DOKUZ EYLÜL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

EXECUTING CODES FOR IA-32 (x86) ARCHITECTURE ON JAVA VIRTUAL MACHINES

by
Erdem GÜVEN

January, 2009
İZMİR

EXECUTING CODES FOR IA-32 (x86) ARCHITECTURE ON JAVA VIRTUAL MACHINES

A Thesis Submitted to the
Graduate School of Natural and Applied Sciences of Dokuz Eylül University
In Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

by
Erdem GÜVEN

January, 2009
İZMİR

M.Sc THESIS EXAMINATION RESULT FORM

We have read the thesis entitled **"EXECUTING CODES FOR IA-32 (X86) ARCHITECTURE ON JAVA VIRTUAL MACHINES"** completed by **ERDEM GÜVEN** under supervision of **ASST. PROF. DR. ŞEN ÇAKIR** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

....

Asst. Prof. Dr. Şen ÇAKIR

Supervisor

....

Asst. Prof. Dr. Güleser K. DEMIR

(Jury Member)

....

Instructor Dr. Canan ATAY

(Jury Member)

Prof. Dr. Cahit HELVACI

Director

Graduate School of Natural and Applied Sciences

ACKNOWLEDGMENTS

I would like to thank to my supervisor, Asst. Prof. Dr. Şen ÇAKIR for her guidance and assistance in this thesis.

I also want to thank to my family for their endless support and encouragements.

Erdem GÜVEN

EXECUTING CODES FOR IA-32 (x86) ARCHITECTURE ON JAVA VIRTUAL MACHINES

ABSTRACT

Today, the IA-32 architecture is ubiquitous among PCs. Much more softwares have been developed for this architecture than for any of others. To benefit from these softwares on the platforms which use other architectures, the softwares must be ported to that platforms or must be redeveloped.

A Java Virtual Machine (JVM), is a set of computer software programs that enables running Java applications. JVM isn't specific to any processor or operating system. It's implemented for various hardware and operating systems so that Java programs can run identically on all of them.

Being able to execute IA-32 specific softwares on JVMs, makes those softwares available for platforms that a JVM is implemented for. This is an easy way to make use of IA-32 softwares on other platforms.

There are many methods to run an IA-32 application on a JVM: the application can be redeveloped for Java, the source code of the application can be converted by automatic tools, the application binary can be executed on an emulator and others. Regardless of which method is used, software services that are needed by the application in question must be somehow supplied. Software services are provided by other softwares. These softwares can be ported to the Java platform with the methods that are mentioned before or the needed services can be supplied by using existent Java services.

Generally most of the needed software services are provided by operating systems (OS). It's hard and very resource consuming to execute a complete IA-32 OS on a Java platform. It affects all of the system performance while running most of the

IA-32 applications. To get rid of this bottleneck, OS services may be provided by a Java software.

In this study, a Linux compatibility layer for Java was developed. This layer provides Linux operating system services under Java platform. It was shown that the Linux compatibility layer facilitates executing codes for IA-32 architecture on Java platforms.

Keywords: IA-32, x86, Linux, Software, Compatibility Layer, Java, Platform Independence

IA-32 (x86) MİMARİSİ İÇİN GELİŞTİRİLMİŞ KODLARIN JAVA SANAL MAKİNASINDA KOŞTURULMASI

ÖZ

Günümüzde IA-32 işlemci mimarisi kisisel bilgisayarlarda yaygın olarak kullanılmaktadır. Bu mimari için diğer mimarilerden daha çok yazılım geliştirilmiştir. Bu yazılımlardan diğer mimarileri kullanan platformlarda da faydalanabilmek için bu yazılımların bu platformlara taşınması yada sıfırdan tekrar geliştirilmesi gerekmektedir.

Java Sanal Makinası (JVM), Java uygulamalarını çalıştırmak için gerekli olan uygulamalar bütünüdür. JVM herhangi bir işlemciye yada işletim sistemine bağımlı değildir. Birçok donanıma ve işletim sistemine taşınmıştır; böylece Java uygulamaları bu platformlarda aynı şekilde çalışabilmektedir.

IA-32'ye özel yazılımların JVM üzerinde çalıştırılabilmesi, bu yazılımları JVM bulunan bütün platformlarda çalışır kılar. Bu IA-32 yazılımlarını diğer platformlarda kullanmak için kolay bir yoldur.

Bir IA-32 uygulamasını bir JVM üzerinde çalıştırmak için bir çok yol bulunmaktadır. Uygulama Java için yeniden geliştirilebilir. Uygulamanın kaynak kodu otomatik araçlarla Java koduna dönüştürülebilir. Uygulama bir emulator üstünde koşturulabilir. Bunlar en çok kullanılan bir kaç yöntem. Hangi yöntem kullanılırsa kullanılsın, uygulama tarafından gerek duyulan yazılım servislerinin sağlanması gerekmektedir. Yazılım servisleri başka yazılım parçaları tarafından sağlanmaktadır. Bu servis yazılımları da üste bahsedilen yöntemlerle Java platformuna taşınabilir yada bu servisleri sağlayacak başka yapılar kurulabilir.

Genellikle ihtiyaç duyulan yazılım servisleri işletim sistemi tarafından sağlanmaktadır (IS). Eksiksiz bir IA-32 IS'yi Java platformunda çalıştırmak çok iş ve kaynak gerektiren bir iştir. Bunun sonucunda uygulamanın Java platformundaki

performansını olumsuz yönde etkiler. Bu etkiden kurtulmak için IS servislerini Java yazılımları ile sağlamak tercih edilebilir.

Bu çalışmada, bir Java için Linux uyumluluk katmanı geliştirilmiştir. Bu katman Linux işletim sistemi servislerini Java platformu altında sağlamaktadır. Linux uyumluluk katmanının, IA-32 mimarisi için geliştirilmiş kodların Java platformunda çalıştırılmasını kolaylaştırdığı gösterilmiştir.

Anahtar Kelimeler : IA-32, x86, Linux, Yazılım, Uyumluluk Katmanı, Java, Platform Bağımsızlığı

CONTENTS **Page**

M.Sc THESIS EXAMINATION RESULT FORM..ii
ACKNOWLEDGEMENTS...iii
ABSTRACT...iv
ÖZ...v

CHAPTER ONE - INTRODUCTION...1

1.1 Introduction...1
1.2 Linux Applications on Java Platform..2
1.2.1 Rewriting the Software in Java...2
1.2.2 Porting the Software to the Java Platform......................................4
1.2.3 Running Software and Linux on an Emulator..................................5
1.3 Structure of the Thesis...7

CHAPTER TWO - JINUX...8

2.1 Jinux...8
2.2 Java Jinux Applications...8
2.3 Jinux Native Applications..9
2.4 Emulator Integration...10

CHAPTER THREE - LINUX COMPATIBILITY LAYER FOR JAVA..........13

3.1 Compatibility Layer..13
3.2 Linux Compatibility Layer for Java...13
3.3 System Calls..14
3.3.1 Typical Implementations...15
3.3.2 lcl4j Implementation..16
3.4 Processes..16
3.5 File System...16
3.5.1 Case Sensitivity, File Permissions and Ownership..........................18

3.6 Peripheral devices 18

CHAPTER FOUR - JPC INTEGRATOR..19

4.1 Introduction...19
4.2 Initialization...19
4.3 Virtual Memory Management...20
4.4 Loading Linux Binaries..22
4.4.1 ELF Loader...22
4.4.2 Environment Variables..23
4.4.3 Auxiliary Vector...23
4.4.4 Stack Initialization..24

CHAPTER FIVE - RESULTS...25

5.1 Introduction...25
5.2 gzip Test...25
5.3 md5sum Test...26
5.4 All Results...26

CHAPTER SIX - CONCLUSION & FUTURE WORK.......................................27

REFERENCES...28

APPENDIX A - CODES...30

A.1 org.jinux.Jinux.java...30
A.2 org.jinux.jpc.memory.MemoryManager.java...35
A.3 org.jinux.jpc.memory.DescriptorTable.java..42
A.4 org.jinux.jpc.memory.PageDirectory.java..45
A.5 org.jinux.fs.JnxFileImp.java...49
A.6 org.jinux.fs.Null.java...51
A.7 org.jinux.fs.Pipe.java...51

CHAPTER ONE
INTRODUCTION

1.1 Introduction

IA-32 (x86) is a 32-bit instruction set architecture. It's first implemented in Intel 80386. It is an extension to the earlier 16-bit Intel 8086, 80186 and 80286 processors. It is used on microprocessors that installed in the vast majority of personal computers in the world.

Linux is one of the most popular operating systems that run on IA-32 architecture. It is a Unix-like computer operating system family which uses the Linux kernel. Linux is one of the most prominent examples of free software and open source development. Typically all the underlying source code can be freely modified, used, and redistributed by anyone. It is installed on a wide variety of computer hardware, ranging from embedded devices and mobile phones to supercomputers.

There are huge number of softwares which are designed to work on Linux. These softwares depends on the Linux OS services. To run these applications on any platform, required services must be provided (Cygwin, nd).

Java refers to a series of computer software products and specifications from Sun Microsystems, which together provide a system for the development of application software and use in a cross-platform environment. Java is used in a wide variety of computing platforms.

The Java platform is the name for a bundle of related programs which allows for developing and running programs written in the Java programming language. The platform is not specific to any processor or operating system. Its execution engine (called a virtual machine) and compiler with a set of standard libraries are implemented for various hardware and operating systems so that Java programs can

run identically on all of them. It can be thought as another architecture which provides libraries.

Standard Java libraries depend on host system for operating system functionality. There have been project to create stand-alone Java operating systems (JavaOS, nd; Jnode, nd; Golm, Felser, Wawersich & Kleinöder, 2002).

Executing Linux applications on Java Platforms has many advantages. Java is already ported to many architectures. An application which is able to run on Java platform can be used on all of these architectures. All the hard work that have done to develop those software and to make them mature, would not be needed to redone.

In this thesis, Jinux is presented. Jinux consists of a Linux compatibility layer for Java and necessary tools to cooperate with present tools. It facilitates porting Linux code to Java Platform and running Linux software on emulators without Linux.

1.2 Linux Applications on Java Platform

Running Linux applications on Java Platform isn't a trivial task and it requires a lot of effort. There are some ways to accomplish this task:

- Rewriting the software in Java.
- Porting the software to Java platform.
- Running software and Linux on an emulator.

All of these methods have to deal with providing necessary Linux services to the resulting processes. This is a common bottleneck in executing Linux applications on Java platforms. Linux, as a whole, is a complex software and it is very importing to find an efficient Java equivalent.

1.2.1 Rewriting the Software in Java

First option is redeveloping the software from the ground up for Java platforms (Figure 1.1). This process requires nearly same effort that have been done to develop

original software. It is a hard and long work even if we have the source code of the software.

