A Guide to Nachos 5.0j

Dan Hettena

Rick Cox

rick@rescomp.berkeley.edu

We have ported the Nachos instructional operating system [1 (http://http.cs.berkeley.edu/~tea/nachos/nachos.ps)] to Java, and in the process of doing so, many details changed (hopefully for the better). [2 (http://www.cs.duke.edu/~narten/110/nachos/main/main.html)] remains an excellent resource for learning about the C++ versions of Nachos, but an update is necessary to account for the differences between the Java version and the C++ versions.

We attempt to describe Nachos 5.0j in the same way that [2 (http://www.cs.duke.edu/~narten/110/nachos/main/main.html)] described previous versions of Nachos, except that we defer some of the details to the Javadoc-generated documentation. We do not claim any originality in this documentation, and freely offer any deserved credit to Narten.

Table of Contents

1. Nachos and the Java Port

The Nachos instructional operating system, developed at Berkeley, was first tested on guinea pig students in 1992 [1 (http://http.cs.berkeley.edu/~tea/nachos/nachos.ps)]. The authors intended it to be a simple, yet realistic, project for undergraduate operating systems classes. Nachos is now in wide use.

The original Nachos, written in a subset of C++ (with a little assembly), ran as a regular UNIX process. It simulated the hardware devices of a simple computer: it had a timer, a console, a MIPS R3000 processor, a disk, and a network link. In order to achieve reasonable performance, the operating system kernel ran natively, while user processes ran on the simulated processor. Because it was simulated, multiple Nachos instances could run on the same physical computer.

1.1. Why Java?

Despite the success of Nachos, there are good reasons to believe that it would be more useful in Java:

- Java is much simpler than C++. It is not necessary to restrict Nachos to a subset of the language; students can understand the whole language.
- Java is type-safe. C++ is not type-safe; it is possible for a C++ program to perform a legal operation (e.g. writing off the end of an array) such that the operation of the program can no longer be described in terms of the C++ language. This turns out to be a major problem; some project groups are unable to debug their projects within the alotted time, primarily because of bugs not at all related to operating systems concepts.
- It is much more reasonable to machine-grade a Java project than a C++ project.
- Many undergraduate data structures classes, including the one at Berkeley, now use Java, not C++; students know Java well.
- Java is relatively portable. Nachos 4.0 uses unportable assembly to support multithreading. Adding a new target to Nachos 4.0 required writing a bit of additional code for the port.

1.2. Will it work?

One of the first concerns many people have about Java is its speed. It is an undebatable fact that Java programs run slower than their C++ equivalents. This statement can be misleading, though:

- Compiling is a significant part of the Nachos 4.0 debug cycle. Because javac compiles as much as it can everytime it is invoked, Nachos 5.0j actually compiles faster than Nachos 4.0 (running on a local disk partition with no optimizations enabled).
- Generating large files on network partitions further slows down the debug cycle. Nachos 5.0j's .class files are significantly smaller than Nachos 4.0's .o files, even when compiling with -Os. This is in part due to C++ templates, which, without a smart compiler or careful management, get very big.
- Type-safe languages are widely known to make debugging cycles more effective.

Another common concern is that writing an operating system in a type-safe language is unrealistic. In short, it *is* unrealistic, but not as unrealistic as you might think. Two aspects of real operating systems are lost by using Java, but neither are critical:

- Since the JVM provides threads for Nachos 5.0j, the context switch code is no longer exposed. In Nachos 4.0, students could read the assembly code used to switch between threads. But, as mentioned above, this posed a portability problem.
- The kernel can allocate kernel memory without releasing it; the garbage collector will release it. In Linux, this would be similar to removing all calls to `kfree`. This, however, is conceptually one of the simplest forms of resource allocation within the kernel (there's a lot more to Linux than `kmalloc` and `kfree`). The Nachos kernel must still directly manage the allocation of physical pages among processes, and must close files when processes exit, for example.

