

# GeekOS — A template for booting and running programs on an x86 PC

David Hovemeyer

<http://www.cs.umd.edu/projects/shrug/>



# Outline

- Goals / background
- Implementation
  - ▷ Makefile
  - ▷ boot sector and setup
  - ▷ initialization
  - ▷ operation
- Conclusions

# General goals

- Find out how to boot a C program on bare hardware
- Learn about low-level programming on the PC
  - ▷ protected mode
  - ▷ interrupt handling
  - ▷ hardware devices
  - ▷ threads / context switching
- Learn about tools for OS development
- Evaluate bochs/gcc/nasm as platform for learning about operating systems

# Approach

- Instead of running on real hardware, use bochs, a freeware PC emulator
  - ▷ <http://www.bochs.com/>
  - ▷ Can be restarted in seconds (unlike a real PC)
  - ▷ Runs on Windows/Mac/BeOS/UNIX, etc.
  - ▷ Instruction-level debugging
- This greatly sped up the edit/compile/debug cycle
- Bochs is really cool!
- In `/fs/unsupported/bochs` on junkfood machines

# Tools

- Bochs (already mentioned)
- Nasm, an excellent free x86 assembler
  - ▷ <http://www.web-sites.co.uk/nasm/>
- gcc, GNU linker, under Linux/x86
- GNU objcopy: a simple way to transform an executable into a form suitable for loading directly into memory
- Development hosted under Linux/x86
- Bochs run on Linux/x86 and Solaris/Sparc

# Information sources

- Intel 486 manual
- *Protected Mode Software Architecture* by Tom Shanley, ISBN 020155447X.
- *The Undocumented PC* by Frank van GILLUWE, ISBN 0201479508.
- The Intel 8259A datasheet,  
<http://support.intel.com/support/controllers/peripheral/231468.htm>
- *System BIOS for IBM PCs, Compatibles, and EISA Computers*, Phoenix Technologies Ltd., second edition, ISBN 0201577607.

# Information sources (continued)

- FreeVGA Project:  
`http://sf.znet.com/~vhold/FreeVGA/home.htm`
- Kernel Toolkit 0.2
  - ▷ `http://web.tiscalinet.it/luigisgro/ktk.html`
  - ▷ Similar to my project; a very simple 'kernel' that boots on bare hardware. I stole lots of tricks from it.
- Linux kernel source, `http://www.kernel.org/`
- Newsgroups:
  - ▷ `alt.lang.asm.x86`
  - ▷ `alt.os.development`

# Development goals

- I wanted to write (almost) all of the code myself:
  - ▷ boot loader
  - ▷ switch into protected mode
  - ▷ memory management
  - ▷ interrupt handling
  - ▷ device support (VGA text screen, keyboard)
  - ▷ threads
- I did steal bits of code from Kernel Toolkit and Linux
- At the same time, I wanted to keep things as simple as possible, *e.g.*, no segmentation, no paging, no user/kernel interface

# Result

- About 1000 lines of assembly and 2000 lines of C code, with lots of comments
- Everything done as simply as possible
- 32 bit flat address space, in ring 0
- cooperatively scheduled threads, with `Yield()`, `Wait()` and `Wake_Up()` scheduling primitives
- VGA text screen support (with `printf()`-style formatted output)
- Keyboard device driver with event (thread) driven interface

# Disclaimer

- There are almost certainly bugs in the code and inaccuracies in my comments and documentation
- I am not an expert
- You have been warned
- Also note that I haven't actually run this code on real hardware :-)

# Background assumptions

- I assume that you
  - ▷ are relatively familiar with basic x86 architecture, registers, and assembly syntax
  - ▷ know about x86 calling conventions
  - ▷ are familiar with assemblers, compilers, and linkers
  - ▷ are familiar with general OS concepts such as interrupts and threads

# Outline

- Goals / background
- Implementation
  - ▷ **Makefile**
  - ▷ boot sector and setup
  - ▷ initialization
  - ▷ operation
- Conclusions

