Memory Management in JikesNode Operating System

A thesis submitted to the University of Manchester for the degree of Master of Science in the Faculty of Science and Engineering

2005

Yun Zhang

School of Computer Science

List of Contents

List of Figures

Abstract

Providing an interface between computer and user, the research on operating system is almost as old as the computer itself. As an important subsystem, the memory management system provides a mechanism for the applications and user to operate the data or code stored in the storage media, such as RAM or hard disk. This mechanism usually depends on the support of processors and other computer components. JikesNode, a Java operating system which combines Jikes Research Virtual Machine with the JNode operating system, aims to provide the Jamaica group with a tool for the further study of chip multi-processors and parallelism. This project improves the memory management subsystem in JikesNode by importing a garbage collection and modifying the memory mapping mechanisms, which are based on the Intel IA-32 architecture processors.

Declaration

No portion of the work referred to in the thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgement

I would like to thank the people who contributed most to this project and thesis:

My supervisor, Dr. Chris Kirkham for his sound supervision, theoretical and practical guidance throughout my M.Sc. project, especially his suggestion and proofreading to my M.Sc. thesis and seminar.

Ian Rogers, for his helping during my design and implement the system. Ian helped me set up the system and tools at the beginning of the project, and gave the ideas and guidance for my design. He would immediately give up what he was up to whenever I asked for help.,

And finally to my friends for their encouragement and technical help. They are Mr Yiming Wang, Mr. Peihong Ke, Mr Yi Zhu, Mr Jie Zhao and Mr Yu Cao.

Chapter 1
Introduction

1.1. Background

The Jamaica group at the University of Manchester is investigating the design of chip multi-processors (CMPs) and their accompanying parallel software environments. To efficiently utilise multi-processors by client applications, the Jamaica group needs an operating system which can run massively multithreaded applications and support advanced compiler technology to automate parallelisation and the distribution of jobs. But the currently available operating systems can not run on the Jamaica chip and do not support the development of advanced compiler technology. So the Jamaica group decided to develop a new operating system to be able to test and further develop the design and the implementation of the Jamaica chip.

We call this operating system JikesNode, which uses JikesRVM [5] as the java virtual machine and JNode [3] for device drivers, file system etc. Although we design the JikesNode to be used in Jamaica research, we also want this system will be able to run on conventional hardware configurations, such as Intel-based PCs, and provide the benefits of optimizing compilers on these systems too. So the implementation and research on JikesNode is on the Intel i386 platform in this thesis.

1.2. Motivation

Much implementation of JikesNode has been done before this project, such as creating a nanokernel, successfully building and booting JikesRVM. But now, JikesNode runs in the no garbage collection environment. Without garbage collection, the memory used by objects will not be freed when the objects are no longer used. Therefore, the memory in the system will be used up soon if many objects are created. Because JikesNode runs in a limited memory now, it's important to import a garbage collection into JikesNode to make the memory reusable. One of the simple methods is to utilize the garbage collection mechanism implemented in JikesRVM. In addition, after the Georgios' M.Sc. project in 2004 [1], many modifications have been done, which makes the system not work. So, at the beginning of this project, we have following goals to be finished:

- Debug the current version of JikesNode.
- Import a proper garbage collection provided by JikesRVM into JikesNode.
- Modify the memory management system to support the selected garbage collection.

1.3. Garbage collection

Garbage collection (also known as GC) is a form of automatic memory management. It attempts to reclaim the memory used by the objects that will never be used again by the application. The basic principle of how a garbage collector works is:

- Detect what data objects in a program will not be accessed in the future
- Reclaim the storage used by those objects

The object detection is usually accomplished by defining a set of roots and determining reachability from the roots [19, Chapter 9]. If the system can access an object by some path of references from the roots, this object is reachable and considered as in use. On the contrary, if an object is not reachable, it's a "garbage" object, whose memory will be reclaimed.

By now, many garbage collection algorithms have been implemented, such as Reference Counting, Mark and Sweep, Compacting, Copying, Generational and Adaptive collectors [19, Chapter 9]. Some of them have been supported by the JikesNode.

1.4. Outline

Chapter 2 is about the JikesNode, the Java operating system implemented by the Jamaica Group. The system architecture and two systems used in JikesNode are described first. Then previous work done by the Jamaica Group is represented.

Chapter 3 gives a description of the memory management mechanism in IA-32 Intel, from its system architecture to the segment and paging supported by IA-32 Intel processor.

Chapter 4 describes the memory management implemented in current JikesNode in detail. First, an overview of MMTk (Memory Management Toolkit [4]) and object model used in JikesRVM is represented. Following this is the memory management subsystem in JikesNode.

Chapter 5 lists the work done in this project and some of the author’s personal opinions on what should be done in the future.

Chapter 2

JikesNode operating system

JikesNode operating system is a Java operating system implemented by the Jamaica Group at the University of Manchester. It uses the Jikes Research Virtual Machine (Jikes RVM) as the virtual machine and Java New Operating System Design Effort (JNODE) as the Java operating system.

This chapter is a detailed description of the implementation of the JikesNode operating system. Firstly, two systems (JikesRVM and JNODE) integrated in JikesNode are described. Then I represent the current work Jamaica Group has done on this system.

2.1. System Architecture

As mentioned above, JikesNode integrates JikesRVM and JNode to implement a new Java operation system. Jikes RVM is used as the basic Java Virtual Machine (JVM) to run Java programs. The JNODE is imported to support the functions of operations system such as device drivers, file system and shell. The following figure displays the architecture of the whole system.

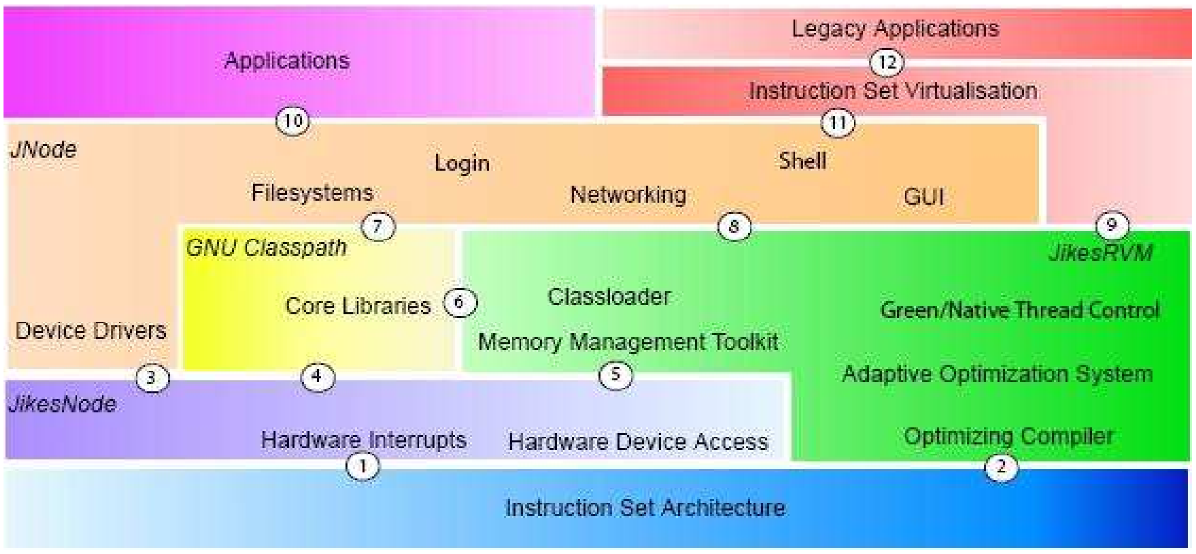

Figure 2-1. JikesNode architecture [6]

In this figure, JikesNode kernel provides an abstract layer between the hardware and other parts in the system; it provides the hardware interrupts, and thread control mechanism. Jikes RVM is the platform responsible for running all the system generated executing code. And JNode gives a system interface for users and applications. All three parts of the system need some core libraries, for which we use GNU Classpath.

2.2. Jikes Research Virtual Machine (Jikes RVM)

The Jikes RVM is a Java Virtual Machine mostly written in Java language [8]. It is built from an IBM internal project called Jalapeño [7] and was made open source in 2001. As a JVM, it has many advanced features such as optimizing compiler, several GC strategies and a sophisticated thread execution mechanism [1].

Jikes RVM Structure

There are four major components in the JikesRVM [4 Chapter 6].

Core runtime is a service platform to execute applications and interface with libraries. It consists of thread scheduler, class loader, library support etc. Most classes of this component are contained in com.ibm.JikesRVM and com.ibm.JikesRVM.classloader packages.

Compiler in Jikes RVM is a Just-In-Time compiler response for building native code from the bytecode. Now, the Jikes RVM has two different compilers, baseline compiler and optimizing compiler.

Memory managers are responsible for managing the objects created during executing applications. In the latest version, the Jikes RVM uses a new memory management framework called MMTk. This part of classes have been modified and imported into the packages org.mmtk.vm and com.ibm.JikesRVM.memoryManagers.mmInterface.

Adaptive optimization system provides a mechanism to an optimizing compiler for applications to improve their performance.

In addition to these four parts, JikesRVM also has a part called native runtime [1, Chapter4] which is not written in Java. The main work of its part includes loading the JikesRVM image into memory, exception pre-processing and providing interfaces between hardware and JVM.

Boot Image

In order to run itself without a second virtual machine, JikesRVM introduces the concept of a boot image, which contains and saves the location of a frozen instance of the initial VM. A program called BootImageWriter performs all the building process. Firstly, it uses an external JVM (currently, the Donor VM) to compile all core classes of the JikesRVM. Then the compiled native code is imported into the image file and some important system components are appended.

2.3. JNode operating system

JNode is a relatively new open source project to create a Java operating system for personal use [3, Goals]. We choose it as our operating system platform for the following reasons.

- A new project and with small size and therefore not too tough to modify
- A loose integration between the operating system and the Java virtual Machine makes it easy to factor the operating system from its own JVM into JikesRVM.
- JNode has an appropriate bootloader, nanoKernel and plug-in architecture.

All JNode systems can be divided into 4 important parts [3, Developer guide]. Common part contains the fundamental functions to boot and run the system. **VM** part has a JVM for the system to compile and run the Java bytecode. **JNode Operating System** provides the functions of an operating system, such as file system, shell and device drivers. And the last part contains the core library JNode depends on.

Plug-in Architecture

In JNode, every module is a Plug-in, except the virtual machine, operating system and plug-in manager framework itself. These plug-ins can be divided into two types, normal plug-in which can be loaded, unloaded and reloaded [3], and system plug-in which must exist during the whole system lifetime. Every Plug-in has a descriptor file written in XML and is contained in a JAR file. The descriptor file defines all the information to load the plug-in, such as the required classes, the location of the associated JAR files. All plug-ins can define their extension points which can be accessed by system or other related applications. Every plug-in has a specific class loader and access permissions

In order to manage all plug-ins, JNode implements a new conception named PluginManager shared by the Virtual machine and operating system. In JNode, PluginManger is a central component and started after initializing the Java virtual machine during system booting. Its main work includes holding information on plug-ins (implemented by PluginRegistry) and plug-in lifecycle management (implemented by PluginLoaderManager)

GRUB boot loader

During booting, most operating systems need a boot loader to load themselves into memory and provide the information about the hardware platform. In JNode, GRUB boot loader [22] is used, which is also used in JikesNode now because we want to be able to boot JikesNode on machines with many operating systems.

