

Building an Operating System Project 1

System Set-up & Echoing Keypresses



Operating Systems

Table of Contents

The VirtualBox - GeekOS Environment

Exploring the GeekOS Source Code	3
Start-up Sequence.....	3
GeekOS Source Code Organization	5

Booting Your VirtualBox VM to GeekOS

Building the GeekOS Kernel.....	8
Creating Your GeekOS VirtualBox VM.....	8

GeekOS Project 1 - Echoing User Keypresses

GeekOS Project1 (GR01) Rubric

GeekOS Performance.....	12
Project Report	12

1. The VMWare - GeekOS Environment



Exploring the GeekOS Source Code

In order to gain a deep understanding of operating system concepts, we will build our own OS. The starting point is the **GeekOS** kernel, an educational operating system kernel to which we will add features to manage processes, memory, files, devices and I/O.

The **GeekOS** distribution provides the minimum framework necessary for an operating system to run on an x86 (Intel IA32) personal computer. We will use a version that includes some significant changes to customize it for this course. You can download it from the course web site. For our first project, the archive is **gr01-starter.zip**

Place the distribution archive in one of your **GeekOS VM** directory and expand it:

```
~/geekos $ unzip gr01-starter.zip
```



This will create the **/gr01** directory. Inside this directory you'll find a **src** directory containing starter code for each project and an **include** directory with the associated header files and some macros.

Start-up Sequence

When you launch the **VMware**, the host operating system loads the **VMware** program into the memory of your machine. To initialize itself, it reads the **.vmx** file to discover the hardware configuration. Once the BIOS is loaded, the **VMware** machine behaves similar to a real hardware computer. After initial hardware tests are performed, control of the CPU passes from a number of different system entities before the operating system gains control.

Our **VMware/GeekOS** environment goes through the same steps as most hardware/operating system pairings. First the BIOS copies the boot sector from the boot disk into memory then executes that program. The bootstrap program is a short piece of code that loads a set-up program into memory. The set-up program loads the operating system kernel, initializes a few low-level data structures then passes control to the operating system kernel.

The operating system initializes its data structures and then passes control to the initial kernel thread object. The table below gives details on how the **VirtualBox/GeekOS** environment boots up.

 Control	 Action
Hardware	Power on
	Hardware systems checks performed
BIOS	BIOS executes and reads the boot sector (Track 0, Sector 0) from the floppy disk into memory location 0x07C0 (BOOTSEG). This single sector (512 bytes) contains the entire Bootstrap Loader program (fd_boot.asm).



Control



Action

Once the Bootstrap Loader is in memory, the BIOS code jumps to BOOTSEG and the CPU begins executing the Bootstrap Loader at that address.

Bootstrap Loader

The Bootstrap Loader starts by copying itself higher into memory, 0x9000 (INITSEG), so that it can load the operating system kernel at the lower memory address. Once this code is moved, it jumps to the new location and begins starts loading the set-up program (**setup.asm**).

The Setup Program is loaded at 0x9020 (SETSEG) and the kernel image is loaded at 0x1000 (KERNSEG). Execution then jumps to the set-up program.

Set-Up

Set-up begins by determining the memory size, blocking interrupts (since they can't be handled yet) and setting protected mode.

Now that the environment is "safe" to work in, set-up creates the kernel's stack and the initial kernel thread. Execution then jumps to the kernel at **Main()**.

Kernel (Main)

The kernel begins by initializing several of it's data structures and system managers:

Init_BSS	Cleans up the Boot Sector Segment (the location in memory of the bootstrap loader program) by filling it with nulls.
Init_Screen	Clears the screen by copying the "Clear Screen" character to the screen buffer and sets the cursor to screen position 0,0
Init_Mem	Initializes Memory Management data structures such as the kernel heap and page table.
Init_TSS	Initializes the Task State Segment (TSS). This is a structure required by the i386 Architecture to management the currently executing thread.
Init_Interrupts	Initializes the Interrupt Descriptor Table (IDT) which maps hardware interrupts to the interrupt handling routines. The initial records in the table set the privilege and base address of the interrupt handler.
Init_Scheduler	Creates a thread structure for the kernel that is managed by the kernel. It also creates the Idle Thread and the Reaper Thread and places them on the All Thread List.
Init_Traps	Initializes trap addresses. Traps are software initiated interrupts.