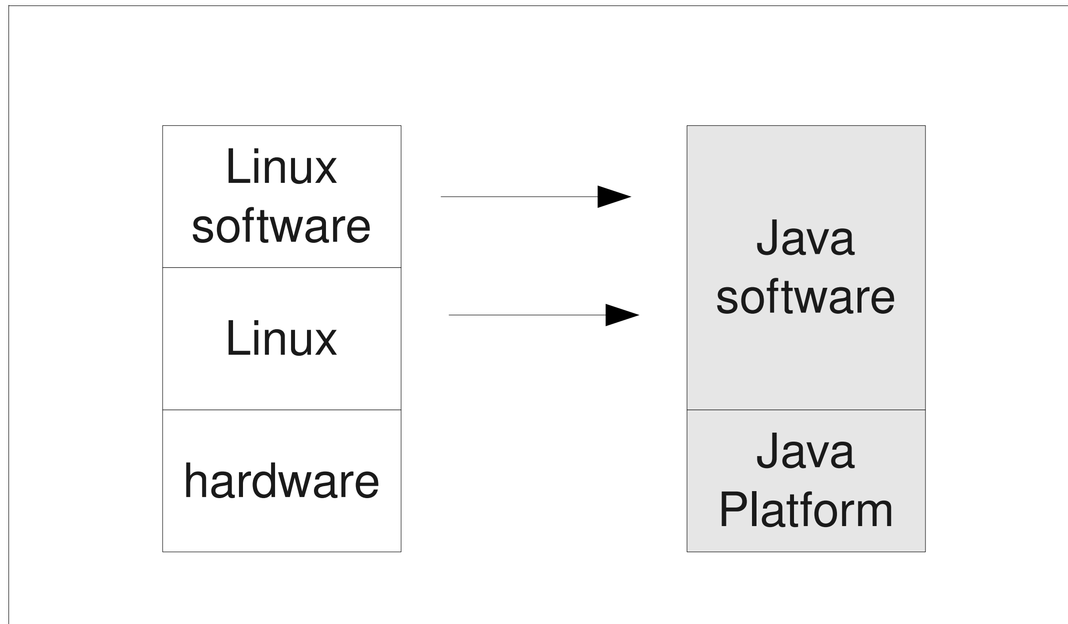

Figure 1.1 Rewriting the software for Java.

Linux applications are generally developed in C or C++. These programming languages have a similar syntax with the Java language but also have important differences such as memory usage. There are some tools to facilitate convertion of C/C++ code to Java code (Jazillian, nd; Malabarba, Devanbu & Stearns, 1999).

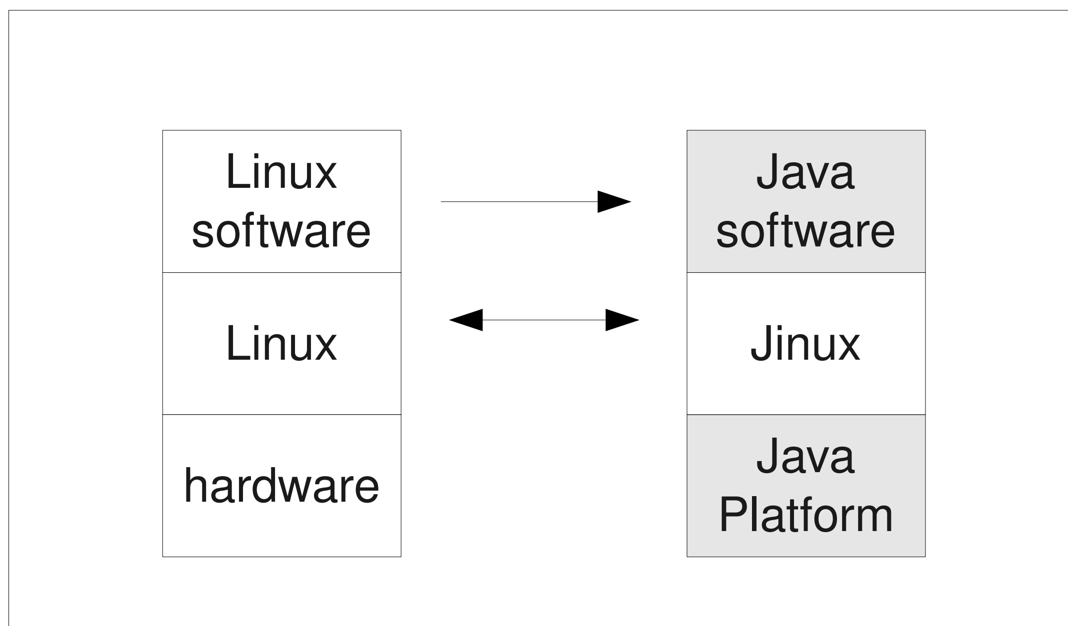

Figure 1.2 How to take advantage of Jinux.

Using Jinux, reduces the work needed to rewrite the software (Figure 1.2).

1.2.2 Porting the Software to the Java Platform

Porting process adapts a software to create an executable program for a computing environment that is different from the one which it was originally designed for. If the cost of porting software to a new platform is less than the cost of writing it from scratch, software is referred as portable.

There are C to Java binary compilers which compile C codes and output Java binaries. So without converting C code to Java code, necessary porting can be done. Linux oriented requirements can not be supported by these compilers. Input source code must be modified to get rid of these dependencies. This process is error prone and likely to cause performance lost. Resulting binary must be tested completely (Figure 1.3).

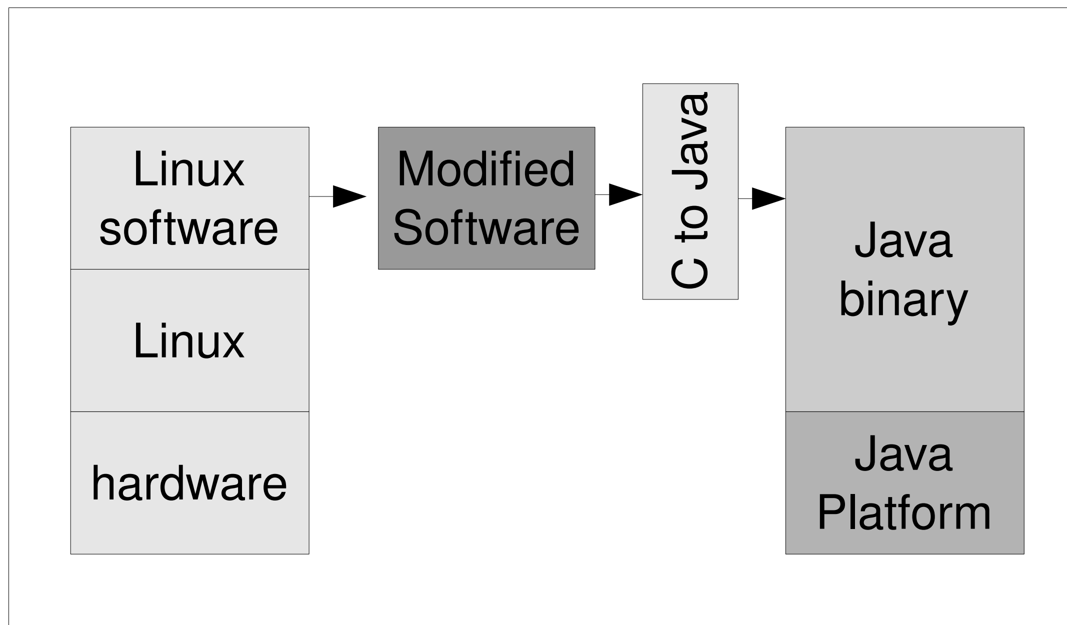

Figure 1.3 C to Java binary.

If the Jinux is used, there will be no need for modification (Figure 1.4). All of the platform depended requirements of application will be provided by Jinux.

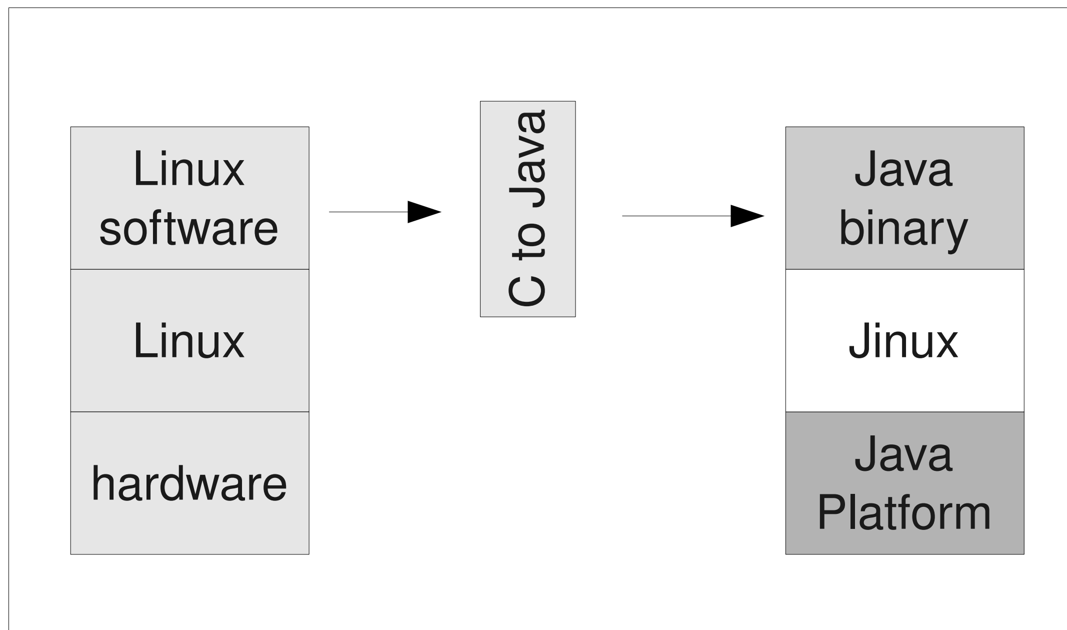

Figure 1.4 No need to modify with Jinux.

1.2.3 Running Software and Linux on an Emulator

Binary form of the software and Linux for some architecture can be executed on an emulator which is developed to emulate that architecture on Java platform (Figure 1.5). Neither modification to the software is needed nor to the Linux. This is the most reliable, feasible and fast of all these three methods but execution of the result is the slowest and the most resource consuming.

There are two projects which use this method:

- Pearcolator (Pearcolator, nd).
- JPC (JPC - Computer Virtualization in Java, nd).

Pearcolator is an open source dynamic binary translator written in Java. It has an incomplete x86 emulator. In Jinux project, some codes are based on the Pearcolator.

JPC is a pure Java emulation of an x86 PC with fully virtual peripherals. It can run a whole Linux installation. In Chapter 4, JPC Integrator tool is described which enables the cooperation between JPC and Jinux (Figure 1.6).