# Makefile

- This is actually one of the more challenging issues: how to get the code in a suitable format for loading?
- For user programs, the compiler, linker, and dynamic loader automatically take care of the messy details
- For programs running on hardware, you need to carefully control details such as the base address for which the program is linked
- You probably don't want to leave relocations in the image, unless your boot loader can fix them up
- My solution was to copy Kernel Toolkit's Makefile :-)

# Makefile (continued)

- Line 89: linking the kernel
  - ▷ We specify that the `.text` section should be located at address `0x10000` (64K), and that the `Main()` function should be the entry point
  - ▷ Results in `Main()` being located at `0x10000`
  - ▷ Note that the kernel executable is in ELF format, so it can't be directly loading into memory
- Line 82: producing the kernel image file
  - ▷ Use `objcopy` to strip out unneeded sections, and produce a flat binary image
  - ▷ Pad to multiple of one sector (512 bytes)
  - ▷ This file can be loaded directly into memory

# Makefile (continued)

- Line 74: assemble the setup code
  - ▷ Note that we instruct nasm to use the 'bin' format, which produces a 'flat' output file
  - ▷ I.e., no special headers, section information, symbol tables, etc.
  - ▷ Pad to multiple of one sector (512 bytes)
- Line 95: assemble the boot sector
  - ▷ Also uses 'bin' format
  - ▷ The rule depends on the setup code and kernel image files, so we know how many disk sectors they occupy
  - ▷ Note that it is always exactly 512 bytes, due to an assembler directive at the end of bootsect.asm

## Makefile (continued)

- Line 64: produce the floppy image file from which Bochs will boot
- It's just a concatenation of the boot sector, setup code, and kernel image file

# Outline

- Goals / background
- Implementation
  - ▷ Makefile
  - ▷ boot sector and setup
  - ▷ initialization
  - ▷ operation
- Conclusions

# Initial machine state

- x86 PCs boot in 'real mode'
- Using the infamous Intel segment/offset address scheme
- Can only directly address low 1MB of memory
- Operand sizes are (by default) 16 bits
- BIOS services are available
  - ▷ disk I/O, keyboard I/O, screen output, etc.

# The boot sector

- I chose to boot Bochs from a virtual floppy
  - ▷ This is the easiest way to boot
- Boot sector is the first sector of the floppy
- Loaded by BIOS at address 07C0:0000
- Files: bootsect.asm, defs.asm
- Note: the design of my boot sector was heavily influenced by Kernel Toolkit's boot sector, which in turn is a simplified version of Linux's boot sector

# Purpose of boot sector

- Load the 16-bit setup code and kernel image from the floppy, into memory
- Jump to the setup code
- That's it!
- We have only 510 bytes to work with, so it has to be simple

# Boot sector operation

- Line 31: we move the boot sector up to INITSEG (address 0x90000)
  - ▷ Why? Because Kernel Toolkit and Linux do it that way.
- Line 46: make default data segment (ds) the same as the new location of boot sector
- Line 50: move stack segment (ss) to old location of boot sector
  - ▷ There's nothing useful there, now
- Line 56: a loop to read the 16-bit setup code (setup.asm) into memory in SETUPSEG

# Boot sector operation (continued)

- Line 78: a loop to read the kernel image into memory at KERNSEG (address 0x10000 == 64K)
  - ▷ Recall that we linked the kernel at base address 0x10000
- Line 111: Jump to setup code!
- Line 113: The `Read_Sector` function reads a single sector of the floppy (addressed 'logically') into memory in given segment and offset
  - ▷ A very simple and slow approach
  - ▷ A better approach would be to read an entire track at a time

# Setup code

- perform initialization needed to execute 32 bit C code
- can occupy an arbitrary number of sectors
  - ▷ so code can be larger and more sophisticated than boot sector
- can use BIOS services

# Setup tasks

- Tasks performed by setup code:
  - ▷ detect amount of memory installed
  - ▷ detect other hardware (currently not implemented)
  - ▷ create initial IDT and GDT
    - these data structures are needed by the processor for protected mode operation
  - ▷ initialize interrupt controllers
  - ▷ enable A20 address line
  - ▷ switch to protected mode
  - ▷ jump to kernel entry point