The GRUB boot loader aims to be a boot loader that supports all existing operating systems. It has two stages during booting an operating system. The first stage is contained in the Master Boot Record (MBR), which is 512 bytes on i386. Because of its limited size, stage 1 does little work and then loads the second stage of GRUB. In this stage, GRUB provides the user with a boot menu, loads the selected kernel of the OS and passes control to the kernel. GRUB uses a configuration file named menu.lst to save information about the possible kernel and some command lines, if needed. During booting, the configuration file is read, then GRUB loads the selected kernel into a consecutive memory starting from 0x100000 (1MB) and passes control to the kernel entry point.

2.4. Current implementation

To integrate JikesRVM and JNode together, we need some modifications to both systems to let them interact with each other seamlessly. For example, unifying the classpath of Java library, modifying the build system and changing the VM of JNode. The following describes the work that has been done by the Jamaica Group.

Nanokernel

Nanokernel is a term describing an operating system core which is strictly limited in it size and/or functionality [1, Chapter 3]. It usually provides an abstraction layer to link the hardware and operating system. But in JikesNode, it also has an extra target: providing a runtime environment for the JikesRVM [1, Chapter 3.3]. The current work on the nanokernel is to extend the nanokernel in JNode to meet our requirements. The JNode nanokernel is entirely implemented in assembly language. But in JikesNode, we try to avoid using assembler and write the kernel in C language instead of assembly language as much as possible.

Mutliboot information header

Using the GRUB boot loader, the Multiboot information header must be written in the first byte of the kernel. Because we use NASM i386 as our assembler, which compiles the code sequentially, the first compiled file start.s, should contain the Multiboot information header at the begin. Considering the memory management in JikesNode, we set the header's flag to align all boot modules on 4KB page boundary.

Memory management initialization

All memory management initialization is implemented in mm.s file. The Global Descriptor Table (GDT) [Section 3.3] is initialized by filling with five segments: Kernel Data Segment, Kernel Code Segment, User Date Segment, User Code Segment and a Task-State Segment [Chapter 3]. The memory space is paged. We create a page directory and one page table. The first 4 MBytes of memory uses 4-KByte Pages and the rest uses 4-MByte pages. The pages located in the kernel area are set as read only. At last, the stack pointer is set.

Interrupt handler initialization

In JikesNode, the initialization is included in **ints.s** and **interrupts.c** file. It uses various macros to prepare and set the interrupt handler. For example, **int_noerror** maps the error interrupt to specific handler address. **intport** sets the interrupt entry in IDT table. **int_irq** maps the interrupt requests (IRQs) with their handler functions, which are written in C.

Hardware component initialization

Before transferring into the user mode, the nanokernel initializes some fundamental hardware, including serial port, PIT and FPU.

System console and debugging support

During the system initialization, console output is supported in kernel mode. It is implemented by writing the characters directly to video memory address 0xB80000. Some basic functions such as scrolling are implemented in console.c, and some complex outputting functions are written in separate files and saved in **klib**.

Native Runtime

We use the native runtime from JikesRVm as the basis of JikesNode native runtime. The main files include **sys.C**, **libVM.C** and **cmdline.h**. But because JikesRVM runs on a Linux platform, it calls many low-level functions that have been implemented in the operating system. For a new operating system, we must write these call back functions (C Stub [6]) independently based on our requirements. All necessary functions are saved in **klib** directory, the implemented ones are written in separate files, while all others are saved in fake.c file. The choice of the functions and their targets are from The Open Group Base Specifications [9]

Others

Besides the above implementation in the JikesNode, the Jamaica Group has done much work on the integration between JikesNode and JNode systems. Different versions of GNU classpath used in both systems have been unified. The VM used in JNode has been mostly removed. Various build systems have also been added to support the different requirements.

Chapter 3

Memory Management in IA-32 Intel

Currently, JikesNode is running on an IA-32 Intel platform. So, to implement the memory management system in JikesNode, it's important to understand the underlying mechanism supported by Intel IA-32 processors.

The following will concentrate on the memory management implemented in IA-32 Intel. First, it describes the memory management facilities in protected-mode, including the segment and page mechanism. The second part is the definition of system registers and memory entries used in such facilities.

3.1. Overview

System architecture

Intel IA-32 architecture processors include the Intel Pentium processors, the P6 family processors, the Pentium 4 processors, the Intel Xeon™ processors, and the Pentium M processors [2]. It provides a set of registers, data structure and a series of instructions for the developer to perform the system-level operations such as memory management and task management. Figure 3-1 gives a summary of its registers and data structures.

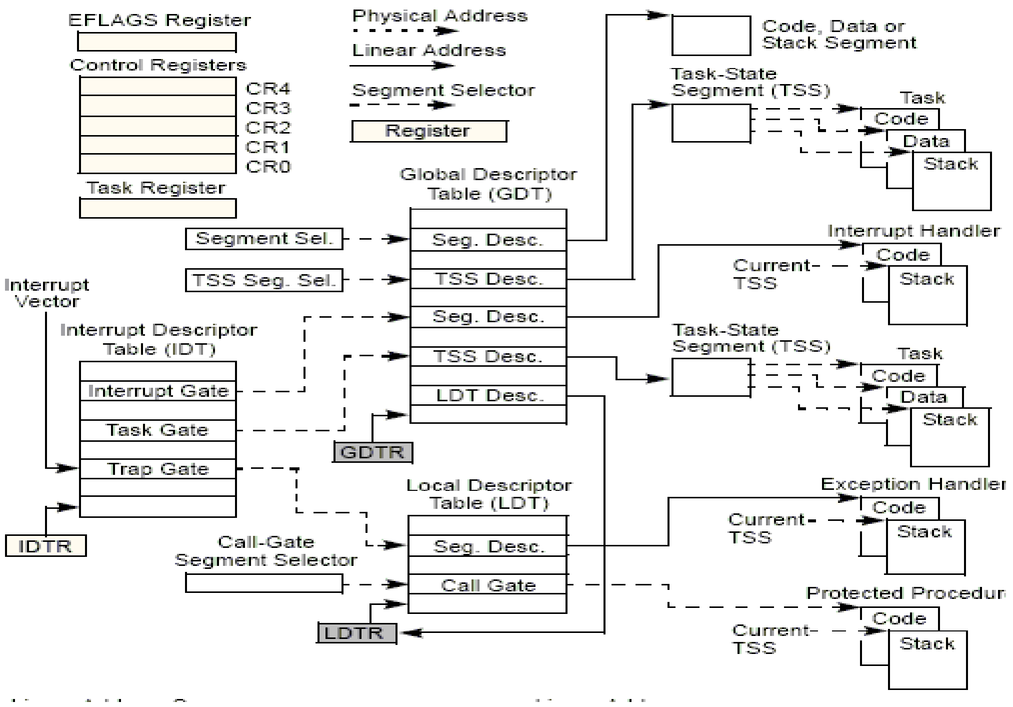

Figure 3-1. System Structure in Intel IA-32 [2, Figure 2.1]

Operation Modes

In IA-32 architecture, we have four operation modes, protected mode, real-address mode, system management mode (SMM) and virtual-8086 mode

- Protected mode is the native mode of IA-32 processors. It provides all instructions and features available in IA-32. The JikesNode operating system is also developed in this mode.

- Real-address mode is the initial mode that a processor is placed in when being powered up or reset. It provides an Intel 8086 programming environment for the user. Running in real-address mode, the processor only supports 1 MBytes (20 bits) physical address space [2, Chapter 16]. It shifts the segment selector left by 4 bits and adds the 16 bits effective address to form a physical address.

- System management mode (SMM) is a special operation mode used to monitor and manage the system resources. In this mode, processors use a separate address space call SMRAM to save the codes and data, whose default size is 64 KBytes and can be up to 4 GBytes.

- Virtual-8086 mode is a quasi-operating mode supported by processor. Actually it's a special task that runs in protected mode. It allows executing 8086 software in a protected and multitasking environment. The execution environment of virtual-8086 mode is the same as real-address mode, except the virtual-8086 can use some features in protected mode.

3.2. Memory management in protected mode

Overview

The memory management facilities of the IA-32 architecture consist of segmentation and paging two parts [2, Chapter 4]. Segmentation provides mechanism of isolating individual code, data and stack modules for multi tasks to run on a processor. Paging provides a mechanism to implement a virtual-memory system and map it into physical memory address. In protected mode, segmentation must be used, while paging can be disabled by setting particular bits of the registers.

As shown in the figure 3-1 and 3-2, segmentation divides the processor's memory space into small pieces of protected address space called segments. These segments can be used to save the code, data, stack, task states and some system data structure [2 Chapter 3]. In the segment phase, processor translates logical address to a linear address by using the segment selector and address offset

The paging mechanism can map a large linear address into a small real memory size. If paging is not used, processor maps the linear address directly to the physical address. When using paging, each segment is divided into pages, and a page directory and page table are used to translate the linear address to the physical address.

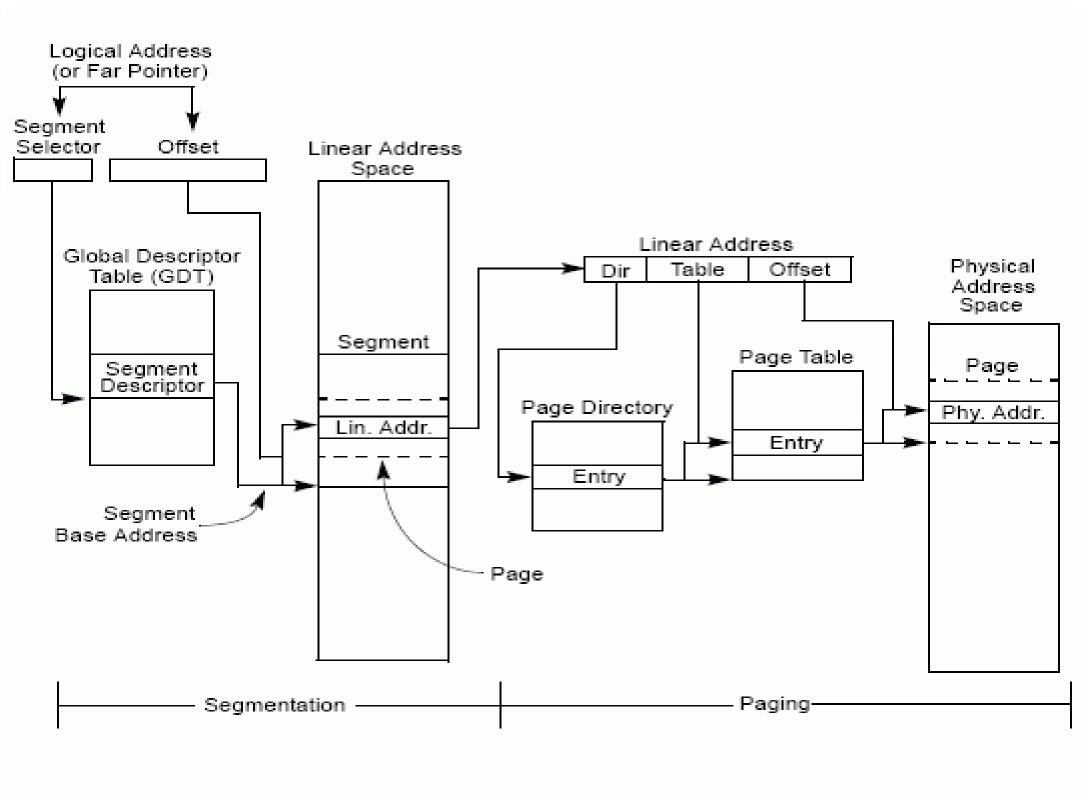

Figure 3-2. Address Translation

Segmentation

Segmentation Model

Segmentation supported by IA-32 architecture can by implemented by different ways, from basic flat model, protected flat model to multi-segment model.