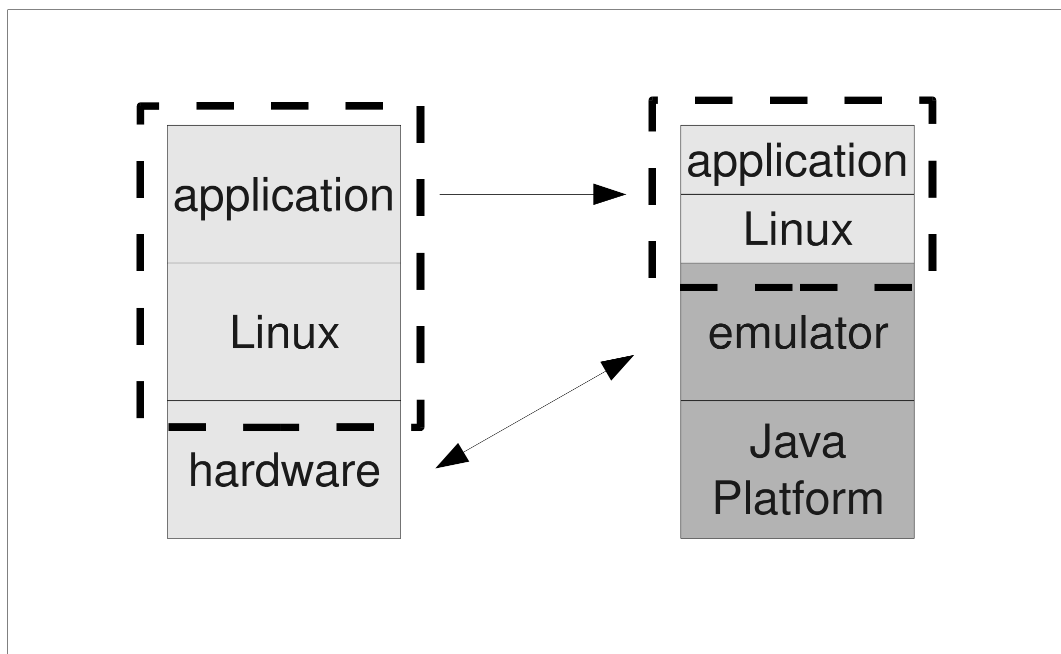

Figure 1.5 Emulation

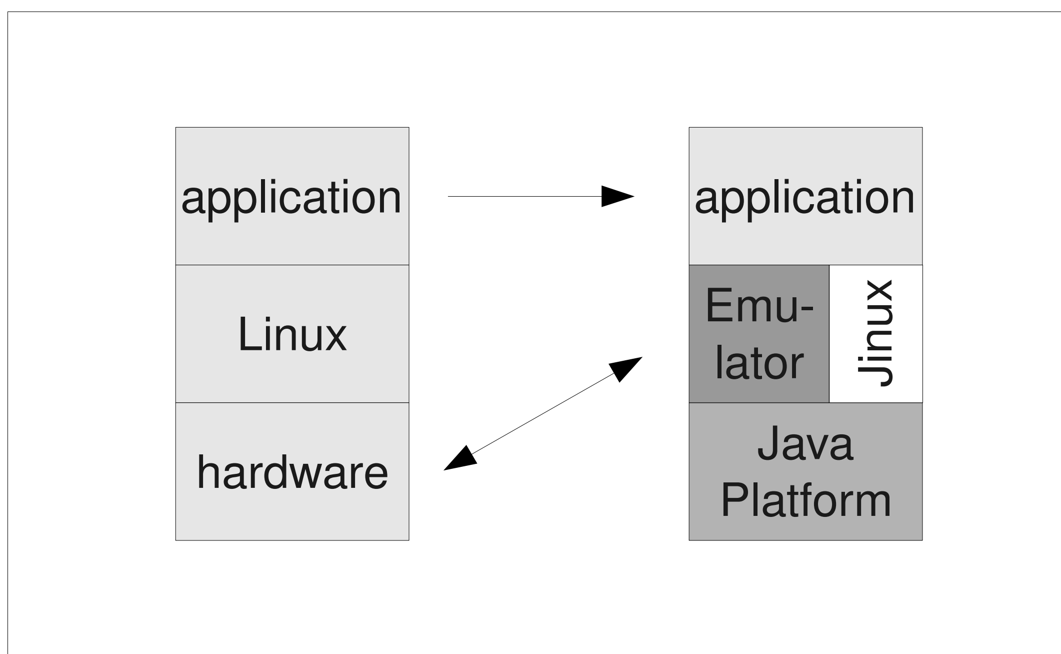

Figure 1.6 Emulation with Jinux.

1.3 Structure of the Thesis

The contents of this thesis are constructed according to this outline:

In Chapter 2, Jinux is introduced and its usage methods are examined.

In Chapter 3, Linux Compatibility Layer for Java and its sub-modules are described.

In Chapter 4, JPC integrator is introduced and it sub-modules are examined.

Chapter 5 is conclusion.

CHAPTER TWO
JINUX

2.1 Jinux

Jinux consists of Linux Compatibility Layer for Java (lcl4j) and helper modules. It is a pure Java application. No Java Native Interface (JNI) method is used so it can run on any platform which a Java virtual machine exists.

There are mainly three different cases to use Jinux:

- Java Jinux applications
- Native Linux applications
- Emulator integration

In the first case, applications that are prepared, Jinux in the mind, or converted from a real Linux application source code are examined.

Second case is for native applications that can be executed on host cpu directly.

In the third case, a general and easy solution for running Linux application without recompilation is concentrated.

2.2 Java Jinux Applications

Java applications that interact with Jinux and use its services can be developed.

Some Java Jinux application examples:

- Jinux utilities:

  These applications are developed to manage Jinux environment. There is Shell Java application inside Jinux which operates as a shell for Jinux (Figure 2.1).

- Applications that need to interact with other Jinux clients:

Jinux supports interprocess communication. Applications that use Jinux, can communicate through pipes or system calls (msgget, msgsnd).

A pipe is a method of connecting the standard output of one process to the standard input of another.

- Applications that are converted to Java applications or ported to Java Platform:

In this cases Jinux eliminates the need of finding equivalents of system calls that are used in the original source. This kind of Java applications can be generated from original Linux application by automatic C to Java converters or C to Java binary compilers.

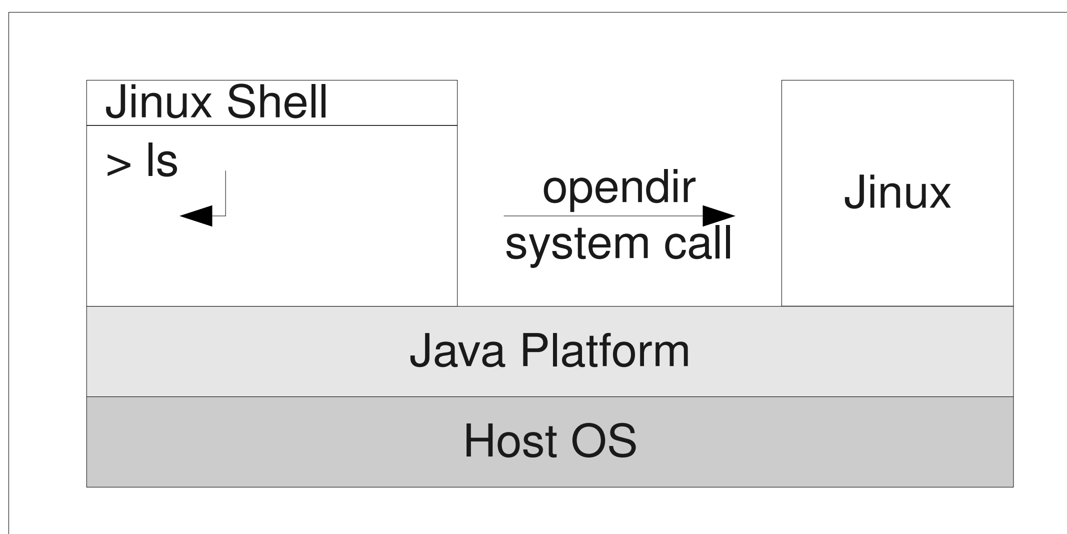

Figure 2.1 Jinux Java Application (Shell.java)

2.3 Jinux Native Applications

Linux application softwares are compiled for some architecture. Generated executables can be run on non-Linux systems with same architecture with the help of Jinux and a system call forwarder (SCF). A SCF intercepts any Linux system calls that was attempted by the native software and forwards them to the Jinux.

A SCF for Windows operating system is planned to be developed.

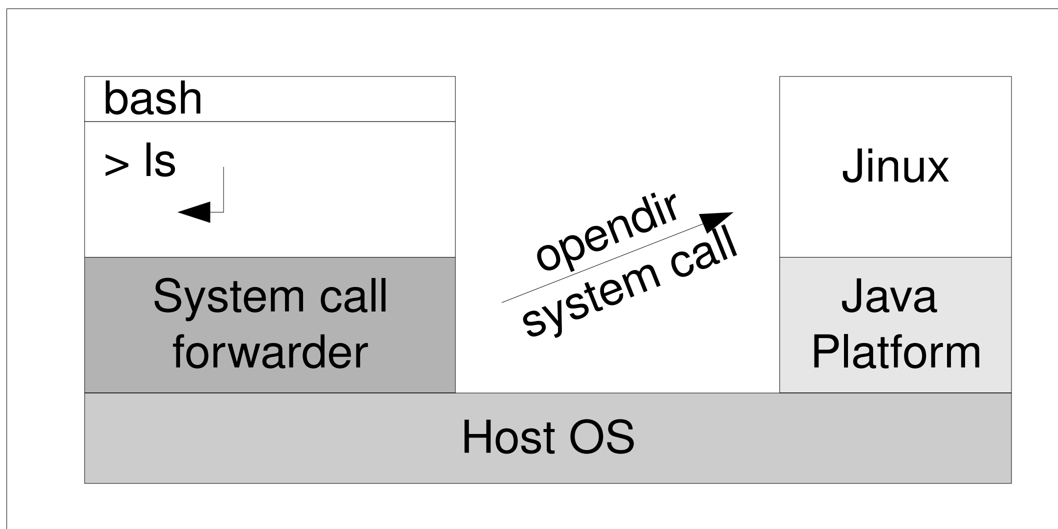

Figure 2.2 Jinux native application.

2.4 Emulator Integration

An emulator duplicates the functions of one system using a different system, so that the second system behaves like the first one.

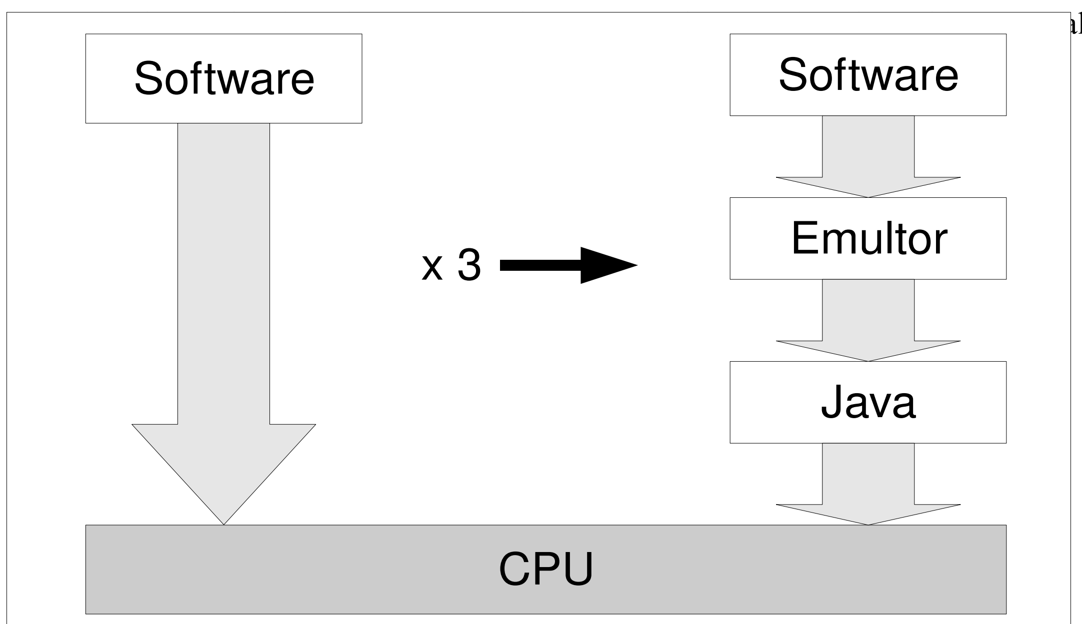

Figure 2.3 Emulator in Java.

A complex system like Linux is even hard to emulate in this nested emulation. Jinux saves emulators from dealing with Linux emulation. All of the needed services by Linux application can be provided by Jinux. This way emulators can archive reasonable speeds.

Architecture specific issues must be handled with emulator integrator. Typically, binary loading must be handled with a separate utility (Figure 2.4).