Control



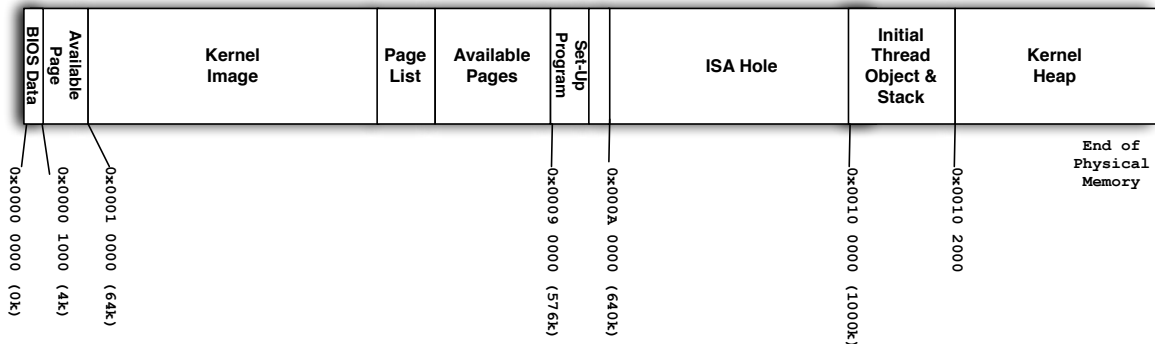
Action

Init_Timer

Initializes timer delay and sets the timer interrupt request.

Init_Keyboard

Initializes the keyboard buffer and sets the hardware interrupt request for the keyboard.



GeekOS Source Code Organization

The operating system is a large program or set of programs for coordinating the computer's individual components so that they work together according to a single plan. The operating system is responsible for managing the CPU, memory, disk drives, keyboard, mouse, screen and any other hardware that is part of the computer, including the network connection. Each of these is considered a limited resource and therefore access to it must be coordinated among the various programs requesting them.

Resources are generally divided into four categories for management: Process, Memory, Device and I/O, and Files. The following sections give a short description of each manager and files from the first **GeekOS** project that implement that manager's functionality.



Process Management

Process management is responsible for managing CPU access. A process is any entity that requests the attention of the CPU, such as a program. Modern operating systems allow multiple processes to be in memory, "running" at the same time, each requesting the CPU. Only one process can have control of the CPU at any instant and the process management functions decide which process that will be.

kthread.c

Functions for thread and process management.

timer.c

Functions to initialize the process timer and to handle timer interrupts.

tss.c

Functions to initialize and handle the TSS (Task State Segment).

Memory Management

Computers have a fixed amount of internal memory (RAM). Memory can hold programs and data and must be shared between processes. Processes can request additional memory while executing and they must free up memory when they complete. Each process must also be protected from another process accessing its memory, either accidentally or maliciously.

bget.c	This is an open source memory management package that doesn't rely on any C library code. It provides for the allocation and release of buffers, buffer pool management and memory consolidation to minimize fragmentation.
memory.c	Functions to initialize and manage the page table and allocate memory pages.
malloc.c	Implementations of Malloc() and Free() so that our code is independent of C library calls. (Note the capitalization)
segment.c	Functions for initializing and management the Local Descriptor Table (LDT) and Code and Data segments.
gdt.c	Functions to initialize and manage the kernel's Global Descriptor Table (GDT)

Device and I/O Management

Device management handles requests by processes to use hardware attached to the computer, such as keyboards, screen, disk drives, etc. Device management coordinates requests to access these devices. Devices commonly need to be accessed for retrieval of data, such as from a disk drive, or sending data, such as thru network connections or to a screen. The device manager must buffer requests if the device is not ready or buffer data coming from the device if the process is not prepared to handle the data.