- Basic flat model is the simplest memory model. In it, all data, codes and stack are located in a continuous and unsegmented address, which means the segment mechanism is hidden from the operating-system and application. To implement it, a code segment descriptor and a data segment descriptor should be created, and both must have the same base address of 0 and limit of 4 GBytes. All segment registers must be set to point to these two descriptors.
- Protected flat model is similar to the basic flat model, except it sets the segment limit to the real physical memory size. If necessary, this model can also be implemented more complex. For example, we can define different

segment with different access level.

- Multi-segment model uses the full capabilities of the segmentation mechanism to provide hardware enforced protection of code, data structures and programs and tasks [2, Chapter 3]. In this mode, every program or task has its own segments and these segments can be set as various private level.

In protected mode, to access a physical memory address, the processor must uses two stages: logical-address translation and linear address space paging [2, Chapter 3]. The segmentation is used in first stage.

When given a logical address (consisting of a segment selector and an offset), the processor first uses segment selector to locate the segment descriptor in the Global Descriptor Table (GDT) or Local Descriptor Table (LDT) and check the access rights. Then processor adds the base address in the selected segment descriptor to the offset to form a linear address. Figure 3-3 displays the translation

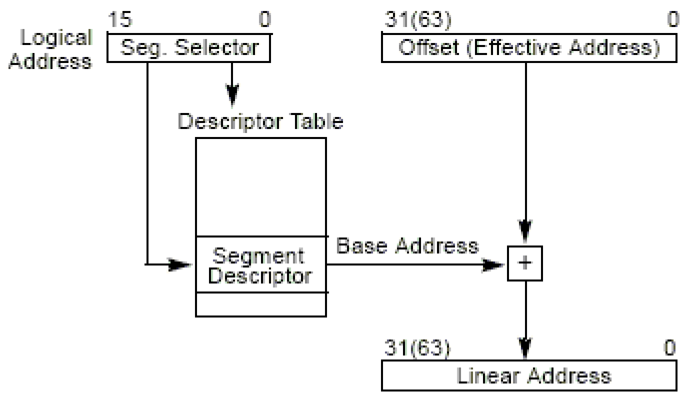

Figure 3-3. Logical-address Translation

Paging

After getting the linear address, the processor uses paging to map a linear address to a physical address. If paging is disabled, the linear address is the physical address. When paging is used, the processor translates the linear address into the physical address by using different mapping mechanisms according to various paging options.

By setting the flags in the control registers and system data structure, we can use different paging translation mechanism: 4-KByte with 32-Bit address, 4-MByte with 32-Bit address, 4-KByte with 36-Bit address, 4-MByte with 36-Bit address and 2-MByte with 36-Bit address. Following describe detailed implementation of these translations.

- **4-KByte page with 32-Bit physical address**

  This uses page directory and page table hierarchy to map linear address to 4KB pages. A linear address is divided into three parts: the lowest 12 bits define the offset in a page, next 10 bits are the page-table offset, and the bits 22 to 31 provide the page-directory offset.

  First, the processor reads the control register CR3 [Section 3.3] to get the base address of the page directory and adds the page-directory offset to get the base address of page table. Then the page-table offset is added to the base address of page table to get the base address of 4-KByte page. Finally physical address is created by adding the base address of the page to the page offset.

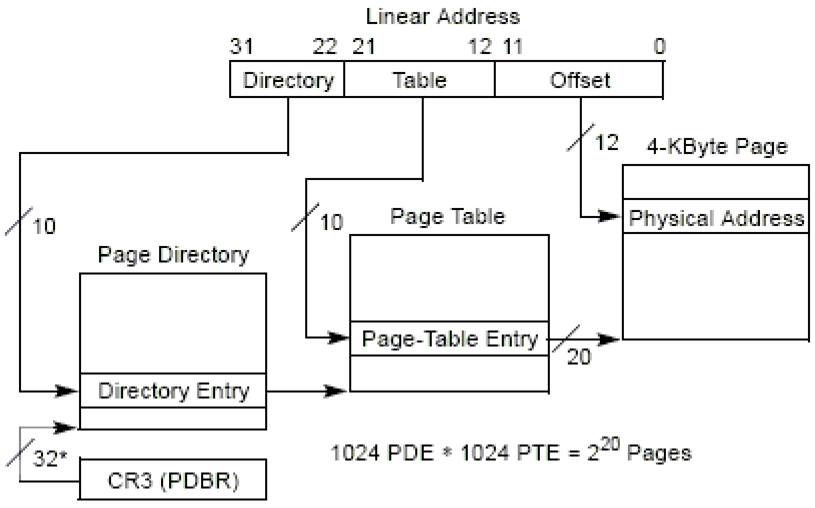

Figure 3-4. 4-KByte Page Translations

- **4-MByte page with 32-bit physical address**

This only uses the page directory to map linear addresses to 4-MByte pages. When using 4-MByte mapping, the linear addresses have two sections. The lower 22 bits are the page offset and the bits 22 to 31 define the offset in the page directory to get a page entry.

First, the processor adds the base address saved in CR3 to the page-directory offset to get the entry point of the page. Then the physical address is created by adding the base address of the page to the 22 bits offset.

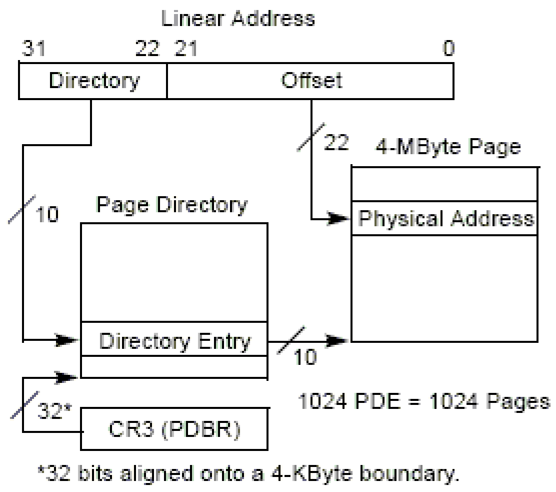

Figure 3-5. 4-MByte Page Translation

- **4-KByte page with 36-bit physical address**

  This uses page-directory-pointer, page-directory and page-table hierarchy and divides the linear address into four parts. The lowest 12 bits defines a page offset. Bits 12 to 20 are page-table offset. Bits 21 to 29 are page-directory offset. And the highest 2 bits provide a offset in the page-directory-pointer table. Figure 3-6 shows how the translation works

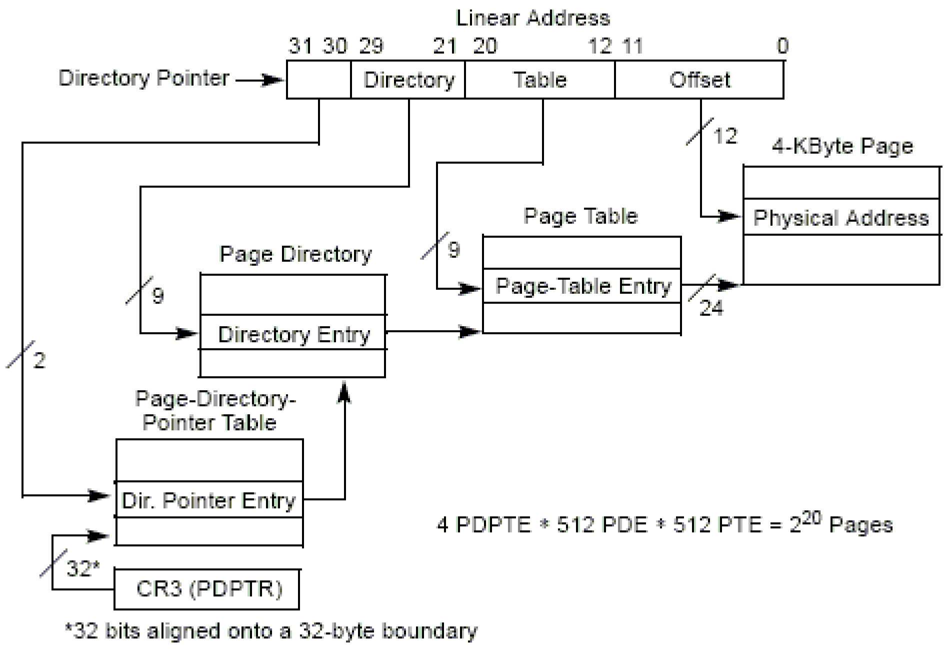

Figure 3-6. 4-KByte pages with extended 36 Bits address

- 2-MByte page with 36-bit physical address

  2MByte paging can only be used with 36bits physical address. To implement this paging mechanism, the processor uses the page-directory-pointer and page-directory. The linear address consists of three sections. The lowest 21 bits are the offset in 2-MByte page. Bits 21 to 29 provide an offset in the page directory. And bits 30 and 31 define the offset in the page-directory-pointer table. Figure 3-7 shows the mapping mechanism.

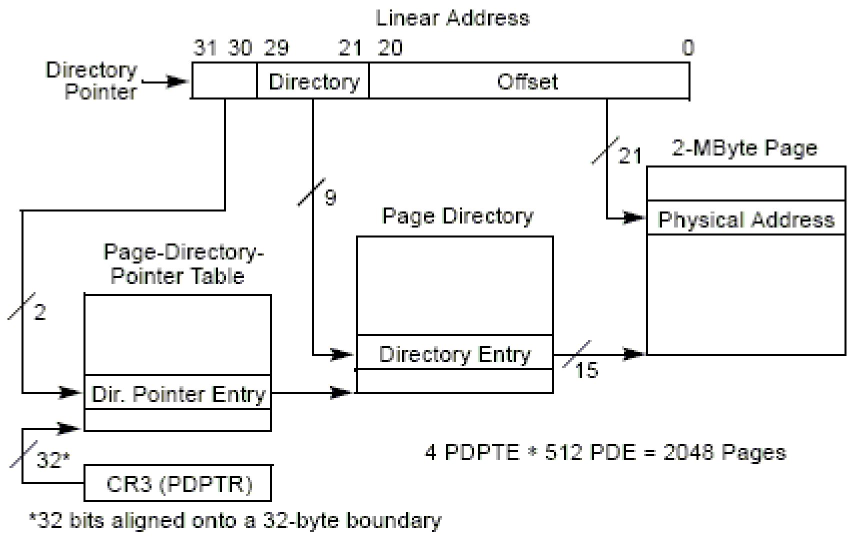

Figure 3-7. 2-MByte pages with extended 36 Bits address

- 4-MByte page with 36-bit physical address

  Its mapping mechanism is the same as the 4-MByte page with 32-bit address. The only difference between them is the data structure defined in the page directory.