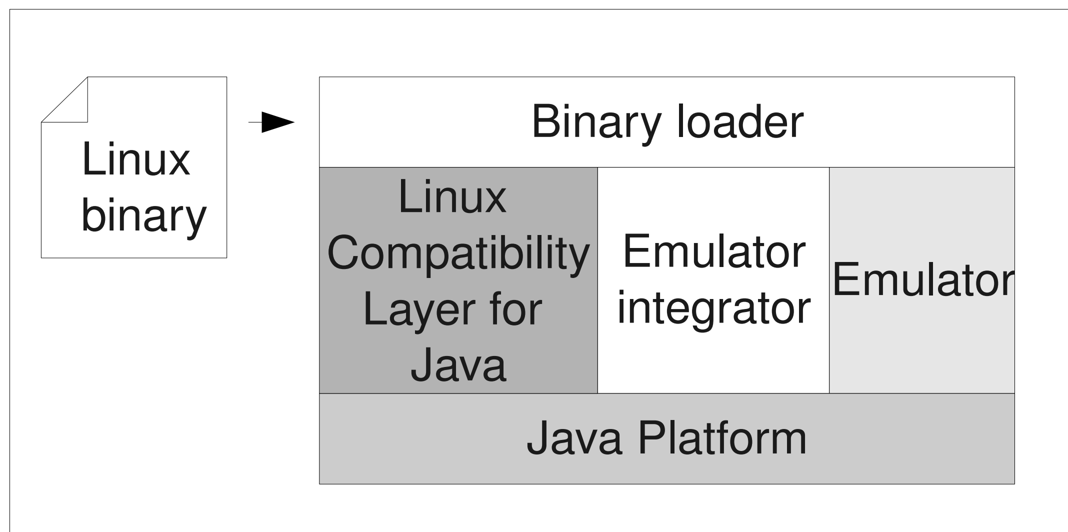

Figure 2.4 Emulator integration

CHAPTER THREE
LINUX COMPATIBILITY LAYER FOR JAVA

3.1 Compatibility Layer

A compatibility layer allows binaries of a different system to run on a host system. It translates system calls for the other system into native system calls for the host. Porting some libraries of the foreign system to host system, will often be sufficient to run the binaries on the host system.

Some compatibility layer examples:

- The Linux compatibility layer on the BSDs, which enables binaries built specifically for Linux to run on BSD (Handy & Murphey, nd).
- Wine, which runs some Microsoft Windows binaries on Unix-like systems using a program loader and the Windows API implemented in DLLs.
- Windows XP's use of compatibility layers to attempt to better run Windows 98 and MS-DOS applications (Russel, 2002).

A compatibility layer avoids both the speed penalty and the complication of full hardware emulation. Some programs may even run faster than the original.

A compatibility layer requires the host system's CPU to be compatible to that of the foreign system. Thus, for example, an MS Windows compatibility layer is not possible on PowerPC hardware, since MS Windows requires an x86 CPU; in that case, full emulation is needed.

3.2 Linux Compatibility Layer for Java

Linux Compatibility Layer for Java (lcl4j) tries to mimic Linux OS view to a Jinux client. It supports Linux system calls, processes, file system and peripheral devices.

Jinux is planed be used as a server and a library. Server mode isn't developed yet. In server mode it will able to communicate with clients through network sockets. In library mode Jinux library dynamically linked with client application in load phase.

In development, general approach was to make Jinux thin as possible as it could be. This means, lcl4j does minimum that is enough to be compatible with Linux. Linux services is provided by standard Java libraries.

Source code is mainly based on x86 extension to the Pearcolator project (Burcham, 2005). Pearcolator is an emulator written in Java. It supports PowerPC and x86 processors.

3.3 System Calls

A system call is a request from a program to the operating system for the implementation of a predefined tasks that the program does not have the necessary privileges to perform in its own flow of execution. The interface between a process and the operating system is provided by system calls. Most operations interacting with the system require permissions that don't available to a user level process, i.e. any I/O performed with any arbitrary device present on the system or any form of communication with other processes requires the use of system calls.

Linux has 319 different system calls. Currently, lcl4j implements 19 of the most important ones: sys_exit, sys_read, sys_write, sys_writev, sys_open, sys_close, sys_geteuid, sys_getuid, sys_getegid, sys_getgid, sys_brk, sys_fstat64, sys_stat64, sys_fcntl64, sys_uname, sys_mmap, sys_mremap, sys_munmap, sys_exitgroup.

Some of the calls are emulated by translating system call to a Java method invocation (i.e. sys_read) and some of others just returns some internal value (i.e. sys_getuid). Others are handled by modules in charge (i.e. sys_exit) (Table 3.1).

Table 3.1 System call handling examples.

System call:	Jinux response:
sys_read(...)	java.io.RandomAccessFile file = ...; file.read(...);
sys_getuid(...)	return PREDEFINED_UID;
sys_exit();	current_process.exit();

3.3.1 Typical Implementations

Implementing system calls requires a control transfer between application and OS. This involves some sort of architecture specific feature. A typical way is to use a software interrupt or trap. Interrupts transfer control to the kernel. So software simply needs to set up some register with the system call number and some others with system call parameters. After then it executes the software interrupt.

On x86, there are 256 software interrupts. Linux just uses 0x80 interrupt for system calls. The system call number is stored in register EAX. If the number of arguments is five or less, the arguments in order are passed by EBX, ECX, EDX, ESI, and EDI (Table 3.1). If the number of arguments is greater than five, EBX contains a pointer to the list of arguments.

Table 3.2 Some of the system calls and their parameter assignments.

eax	**Name**	**ebx**	**ecx**	**edx**	**esx**	**edi**
1	sys_exit	int	-	-	-	-
2	sys_fork	struct pt_regs	-	-	-	-
3	sys_read	unsigned int	char *	size_t	-	-
4	sys_write	unsigned int	const char *	size_t	-	-
5	sys_open	const char *	int	int	-	-
6	sys_close	unsigned int	-	-	-	-
7	sys_waitpid	pid_t	unsigned int *	int	-	-
8	sys_creat	const char *	int	-	-	-

3.3.2 lcl4j Implementation

In lcl4j, LinuxSystemCalls class is responsible of handling system calls. It have SystemCall typed sub-class for each supported system call. When a system call received, LinuxSystemCalls instance calls corresponding object's doSysCall() method. There is one additional SystemCall instance which is UnknownSystemCall. It is for unsupported/unknown system calls.

3.4 Processes

In Jinux, processes are used to define client sessions. Client specific information is managed through processes. Every Jinux client is assigned with a process.

A client must obtain a process to start using Jinux services. Processes are created by a sys_fork system call so a process can be created only by another process as in Linux. There is an initial process which is named 'init'. All other processes are forked from it.

Jinux doesn't make any kind of scheduling between processes for now. A priority order between system request may be implemented.

3.5 File System

Linux provides a standard file structure which has only one root ('/'). All other files and devices are children of this root.

Because of many Linux applications depend on this standard file structure, Jinux provides a virtual file system. It maps a specified directory in host file system, to the virtual root (Figure 3.1). Contents of this directory can be accessed and modified by the host.

There are also non-mapped virtual files and directories like: '/proc', '/dev/null'. These files doesn't exist in the host file system. File access operations to this files are answered by specific classes. For example, the null device is a special file that

discards all data written to it (but reports that the write operation succeeded), and provides no data to any process that reads from it (it returns EOF). The org.jinux.fs.Null class emulates this behavior. All file operations to /dev/null file are forwarded to this class.

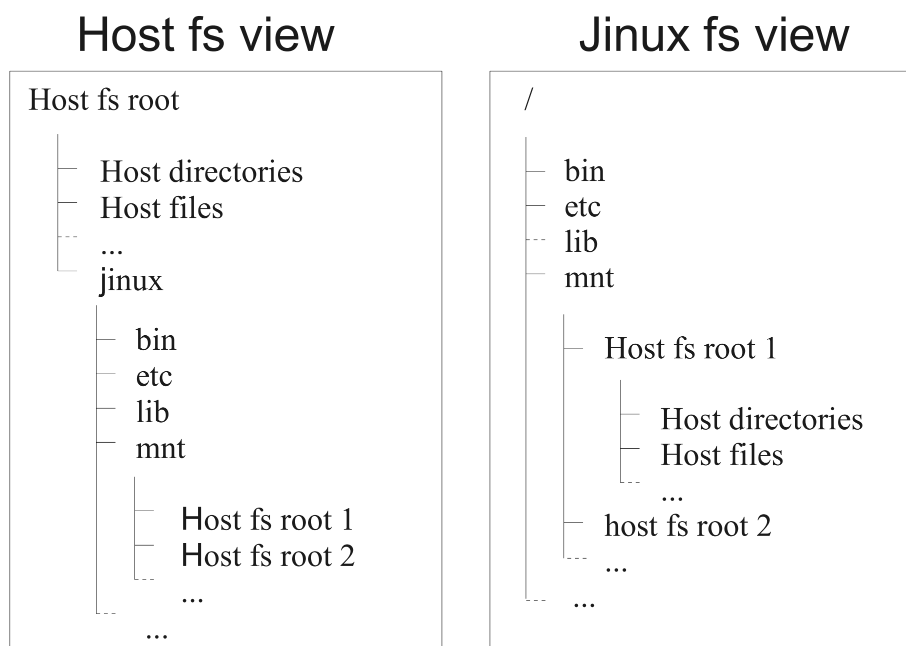

Figure 3.1 File mapping.

Host file system roots are mounted to some folders under /mnt folder which is generally used for mounting.

Linux uses Posix style paths. In Java, path style changes according to host. Mapped files' paths must be converted while accessing original files on the host (Table 3.2).

Table 3.3 Path convertion example

	Jinux path	Windows Host path	Linux Host path
Root dir.	/	c:\jinux	/jinux
Path	/root/test/test1	c:\jinux\root\test\test1	/jinux/root/test/test1

3.5.1 Case Sensitivity, File Permissions and Ownership

Linux applications are generally developed considering a case sensitive, permission and ownership assignable file system but this isn't the default case in Java. Since Java file operations depends on the host system; an assumption about these file system properties, can not be made. So case sensitivity depends on the host file system.

A general solution for this case is using a loop device. A loop device is a pseudo-device that makes a file accessible as a block device. Any file system can be installed on these devices. In this case, a case sensitive file system can be used on this pseudo device. With the use of this solution; ability of host system access to Jinux files is complicated.

Another method that was used on UMSDOS file system is storing file properties for directory entries, like mode and owner, in a special file. Disadvantage of this method is the file that holds additional info becomes bigger in time. It becomes slower to manage this file and to access needed info (UMSDOS, nd).

Currently lcl4j doesn't support ownership and case sensitivity. File read/write permissions are gathered from java.io.File class.

3.6 Peripheral devices

In Linux, general way to access peripheral devices is to use pseudo files that reside in the '/dev' folder.

Jinux doesn't emulate any devices. It just let Jinux applications access devices through the file system and system calls. It creates virtual files for supported devices. Application access to these devices through these files.

For now just serial port is supported but any device such as audio card, hard disk or usb devices can be supported.

CHAPTER FOUR
JPC INTEGRATOR

4.1 Introduction

JPC is a pure Java emulation of an x86 PC with fully virtual peripherals. It can emulate a whole Linux system with the kernel, drivers and applications.

JPC uses a number of optimization strategies to achieve an acceptable speed. It runs up to 20% of native speed (JPC - Technology Overview, nd).

JPC Integrator (JPCI) is a submodule of Jinux. It is developed to make use of JPC to run x86 Linux binaries on pure Java environments with the help of Jinux. Jinux and JPCI combination take the place of a real Linux system in the emulation. All the services that application needs, are provided by these. JPCI is responsible of boot process, memory management and application loading.

As JPC is developed full emulation in mind; it has many parts that emulates a part of a real x86 architecture. This parts are accessed by emulated software during emulation. They change their states due to other parts' states. During boot process, Linux make many settings to prepare cpu and other peripherals to work as needed. In the absence of Linux, this setup is done by JPCI.

JPCI currently can handle only one application at a time. It is planned to develop multiple application support.

4.2 Initialization

During initializing JPC, some configuration must be done to prepare JPC for Linux applications execution. This can be thought as simplified form of a Linux boot process. Most important part of configuration is initializing protected mode.

Protected mode, is an operational mode of x86 compatible central processing units. Protected mode allows system software to utilize features such as virtual memory, memory protection, safe multi-tasking, and other features designed to increase an operating system's control over application software.