2. Nachos Machine

Nachos simulates a real CPU and harware devices, including interrupts and memory management. The Java package nachos.machine provides this simulation.

2.1. Configuring Nachos

The nachos simulation is configured for the various projects using the `nachos.conf` file (for the most part, this file is equivelant to the BIOS or OpenFirmware configuration of modern PCs or Macintoshes). It specifies which hardware devices to include in the simulation as well as which Nachos kernel to use. The project directories include appropriate configurations, and, where neccessary, the project handouts document any changes to this file required to complete the project.

2.2. Boot Process

The nachos boot process is similar to that of a real machine. An instance of the nachos.machine.Machine class is created to begin booting. The hardware (Machine object) first initializes the devices including the interrupt controller, timer, elevator controller, MIPS processor, console, and file system.

The Machine object then hands control to the particular AutoGrader in use, an action equivelant to loading the bootstrap code from the boot sector of the disk. It is the AutoGrader that creates a Nachos kernel, starting the operating system. Students need not worry about this step in the boot process - the interesting part begins with the kernel.

A Nachos kernel is just a subclass of nachos.machine.Kernel. For instance, the thread project uses nachos.threads.ThreadedKernel (and later projects inherit from ThreadedKernel).

2.3. Nachos Hardware Devices

The Nachos machine simulation includes several hardware devices. Some would be found in most modern computers (e.g. the network interface), while others (such as the elevator controller) are unique to Nachos. Most classes in the `machine` directory are part of the hardware simulation, while all classes outside that directory are part of the Nachos operating system.

2.3.1. Interrupt Management

The nachos.machine.Interrupt class simulates interrupts by maintaining an event queue together with a simulated clock. As the clock ticks, the event queue is examined to find events scheduled to take place now. The interrupt controller is returned by `Machine.interrupt()`.

The clock is maintained entirely in software and ticks only under the following conditions:

- Every time interrupts are re-enabled (i.e. only when interrupts are disabled and get enabled again), the clock advances 10 ticks. Nachos code frequently disables and restores interrupts for mutual exclusion purposes by making explicit calls to `disable()` and `restore()`.
- Whenever the MIPS simulator executes one instruction, the clock advances one tick.

> **Note:** Nachos C++ users: Nachos C++ allowed the simulated time to be advanced to that of the next interrupt whenever the ready list is empty. This provides a small performance gain, but it creates unnatural interaction between the kernel and the hardware, and it is unnecessary (a normal OS uses an idle thread, and this is exactly what Nachos does now).

Whenever the clock advances, the event queue is examined and any pending interrupt events are serviced by invoking the device event handler associated with the event. Note that this handler is *not* an interrupt handler (a.k.a. interrupt service routine). Interrupt handlers are part of software, while device event handlers are part of the hardware simulation. A device event handler will *invoke* the software interrupt handler for the device, as we will see later. For this reason, the Interrupt class disables interrupts before calling a device event handler.

> **Caution**
>
> Due to a bug in the current release of Nachos, *only the timer interrupt handler may cause a context switch* (the problem is that a few device event handlers are not reentrant; in order for an interrupt handler to be allowed to do a context switch, the device event handler that invoked it must be reentrant). All interrupt handlers besides the timer interrupt handler must not directly or indirectly cause a context switch before returning, or deadlock may occur. However, you probably won't even want to context switch in any other interrupt handler anyway, so this should not be a problem.

The Interrupt class accomplishes the above through three methods. These methods are only accessible to hardware simulation devices.

- `schedule()` takes a time and a device event handler as arguments, and schedules the specified handler to be called at the specified time.
- `tick()` advances the time by 1 tick or 10 ticks, depending on whether Nachos is in user mode or kernel mode. It is called by `setStatus()` whenever interrupts go from being disabled to being enabled, and also by `Processor.run()` after each user instruction is executed.
- `checkIfDue()` invokes event handlers for queued events until no more events are due to occur. It is invoked by `tick()`.