3.3. System Registers and data structure

In IA-32 architecture, processor uses some system registers and data structures to perform the memory management facilities effectively. Here is a list of these registers and structures and a description of some important flags in them.

Control Register 0 (CR0)

Bit 31 of CR0 indicates whether paging is used. When it is set, paging is enabled while it is cleared to disable paging.

Control Register 3 (CR3)

When enabling paging, CR3 is used to save the base address of the page directory.

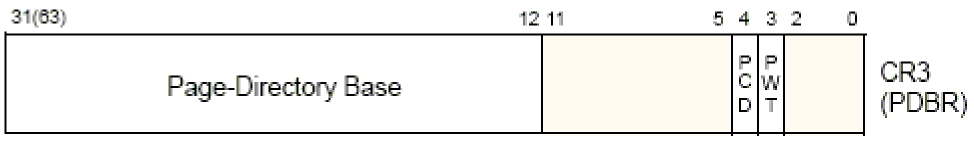

Figure 3-8. CR3

Control Register 4 (CR4)

Bit 4 of CR4 (PSG flag) indicates whether the 4MB page can be used. If set, the processor can use both 4MB and 4KB page, while only 4KB page can be used when it is cleared.

Bit 5 of CR4 is the flag of page address extension. It enables the 36-bit physical address when set.

Segment selector

Segment Selector is a 16-bit identifier used to find the segment descriptor in a descriptor table. Bits 3 to 15 provide the offset in a descriptor. Bit 2 directs the processor to search in Global Descriptor Table (GDT) or Local Descriptor Table (LDT). And the first 2 bits are used to specify the privilege level.

Segment Registers

To reduce address translation time and coding complexity, the IA-32 processor also provides 6 registers to hold the segment selector. Before accessing a segment, its segment selector must be loaded into one of these 6 register. So at any time, only 6 segments are available immediately. Each segment register consists of a “visible” part and a “hidden” part [2, Chapter 3]. The “visible” part saves the segment selector, while the “hidden” part looks like a cache, which saves the corresponding information such as base address and limit.

Segment Descriptor

A segment descriptor is a 64-bit data structure saved in GDT or LDT. It saves all the necessary information of the segment it points to. Figure 3-9 lists its data format.

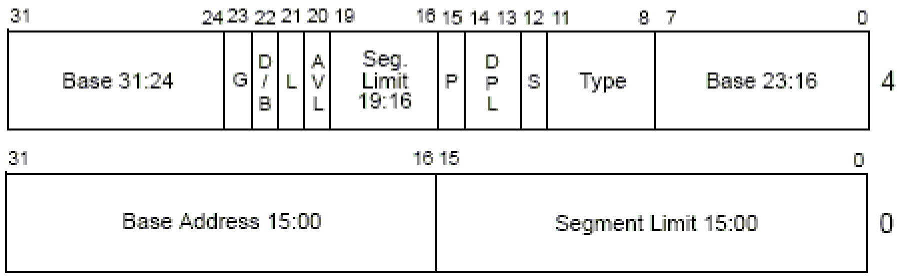

Figure 3-9. Segment Descriptor

It has a 32-bit address which points to the base address of the segment it describes, a 20-bit data defining the segment size and some flags.

Segment Descriptor Table

A segment descriptor table is used to save segment descriptors contiguously. It has two types: the global descriptor (GDT) and the local descriptor table (LDT).

Processor uses the GDTR register and the LDTR register to save the base address of the descriptor table. In GDT, the first descriptor (the first 8 bytes) must remain empty.

GDTR and IDTR

These two registers specify the locations of descriptor table. The GDTR refers to the GDT while IDTR refers to the IDT. Both registers are 48 bits, of which the highest 32 bits save the linear address of the descriptor table and the lower 16 bits indicate the limit size of the table.

Page directories and tables

When paging is enabled, page directories and tables are used to translate a linear address to a physical address and both save a series of entries. Depending on the paging mechanism, the page directory and table entries have different data formats.

When using 4Kbyte page, both page directory and table are used. Their data format are shown in Figure 3-10 and 3-11

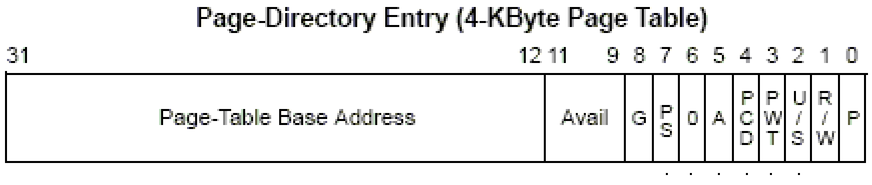

Figure 3-10. 4-KByte page-directory entry.

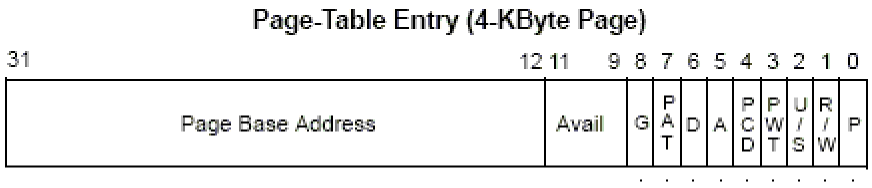

Figure 3-11. 4-KByte page-table entry

In the page-directory entry, the highest 20 bits provides the base address of the page table. The 7 bits indicates the page size, which must be set 0 in 4KB page.

In the page-table entry, the highest 20 bits describes the base address of the page. The other bits define the attribute of this page.

When using 4MByte page, only page directory will be used. Figure 3-12 describes the data format of an entry.

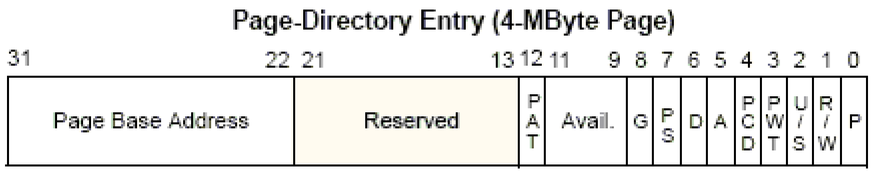

Figure 3-12. 4-MByte page-directory entry

The bits 22 to 31 indicate the base address of a page. It will be shifted left 22 bits to create the real 32-bit page base address during translating an address. The 7 bits (page size flag) should always be set 1.

Chapter 4
Memory system in JikesNode

Unlike other programming languages, Java has some distinctive features such as automatic memory management, support for multithreading, the existence of architecture-neutral intermediate codes (bytecodes), etc. [10]. With the increasing gap between the speeds of CPU and memory, the memory system has become a major performance bottleneck in modern computer systems [10]. As a new Java operating system, JikesNode needs an effective memory system to increase its performance.

In this Chapter, we describe the memory system implemented in JikesNode. First, we have a look at MMTk, the memory management component used in JikesRVM. Following MMTk, there is a detailed description of the memory initialization in the JikesNode nanokernel. Finally, we present the memory model in JikesNode without a Garbage Collector (GC).

4.1. MMTk (Memory Management Toolkit)

MMtk is a memory management toolkit written in and for Java. It is developed from the JMTk (Java Memory Management Toolkit), which is also written in Java and especially for JikesRVM. It provides a series of reusable, efficient, extensible components for garbage collectors. Now the JNode project has begun to integrate

MMTk into JNode system. So using MMTk as our memory management toolkit is quite a natural choice.

Policy

Policy contains the garbage collection algorithms that couple memory, which is grouped into spaces in MMTk, with an allocation and collection mechanism. A whole heap collector uses one policy for most objects, while a generational collector always uses one or more policies. For generational collector, a write barrier [11] is used to remember all references to the objects into an independently collected space. Before the program accesses the object, the barrier will be triggered and perform some necessary work before operating on the object.

Following is the basic allocation and collections mechanisms support by MMTk

- Bump Pointer Allocator: All memory is grouped in a contiguous space. A cursor called the bump pointer is used to record the start address of free memory. When creating a new object, the allocator appends the object from the cursor, and increment the cursor by the size of the created object.
- Free-List Allocator: The memory is organized into some size-segregated free-list so that all memory is divided into blocks. The block that has just size to accommodate the new object is used to save it.
- Tracing Collector: Uses a transitive closure from the roots to identify the live objects. When reclaiming space, MMTk moves the data out of the space, or frees untraced objects.
- Reference Counting Collector: A reference count is maintained for each object. When creating a new object, a reference to this object is also created and the object’s reference count is set to one. When the object is referenced by other values, its reference count is incremented. When a reference to the object is deleted or assigned a new value, its reference count is decremented. Whenever an object’s reference count is zero, it will be reclaimed.

In addition to these mechanisms, JikesRVM has supported some advanced real-time garbage collection called treadmill.

With the above mechanism, MMTk creates five policies.

- Copy space has a bump-pointer allocation and a tracing collection by moving live objects out of the space.
- MarkSweep space has a free-list allocation and a tracing collection that frees untraced object
- RefCount space has a free-list allocation and a reference counting collection.
- Immortal space: bump-pointer allocation and no collection.
- Large object space: coarse-grained free-list allocation and treadmill collection [20].

From these five policies, MMTk forms the following collectors.

- SemiSpace uses two copy spaces. Every time, one space is used to save objects. Once full, the live objects in the used space are copied to the other.
- MarkSweep uses one mark-sweep space. During allocating a new object, it traces and marks the live objects and reclaims dead objects.
- RefCount uses Refcount space, but the collection is deferred. During mutation, it buffers the counts of object references. The collector periodically processes the buffer, saves the changing for deferred objects, and then clears objects with a zero count.
- GenCopy: The classic copying generational collector [21] allocates into a (nursery) Copy space, and promotes survivors into an old SemiSpace [11]. When the nursery is full, it collects and sets the size of copy space to the size of all live objects. When the SemiSpace is full, it collects the entire space.
- GenMS is like GenCopy, excepting replacing the SemiSpace by MarkSweep.
- GenRC: uses Ulterior Reference Counting to combine a copying nursery with a RefCount mature space.[11]

Plan

MMTk defines collectors through the composition of policies and mechanism. Plans just perform the highest level of this composition, defining the rules by which policies are composed. The key functions are [4]:

- Identifying a virtual memory layout (using VMResources).
- Providing allocation by binding suitable allocators to different VMResources.
- Invoking collection when necessary through the use of a *polling* mechanism.
- Applying the appropriate collection policies to objects encountered during the collection process (objects may be subject to different collection regimens depending on where they reside in memory).
- Implementing read and write barriers if necessary.

In the latest version, MMTk implements eight different plans. In addition to the plans that simply compose the mentioned policies, another three are also implemented.
CopyMS composes a full-heap collector with a copying nursery and mark-sweep mature space. The collector has no write barrier and no remembered set
GenCopy implements a standard two-generation copying collector.
NoGC only has a simple allocator but no collectors

4.2. Object model

The term *object model* refers to the way objects (both in the specific sense of object oriented languages and the more general sense of heap objects in other languages) are laid out in memory, how their type is determined and how they are manipulated by the runtime and memory management systems.[13, 4.2.3]. From the memory manager, an object consists of a size and some metadata fields which are used for managing.