# Setup (details)

- Source file: setup.asm
- Line 25: use BIOS to detect amount of memory installed
- Line 37: disable interrupts, since we're not ready to deal with them
  - ▷ means that we can't use the BIOS anymore
- Line 39: load initial (temporary) IDT and GDT
  - ▷ the IDT is empty, since we're not dealing with interrupts yet
  - ▷ the GDT defines 32 flat address spaces for kernel code and data
  - ▷ Actual GDT at line 170

# Setup: PIC initialization

- Line 45: call `Init_PIC` function
- The interrupt controllers (master and slave Intel 8259A's) take interrupts from hardware devices, and deliver them to the processor in a controlled manner
- Term: PIC
  - ▷ stands for 'Programmable Interrupt Controller'
- Pentium and above have a more sophisticated built-in interrupt controller, the I/O APIC
  - ▷ which by default operates in 8259A-compatible mode :-)

# Setup: PIC initialization (continued)

- The reason we need to reprogram the PICs is that the BIOS programs them to route IRQ lines to interrupts 8-23
  - ▷ But some of these are reserved by Intel for internal processor-generated interrupts!
- We reprogram them to route hardware IRQs to generate interrupts 32-47
- Term: IRQ
  - ▷ stands for 'Interrupt ReQuest'
  - ▷ An IRQ line is a signal connected to an input pin of the PICs
  - ▷ Typical x86 PCs have 16 IRQs

# Setup: PIC initialization (continued)

- Line 100: The code to reprogram the PICs
  - ▷ Was stolen directly from Linux
  - ▷ See Intel 8259A datasheet for details
  - ▷ Note that all IRQs are masked (blocked) initially

# Setup: A20 address line

- Line 46: call `Enable_A20` function
- Using segment/offset real mode addressing, it is possible to generate physical addresses above 1MB
  - ▷ e.g., `FFFF:0100` yields physical address `0x1000F0`
- On the original IBM PC and XT, such addresses wrapped around to low 64KB
- Some software depended on this behavior!

## Setup: A20 address line (continued)

- So for compatibility, the AT disabled address line 20 by default!
  - ▷ The AT had 24 physical address lines, for 16MB of memory max
- A spare pin in the keyboard controller was used to enable the A20 line
- Line 139: the code to enable the A20 line

# Setup: Entering protected mode!

- Line 48: protected mode is enabled by setting bit 0 in the MSW (Machine Status Word), which is the low 16 bits of the CR0 register
- However, the processor continues executing in 16-mode until we make an explicit jump to a 32-bit code segment
- Line 52: a far jump into 32 bit code
  - ▷ Uses the kernel code segment described in the GDT
  - ▷ `KERNEL_CS` is a 'segment selector', which references entry 1 in the GDT
  - ▷ A 'far' jump is one that explicitly specifies the code segment

# Setup: Executing 32 bit code

- Line 62: start of 32 bit code
- Line 65: reload all data segment registers to refer to the kernel data segment (otherwise they would still refer to 16 bit data segments)
- Line 73: set up initial kernel stack
- Line 75: build a `Boot_Info` data structure to pass to kernel entry point
  - ▷ defined in file `bootinfo.h`
  - ▷ can be used to pass useful information to kernel
  - ▷ currently, just the amount of memory

# Setup: Executing 32 bit code (continued)

- Line 89: push a return address on the stack (in case we want to return from the kernel entry point)
  - ▷ I don't know why we would
- Line 92: Jump to the kernel entry point!