The Interrupt class also simulates the hardware interface to enable and disable interrupts (see the Javadoc for Interrupt).

The remainder of the hardware devices present in Nachos depend on the Interrupt device. No hardware devices in Nachos create threads, thus, the only time the code in the device classes execute is due to a function call by the running KThread or due to an interrupt handler executed by the Interrupt object.

2.3.2. Timer

Nachos provides an instance of a Timer to simulate a real-time clock, generating interrupts at regular intervals. It is implemented using the event driven interrupt mechanism described above. `Machine.timer()` returns a reference to this timer.

Timer supports only two operations:

- `getTime()` returns the number of ticks since Nachos started.
- `setInterruptHandler()` sets the timer interrupt handler, which is invoked by the simulated timer approximately every `Stats.TimerTicks` ticks.

A Guide to Nachos 5.0j

The timer can be used to provide preemption. Note however that the timer interrupts do not always occur at exactly the same intervals. Do not rely on timer interrupts being equally spaced; instead, use `getTime()`.

2.3.3. Serial Console

Nachos provides three classes of I/O devices with read/write interfaces, of which the simplest is the serial console. The serial console, specified by the SerialConsole class, simulates the behavior of a serial port. It provides byte-wide read and write primitives that never block. The machine's serial console is returned by `Machine.console()`.

The read operation tests if a byte of data is ready to be returned. If so, it returns the byte immediately, and otherwise it returns -1. When another byte of data is received, a receive interrupt occurs. Only one byte can be queued at a time, so it is not possible for two receive interrupts to occur without an intervening read operation.

The write operation starts transmitting a byte of data and returns immediately. When the transmission is complete and another byte can be sent, a send interrupt occurs. If two writes occur without an intervening send interrupt, the actual data transmitted is undefined (so the kernel should always wait for a send interrupt first).

Note that the receive interrupt handler and send interrupt handler are provided by the kernel, by calling `setInterruptHandlers()`.

Implementation note: in a normal Nachos session, the serial console is implemented by class StandardConsole, which uses stdin and stdout. It schedules a read device event every `Stats.ConsoleTime` ticks to poll stdin for another byte of data. If a byte is present, it stores it and invokes the receive interrupt handler.

2.3.4. Disk

The file systems project has not yet been ported, so the disk has not been tested.

2.3.5. Network Link

Separate Nachos instances running on the same real-life machine can communicate with each other over a network, using the NetworkLink class. An instance of this class is returned by `Machine.networkLink()`.

The network link's interface is similar to the serial console's interface, except that instead of receiving and sending bytes at a time, the network link receives and sends packets at a time. Packets are instances of the Packet class.

Each network link has a *link address*, a number that uniquely identifies the link on the network. The link address is returned by `getLinkAddress()`.

A packet consists of a header and some data bytes. The header specifies the link address of the machine sending the packet (the source link address), the link address of the machine to which the packet is being sent (the destination link address), and the number of bytes of data contained in the packet. The data bytes are not analyzed by the network hardware, while the header is. When a link transmits a packet, it transmits it only to the link specified in the destination link address field of the header. Note that the source address can be forged.

The remainder of the interface to NetworkLink is equivalent to that of SerialConsole. The kernel can check for a packet by calling `receive()`, which returns `null` if no packet is available. Whenever a packet arrives, a receive interrupt is generated. The kernel can send a packet by calling `send()`, but it must wait for a send interrupt before attempting to send another packet.

3. Threads and Scheduling

Nachos provides a kernel threading package, allowing multiple tasks to run concurrently (see nachos.threads.ThreadedKernel and nachos.threads.KThread). Once the user-prcesses are implmented (phase 2), some threads may be running the MIPS processor simulation. As the scheduler and thread package are concerned, there is no difference between a thread running the MIPS simulation and one running just kernel Java code.

3.1. Thread Package

All Nachos threads are instances of nachos.threads.KThread (threads capable of running user-level MIPS code are a subclass of KThread, nachos.userprog.UThread). A nachos.machine.TCB object is contained by each KThread and provides low-level support for context switches, thread creation, thread destruction, and thread yield.