In java language, values are either primitive (e.g., int, double, etc.) or references to objects. JikesRVM divides all objects into two types, **arrays** which consist of a set of components and **scalars** which only have fields. These two types have different memory layout shown in figure 4-1 [14, Figure 1]. An array object grows up from its reference, while a scalar object grows down from its reference. Each object has a two-word object header to support various operations such as dynamic type checking, memory management, synchronization, hashing, etc.

One word of the header describes the status of objects. It is divided into three parts. The first part is used for locking. The second one holds the default hash value. The last one is for the memory management.

The other word is a reference to the Type Information Block (TIB) for the object's class [14]. A TIB is an array of object references. The first component describes the object's class such as its superclass and interface. The remainder is compiled code of the virtual methods of the class.

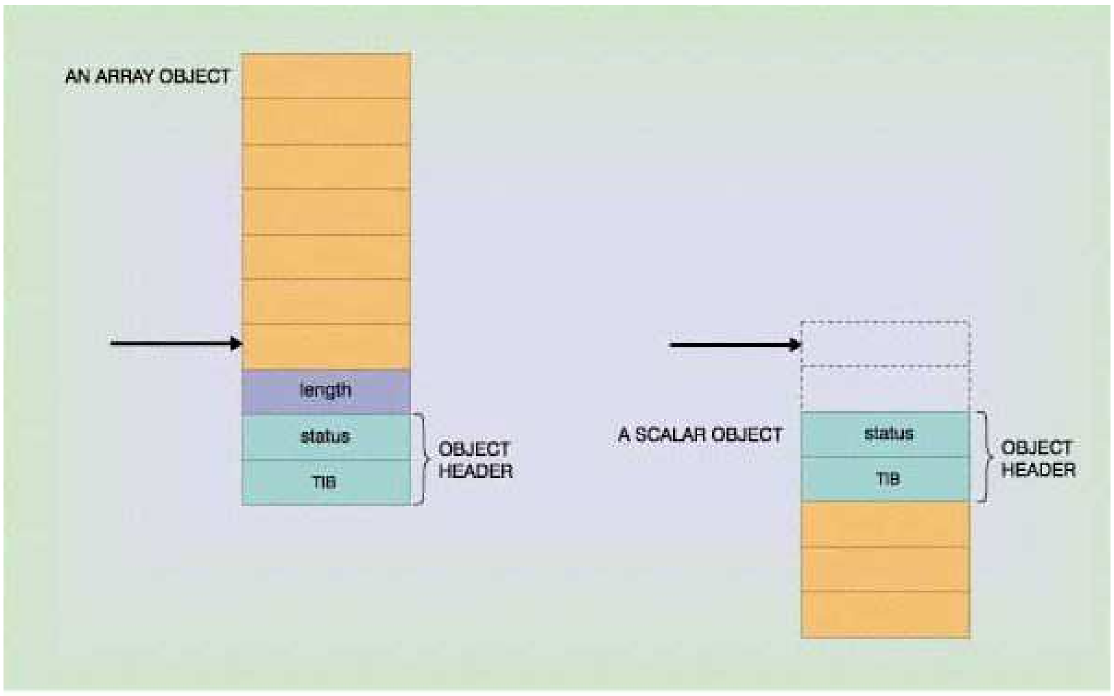

Figure 4-1. Layout of Object

4.3. Memory initialization

As mentioned in Chapter 2, during booting, the nanokernel does much work in initializing the memory management, for example the segment and paging, GDT and LDT tables and etc. Now, we will represent the initialization in detail.

In JikesNode, all initialization is done in protected mode, which means the bit 17 in EFLAGE [2, Section 2.3] must be cleared first. The whole process of memory initialization is implemented in three steps: paging initialization, GDT initialization and setting TSS (the LDT is not used now).

Paging initialization

Because a combination of 4-KByte and 4-MByte pages [Section 2.4] is used in the system, we must initialize both page directory and page tables during initialization. To enable the mixed paging mechanism, the PG flag (bit 31 of CR0) [Section 3.3] and PSE flag (bit 4 of CR4) [Section 3.3] should be set.

Currently, JikesNode supports maximum 4 GBytes memory space, which is paged by 4 MBytes. The first 4 MBytes memory uses the 4-KByte paging mechanism. The only exception is that the first 4-KBytes page (from address 0 to address 0x0FFF) is not present, in order to detect the nullpoint exception.

The remaining memory from the address 0x400000 to maximum 0xFFFFFFFF is set using 4MB pages. But the last 4MB page is not initialized, again to detect the nullpoint exception. In addition, those pages which hold the kernel (between the kernel_begin and kernel_end label) are marked as read-only. But all paging sets the virtual address equal to the real address.

During booting, a page directory and a page table are created. At first, a page directory is located at address 0x1000 and has 1024 entries, in which the first one is set 4KB paging (the PS flag in the entry is cleared) and the others are set 4MB paging (the PS flag is set). Then a page table is located at 0x2000 and set in the page directory. In this step, the page privilege level is set. After finishing initialization, we set the CR0 and CR4 to enable paging and load the address of page directory into CR3.

GDT initialization

There are six entries created in the GDT. The first entry is an empty entry which is never used. The other five include two data segments (Kernel Data Segment and User Data segment), two code segments (Kernel Code segment and User Code segment) and a TSS.

Both code segments and code segments are set 4GB limited and start from address 0x0. But the kernel segments are set to the highest privilege level while the user segments are set to the lowest.

TSS Setting

In protected mode, all programs execute within the context of a task [2 Chapter 2], of which the execution environment is defined in TSS. The structure of TSS is shown in Figure 4-2 [2, Figure6-2].

In the TSS, CS field is set in the user code segment and other segment fields are to the user data segment.

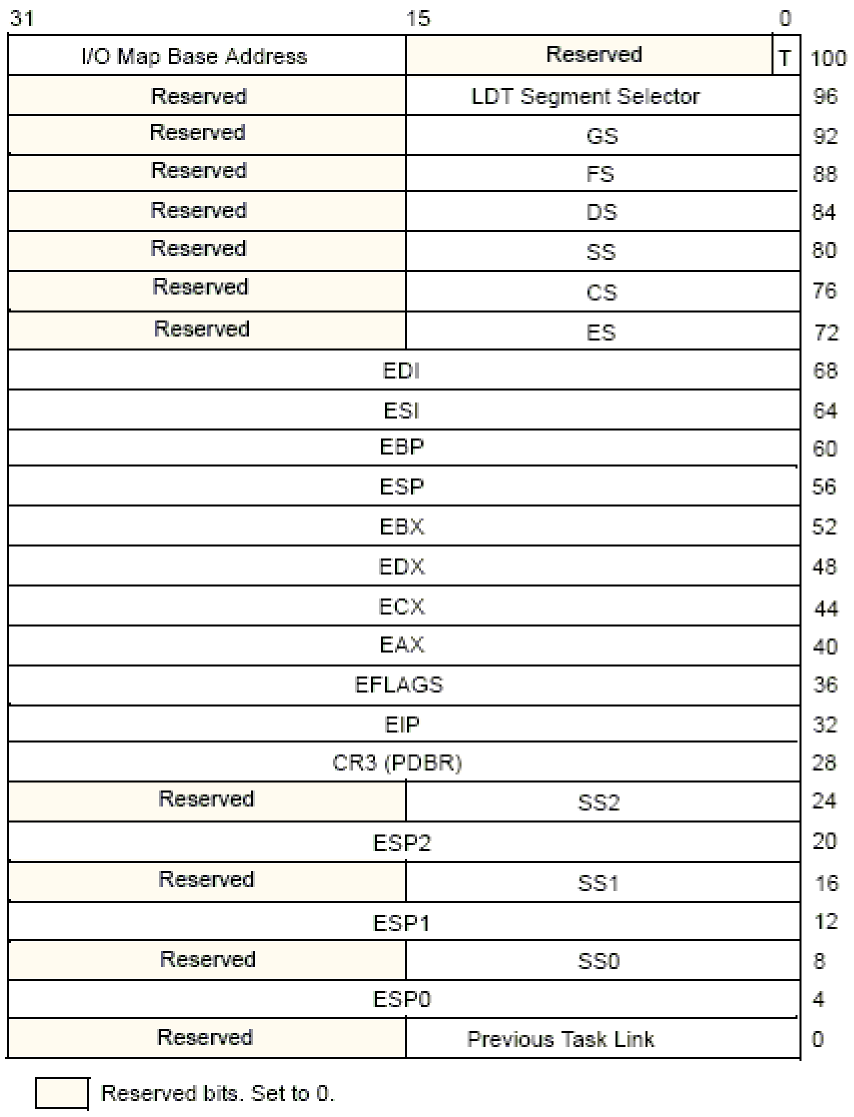

Figure 4-2. TSS Structure

An overview of the memory layout after initialization is shown in Figure 4-3 [2, Figure 6.2].

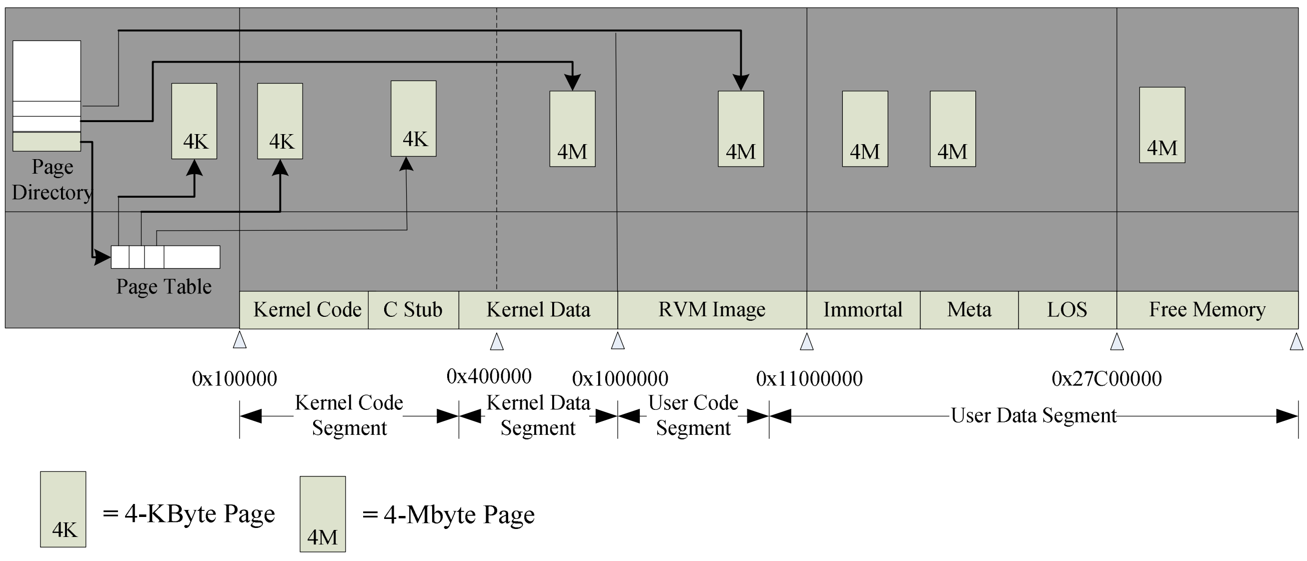

Figure 4-3. Memory layout

From the operation to the TSS and GDT, we find that JikesNode separates the memory to code segment and data segment. But because all segments are set the same start address and size, the operating system and programs are actually located in a continuous and unsegmented memory space.