idt.c	Functions to initialize and manage the Interrupt Descriptor Table (IDT)
interrupts.c	Functions to initialize interrupt handling and to enable and disable interrupts.
io.c	Low level code to read and write bytes and word to I/O ports.
irq.c	Initializes, enables and disables Interrupt request (IRQ) handling. Serves as a device driver for the interrupt system.
keyboard.c	Functions for handling input from a keyboard.
screen.c	Functions for handling output to a screen (monitor). Includes definition of Print()
lowlevel.asm	These are low level routines to establish interrupts and the descriptor tables.

util.asm	Set of short routines to display characters, digits and new lines.
-----------------	--

File Management

Files are blocks of data or information that are handled as a single virtual resource. This resource must be moved from external storage, such as disk drives or network connections to internal storage, memory, and back again. The file manager provides structures to track files so that they can be quickly stored, modified and retrieved from whichever device they may be residing on.

This project does not have any files directly related to file management functionality.

General Operating System Files

These files perform initialization or general functionality and don't fall under any particular operating system manager.

fd_boot.asm	This code is stored in the boot sector of the floppy disk. The PC's BIOS checks here first for program code to load. Once loaded, fd_boot is executed to load the setup program and the operating system code into memory.
defs.asm	Common definitions of constants. Many of these identify the memory locations where various parts of the operating system will be loaded.
setup.asm	This short application is called after fd_boot finishes loading it and the kernel image into memory. Setup established basic structures and memory regions required by the operating system. Among these are the Global Descriptor Table (GDT), IDT, the kernel code and data segments and the kernel stack. When this routine is finished, it jumps to the start of the kernel – Main() .
main.c	Contains the Main() function which is the start point for the kernel. Main() initializes operating system memory and data structures and then starts the initial kernel thread.
crc32.c	Provides 32-bit CRC error check.
traps.c	Functions for initializing and handling traps (software interrupts).
kassert.h	KASSERT , PAUSE and TODO macro definitions

2. Booting Your VMWare VM to GeekOS

Building the GeekOS Kernel

In the directory for the first GeekOS project, **gr01**, you can find the following sub-directories:

- ▶ **build** – this directory contains the **makefile** and should be your working directory when you build the project. When the build process is complete, there will be a disk image (or two) in this directory that contains your new operating system.
- ▶ **include** – header files for the operating system
- ▶ **scripts** – Perl scripts used by the **makefile** to build the final executable and disk image.
- ▶ **scr** – source files for the operating system. Most of your work will be done in this directory. Occasionally, you'll need to edit a header file.

When you compile and build each **GeekOS** project, your working directory should be the **build** directory for the project on which you're working. So change your working directory **gr01**'s **build** directory. Your path should be:

```
~/geekos/gr01/build
```

Each **GeekOS** project has a **makefile** in its **build** directory. A **makefile** describes the dependencies and steps for building a project. These steps are carried out by the **make** utility. To build your first GeekOS project execute the **make** program:

```
~/geekos/gr01/build $ make
```

The **make** utility uses information in the **makefile** to compile and link the project files. It then creates a floppy disk image, **fd-gr01.img**, in the **build** directory to be used by VMWare that contains the operating system kernel just built.

Creating Your GeekOS VMware

Setting up a virtual machine to run our operating system is fairly simple.

- In the **VMware**, click on the **File>New Virtual Machine** button to create a new machine.
- Select "I will install the operating system later."
- Select an operating system of **Other** and version **Other**.
- Name the VM as **GR01** and give the VM **32MB** of memory.
- We will boot from a floppy drive, so need to "**Customize Hardware**"
- Click "**Add...**" and "**Floppy Drive**"

Confirm this is the virtual machine that you want and it will be created for you. Now we need to place the bootable floppy with our version of the GeekOS in the floppy drive so that our machine will boot from it.