Every KThread has a `status` member that tracks the state of the thread. Certain KThread methods will fail (with a `Lib.assert()`) if called on threads in the wrong state; check the KThread Javadoc for details.

`statusNew`

A newly created, yet to be forked thread.

`statusReady`

A thread waiting for access to the CPU. `KThread.ready()` will add the thread to the ready queue and set the status to `statusReady`.

`statusRunning`

The thread currently using the CPU. `KThread.restoreState()` is responsible for setting `status` to `statusRunning`, and is called by `KThread.runNextThread()`.

`statusBlocked`

A thread which is asleep (as set by `KThread.sleep()`), waiting on some resource besides the CPU.

`statusFinished`

A thread scheduled for destruction. Use `KThread.finish()` to set this status.

Internally, Nachos implements threading using a Java thread for each TCB. The Java threads are synchronized by the TCBs such that exactly one is running at any given time. This provides the illusion of context switches saving state for the current thread and loading the saved state of the new thread. This detail, however, is only important for use with debuggers (which will show multiple Java threads), as the behavior is equivelant to a context switch on a real processor.

3.2. Scheduler

A sub-class (specified in the `nachos.conf`) of the abstract base class nachos.threads.Scheduler is responsible for scheduling threads for all limited resources, be it the CPU, a synchronization construct like a lock, or even a thread join operation. For each resource a nachos.threads.ThreadQueue is created by `Scheduler.newThreadQueue()`. The implementation of the resource (e.g. nachos.threads.Semaphore class) is responsible for adding KThreads to the ThreadQueue (`ThreadQueue.waitForAccess()`) and requesting the ThreadQueue return the next thread (`ThreadQueue.nextThread()`). Thus, all scheduling decisions (including those regarding the CPU's ready queue) reduce to the selection of the next thread by the ThreadQueue objects[1].

Various phases of the project will require modifications to the scheduler base class. The nachos.threads.RoundRobinScheduler is the default, and implements a fully functional (though naive) FIFO scheduler. Phase 1 of the projects requires the student to complete the nachos.threads.PriorityScheduler; for phase 2, students complete nachos.threads.LotteryScheduler.

3.3. Creating the First Thread

Upto the point where the Kernel is created, the boot process is fairly easy to follow - Nachos is just making Java objects, same as any other Java program. Also like any other single-threaded Java program, Nachos code is executing on the initial Java thread created automaticaly for it by Java. `ThreadedKernel.initialize()` has the task of starting threading:

```
public void initialize(String[] args) {
      ...
      // start threading
      new KThread(null);
      ...
}
```

The first clue that something special is happening should be that the new KThread object created is not stored in a variable inside `initialize()`. The constructor for KThread follows the following procedure the first time it is called:

1. Create the ready queue (`ThreadedKernel.scheduler.newThreadQueue()`).
2. Allocate the CPU to the new KThread object being created (`readyQueue.acquire(this)`).
3. Set `KThread.currentThread` to the new KThread being made.
4. Set the TCB object of the new KThread to `TCB.currentTCB()`. In doing so, the currently running Java thread is assigned to the new KThread object being created.
5. Change the `status` of the new KThread from the default (`statusNew`) to `statusRunning`. This bypasses the `statusReady` state.
6. Create an idle thread.
    a. Make another new KThread, with the target set to an infinite `yield()` loop.
    b. Fork the idle thread off from the main thread.

After this procedure, there are two KThread objects, each with a TCB object (one for the main thread, and one for the idle thread). The main thread is not special - the scheduler treats it exactly like any other KThread. The main thread can create other threads, it can die, it can block. The Nachos session will not end until all KThreads finish, regardless of whether the main thread is alive.