4.4. Garbage Collection

In order to simplify the system, JikesNode doesn't use any Garbage Collection mechanism. So during the building phase, MMTk chooses the NoGC Plan.

In the NoGC plan, there are three policies used by the virtual machine and a policy for user data. Three policies for VM include an immortal space, a RawPage Space for the metadata of the VM, and a LargeObject (LOS) Space for some large objects. The user policy is immortal space, which only has a bump-point allocation and no Garbage Collection.

This allocation mechanism is simple but fast, only requiring a load, comparison and store. In addition, bump-point allocation supports a linear scan through both the allocated objects and a single contiguous space.

In JikesNode, the regions memory used by the policy is pre-set during the building phase. The size and address of user immortal space dynamically depends on the total size of the virtual memory defined in JikesRVM.

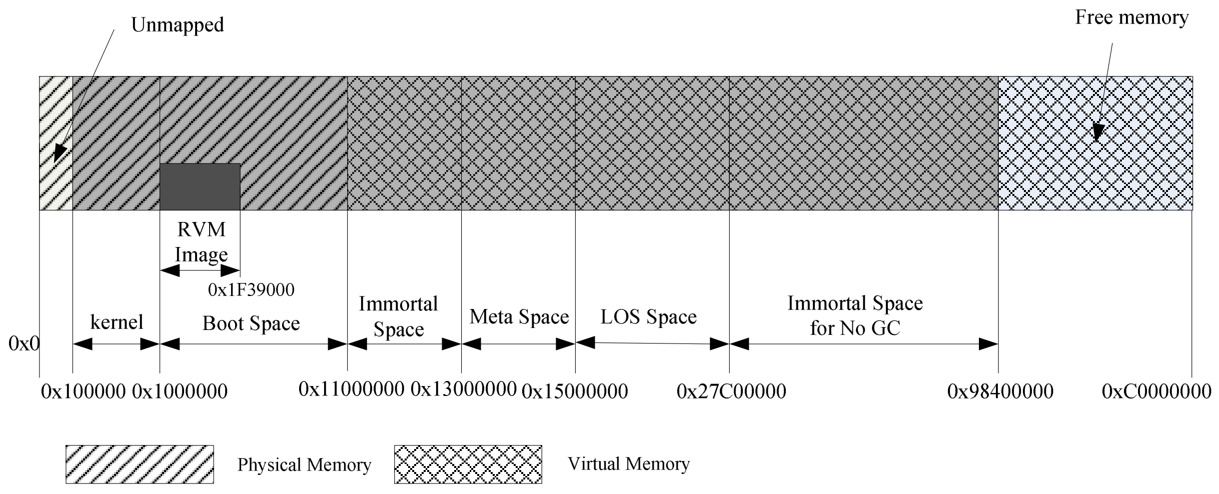

Figure 4-4. Memory space in NoGC

Chapter 5

Design and Implementation

5.1. Debug

After Georgios's work [1], the Jamaica group has done much work, such as forcing the RVM image to be saved from the address 0x1000000, initializing the necessary classes and libraries for the VM to load JNode and implementation of the klib functions. So at the beginning of the project, we must make the system work. During debugging the system, we find there are two main problems: overlap between the stack and the code, and the unfinished implementation of system functions.

Stack and code

When the system is running, sometimes the memory that holds the kernel code is re-written, which makes the system crash. After tracing the memory, we find that it is because the size of the user stack is too small and makes the stack value be saved into the code memory.

During initializing the system memory, the JikesNode creates two 16 KBytes memory spaces for kernel stack and user stack, after the kernel code. But some functions such as **vfprintf** will require more memory than 16 KBytes. Therefore stack region grows down and overwrites the code memory.

There are two methods to resolve the problem. The easy way is just extend the size of the stack. The other one is using a stack segment to separate the stack memory from the data memory and the code memory. Because current system only has code segments and data segments, the stack and other data share the same memory regions. In addition, the data segments and code segments are set to uses the same memory region. So actually, the boundary and privilege check in segments is not useful. If using a separated segment, the system will report an exception and prevent from writing to the memory in another segment.

Considering that the JikesNode is in an early stage, using many segments will make the further work more complex, and we choose the first method and extend both the user stack and the kernel stack to 64 KBytes

Klib function

Without system support, some functions in the system have not been implemented and force the system to stop. During debugging, the author modifies these functions just to check the status of the system. The following tables list the modified functions and codes.

Functions	Modifications	Reasons
gettimeofday (fake.c)	Return the value of fake_time instead of zero	The system uses this function to check the time gap in different boot stages. Return 0 will make the system halt.
setTimeSlicer (sys.C)	Modify “assert(false)” to “assert(true)”	“assert(false)” makes the system halt
finishbooting (VM.java)	Comment the code **runClassInitializer("java.lang.Math"), System.loadLibrary("javaio"), runClassInitializer("gnu.java.nio.channels .FileChannelImpl"), runClassInitializer("java.lang.Double"), runClassInitializer("java.lang.VMDouble "), runClassInitializer("com.ibm.JikesRVM. VM_Process") JikesRVMSocketImpl.boot();**	Need the support from JNI, which has not been implemented in the system.

Table 5-1. Modified functions

5.2. Import Garbage Collection

After debugging the system in NoGC, we start to import a garbage collection into JikesNode to make the memory in the system reusable. As listed in Section 4.1, MMTk used in JikesRVM supports eight plans for garbage collection. So we must choose a suitable plan before importing it.

Some papers [15, 16, 17] show that the generational collectors provide better performance than the whole heap collectors, such as semispace and marksweep, in virtually all circumstances [15, Chapter 1]. In the three generational collectors, GenMS has lower garbage collection costs than GenCopy because of its space efficiency and the implementation of GenRC is still immature [15 Section 5.4]. So finally, we use GenMS as the garbage collection in JikesNode.

GenMS

GenMS is a hybrid generational collector which uses a MarkSweep policy for the mature generation and a copying space for nursery. In its copying space, a bump pointer is used to trigger a nursery collection when the nursery is full. Normally, the nursery collection only works when either the nursery or the heap is full.

As in NoGC, in addition to its own space, there are three policies used by the virtual machine. The address and size of the MarkSweep policy and the copy space are pre-set during the building phase and depend on the size of virtual memory which is defined in configuration files. The space used for nursery copy space is located from the bottom of system memory and on the LOS space [Section 4.4]. The memory for the MarkSweep is located in the top area of the all memory and its start address is the maximum memory address minus its size. Figure 5-1 displays the overview of the space after building the system, whose memory is 0xC0000000

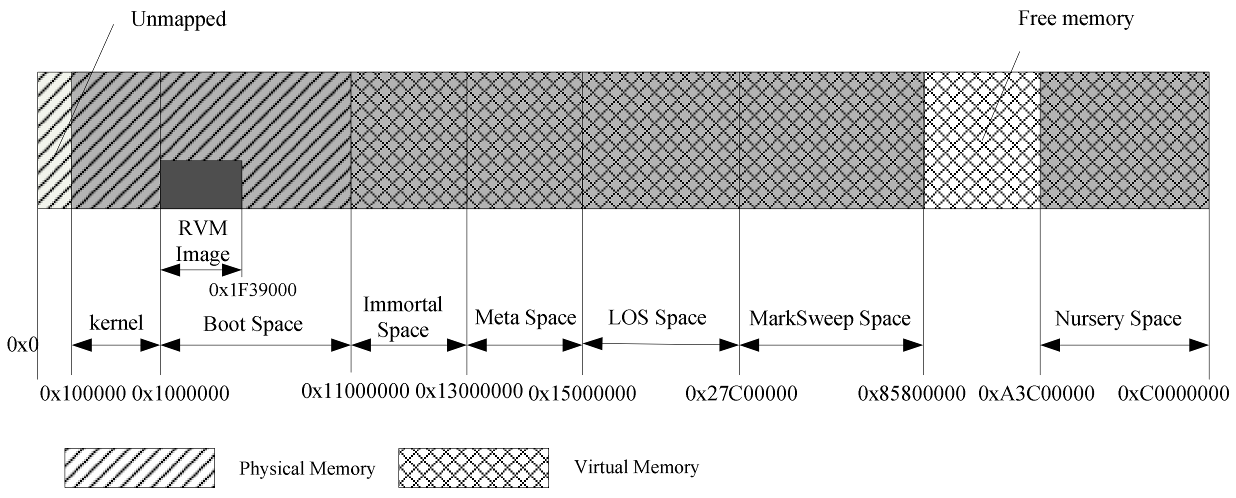

Figure 5-1. Memory space in GenMS

5.3. Memory Mapping

Using GenMS as our garbage collection leads to a new problem. Because the Marksweep space is allocated from the top of the memory and the size of virtual memory is usually much large than real memory, objects located in MarkSweep can not be saved to the real memory in current paging setting. So, when creating a new object, the page table or page directory must be modified to map the virtual address to available real address.

From the code, we find that when declaring or initializing a new object, the system calls **malloc** function to find a block of memory space for the object. And in **malloc** function, a function named **mmap** is called, which request the memory space from the kernel. So modifying the code in mmap to implement paging setting is sensible.

mmap

The **mmap** function establishes a mapping between a memory space and an operating resource. The resource can be a file, a shared memory object, or a typed memory object. This is the declaration of this function:

*void * mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset);*

- **start** indicates the start address of the memory to be mapped
- **length** specifies the size of the mapped memory, which is in bytes
- **prot** specifies the permission of the memory
- **flags** specifies attributes of the mapped region
- **fd** contains the file descriptor of the mapped object
- **offset** specifies the file byte offset at which the mapping starts

In these six parameters, the *prot* and *flags* are the most important, which will affect the property of the page and the mapping mechanism. Because the JikesNode is very simple now and many functions haven't been implemented, we only can implement some basic functions defined in mmap function (the full description described in [18]).

prot defines four access options:

- PROT_READ: Region can be read.
- PROT_WRITE: Region can be written.
- PROT_EXEC: Region can be executed.
- PROT_NONE: Region cannot be accessed

When prot is PROT_EXEC，it means the memory should be located in the memory in a code segment. But now all segments in JikesNode are defined in the same memory area. So the PROT_EXEC option is meaningless in current **mmap** function. So, only PROT_WRITE and PROT_READ options are implemented.

In *flags* parameter, we only implement the MAP_VARIABLE and MAP_FIXED options because the others need some operation on file system, which have not implemented in JikesNode. The definition of these two options is listed below.

- MAP_VARIABLE: When this option is set, system can select an address for the new memory, if the memory space indicated by the addr parameter can be mapped, or the addr parameter is null.
- **MAP_FIXED**: If this option is set, system must place the mapped space at the address specified by the **addr** parameter and replace all previous mappings for the pages located in the mapped memory.

munmap

With mmap, there is another function called **munmap**, which performs an opposite operation. It unmaps a mapped file region or anonymous memory region.