- Select the **GR01** VM and double click on the **Floppy** and select **Use floppy image file**.
- Click on the **Browse...** button and select the file **fd-gr01.img**.

Your hardware is now configured and you have the floppy disk with your operating system in the floppy drive. **Boot-up your system!**

At this point, when control passes from the hardware BIOS and bootstrap program to our **GeekOS** code and you should see something similar to:

```
32768KB memory detected, 7802 pages in freelist, 1048576bytes in kernel heap
Initializing IDT...
Initializing timer...
Delay loop: 33890645 iterations per tick
Initializing keyboard...
Welcome to GeekOS!
Unimplemented feature: Start a kernel thread to echo pressed keys
```

Don't worry about the **Unimplemented Feature** error. You'll be fixing that as part of your first project!

3. GeekOS Project 1 - Echoing User Keypresses

The Task

As you saw in the last section, the current version has some unimplemented features. Each project contains placeholders indicating where you need to add code. These placeholders use the **TODO** macro defined in `include/geekos/kassert.h`. This project has a single **TODO** which can be found in `src/geekos/main.c`:

```
TODO("Start a kernel thread to echo pressed keys");
```

For this project you are to add code to the kernel that creates a new kernel mode thread. A kernel thread can simply be a kernel function controlling user interaction. You need to create a function, **Echo_Keypress()**, to serve as the body of this kernel thread. This function is launched as a kernel thread by passing it as a parameter to **Start_Kernel_Thread()**. The function should be placed in `src/geekos/main.c`.

This kernel mode thread should do the following:

- Display "**Hello from xxx!**" where **xxx** is your group name.
- Call the keyboard input routine **Wait_For_Key()** repeatedly and echo each character entered.
- When the **RETURN** key is pressed, the cursor should move down one line and to the left-most column.
- When the **BACKSPACE** key is pressed, the cursor should move a single space to the left and erase the character entered
- When the termination character (**control-d**) is entered, print a good-bye message and exit.

Important Functions Available in the OS

Here are several functions and constants you may find helpful in completing this project:

Keycode **Wait_For_Key(void)**
[found in `keyboard.c`]

Waits for user to type a key then returns the **Keycode** of that key. **Keycode** is an unsigned two-byte value that has the ASCII code of the key in the low byte and special or modifier key code in the high byte. You can determine the specific keys by using a bit-wise **AND**. Keycodes are sent based on the key repeat rate as long as the key is pressed down.

KEY_RELEASE_FLAG 0x8000 [found in keyboard.h]	To keep things simple, we are not going to worry about the key repeat rates on the various systems. Instead, we will wait until the user releases the key, thereby setting the KEY_RELEASE_FLAG and process that key. keyboard.h has many other important character constants that may be useful.
void Get_Cursor(int* row, int* col) bool Put_Cursor(int row, int col) [found in screen.c]	These functions are useful in determining and setting the cursor position.
void Print(const char *fmt, ...) [found in screen.c]	Because our kernel is the operating system on this machine, normal C functions, such as printf() are not available. A simple Print() function that takes a parameter list similar to printf() is supplied to provide output in GeekOS.
struct Kernel_Thread* Start_Kernel_Thread (Thread_Start_Func startFunc, ulong_t arg, int priority, bool detached) [found in kthread.c]	Starts a kernel-mode-only thread, using function, startFunc as its body. It returns a pointer to the new thread if successful and null otherwise. In this project, you need to catch the return value, no processing of that value is necessary. The parameter list for the function is passed in arg , you can leave that null. The priority of this thread should be PRIORITY_NORMAL (found in kthread.h). Kernel mode threads, such as this one, should have detached set as false .
KASSERT(cond) [found in kassert.h]	KASSERT is a macro that performs a “Kernel Assertion.” If cond is false, the kernel program is halted and an error message is displayed on the screen giving the failed assertion and where it failed. This is the recommended way to test for precondition in all of your functions.

The **Print()** function and **KASSERT()** macro can provide useful debugging information.