For the most part the idle thread is also a normal thread, which can be contexted switched like any other. The only difference is it will never be added to the ready queue (`KThread.ready()` has an explicit check for the idle thread). Instead, if `readyQueue.nextThread()` returns `null`, the thread system will switch to the idle thread.

Note: While the Nachos idle thread does nothing but `yield()` forever, some systems use the idle thread to do work. One common use is zeroing memory to prepare it for reallocation.

3.4. Creating More Threads

Creating subsequent threads is much simpler. As described in the KThread Javadoc, a new KThread is created, passing the constructor a Runnable object. Then, `fork()` is called:

```
KThread newThread = new KThread(myRunnable);
...
newThread.fork();
```

This sequence results in the new thread being placed on the ready queue. The currently running thread does not immediatly yield, however.

3.4.1. Java Anonymous Classes in Nachos

The Nachos source is relatively clean, using only basic Java, with the exception of the use of anonymous classes to replicate the functionality of function pointers in C++. The following code illustrates the use of an anonymous class to create a new KThread object which, when forked, will execute the `myFunction()` method of the encolosing object.

```
Runnable myRunnable = new Runnable() {
public void run() {
    myFunction();
}
    };
KThread newThread = new KThread(myRunnable);
```

This code creates a new object of type Runnable inside the context of the enclosing object. Since `myRunnable` has no method `myFunction()`, executing `myRunnable.run()` will cause Java to look in the enclosing class for a `myFunction()` method.

3.5. On Thread Death

All threads have some resources allocated to them which are neccessary for the thread to run (e.g. the TCB object). Since the thread itself cannot deallocate these resources while it is running, it leaves a virtual will asking the next thread which runs to deallocate its resources. This is implemented in

`KThread.finish()`, which sets `KThread.toBeDestroyed` to the currently running thread. It then sets current thread's `status` field to `statusFinished` and calls `sleep()`.

Since the thread is not waiting on a ThreadQueue object, its sleep will be permanent (that is, Nachos will never try to wake the thread). This scheme does, however, require that after every context switch, the newly running thread must check `toBeDestroyed`.

> **Note:** In the C++ version of Nachos, thread death was complicated by the explicit memory deallocation required, combined with dangling references that still pointing to the thread after death (for example, most thread `join()` implementations requires some reference to the thread). In Java, the garbage collector is responsible for noticing when these references are detached, significantly simplifying the thread finishing process.

4. The Nachos Simulated MIPS Machine

Nachos simulates a machine with a processor that roughly approximates the MIPS architecture. In addition, an event-driven simulated clock provides a mechanism to schedule events and execute them at a later time. This is a building block for classes that simulate various hardware devices: a timer, an elevator bank, a console, a disk, and a network link.

The simulated MIPS processor can execute arbitrary programs. One simply loads instructions into the processor's memory, initializes registers (including the program counter, regPC) and then tells the processor to start executing instructions. The processor then fetches the instruction that regPC points at, decodes it, and executes it. The process is repeated indefinitely, until either an instruction causes an exception or a hardware interrupt is generated. When an exception or interrupt takes place, execution of MIPS instructions is suspended, and a Nachos interrupt service routine is invoked to deal with the condition.

Conceptually, Nachos has two modes of execution, one of which is the MIPS simulator. Nachos executes user-level processes by loading them into the simulator's memory, initializing the simulator's registers and then running the simulator. User-programs can only access the memory associated with the simulated processor. The second mode corresponds to the Nachos "kernel". The kernel executes when Nachos first starts up, or when a user-program executes an instruction that causes an exception (e.g., illegal instruction, page fault, system call, etc.). In kernel mode, Nachos executes the way normal Java programs execute. That is, the statements corresponding to the Nachos source code are executed, and the memory accessed corresponds to the memory assigned to Nachos variables.

4.1. Processor Components

The Nachos/MIPS processor is implemented by the Processor class, an instance of which is created when Nachos first starts up. The Processor class exports a number of public methods and fields that the Nachos kernel accesses directly. In the following, we describe some of the important variables of the Processor class; describing their role helps explain what the simulated hardware does.