Implementation

To monitor system memory, we need some structures to save the status and layout of memory. A dynamic linked list and a static array are two choices. Basing on the following reasons, we use arrays.

- To create and initialize a linked structure, **mmap** function calls **malloc** function to allocate some memory, which will call mmap. So special care is needed to stop them being an endless loop.
- Currently, many operation systems or applications run with a large memory. So, compared to the size of system memory, the memory used for static arrays is not costly. In addition, for processor, operations on an array are simpler and more efficient than operations on a linked list.

We created two static arrays to log the free memory in the system, one named **free_regions** used for virtual memory and the other name **free_map_regions** used for physical memory. Another array is created to save the mapping between virtual memory and physical memory.

When called, **mmap** checks the **flags** parameter. If **MAP_FIXED** is set, all mapping for the virtual address located in the region [**start, start+ length**] is cleared. Then find a free physical memory from **free_map_regions**, which can match specified size to map to the specified size. The mapping virtual memory and physical memory are also removed from **free_regions** and **free_map_regions**. If **MAP_VARIABLE** is set, **mmap** first checks whether the specified memory region has been used. If it has not been used, the specified virtual address is used and mapped to the available physical memory. If the memory has been used or no memory is specified, **mmap** finds another available memory region from **free_regions** and **free_map_regions** and maps them. For the other conditions, any available virtual memory and physical memory can be used. Figure 5-2 shows the operation sequence of mmap.

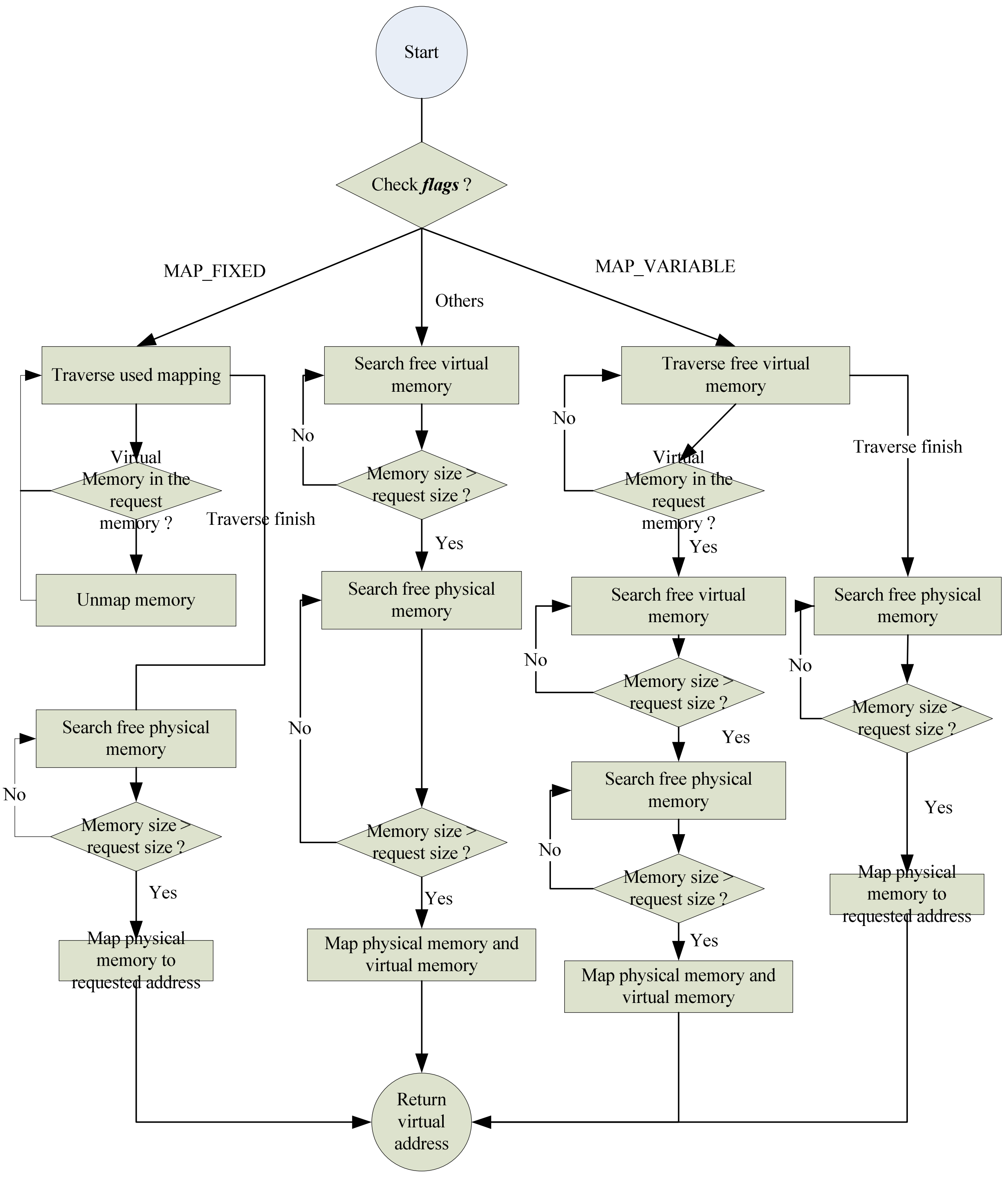
Start
Check *flags* ?
MAP_FIXED
Others
MAP_VARIABLE
Traverse used mapping
Search free virtual memory
Traverse free virtual memory
No
No
Virtual Memory in the request memory ?
Memory size > request size ?
Virtual Memory in the request memory ?
Traverse finish
Traverse finish
Yes
Yes
Unmap memory
Search free physical memory
Search free virtual memory
Search free physical memory
No
No
No
Memory size > request size ?
Memory size > request size ?
Yes
Search free physical memory
No
Search free physical memory
Yes
No
Memory size > request size ?
Memory size > request size ?
No
Yes
Memory size > request size ?
Map physical memory to requested address
Yes
Yes
Map physical memory to requested address
Yes
Map physical memory and virtual memory
Map physical memory and virtual memory
Return virtual address

Figure 5-2. Logic Diagram in mmap

5.4. Paging

In Section 2.3 and Section 4.3, we have described the memory management in

JikesNode. The mixing of 4-KByte and 4-MByte pages is used in the system and the first 4 MB memory uses 4-KByte pages. Currently, JikesNode sets the kernel start from 0x100000 and the JikesRVM image start from 0x1000000. Between kernel and JikesRVM image, there are the C stub functions and Stack. Therefore, the kernel and system components are placed in both 4-KByte page and 4-MByte page.

To reduce the TLB [2, Section 10.9] misses and improve overall system performance, the system operating system and kernel should be placed in a large page [2 Section 3.7.3]. But now all kernel and system is in a small page, so we change the first 4 MB memory to use 4-MByte page.

This modification is not very complex. The first thing is to remove the code that initializes 4-KByte pages and sets them in page directory. Then we should add some code to check and set the read/write privileges for the pages, which is originally implemented in page table setting. All pages holding the kernel should set read-only, while others are read and write.

Chapter 6
Summary

Research on Operating system is as old as computing itself, but Java Operating System is still a new researching area. The JikesNode aims to provide user a higher performance, dynamically optimising, operating system architecture.

This project is to do some improvement on the memory management subsystem in JikesNode. We import the GenMS garbage collection provided by JikesRVM into the system and modify the memory mapping and initialization mechanism. Because currently the size of physical memory is much smaller than the size of memory required by GenMS, we can’t trigger a nursery collection now. But from the output in building phase [Appendix A] and the location of object during running [Appendix B], we can find that the GenMS garbage collection works.

Future Work

With only 4 months, it’s impossible to implement all JikesNode components. By now, JikesRVM is successfully loaded and many classes can be initialized. From the author’s point of view, in the future, the main work is to extend system stub functions and integrate JNode into JikesRVM.

- JNI mechanism should be set up after booting the VM, so that the java system can utilize the native function in the java library. JikesRVM has implemented a JNI package to interact with native code. Modification of JNode to use the JNI package is critical.

- Many C methods created in klib to replace the C stub in JikesRVM. In addtition, these methods also define a set of callbacks which provide a bridge between the java call and underlying system functions. The definition and description of these methods can be found in [9]. In addition, JikesRVM uses VM_Syscall to invocate the C functions, while JNode uses **Unsafe** class to perform hardware access. How to unify these two ways is an important task.
- A C function has been created for interrupt handler and IRQ handler. Much work should be done to implement this function and transfer the interrupt or IRQ to the JNode IRQ management subsystem. This would probably require changing the VM_Runtime class to provide an entry point for system call.
- How to build JNode boot image and make JikesRVM to load this boot image after finish booting VM. One solution is to build two images together and JikesRVM directly intiliaze JNode after finishing booting VM. But using this way, the image file will be very large and take a long time to load before booting. Another technique is to build JNode to a Jar file, specify the file as JikesRVM command line and load it after booting VM.

Bibliography

[1] G. I. Gousios: JIKESNODE: A JAVA OPERATING SYSTEM, MSc thesis, Dept. of Computer Science, University of Manchester, 2004.

[2] Intel Corporation: IA-32 Intel® Architecture Software Developer's Manual Volume 3: System Programming Guide, 2005

[3] The Jnode operating system, 2004. http://jnode.sourceforge.net.

[4] IBM. The Jikes Research Virtual Machine User's guide, 2004. Manual accompanying the Jikes RVM source distribution. http://jikesrvm.sourceforge.net/

[5] The Jikes Research Virtual Machine (RVM), http://oss.software.ibm.com/developerworks/oss/Jikes RVM/.

[6] I. Rogers and C. Kirkham: JikesNode and PearColator: A Jikes RVM Operation System and Legacy Code Execution Environment, 2nd ECOOP Workshop on Programming Languages and Operating Systems (ECOOP-PLOS'05), Glasgow, July 26, 2005.

[7] B. Alpern, C. R. Attanasio and J. J. Burton: The Jalapeño virtual machine. IBM System Journal, Vol 39, No 1, February 2000.

[8] B. Alpern, S. Augart, S.M. Blackburn, M. Butrico, A. Cocchi, P Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K.S. McKinley, M. Mergen, J.E.B. Moss, T. Ngo, V.

Sarkar, and M. Trapp: The Jikes Research Virtual Machine project: Building an open-source research community. IBM Systems Journal, Vol 44, No 2, 2005.

[9] The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition http://www.opengroup.org/onlinepubs/009695399/frontmatter/preface.html

[10] J.-S. Kim and Y. Hsu: Memory System Behavior of Java Programs: Methodology and Analysis, ACM SIGMETRICS 2002

[11] S. M. Blackburn, P. Cheng and K. S. McKinley: Oil and Water? High Performance Garbage Collection in Java with MMTk, In Proceedings of ICSE 2004, 26th International Conference on Software Engineering

[12] R. E. Jones and R.D. Lins: Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Wiley, July 1996.

[13] R. J, Garner: JMTk: A Portable Memory Management Toolkit. Honours thesis,.Computer Science, ANU. 2003

[14] IBM Staff: The Jikes Research Virtual Machine (RVM) Independently developed as part of the Jalapeno research project at Thomas J. Watson Research Center. 01 Feb 2000. http://www-128.ibm.com/developerworks/java/library/j-jalapeno/index.html#h6

[15] S. M. Blackburn, P. Cheng and K. S. McKinley: Myths and Realities:The Performance Impact of Garbage Collection, Technical Report TR-CS-04-04, Dept. of Computer Science, Australian National University, 2004.