Memory protection is a way for controlling memory usage on a computer, and it is very important for operating systems. The main purpose of the memory protection is to prevent a process from accessing memory other than allocated to it. This prevents any bug within the process's software from affecting other processes, and also prevents malicious software from gaining unauthorized access to the system.

In protected mode, segmentation is used to achieve memory protection. Segmentation refers to dividing a computer's memory into segments. Each segment has its own base address, range, type (code,data...) and write or execution permission. All memory access made by one of defined segments. Out of segment range access or a non permitted operations causes a hardware exception to be delivered. Segment descriptors are stored in a GDT (global descriptor table) and can be also stored in an LDT (local descriptor table).

Linux uses 6 segments. All of them addresses same base address (0x00000000) and have same range (0xFFFFFFFF) for simplicity reasons.

Two of the segments are user code and data segments. These segment are enough for execution of a single Linux application. JPCI configures just these segments.

In JPCI initializing process, MemoryManager class initializes the GDT by DescriptorTable class. It then configures JPC to switch to protected mode.

4.3 Virtual Memory Management

Virtual memory is a computer system technique which gives an application program the impression that it has contiguous working memory, while in fact it may

be physically fragmented and may even overflow on to disk storage (Figure 4.1). Systems that use this technique make programming of large applications easier and use real physical memory (e. g. RAM) more efficiently than those without virtual memory.

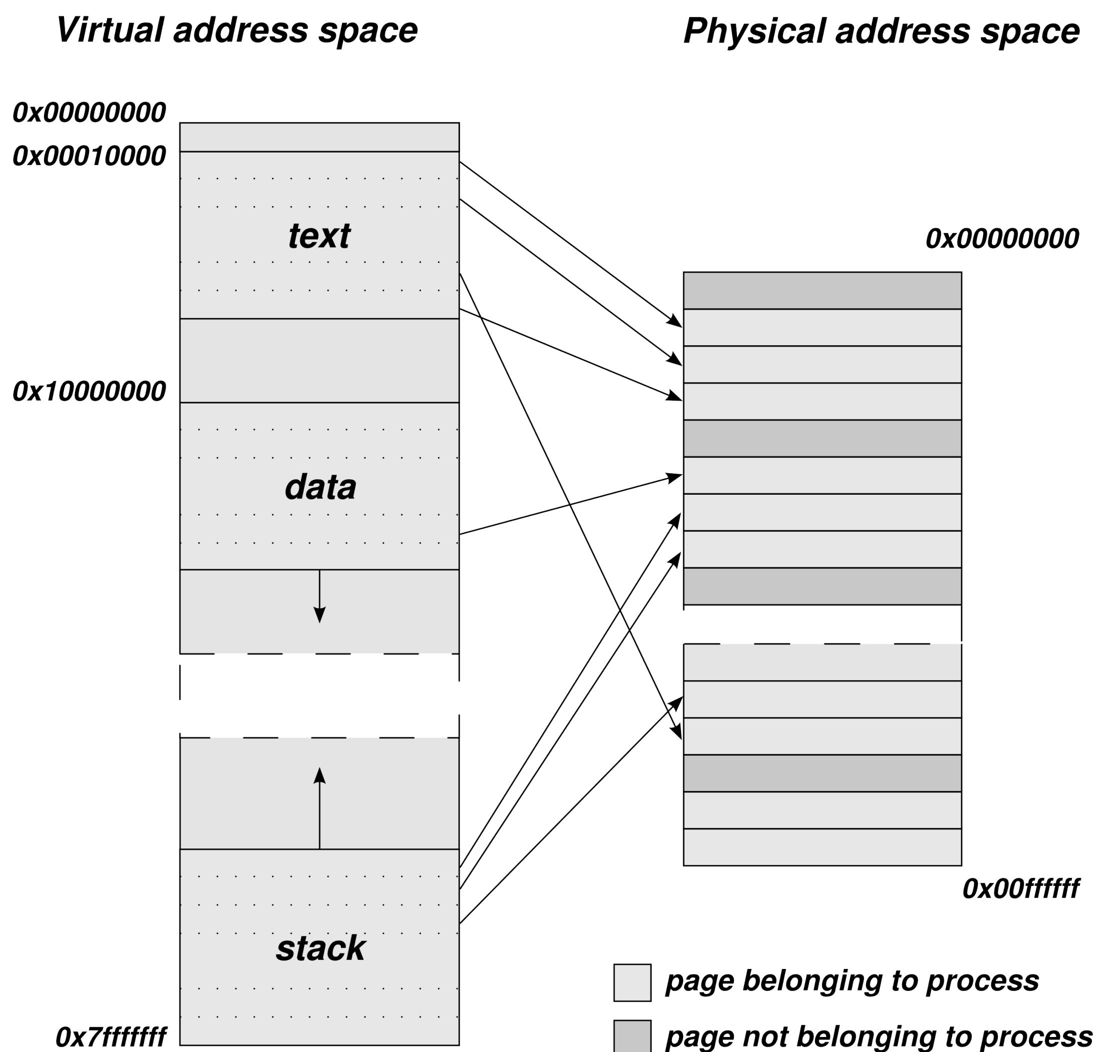

Figure 4.1 Address mapping.

Virtual memory depends on mapping arbitrary virtual addresses to arbitrary physical memory addresses or to nothing. Memory pages are the basic units of this mapping. Pages are blocks of contiguous memory addresses. They are usually at least 4K bytes in size, and systems with large virtual address ranges or large amounts of real memory generally use larger page sizes.

A page table is a look up table for memory pages. It is used by a virtual memory system in an OS to store the mapping between virtual addresses and physical addresses.

MemoryManager class handles memory mapping and virtual memory management with the help of PageManager class.

4.4 Loading Linux Binaries

Executable binaries must be loaded to memory prior to execution. Loader is a special utility that uses relocation information inside binary format to place code and data blocks to correct places. There are different loaders for each of binary formats in Linux.

JPC has no support for executable binary loading so JPCI includes an ELF loader.

4.4.1 ELF Loader

Executable and Linking Format (ELF) is a common standard file format for executables, object codes, shared libraries, and core dumps. It is the standard binary file format for Linux systems on x86. ELF format includes dynamic linking information.

Dynamic linking is the process of linking executable and libraries just before the every execution rather than statically linking them once. This allows sharing libraries between applications and results in reduced binary size. Otherwise every application must include its own instance of library. Generally Linux applications uses many shared libraries.

Elf_Loader class handles loading of ELF binaries to JPC. Main source code of this class is evolved from Pearcolator code. Dynamic linking ability for x86 architecture is added by JPCI.

4.4.2 Environment Variables

Environment variables are a set of dynamic values that can affect the way running processes will behave on a computer.

In Linux systems, each process has its own private set of environment variables. By default, when a process is created it inherits a duplicate environment of its parent process, except for explicit changes made by the parent when it creates the child.

In Linux, the environment variables are normally initialized during system startup by the system init scripts. Jinux uses host environment variables for initial values. Variables that are defined in configuration file, are overrides hosts'. Variables can change dynamically during execution.

4.4.3 Auxiliary Vector

The auxiliary vector conveys information from the operating system to the application. To terminate vector, null auxiliary vector entry is required. This vector is an array of the following structures.

```
typedef struct
{
   long int a_type; /* Entry type */
   union
   {
        long int a_val; /* Integer value */
        void *a_ptr; /* Pointer value */
        void (*a_fcn) (void); /* Function pointer
                                 value */
   } a_un;
} auxv_t;
```

The application shall interpret the a_un value according to the a_type. Other auxiliary vector types are reserved.

JPCI uses static values to initialize auxiliary vector.

4.4.4 Stack Initialization

At startup of an application, stack is filled with everything needed to access environment variables, command line arguments and the aux vector.

LinuxStackInitizer class prepares stack prior to program emulation start.

CHAPTER FIVE
RESULTS

5.1 Introduction

To show the effect of Jinux, many measurements have been done on test cases. Main purpose of Jinux project is to run Linux applications on Java platform with less resources so testes are done with real applications. Some common utility applications have been used in non-interactive mode to compare execution times on Jinux and full emulation cases.

All of the tests have been done on a computer that has 1.8GHz AMD Turion64 cpu (executed in 32-bit mode) and 1GB ram. Ubuntu 8.04 OS was used.

In full emulation mode 'Qemu Linux test distribution', which is based on Redhat 9, was used. It is a very light and reduced distribution.

Other then the length of application execution time, length of initialization time is also important for the end user. In full emulation case, spend time includes whole Linux boot up time (Table 5.1).

Table 5.1 Initialization times.

	Jinux	Full emulation
Initialization time	0.976s	80.700s

5.2 gzip Test

gzip is a well known, open source command line data stream compressor and archiver. gzip is short for GNU zip.

'gzip -c input.dat > output.zip' command executed to compress a 10MB zero filled file and a 10MB randomly filled file (Table 5.2).

Table 5.2 Compression times.

	Jinux	Full emulation
Zero filled	18.828s	43.300s
Random filled	78.520s	110.200s

5.3 md5sum Test

md5sum is a utility that calculates and verifies 128-bit MD5 hashes, as described in RFC 1321 (RFC 1321, nd). The MD5 hash (or checksum) functions as a compact digital fingerprint of a file.

'md5sum input.dat' command executed with a 10MB randomly filled file (Table 5.3).

Table 5.3 md5sum execution times.

	Jinux	Full emulation
md5sum	9.725s	13.700s

5.4 All Results

In Figure 5.1 comparison of execution times can be seen.

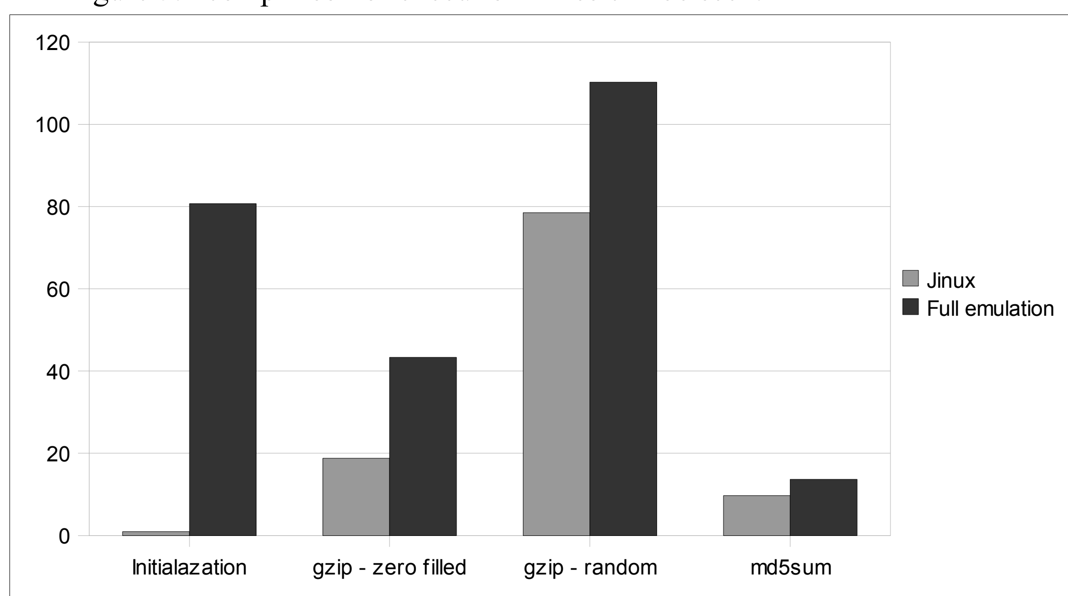

Figure 5.1 All measurements in graphics.

CHAPTER SIX
CONCLUSION & FUTURE WORK

Today IA-32 compatible systems and Java has very important roles in computer technology. IA-32 systems has a huge software repository and Java brings platform independence to softwares. Giving this independence to IA-32 softwares, promises many usage cases.