The processor provides registers and physical memory, and supports virtual memory. It provides operations to run the machine and to examine and modify its current state. When Nachos first starts up, it creates an instance of the Processor class and makes it available through `Machine.processor()`. The following aspects of the processor are accessible to the Nachos kernel:

Registers

: The processor's registers are accessible through `readRegister()` and `writeRegister()`. The registers include MIPS registers 0 through 31, the low and high registers used for multiplication and division, the program counter and next program counter registers (two are necessary because of branch delay slots), a register specifying the cause of the most recent exception, and a register specifying the virtual memory address associated with the most recent exception. Recall that the stack pointer register and return address registers are general MIPS registers (specifically, they are registers 29 and 31, respectively). Recall also that r0 is always 0 and cannot be modified.

Physical memory

: Memory is byte-addressable and organized into 1-kilobyte pages, the same size as disk sectors. A reference to the main memory array is returned by `getMemory()`. Memory corresponding to physical address `m` can be accessed in Nachos at `Machine.processor().getMemory()[m]`. The number of pages of physical memory is returned by `getNumPhysPages()`.

Virtual memory

: The processor supports VM through either a single linear page table or a software-managed TLB (but not both). The mode of address translation is actually used is determined by `nachos.conf`, and is returned by `hasTLB()`. If the processor does not have a TLB, the kernel can tell it what page table to use by calling `setPageTable()`. If the processor does have a TLB, the kernel can query the size of the TLB by calling `getTLBSize()`, and the kernel can read and write TLB entries by calling `readTLBEntry()` and `writeTLBEntry()`.

Exceptions

: When the processor attempts to execute an instruction and it results in an exception, the kernel exception handler is invoked. The kernel must tell the processor where this exception handler is by invoking `setExceptionHandler()`. If the exception resulted from a syscall instruction, it is the kernel's responsibility to advance the PC register, which it should do by calling `advancePC()`.

At this point, we know enough about the Processor class to explain how it executes arbitrary user programs. First, we load the program's instructions into the processor's physical memory (i.e. the array returned by `getMemory()`). Next, we initialize the processor's page table and registers. Finally, we invoke `run()`, which begins the fetch-execute cycle for the processor.

`run()` causes the processor to enter an infinite fetch-execute loop. This method should only be called after the registers and memory have been properly initialized. Each iteration of the loop does three things:

1. It attempts to run an instruction. This should be very familiar to students who have studied the generic 5-stage MIPS pipeline. Note that when an exception occurs, the pipline is aborted.
   a. The 32-bit instruction is fetched from memory, by reading the word of virtual memory pointed to by the PC register. Reading virtual memory can cause an exception.
   b. The instruction is decoded by looking at its 6-bit `op` field and looking up the meaning of the instruction in one of three tables.
   c. The instruction is executed, and data memory reads and writes occur. An exception can occur if an arithmetic error occurs, if the instruction is invalid, if the instruction was a syscall, or if a memory operand could not be accessed.
   d. The registers are modified to reflect the completion of the instruction.
2. If an exception occurred, handle it. The cause of the exception is written to the cause register, and if the exception involved a bad virtual address, this address is written to the bad virtual address register. If a delayed load is in progress, it is completed. Finally, the kernel's exception handler is invoked.
3. It advances the simulated clock (the clock, used to simulate interrupts, is discussed in the following section).

Note that from a user-level process's perspective, exceptions take place in the same way as if the program were executing on a bare machine; an exception handler is invoked to deal with the problem. However, from our perspective, the kernel's exception handler is actually called via a normal procedure call by the simulated processor.

The processor provides three methods we have not discussed yet: `makeAddress()`, `offsetFromAddress()`, and `pageFromAddress()`. These are utility procedures that help the kernel go between virtual addresses and virtual-page/offset pairs.

4.2. Address Translation

The simulated processor supports one of two address translation modes: linear page tables, or a software-managed TLB. While the former is simpler to program, the latter more closely corresponds to what current machines support.