# Outline

- Goals / background
- Implementation
  - ▷ Makefile
  - ▷ boot sector and setup
  - ▷ **initialization**
  - ▷ operation
- Conclusions

# Kernel entry point

- File `main.c`, function `Main()`
- Call initialization functions
- As a demonstration, read keystrokes and print them to screen
- Also, test thread creation and destruction

# Kernel entry point (continued)

- Lines 18, 41: initialize the kernel `.bss`
  - ▷ The part of the executable image for uninitialized static and global data
- At this point, we are completely ready to execute arbitrary code, call functions, return values, etc.
  - ▷ Shows that C's runtime environment has very minimal requirements
  - ▷ Which is why it's so frequently used in OS development

# Kernel initialization

- Initialization process:
  - ▷ Screen output
  - ▷ Memory
  - ▷ Interrupts
  - ▷ Threads
  - ▷ Keyboard driver

# Screen output

- In files `screen.{h,c}`
- Implements a very simple scrolling terminal
- VGA text mode:
  - ▷ mapped into memory at address `0xB8000`
  - ▷ even bytes are characters
  - ▷ odd bytes are attributes (foreground and background colors)
- The hardware cursor can be manipulated with port I/O to VGA registers
  - ▷ `screen.c`, line 111: `Update_Cursor()`

# Screen output (continued)

- screen.c, line 265: the `Print()` function
  - ▷ `printf()`-style formatted output
  - ▷ essential for debugging :-)

# Memory initialization

- Files mem.{h,c}
- Initializes GDT (again), files gdt.{h,c}
  - ▷ Mostly so GDT lies within kernel, rather than setup code (which may be overwritten)
- Create an array of Page structs, one for each page of physical memory
  - ▷ mem.h, line 29
  - ▷ Status flags
  - ▷ Next free page, for kernel freelist

# Memory initialization (continued)

- `mem.c`, Line 94: `Init_Mem()`
  - ▷ Classify memory pages as unused, available, kernel, or hardware
  - ▷ Organize available pages into a freelist
- The 'ISA hole'
  - ▷ Physical addresses from 640KB - 1MB used for VGA, BIOS, etc.
- Once memory and interrupts are initialized, can allocate memory
  - ▷ `mem.c`, line 168: `Alloc_Page()`, `Alloc_Page_Atomic()`
  - ▷ `mem.c`, line 200: `Free_Page()`, `Free_Page_Atomic()`
  - ▷ regular versions functions are used with interrupts disabled
  - ▷ `_Atomic` versions of functions are used with interrupts enabled

# Interrupts

- Files `int.{h,c}`, `lowlevel.asm`
- Build a real IDT: files `idt.{h,c}`
- Term: IDT
  - ▷ ‘Interrupt Descriptor Table’
  - ▷ Contains entries corresponding to interrupts that may be generated
  - ▷ Each descriptor specifies a procedure or task to handle the interrupt

# Interrupts (continued)

- `idt.c`, line 39: `Init_IDT()`
  - ▷ the IDT is initialized using handler entry points defined in `lowlevel.asm`
  - ▷ For each interrupt, we define an interrupt gate that refers to the appropriate handler entry point
  - ▷ `idt.h`, line 24: `Interrupt_Gate` struct shows how a C struct with bitfields can be used to represent a hardware data structure

# Low-level interrupt handling

- File: `lowlevel.asm`
- What happens when an interrupt is generated?
  - ▷ processor pushes `eflags`, `cs`, and `eip` registers onto current stack
  - ▷ interrupts are disabled (as if a `cli` instruction were executed)
  - ▷ some interrupts additionally push an error code
- Line 169: table of entry points, one for each interrupt

# Low-level interrupt handling (continued)

- Building the handler entry points
  - ▷ Lines 49, 59: `Int_With_Err` and `Int_No_Err` macros
  - ▷ `Int_No_Err` macro pushes a 'fake' error code, so stack layout is always the same, regardless of interrupt type
  - ▷ Push the interrupt number
  - ▷ Jump to common interrupt handling code: `Handle_Interrupt`, line 134

# Handle\_Interrupt

- Line 138: Save registers that may be modified by handler
  - ▷ Line 18: definition of `SaveRegisters` macro
- Line 141: set a global variable to indicate that an interrupt is in progress
- Line 146: get address of C function to handle interrupt from a table (`g_interruptTable`, defined at `idt.c`, line 29)
- Line 153: push address of `Interrupt_State` struct (which is just a description of the stack, defined in `int.h`) as argument to C handler function
- Line 154: Call the C handler function