In this study, a Linux compatibility layer for Java introduced (Jinux). It is shown that this layer facilitates executing codes for IA-32 (x86) architecture on Java platforms and decrease resource requirements.

Currently Jinux is in preliminary stages but able to satisfy operating system needs of simple applications. With the increasing number of supported system calls, number of supported Linux software will increase.

To extend the efficiency of JPCI module, multi application support can be added.

File system case-sensitivity and permission support must be added to increase the Linux compatibility.

REFERENCES

Burcham, J. K. W. (2005). *An X86 emulator written using Java*. Manchester: University of Manchester.

Cygwin (n.d.). Retrieved November 10, 2008, from http://www.cygwin.com/index.html

Golm, M., Felser, M., Wawersich, C., & Kleinöder, J. (2002). The JX Operating System.

Handy, B. N., & Murphey, R. (n.d.). Linux Binary Compatibility. *FreeBSD Handbook.* Retrieved August 7, 2008, from http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/linuxemu.html

JavaOS (n.d.). Retrieved November 10, 2008, from http://en.wikipedia.org/wiki/JavaOS

Jnode (n.d.). Retrieved November 10, 2008, from http://www.jnode.org/index.html

Jazillian (n.d.). Retrieved November 10, 2008, from http://www.jazillian.com/index.html

JPC - Computer Virtualization in Java (n.d.). Retrieved November 10, 2008, from http://www-jpc.physics.ox.ac.uk/index.html

Malabarba, S., Devanbu, P., & Stearns, A. (1999). MoHCA-Java: A Tool for C++ to Java Conversion Support

Pearcolator (n.d.). Retrieved November 10, 2008, from http://pearcolator.wiki.sourceforge.net/index.html

Raymond, E. (n.d.). Retrieved November 7, 2008, from http://www.catb.org/~esr/jargon/html/G/grep.html

RFC 1321 (n.d.). Retrieved November 7, 2008, from http://tools.ietf.org/html/rfc1321

Russel, C. (2002). Application Compatibility in Windows XP. Microsoft *MVP for Windows Server and Tablet PC*. Retrieved August 7, 2008, from http://www.microsoft.com/windowsxp/using/helpandsupport/learnmore/russel_02february18.mspx.

JPC - Technology Overview (n.d.). Retrieved August 7, 2008, from http://www-jpc.physics.ox.ac.uk/Technology.html.

UMSDOS (2008). Retrieved August 7, 2008, from http://en.wikipedia.org/wiki/UMSDOS

APPENDIX A
CODES

A.1 org.jinux.Jinux.java

```
package org.jinux;

import java.io.IOException;
import java.io.InputStream;
import java.io.PrintStream;
import java.util.Enumeration;
import java.util.InvalidPropertiesFormatException;
import java.util.LinkedList;
import java.util.Properties;

import org.binarytranslator.DBT_Options;
import org.binarytranslator.generic.os.abi.linux.LinuxConstants.fcntl;
import org.binarytranslator.generic.os.loader.elf.ELF_Loader;
import org.binarytranslator.generic.os.process.ProcessSpace;
import org.jinux.fs.FileSystem;
import org.jinux.fs.JSystemIO;
import org.jinux.fs.Null;
import org.jinux.jpc.JnxPC;
import org.jinux.jpc.memory.MemoryManager;
import org.jinux.jpc.memory.OutOfPageException;
import org.jinux.kernel.SysCallHandler;
import org.jpc.emulator.processor.Processor;
import nasm.Disassembler;

public class Jinux
{

    public static FileSystem fileSystem;
    private static Properties properties;
    public static SysCallHandler syscallHandler;

    static {
        properties = new Properties();
        InputStream is = Jinux.class.getClassLoader().
                         getResourceAsStream("jinux.properties");
        try
        {
            properties.load(is);
        } catch (InvalidPropertiesFormatException e)
```

```
            {
                e.printStackTrace();
            }
            catch (IOException e)
            {
                e.printStackTrace();
            }

            fileSystem = new FileSystem();
            fileSystem.map("/dev/tty", Null.class);
            fileSystem.map("/dev/null", Null.class);

            try
            {
                fileSystem.setFile(0, JSystemIO.in);
                fileSystem.setFile(1, JSystemIO.out);
                fileSystem.setFile(2, JSystemIO.err);
            } catch (Exception e)
            {}

            syscallHandler = new SysCallHandler();
        }

        public static String getProperty(String key)
        {
            return properties.getProperty(key);
        }

        public static String[] getEnvironment()
        {
            Enumeration<?> propertyNames = properties.propertyNames();
            LinkedList<String> list = new LinkedList<String>();
            while( propertyNames.hasMoreElements() ) {
                String key = (String) propertyNames.nextElement();
                if(key.startsWith("env.")) {
                    list.add(key.substring(4)+"="+properties.getProperty(key));
                }
            }

            String env[] = new String[list.size()];
            return list.toArray(env);
        }

        public static int run(String executable, String... args) throws IOException,
OutOfPageException
        {
            long init = System.currentTimeMillis();
```

```
        JnxPC pc = JnxPC.getInstance();
        MemoryManager mm = MemoryManager.getInstance();
        mm.initPaging();
        mm.initProtectedMode();

        DBT_Options.executableFile = executable;
        DBT_Options.executableArguments = args;

        ELF_Loader el = new ELF_Loader();

        Processor cpu = pc.getProcessor();
        int br = 0;

        long start = System.currentTimeMillis();

        ProcessSpace ps = el.readBinary(DBT_Options.executableFile);

        try {
            while (ps.finished == false) {
                /*int ip = cpu.getInstructionPointer();
                if (ip == br) {
                    System.out.println("PC: " + Integer.toHexString(ip));
                }*/
                pc.executeStep();
            }
        } catch(Exception e) {
            int ip = cpu.getInstructionPointer();
            System.out.println("Exception at PC: " + Integer.toHexString(ip));
            dumpRegisters(cpu);
            dumpCode( ip );
            dumpCallStack();

            e.printStackTrace();
        }

        long end = System.currentTimeMillis();
        System.out.println("Initializetion time was "+(start-init)+" ms.");
        System.out.println("Execution time was "+(end-start)+" ms.");

        pc.dispose();
        return ps.exit_code;
    }

    /**
     * @param args
     * @throws IOException
     * @throws OutOfPageException
     */
```

```
	public static void main(String[] args) throws IOException, OutOfPageException
{
		switch(1) {
			case 1:
				run("/root/test/gzip/md5sum","/root/test/gzip/testdata2");
				break;

			case 2:
				run("/root/test/gzip/grep","-c","a","/root/test/gzip/testdata2");
				break;

			case 3:
				fileSystem.close(1);
				fileSystem.open("/root/test/gzip/td2j.gz",
					fcntl.O_WRONLY|fcntl.O_CREAT|fcntl.O_TRUNC);
				run("/root/test/gzip/gzip","-c","/root/test/gzip/testdata2");
				break;

			case 4:
				fileSystem.close(1);
				fileSystem.open("/root/test/gzip/tdj.gz",
					fcntl.O_WRONLY|fcntl.O_CREAT|fcntl.O_TRUNC);
				run("/root/test/gzip/gzip","-c","/root/test/gzip/testdata");
				break;

			case 5:
				run("/root/test/fstat/fstat","0");
				break;
		}
	}

	static void dumpCode(int address) {
		address &= ~0xFF;
		MemoryManager mm = MemoryManager.getInstance();
		byte data[] = new byte[256];
		mm.readV(address, data, 0, data.length);

		PrintStream out = System.out;
		out.println("Code:");

		String[] asms = new String[256];
		byte[] lengths = new byte[256];
		int lc = Disassembler.disassemble(data, 0, address, true, asms, lengths);

		for(int i=0; i<lc; i++) {
			out.printf("%X: %s\n", address, asms[i]);
			address += lengths[i];
		}
```

```
    }

    static void dumpCallStack() {
        JnxPC pc = JnxPC.getInstance();
        Processor cpu = pc.getProcessor();
        MemoryManager mm = MemoryManager.getInstance();

        PrintStream out = System.out;
        out.println("Call Stack:");

        int bp = cpu.ebp;
        int ip = cpu.eip;
        byte data[] = new byte[4];
        try{
            for(int i=0; i<10; i++) {
                out.printf("%d: %X %X\n",i,bp,ip);
                mm.readV(bp+4, data, 0, data.length);
                ip = ((data[3]&0xFF) << 24) |
                    ((data[2]&0xFF) << 16) |
                    ((data[1]&0xFF) << 8) |
                    ((data[0]&0xFF));
                mm.readV(bp, data, 0, data.length);
                bp = ((data[3]&0xFF) << 24) |
                    ((data[2]&0xFF) << 16) |
                    ((data[1]&0xFF) << 8) |
                    ((data[0]&0xFF));
            }
        } catch(Exception e) {
        }

        try{
            data = new byte[256];
            bp = cpu.ebp;
            mm.readV(bp, data, 0, data.length);
            for(int i=0; i<256; i+=4) {
                int v = ((data[i+3]&0xFF) << 24) |
                    ((data[i+2]&0xFF) << 16) |
                    ((data[i+1]&0xFF) << 8) |
                    ((data[i+0]&0xFF));
                out.printf("%d %X: %X\n",i,bp,v);
                bp += 4;
            }
        } catch(Exception e) {
        }
    }

    static void dumpRegisters(Processor cpu) {
        PrintStream out = System.out;
```

```
        out.println("Registers:");
        out.printf("eax: %X\tebx %X\n", cpu.eax, cpu.ebx);
        out.printf("ecx: %X\tedx %X\n", cpu.ecx, cpu.edx);
        out.printf("esi: %X\tedi %X\n", cpu.esi, cpu.edi);
        out.printf("esp: %X\tebp %X\n", cpu.esp, cpu.ebp);
    }
}
```

