Prentice Hall, 2002.
- [2] Comer, Douglas and John C. Lin, A Laboratory Environment For Experimenting With Xinu, *Purdue University CS Technical Report CSD-TR 96-047*, January 1996.

- [3] LTSP website: <http://www.ltsp.org/>
- [4] Kiosk Project website: <http://kiosk.mozdev.org/>
- [5] ELKS website: <http://elks.sourceforge.net>
- [6] DIET-PC website: <http://diet-pc.sourceforge.net>
- [7] LinuxBIOS website: <http://www.acl.lanl.gov/linuxbios>
- [8] FreeBIOS website: <http://freebios.sourceforge.net>
- [9] TinyBIOS website: <http://www.pcengines.com/tinybios.htm>