[16] M. Hertz, Y. Feng and E. D. Berger: PageLevel Cooperative Garbage Collection, Technical Report 2004 UMass CS TR-04-16:

[17] M. Hertz, Y. Feng and E. D. Berger: Garbage Collection Without Paging, PLDI 2005

[18] Technical Reference: Base Operating System and Extensions, Volume 1 http://publib16.boulder.ibm.com/pseries/en_US/libs/basetrf1/basetrf102.htm#wq3521

[19] B. Venners: Inside the Java Virtual Machine published by McGraw-Hill Companies, December 1997

[20] H. G.. Baker: The Treadmill: Real-Time Garbage Collection Without Motion Sickness, SIGPLAN Notices 27(3):66-70, March 1992

[21] A. W. Appel. Simple generational garbage collection and fast allocation, Software Practice and Experience, 19(2):171–183, 1989.

[22] GNU GRUB manual http://www.gnu.org/software/grub/manual/grub.html

Append A

A sample run output

```
RVMmodule: mod_start = 0x1000000, mod_end = 0x1f38534 mod_size=15585kb cmdline=
• P
Kernel end: 0x1f39000
Memory map provided by grub
  base_addr=0x0, length_low=0x9fc00, type = 0x1
  base_addr=0x100000, length_low=0xff00000, type = 0x1
mmap-length_low=0xff00000, mmap_base_addr_low=0x100000 FreeMem: start=0x1f39000
end=0x10400000 size=234268kb usable pages=6969
RunBootImage.main(): VM variable settings
initialHeapSize 20971520
maxHeapSize 104857600
rvm_singleVirtualProcessor 1
bootFileName |JikesNODE|
lib_verbose 1
IA32 jnode build for single virtual processor
Boot record contents:
      bootImageStart:          0x1000000
      bootImageEnd:              0x1f38480
      initialHeapSize:         0x0
      maximumHeapSize:           0x0
      tiRegister:              0x40000
      spRegister:               0x1c3ace0
      ipRegister:              0x14d60e0
      tocRegister:             0x10001d4
      sysWriteCharIP:           0x0
```

```
    ...etc...
post linkage
Boot record contents:
    bootImageStart:          0x1000000
    bootImageEnd:              0x1f39000
    initialHeapSize:         0x1400000
    maximumHeapSize:          0x6400000
    tiRegister:                0x40000
    spRegister:                 0x40000
    spRegister:                 0x1c3ace0
    ipRegister:                0x14d60e0
    tocRegister:               0x10001d4
    sysWriteCharIP:           0x102282
    ...etc...
JikesNODE: here goes...
JikesNODE: here goes2...
Booting
Setting up current VM_Processor
Doing thread initializProcessor
Doing thread initialization
Setting up write barrier
Setting up memory manager: bootrecord = 0x0100000c
i386.c
```

mmap(start=0x11000000, lenght=1048576, prot=7, flags=50, fd=-1, offfset=0

)

free_map_region[0].start=3400000;free_map_region[0].end=10000000

mapping[1].vp_start_address=0x11000000,

vp_end_address=0x11400000;mapped_start=0

x3000000 ,mapped_end=0x3400000

mmap(start=0xa3c00000, lenght=1048576, prot=7, flags=50, fd=-1, offfset=0

```
)free_map_region[0].start=3800000;free_map_region[0].end=10000000
mapping[1].vp_start_address=0x11000000,
vp_end_address=0x11400000;mapped_start=0
x3000000 ,mapped_end=0x3400000
mapping[2].vp_start_address=0xa3c00000,
vp_end_address=0xa4000000;mapped_start=0
x3400000 ,mapped_end=0x3800000
Garbage Collection being used now
$Id: GenMs.java,v 1.4 2005/05/25 14:58:04 irogers Exp $
Create two objects, should be located in Nursery Space
String one
0xa3c00028
String two
0xa3c00038
Stage one of booting VM_Time
Initializing baseline compiler options to defaults
java.lang.Throwable
mmap(start=0x13000000, lenght=1048576, prot=7, flags=50, fd=-1, offfset=0
)free_map_region[0].start=3c00000;free_map_region[0].end=10000000
mapping[1].vp_start_address=0x11000000,
vp_end_address=0x11400000;mapped_start=0
x3000000 ,mapped_end=0x3400000
mapping[2].vp_start_address=0xa3c00000,
vp_end_address=0xa4000000;mapped_start=0
x3400000 ,mapped_end=0x3800000
mapping[3].vp_start_address=0x13000000,
vp_end_address=0x13400000;mapped_start=0
x3800000 ,mapped_end=0x3c00000
java.util.zip.ZipEntry
com.ibm.JikesRVM.classloader.VM_MemberReference
```

```
java.net.URL
java.util.TimeZone
com.ibm.JikesRVM.jni.VM_JNIEnvironment
java.util.ResourceBundle
gnu.java.net.protocol.jar.Connection$JarFileCache
java.lang.Thread
com.ibm.JikesRVM.VM_EdgeCounts
com.ibm.JikesRVM.classloader.VM_Type
com.ibm.JikesRVM.classloader.VM_InterfaceMethodSignature
com.ibm.JikesRVM.jni.VM_JNICompiler
java.lang.System
com.ibm.JikesRVM.VM_DynamicLibrary
com.ibm.JikesRVM.VM_CompiledMethods
java.util.Locale
java.lang.Math
java.util.Calendar
com.ibm.JikesRVM.VM_Statics
java.net.URLConnection
gnu.java.nio.charset.Provider
com.ibm.JikesRVM.classloader.VM_TypeReference
com.ibm.JikesRVM.classloader.VM_TableBasedDynamicLinker
com.ibm.JikesRVM.classloader.VM_Atom
com.ibm.JikesRVM.VM_StackTrace
Fetching command-line arguments
Early stage processing of command line
Collector processing rest of boot options
Initializing bootstrap class loader
Stage two of booting VM_Time
Running various class initializers
running class intializer for gnu.classpath.SystemProperties
```

```
mmap(start=0x15000000, lenght=1048576, prot=7, flags=50, fd=-1, offfset=0
)free_map_region[0].start=4000000;free_map_region[0].end=10000000
mapping[1].vp_start_address=0x11000000, vp_end_address=0
x11400000;mapped_start=0x3000000 ,mapped_end=0x3400000
mapping[2].vp_start_address=0xa3c00000,
vp_end_address=0xa4000000;mapped_start=0
x3400000 ,mapped_end=0x3800000
mapping[3].vp_start_address=0x13000000,
vp_end_address=0x13400000;mapped_start=0
x3800000 ,mapped_end=0x3c00000
mapping[4].vp_start_address=0x15000000,
vp_end_address=0x15400000;mapped_start=0
x3c00000 ,mapped_end=0x4000000
running class intializer for java.lang.Runtime
running class intializer for java.lang.System
running class intializer for java.lang.Void
running class intializer for java.lang.Boolean
running class intializer for java.lang.Byte
running class intializer for java.lang.Short
running class intializer for java.lang.Number
running class intializer for java.lang.Integer
running class intializer for java.lang.Long
running class intializer for java.lang.Float
running class intializer for java.lang.Character
running class intializer for java.util.WeakHashMap
running class intializer for java.lang.ThreadGroup
running class intializer for java.lang.ThreadLocal
running class intializer for java.security.VMAccessController
running class intializer for java.io.File
running class intializer for gnu.java.lang.SystemClassLoader
running class intializer for java.lang.String

running class intializer for java.lang.VMString

running class intializer for gnu.java.security.provider.DefaultPolicy

running class intializer for java.net.URL

running class intializer for java.net.URLClassLoader

running class intializer for gnu.java.net.protocol.jar.Connection$JarFileCache

running class intializer for java.lang.ClassLoader$StaticData

running class intializer for gnu.java.io.EncodingManager

running class intializer for java.nio.charset.CharsetEncoder

running class intializer for java.nio.charset.CoderResult

running class intializer for java.io.PrintWriter

running class intializer for java.io.PrintStream

running class intializer for java.util.SimpleTimeZone

running class intializer for java.util.Locale

running class intializer for java.util.Calendar

running class intializer for java.util.GregorianCalendar

running class intializer for java.util.ResourceBundle

running class intializer for java.util.zip.Inflater

running class intializer for java.util.zip.DeflaterHuffman

running class intializer for java.util.zip.InflaterDynHeader

running class intializer for java.util.zip.InflaterHuffmanTree

running class intializer for gnu.java.locale.Calendar

running class intializer for java.util.Date

Booting VM_Lock

Booting scheduler

Using a time-slice of 20 ms

Initializing JNI for boot thread

Running late class initializers

running class intializer for java.io.FileDescriptor

running class intializer for java.util.PropertyPermission
```

```
VM is now fully booted

Initializing runtime compiler

Late stage processing of command line

[VM booted]

Initializing socket factories

Extracting name of class to execute

vm: Please specify a class to execute.

vm:     You can invoke the VM with the "-help" flag for usage information.

JikesNODE: exit 100

Exit no=100

Halted.
```

Appendix B
Some building output

```
/usr/lib/java/bin/javac -encoding iso-8859-1 -nowarn –g
-classpath .:/home/M04/acs/zhangya/rvmBuild-antigua/RVM.classes:/home/M04/acs/zhangya
/rvmBuild-antigua/RVM.classes/rvmrt.jar:/home/M04/acs/zhangya/rvmBuild-antigua/RVM.c
lasses/mmtk.jar
-bootclasspath .:/home/M04/acs/zhangya/rvmBuild-antigua/RVM.classes:/home/M04/acs/zha
ngya/rvmBuild-antigua/RVM.classes/rvmrt.jar:/home/M04/acs/zhangya/rvmBuild-antigua/R
VM.classes/mmtk.jar GenerateInterfaceDeclarations.java
/usr/lib/java/jre/bin/java -Xmx200M –classpath
  .:/home/M04/acs/zhangya/rvmBuild-antigua/RVM.classes:/home/M04/acs/zhangya/rvmBuil
d-antigua/RVM.classes/rvmrt.jar:/home/M04/acs/zhangya/rvmBuild-antigua/RVM.classes/m
mtk.jar GenerateInterfaceDeclarations -ia 0x1000000 -out
/home/M04/acs/zhangya/rvmBuild-antigua/RVM.scratch/InterfaceDeclarations.h
/**********Following output is implemented in the source code***********
Construct a memory space
Space name=boot
Start Address=16777216
Size=268435456
Construct a memory space
Space name=immortal
Start Address=285212672
Size=33554432
Construct a memory space
Space name=meta
Start Address=318767104
```

```
Size=33554432
Construct a memory space
Space name=los
Start Address=352321536
Size=314572800
Construct a memory space
Space name=nursery
Start Address=2747269120
Size=473956352
JikesNode uses GenMS garbage colleciton
Construct a memory space
Space name=ms
Start Address=666894336
Size=1572864000
Construct GenMS
/****************** END ****************
(wrote interface) 15 s
jbuild.linkImage: (bootimage cleaned)
```