A.2 org.jinux.jpc.memory.MemoryManager.java

```
package org.jinux.jpc.memory;

import java.io.IOException;
import java.io.RandomAccessFile;

import org.jinux.jpc.JnxProcessor;
import org.jpc.emulator.processor.ModeSwitchException;
import org.jpc.emulator.processor.Processor;

public class MemoryManager {

    private static MemoryManager instance;
    public static final int CODE_SEGMENT = 0x08; /* 1 * 8 = 0x08 */

    public static final int DATA_SEGMENT = 0x10; /* 2 * 8 = 0x10 */

    private static final int PD_ADDRESS = 0;
    private static final int PD_SIZE = (1 + 1024) * 1024 * 4; /* MAX PD SIZE */

    private static final int DATA_ADDRESS = 0x10000000;
    private static final int DATA_SIZE = Pages.PAGE_SIZE * 10;
    private static final int GDT_ADDRESS = DATA_ADDRESS;
    private static final int HEAP_START = 0x20000000;

    final Processor processor;
    final DescriptorTable gdt;
    public final PageDirectory pd;
    public final PageManager pm;
    PhysicalMemorySegment pd_ms;
    VirtualMemorySegment dt_ms;
    private boolean mapping_changed;
    private int heap_address;

    protected MemoryManager(Processor processor) throws OutOfPageException {
        this.processor = processor;
```

```
        pm = new PageManager(0, 32 * 1024 * 1024);

        pd = new PageDirectory();
        mapping_changed = true;

        pd_ms = new PhysicalMemorySegment(this, PD_ADDRESS, PD_SIZE);
        dt_ms = new VirtualMemorySegment(this, DATA_ADDRESS, DATA_SIZE,
                PageDirectory.PTGLOBAL);

        heap_address = HEAP_START;

        gdt = new DescriptorTable();

        // Null GDT.  This initial null GDT is needed
        // by the CPU to set their protection scheme.
        gdt.addEntry(0, 0, 0, 0);


        /* The kernel segments */

        // Code Segment.  The base address is 0, the
        // limit is 4GBytes (0xffffffff), it uses
        // 4KByte granularity (page size, 0xcf), uses
        // 32-bit opcodes, and is a Code Segment
        // descriptor (0x9a).
        gdt.addEntry(0, 0xffffffff,
                DescriptorTable.D_CODE | DescriptorTable.D_READ,
                DescriptorTable.D_BIG | DescriptorTable.D_BIG_LIM);


        // Data Segment.  Values are the same but for
        // Data Segment indicator (0x92 insetead of
        // 0x9a).
        gdt.addEntry(0, 0xffffffff,
                DescriptorTable.D_DATA | DescriptorTable.D_WRITE,
                DescriptorTable.D_BIG | DescriptorTable.D_BIG_LIM);


    /* The user segments --- to hell with memory protection :) */
    // stndDesc(0x00000000,0xfffff,(D_CODE+D_READ+D_BIG+D_BIG_LIM+D_DPL3)),
    // stndDesc(0x00000000,0xfffff,(D_DATA+D_WRITE+D_BIG+D_BIG_LIM+D_DPL3)),

    /* The two TSSes 'task state segment' taskstate.h */
    //struct TSS {
    //    UWORD16 back, RESERVED0;      /* Backlink                      */
    //    UWORD32 esp0;                 /* The CK stack pointer          */
    //    UWORD16 ss0,  RESERVED1;      /* The CK stack selector         */
```

```
//    UWORD32 esp1;                 /* The parent KL stack pointer    */
//    UWORD16 ss1,  RESERVED2;      /* The parent KL stack selector   */
//    UWORD32 esp2;                 /* Unused                         */
//    UWORD16 ss2,  RESERVED3;      /* Unused                         */
//    UWORD32 cr3;                  /* The page directory pointer     */
//    UWORD32 eip;                  /* The instruction pointer        */
//    UWORD32 eflags;               /* The flags                      */
//    UWORD32 eax, ecx, edx, ebx;   /* The general purpose registers  */
//    UWORD32 esp, ebp, esi, edi;   /* The special purpose registers  */
//    UWORD16 es,   RESERVED4;      /* The extra selector             */
//    UWORD16 cs,   RESERVED5;      /* The code selector              */
//    UWORD16 ss,   RESERVED6;      /* The application stack selector */
//    UWORD16 ds,   RESERVED7;      /* The data selector              */
//    UWORD16 fs,   RESERVED8;      /* And another extra selector     */
//    UWORD16 gs,   RESERVED9;      /* ... and another one            */
//    UWORD16 ldt,  RESERVED10;     /* The local descriptor table     */
//    UWORD16 trap;                 /* The trap flag (for debugging)  */
//    UWORD16 io;                   /* The I/O Map base address       */
//}
// stndDesc(0x00000000,(sizeof(struct TSS)-1), D_TSS),
// stndDesc(0x00000000,(sizeof(struct TSS)-1), D_TSS),
}

public void initPaging() {
    updatePageDirectory();

    processor.physicalMemory.setGateA20State(true);

    // /* Load %cr3 with kernelPageTables */
    // asm("movl %%eax,%%cr3" : : "a" (((UADDR) pageDir) & 0xfffff000));
    //
    // /* Enable paging */
    // asm volatile (
    //     "movl %%cr0,%%eax       \n"   /* Get current MSW      */
    //     "orl $0x80000000,%%eax  \n"   /* Set PG bit           */
    //     "movl %%eax,%%cr0       \n"   /* Load MSW into %cr0   */
    //     "movl %%cr3,%%eax       \n"   /* Flush TLB            */
    //     "movl %%eax,%%cr3       \n"
    //     "ljmp %0,$1f            \n"   /* Flush prefetch queue */
    //     "1:                     \n"
    //     :
    //     : "p" (0x08)
    //     : "%eax"
    // );

    /* Load %cr3 with PageDir */
    processor.setCR3(PD_ADDRESS);
```

```
            /* Enable paging */
            try {
                processor.setCR0(
                        (processor.getCR0() |
                        Processor.CR0_PAGING |
                        Processor.CR0_PROTECTION_ENABLE) & /* Set PG bit */
                        ~(Processor.CR0_CACHE_DISABLE) );   /* Enable caching */
            } catch (ModeSwitchException e) {
            }

        // do far jump
        }

        public void updatePageDirectory() {
            if (mapping_changed) {
                MemoryRange pd_range = pd_ms.newRange(PD_ADDRESS, PD_SIZE);

                pd.load(pd_range);

                mapping_changed = false;
            }
        }

        public void updateGDT() {
            int size = gdt.getSize();

            MemoryRange gdt_range = dt_ms.newRange(GDT_ADDRESS, size);

            gdt.load(gdt_range);

            int limit = size - 1;
            processor.gdtr = processor.createDescriptorTableSegment(GDT_ADDRESS,
limit);
        }

        public int addGlobalDescriptorEntry(int base, int limit, int access, int
granularity) {
            return gdt.addEntry(base, limit, access, granularity);
        }

        public void initProtectedMode() throws OutOfPageException {
            updateGDT();

            /*
            // Set up the pointer
            DTEntryPtr gdtPtr = new DTEntryPtr();
            gdtPtr.mLimit = sizeof(gdt) - 1;
            gdtPtr.mBase = (int)(&gdt);
```

```
            int GDTPtrAddress = (int)(&gdtPtr);
            // Load it into the CPU
            asm volatile ("lgdt %0" : : "m"(GDTPtrAddress));
            asm volatile ("mov %%ax,0x10" : : );
            asm volatile ("mov %%ds,%%ax" : : );
            asm volatile ("mov %%es,%%ax" : : );
            asm volatile ("mov %%fs,%%ax" : : );
            asm volatile ("mov %%gs,%%ax" : : );
            asm volatile ("mov %%ss,%%ax" : : );
             */

            try {
                processor.setCR0(processor.getCR0() |
Processor.CR0_PROTECTION_ENABLE);
            } catch (ModeSwitchException e) {
            }

            processor.cs = processor.getSegment(CODE_SEGMENT);

            processor.ds = processor.getSegment(DATA_SEGMENT);
            processor.es = processor.getSegment(DATA_SEGMENT);
            processor.fs = processor.getSegment(DATA_SEGMENT);
            processor.gs = processor.getSegment(DATA_SEGMENT);
            processor.ss = processor.getSegment(DATA_SEGMENT);

        //do far jump
        }

        public void write(Pages pages, byte[] data) {
            for (int a = 0; a < pages.getPageCount(); a++) {
                processor.physicalMemory.copyContentsFrom(pages.getPageAddress(a),
data, 0, data.length);
            }
        }

        public void write(Pages pages, int offset, byte[] data, int data_offset, int
length) {
            int s = offset / Pages.PAGE_SIZE;
            int e = (offset + length - 1) / Pages.PAGE_SIZE;
            offset %= Pages.PAGE_SIZE;
            int p = Pages.PAGE_SIZE - offset;
            for (int a = s; a <= e; a++) {
                int len = p > length ? length : p;
                processor.physicalMemory.copyContentsFrom(pages.getPageAddress(a) +
offset, data, data_offset, len);
                data_offset += len;
                length -= len;
                p = Pages.PAGE_SIZE;
```

```
            offset = 0;
        }
    }

    public void read(Pages pages, int offset, byte[] data, int data_offset, int
length) {
        int s = offset / Pages.PAGE_SIZE;
        int e = (offset + length - 1) / Pages.PAGE_SIZE;
        offset %= Pages.PAGE_SIZE;
        int p = Pages.PAGE_SIZE - offset;
        for (int a = s; a <= e; a++) {
            int len = p > length ? length : p;
            processor.physicalMemory.copyContentsInto(pages.getPageAddress(a) +
offset, data, data_offset, len);
            data_offset += len;
            length -= len;
            p = Pages.PAGE_SIZE;
            offset = 0;
        }
    }

    public void readP(int address, byte[] buffer, int off, int len)
    {
        processor.physicalMemory.copyContentsInto(address, buffer, off, len);
    }

    public void readV(int addr, byte[] buffer, int off, int len)
    {
        int address = Pages.truncateToPage(addr);
        int offset = addr - address;
        int size = 1;
        if( offset > Pages.PAGE_SIZE - len ) {
                size = 2;
        }
        size *= Pages.PAGE_SIZE;

        VirtualMemorySegment vms =
                VirtualMemorySegment.getSegment(this, address, size);

        vms.read(offset, buffer, 0, len);
    }

    public VirtualMemorySegment map(int address, int length, boolean read,
            boolean write, boolean exec) throws MemoryMapException {
        VirtualMemorySegment vms;
        int flags = PageDirectory.PTUSER;
        if (write) {
            flags |= PageDirectory.PTWRITE;
```

```
            }
            if (address == 0) {
                // TODO: check if heap is still empty
                address = heap_address;
                heap_address += length;
            }
            try {
                vms = new VirtualMemorySegment(this, address, length, flags);
            // TODO permisions
            } catch (OutOfPageException e) {
                throw new MemoryMapException(0,
MemoryMapException.Reason.UNALIGNED_ADDRESS);
            }

            mapping_changed = true;
            return vms;
        }

        public VirtualMemorySegment map(RandomAccessFile file, long file_pos, int
address, int length,
                boolean read, boolean write, boolean exec) throws MemoryMapException,
IOException {
            VirtualMemorySegment vms;
            int flags = PageDirectory.PTUSER;
            if (write) {
                flags |= PageDirectory.PTWRITE;
            }
            if (address == 0) {
                // TODO: check if heap is still empty
                address = heap_address;
                heap_address += length;
            }
            try {
                vms = new VirtualMemorySegment(this, address, length, flags);
            // TODO permisions
            } catch (OutOfPageException e) {
                throw new MemoryMapException(0,
MemoryMapException.Reason.UNALIGNED_ADDRESS);
            }

            file.seek(file_pos);
            byte[] buf = new byte[Pages.PAGE_SIZE];
            for (int a = 0; a < length;) {
                int len = length - a;
                if (len > Pages.PAGE_SIZE) {
                    len = Pages.PAGE_SIZE;
                }
                len = file.read(buf, 0, len);
```

```
            if (len <= 0) {
                break;
            }
            vms.write(a, buf, 0, len);
            a += len;
        }

        mapping_changed = true;
        return vms;
    }

    public static MemoryManager getInstance() {
        if (instance == null) {
            try {
                instance = new MemoryManager(JnxProcessor.getInstance());
            } catch (OutOfPageException e) {
                e.printStackTrace();
            }
        }
        return instance;
    }

    void unmap(int addr, int len) {
        pd.unmap(addr, len);
    }
}
```