In both cases, when translating an address, the processor breaks the 32-bit virtual address into a virtual page number (VPN) and a page offset. Since the processor's page size is 1KB, the offset is 10 bits wide and the VPN is 22 bits wide. The processor then translates the virtual page number into a translation entry.

Each translation entry (see the TranslationEntry class) contains six fields: a valid bit, a read-only bit, a used bit, a dirty bit, a 22-bit VPN, and a 22-bit physical page number (PPN). The valid bit and read-only bit are set by the kernel and read by the processor. The used and dirty bits are set by the processor, and read and cleared by the kernel.

4.2.1. Linear Page Tables

When in linear page table mode, the processor uses the VPN to index into an array of translation entries. This array is specified by calling `setPageTable()`. If, in translating a VPN, the VPN is greater than or equal to the length of the page table, or the VPN is within range but the corresponding translation entry's valid bit is clear, then a page fault occurs.

In general, each user process will have its own private page table. Thus, each process switch requires calling `setPageTable()`. On a real machine, the page table pointer would be stored in a special processor register.

4.2.2. Software-Managed TLB

When in TLB mode, the processor maintains a small array of translation entries that the kernel can read/write using `readTLBEntry()` and `writeTLBEntry()`. On each address translation, the processor searches the entire TLB for the first entry whose VPN matches.

5. User-Level Processes

Nachos runs each user program in its own private address space. Nachos can run any COFF MIPS binaries that meet a few restrictions. Most notably, the code must only make system calls that Nachos understands. Also, the code must not use any floating point instructions, because the Nachos MIPS simulator does not support coprocessors.

5.1. Loading COFF Binaries

COFF (Common Object File Format) binaries contain a lot of information, but very little of it is actually relevent to Nachos programs. Further, Nachos provides a COFF loader class, nachos.machine.Coff, that abstracts away most of the details. But a few details are still important.

A COFF binary is broken into one or more *sections*. A section is a contiguous chunk of virtual memory, all the bytes of which have similar attributes (code vs. data, read-only vs. read-write, initialized vs. uninitialized). When Nachos loads a program, it creates a new processor, and then copies each section into the program's virtual memory, at some start address specified by the section. A COFF binary also specifies an initial value for the PC register. The kernel must initialize this register, as well as the stack pointer, and then instruct the processor to start executing the program.

The Coff constructor takes one argument, an OpenFile referring to the MIPS binary file. If there is any error parsing the headers of the specified binary, an EOFException is thrown. Note that if this constructor succeeds, the file belongs to the Coff object; it should not be closed or accessed anymore, except through Coff operations.

There are four Coff methods:

- `getNumSections()` returns the number of sections in this binary.
- `getSection()` takes a section number, between `0` and `getNumSections() - 1`, and returns a CoffSection object representing the section. This class is described below.
- `getEntryPoint()` returns the value with which to initialize the program counter.
- `close()` releases any resources allocated by the loader. This includes closing the file passed to the constructor.

The CoffSection class allows Nachos to access a single section within a COFF executable. Note that while the MIPS cross-compiler generates a variety of sections, the only important distinction to the Nachos kernel is that some sections are read-only (i.e. the program should never write to any byte in the section), while some sections are read-write (i.e. non-`const` data). There are four methods for accessing COFF sections:

- `getFirstVPN()` returns the first virtual page number occupied by the section.
- `getLength()` returns the number of pages occupied by the section. This section therefore occupies pages `getFirstVPN()` through `getFirstVPN() + getLength() - 1`. Sections should never overlap.
- `isReadOnly()` returns true if and only if the section is read-only (i.e. it only contains code or constant data).
- `loadPage()` reads a page of the section into main memory. It takes two arguments, the page within the section to load (in the range `0` through `getLength() - 1`) and the physical page of memory to write.