A.3 org.jinux.jpc.memory.DescriptorTable.java

```
package org.jinux.jpc.memory;

import java.lang.reflect.Field;
import java.util.ArrayList;

import org.jinux.jpc.util.struct.Struct2;

public class DescriptorTable extends Struct2 {
    /*
     *  Each descriptor should have exactly one of next 8 codes to define the
     *  type of descriptor
     */

    static public final int D_LDT = 0x2; /* LDT segment        */
```

```
        static public final int D_TASK = 0x5; /* Task gate          */

        static public final int D_TSS = 0x9; /* TSS                */

        static public final int D_CALL = 0x0C; /* 386 call gate      */

        static public final int D_INT = 0x0E; /* 386 interrupt gate */

        static public final int D_TRAP = 0x0F; /* 386 trap gate      */

        static public final int D_DATA = 0x10; /* Data segment       */

        static public final int D_CODE = 0x18; /* Code segment       */

        /*
         *  Descriptors may include the following as appropriate:
         */
        static public final int D_DPL3 = 0x60;   /* DPL3 or mask for DPL */

        static public final int D_DPL2 = 0x40;   /* DPL2 or mask for DPL */

        static public final int D_DPL1 = 0x20;   /* DPL1 or mask for DPL */

        static public final int D_PRESENT = 0x80;   /* Present              */

        /*
         *  Segment descriptors (not gates) may include:
         */
        static public final int D_ACC = 0x1;  /* Accessed (Data or Code)          */

        static public final int D_WRITE = 0x2;  /* Writable (Data segments only)
*/

        static public final int D_READ = 0x2;  /* Readable (Code segments only)    */

        static public final int D_BUSY = 0x2;  /* Busy (TSS only)                  */

        static public final int D_EXDOWN = 0x4;  /* Expand down (Data segments only)
*/

        static public final int D_CONFORM = 0x4;  /* Conforming (Code segments only)
*/

        /*
         * granularity
         */
        static public final int D_BIG = 0x40; /* Default to 32 bit mode          */
```

```
	static public final int D_BIG_LIM = 0x80; /* Limit is in 4K units
*/

	/* Structure */
	private final static Field _fields[] = getFields(DescriptorTable.class, new
String[]{
	    "entries"
	});

	public ArrayList<DTEntry> entries;

	//-------------------- GlobalDescriptorTable -----------------
	DescriptorTable() {
	    // Set up an array to hold the entries
	    entries = new ArrayList<DTEntry>();
	}

	/**
	 * @param base
	 * @param limit
	 * @param access
	 * @param granularity
	 * @return          Returns segment address
	 */
	int addEntry(int base, int limit, int access, int granularity) {
	    DTEntry entry = new DTEntry(base, limit, access, granularity);
	    entries.add(entry);

	    return (entries.size() - 1) * DTEntry.SIZE;
	}

	protected Field[] getFields() {
	    return _fields;
	}

	void load(MemoryRange range) {
	    int s = getSize();
	    byte[] buffer = new byte[s];
	    get(buffer, 0);
	    range.write(0, buffer, 0, s);
	}
   }
```

A.4 org.jinux.jpc.memory.PageDirectory.java

```
   package org.jinux.jpc.memory;

   import org.jinux.jpc.util.struct.SIArray;
   import org.jinux.jpc.util.struct.StructItem;

   class PageDirectory extends PageTable {
           /* PTE flags: */
           static protected final int      PTPRESENT       = (1 << 0); /* Present
bit                                 */
           static public final int         PTWRITE         = (1 << 1); /* Write
permission bit                        */
           static public final int         PTUSER          = (1 << 2); /*
User/Supervisor bit (when included, user)   */
           static public final int         PTWRTTRU        = (1 << 3); /* Page-level
write-through                    */
           static public final int         PTCACHEDIS      = (1 << 4); /* Page-level
cache disable                    */
           static protected final int      PTACCESSED      = (1 << 5); /* Accessed
bit                                */
           static protected final int      PTDIRTY         = (1 << 6); /* Dirty bit
*/

/* Pentium and up                                  */
           static protected final int      PTBIGPAGE       = (1 << 7); /* Page size
2MB-4MB pages if set else 4kb     */

/* Pentium pro and up                              */
           static public final int         PTGLOBAL        = (1 << 8); /* Global
page, keeps page from being flushed  */

/* from cache                                      */
           static public final int         PTNOPAGEOUT     = (1 << 9); /* Don't let
the swapper page this page out,   */

/* this is an alliance defined bit             */

           static public final int         PTUSERFLAG1     = (1 << 10); /* 2 bits
available for programmer's use       */
           static public final int         PTUSERFLAG2     = (1 << 11); /* 2 bits
available for programmer's use       */

           static protected final int PUBLIC_FLAGS_MASK =
                   PTWRITE | PTUSER | PTWRTTRU | PTCACHEDIS | PTGLOBAL | PTNOPAGEOUT
|
                   PTUSERFLAG1 | PTUSERFLAG2;
```

```
        protected static final int TABLE_MAP_SIZE = (1 << 22);

        protected static final int TABLE_ENTRY_MASK = (1 << 10) - 1;
        protected static final int TABLE_ENTRY_SHIFT = 22;

        protected PageTable[] tables;

        public PageDirectory() {
                tables = new PageTable[1024];
        }

        public void map(Pages pages, int linearAddress, int flags) {
                //System.out.printf("MAP 0x%08X 0x%08X 0x%08X 0x%04X\n",
                //      linearAddress, linearAddress+pages.getSize(),
pages.getSize(), flags);

                flags &= PUBLIC_FLAGS_MASK;
                flags |= PTPRESENT;

                int pc = pages.getPageCount();
                for( int pi = 0; pi < pc; pi++ ) {
                        int index = (linearAddress >> TABLE_ENTRY_SHIFT) &
TABLE_ENTRY_MASK;
                        PageTable table = getTable(index);
                        table.map(linearAddress, pages.getPageAddress(pi),
flags);

                        int table_flags = getFlags(index);
                        if( table_flags == 0 ) {
                                setFlags(index, flags);
                        } else if( (flags & (PTUSER | PTWRITE)) != 0 ) {
                                if( (table_flags & (PTUSER | PTWRITE)) != (PTUSER
| PTWRITE) ) {
                                        table_flags |= (PTUSER | PTWRITE);
                                        setFlags(index, table_flags);
                                }
                                //TODO: check other permision bits
                        }

                        linearAddress += Pages.PAGE_SIZE;
                }
        }

        public void unmap(int linearAddress, int length) {
                int pc = Pages.pageCount(length);
                for( int pi = 0; pi < pc; pi++ ) {
                        int index = (linearAddress >> TABLE_ENTRY_SHIFT) &
```

```
TABLE_ENTRY_MASK;
                                PageTable table = getTable(index);
                                table.map(linearAddress, 0, 0);
                                linearAddress += Pages.PAGE_SIZE;
                        }
                }

                /**
                 * Returns a Pages object which contains the pages which map to given
address
                 * space. If there is no mapping for a page then page index is setted to
-1.
                 * @param linearAddress
                 * @param length
                 * @return a Pages object.
                 */
                public Pages getPages(int linearAddress, int length) {
                        int pc = Pages.pageCount(length);
                        int[] pages = new int[pc];

                        for( int pi = 0; pi < pc; pi++ ) {
                                long padr = getMappedAddress(linearAddress);
                                if( padr < 0 ) {
                                        pages[pi] = -1;
                                } else {
                                        pages[pi] = (int) (padr / Pages.PAGE_SIZE);
                                }
                                linearAddress += Pages.PAGE_SIZE;
                        }

                        return new Pages(pages);
                }

                @Override
                public long getMappedAddress(int linearAddress) {
                        int index = (linearAddress >> TABLE_ENTRY_SHIFT) &
TABLE_ENTRY_MASK;
                        if( tables[index] != null ) {
                                return tables[index].getMappedAddress(linearAddress);
                        } else {
                                return -1;
                        }
                }

                private PageTable getTable(int index) {
                        if( tables[index] == null ) {
                                tables[index] = new PageTable();
                        }
```

```
                return tables[index];
        }

        @Override
        public void load(MemoryRange mr) {
                int b = mr.getAddress() + getSize();
                for(int a=0; a<tables.length; a++) {
                        if( tables[a] != null ) {
                                int s = tables[a].getSize();
                                setAddress(a, b);
                                tables[a].load(mr.newRange(b, s));
                                b += s;
                        }
                }
                super.load(mr);
        }
}

class PageTable extends SIArray {
        protected static final int ADDR_MASK = 0xFFFFF000;
        protected static final int FLAG_MASK = 0x00000FFF;

        protected static final int ENTRY_MASK = (1 << 10) - 1;
        protected static final int ENTRY_SHIFT = 12;

        PageTable() {
                super(StructItem.UInt(), 1024);
        }

        void map(int linearAddress, int pyhsicalAddress, int flags) {
                int index = (linearAddress >> ENTRY_SHIFT) & ENTRY_MASK;
                setEntry(index, pyhsicalAddress, flags);
        }

        long getMappedAddress(int linearAddress) {
                int index = (linearAddress >> ENTRY_SHIFT) & ENTRY_MASK;
                if( getFlags(index) == 0 ) {
                        return -1;
                }
                return getAddress(index);
        }

        void setEntry( int index, int pyhsicalAddress, int flags ) {
                int v = (pyhsicalAddress & ADDR_MASK) | (flags & FLAG_MASK);
                set(index, v);
        }

        void setAddress( int index, int pyhsicalAddress) {
```

```
                int v = (pyhsicalAddress & ADDR_MASK) | (getFlags(index) &
FLAG_MASK);
                set(index, v);
        }

        void setFlags( int index, int flags ) {
                int v = (getAddress(index) & ADDR_MASK) | (flags & FLAG_MASK);
                set(index, v);
        }

        int getAddress( int index ) {
                long v = get(index);
                return (int) (v & ADDR_MASK);
        }

        int getFlags( int index ) {
                long v = get(index);
                return (int) (v & FLAG_MASK);
        }

        void load(MemoryRange mr) {
                int s = getSize();
                byte[] buffer = new byte[s];
                get(buffer, 0);
                mr.write(0, buffer, 0, s);
        }
}
```

A.5 org.jinux.fs.JnxFileImp.java

```
package org.jinux.fs;

import java.io.IOException;
import java.util.Arrays;
import org.binarytranslator.generic.os.abi.linux.LinuxConstants.fcntl;

public class JnxFileImp implements JnxFile {
    private boolean readable;
    private boolean writable;

    public JnxFileImp(String path, int flags) throws IOException {
        readable = (flags & fcntl.O_ACCMODE) != fcntl.O_WRONLY;
        writable = (flags & fcntl.O_ACCMODE) != fcntl.O_RDONLY;
    }
```

```
    public boolean isReadable() {
        return readable;
    }

    public boolean isWritable() {
        return writable;
    }

    private void checkRead() throws IOException {
        if(!readable) {
            throw new IOException("File didn't opened as readable.");
        }
    }

    private void checkWrite() throws IOException {
        if(!writable) {
            throw new IOException("File didn't opened as writable.");
        }
    }

    public int read(byte[] b) throws IOException {
        checkRead();
        Arrays.fill(b, (byte)0);
        return b.length;
    }

    public int read(byte[] b, int off, int len) throws IOException {
        checkRead();
        Arrays.fill(b, off, off+len, (byte)0);
        return len;
    }

    public void write(byte[] b) throws IOException {
        checkWrite();
    }

    public void write(byte[] b, int off, int len) throws IOException {
        checkWrite();
    }

    public long getFilePointer() throws IOException {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    public long length() throws IOException {
        throw new UnsupportedOperationException("Not supported yet.");
    }
```

```
    public void seek(long pos) throws IOException {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    public void setLength(long newLength) throws IOException {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    public void close() throws IOException {
    }
}
```

A.6 org.jinux.fs.Null.java

```
package org.jinux.fs;

import java.io.IOException;

public class Null extends JnxFileImp {

    public Null(String path, int flags) throws IOException {
        super(path, flags);
    }

}
```

A.7 org.jinux.fs.Pipe.java

```
package org.jinux.fs;

import java.io.IOException;

public class Pipe implements JnxFile {
    private JnxFile file;

    public Pipe(JnxFile file) throws IOException {
        this.file = file;
    }

    public boolean isReadable() {
        return file.isReadable();
    }
```

```
    public boolean isWritable() {
        return file.isWritable();
    }

    public int read(byte[] b) throws IOException {
        return file.read(b);
    }

    public int read(byte[] b, int off, int len) throws IOException {
        return file.read(b, off, len);
    }

    public void write(byte[] b) throws IOException {
        file.write(b);
    }

    public void write(byte[] b, int off, int len) throws IOException {
        file.write(b, off, len);
    }

    public long getFilePointer() throws IOException {
        return file.getFilePointer();
    }

    public long length() throws IOException {
        return file.length();
    }

    public void seek(long pos) throws IOException {
        file.seek(pos);
    }

    public void setLength(long newLength) throws IOException {
        file.setLength(newLength);
    }

    public void close() throws IOException {
        file.close();
        file = null;
    }
}
```