5.2. Starting a Process

The kernel starts a process in two steps. First, it calls `UserProcess.newUserProcess()` to instantiate a process of the appropriate class. This is necessary because the process class changes as more functionality is added to each process. Second, it calls `execute()` to load and execute the program, passing the name of the file containing the binary and an array of arguments.

`execute()` in turn takes two steps. It first loads the program into the process's address space by calling `load()`. It then forks a new thread, which initializes the processor's registers and address translation information and then calls `Machine.processor().run()` to start executing user code.

`load()` opens the executable's file, instantiates a COFF loader to process it, verifies that the sections are contiguously placed in virtual memory, verifies that the arguments will fit within a single page, calculates the size of the program in pages (including the stack and arguments), calls `loadSections()` to actually load the contents of each section, and finally writes the command line arguments to virtual memory.

`load()` lays out the program in virtual memory as follows: first, starting at virtual address 0, the sections of the executable occupy a contiguous region of virtual memory. Next comes the stack, the size of which is determined by the variable `stackPages`. Finally, one page is reserved for command line arguments (that `argv` array).

`loadSections()` allocates physical memory for the program and initializes its page table, and then loads sections to physical memory (though for the VM project, this loading is done lazily, delayed until pages are demanded). This is separated from the rest of `load()` because the loading mechanism depends on the details of the paging system.

In the code you are given, Nachos assumes that only a single process can exist at any given time. Therefore, `loadSections()` assumes that no one else is using physical memory, and it initializes its page table so as to map virtual memory addresses directly to physical memory addresses, without any translation (i.e. virtual address `n` maps to physical address `n`).

The method `initRegisters()` zeros out the processor's registers, and then initializes the program counter, the stack pointer, and the two argument registers (which hold `argc` and `argv`) with the values computed by `load()`. `initRegisters()` is called exactly once by the thread forked in `execute()`.

5.3. User Threads

User threads (that is, kernel threads that will be used to run user code) require additional state. Specifically, whenever a user thread starts running, it must restore the processor's registers, and possibly restore some address translation information as well. Right before a context switch, a user thread needs to save the processor's registers.

To accomplish this, there is a new thread class, UThread, that extends KThread. It is necessary to know which process, if any, the current thread belongs to. Therefore each UThread is bound to a single process.

UThread overrides `saveState()` and `restoreState()` from KThread so as to save/restore the additional information. These methods deal only with the user register set, and then direct the current process to deal with process-level state (i.e. address translation information). This separation makes it possible to allow multiple threads to run within a single process.

5.4. System Calls and Exception Handling

User programs invoke system calls by executing the MIPS `syscall` instruction, which causes the Nachos kernel exception handler to be invoked (with the cause register set to `Processor.exceptionSyscall`). The kernel must first tell the processor where the exception handler is by calling `Machine.processor().setExceptionHandler()`.

The default Kernel exception handler, `UserKernel.exceptionHandler()`, reads the value of the processor's cause register, determines the current process, and invokes `handleException` on the current process, passing the cause of the exception as an argument. Again, for a syscall, this value will be `Processor.exceptionSyscall`.

The `syscall` instruction indicates a system call is requested, but doesn't indicate which system call to perform. By convention, user programs place the value indicating the particular system call desried into MIPS register `r2` (the first return register, `v0`) before executing the `syscall` instruction. Arguments to the system call, when necessary, are passed in MIPS registers `r4` through `r7` (i.e. the argument registers, `a0 ... a3`), following the standard C procedure call convention. Function return values, including system call return values, are expected to be in register `r2` (`v0`) on return.

> **Note:** When accessing user memory from within the exception handler (or within Nachos in general), user-level addresses cannot be referenced directly. Recall that user-level processes execute in their own private address spaces, which the kernel cannot reference directly. Use `readVirtualMemory()`, `readVirtualMemoryString()`, and `writeVirtualMemory()` to make use of pointer arguments to syscalls.

Notes

1. The ThreadQueue object representing the ready queue is stored in the static variable `KThread.readyQueue`.