# Handle\_Interrupt (continued)

- Line 158: Restore registers
  - ▷ Line 33: definition of RestoreRegisters macro
- Line 161: unset global variable, since handler is finished
- Line 164: return from the interrupt. When the saved eflags value on the stack is restored, interrupts are reenabled (as though a sti instruction were executed)

# Note: interrupt handling and privilege levels

- Interrupt handling is more complicated when there are multiple privilege levels involved
- OS must define a TSS (Task State Segment) to define stacks for each privilege level
  - ▷ TSS structures can also be used for task-switching
- Since I only use privilege level 0 (most privileged), I didn't need to worry about it
- Would be required to have a true user/kernel distinction

# Thread/scheduler initialization

- Files `kthread.{h,c}`
- `kthread.c`, line 153: `Init_Scheduler()`
- Creates a thread context object and stack for the initial thread of control (i.e., the one currently executing)
  - ▷ Uses memory pages that are known a priori
  - ▷ The thread of control in `Main()` is scheduled just like any other thread
- Creates idle thread and reaper thread
- I'll discuss thread support in more detail later

# Keyboard initialization

- Files `keyboard.{h,c}`
- `keyboard.c`, line 159: `Init_Keyboard()`
  - ▷ initialize key buffer and shift state status variable
  - ▷ install an interrupt handler for the keyboard IRQ
  - ▷ unmask the keyboard IRQ in the PIC, so it can start delivering keyboard interrupts to the processor
- IRQ and keyboard operation discussed later

# Outline

- Goals / background
- Implementation
  - ▷ Makefile
  - ▷ boot sector and setup
  - ▷ initialization
  - ▷ **operation**
- Conclusions

# Operation

- As a test of the screen, keyboard driver and threads, I wrote some code to read keystrokes and echo characters to the screen
- Makes use of the thread scheduling primitives
- main.c, line 31:
  - ▷ if key event is not a 'special' key, and the event is the key press rather than the release, get ASCII code (stored in low 8 bits) and echo to screen

# The keyboard driver

- Files `keyboard.{h,c}`
- Shows example of how an interrupt associated with an IRQ is handled, and how an interrupt can wake up threads that are waiting for an event

# Handling a keyboard interrupt

- Each keystroke (press and release) generates an interrupt
  - ▷ keyboard.c, line 97: the interrupt handler
- The key's scan code can then be read from the keyboard controller
  - ▷ Line 105: get status of keyboard controller, see if a scan code is available
  - ▷ Line 110: get the scan code

# Handling a keyboard interrupt (continued)

- The keyboard driver then translates scan codes to 'key codes', using lookup tables
  - ▷ The key code tries to encode all useful information about the key event, including the ASCII character code if appropriate
  - ▷ Key code returned also depends on the current shift state (whether the shift keys are down or up)
- keyboard.c, line 144: put keycode in buffer
- keyboard.c, line 148: wake up thread(s) waiting for key event

# Handling a keyboard interrupt (continued)

- Since keyboard interrupts are generated by one of the PICs, some special handling is needed at the end of the handler
- keyboard.c, line 152: call to `End_IRQ()`
  - ▷ This function sends an 'End Of Interrupt' command, or 'EOI', to the relevant PIC(s)
  - ▷ This lets the PIC know that the device was serviced, and that it may proceed to send other IRQ interrupts

# Handling a keyboard interrupt (continued)

- `irq.c`, line 70: definition of `End_IRQ()`
  - ▷ From the interrupt number in the `Interrupt_State()` struct (which was created by the `Handle_Interrupt()` code in `lowlevel.asm`), we determine which IRQ was handled
  - ▷ Based on the IRQ, we send an EOI to the master (IRQs 0-7), or both the slave and master (IRQs 8-15)

# Threads

- Files `kthread.{h,c}`
- A thread consists of a thread context object (struct `Kernel_Thread`) and a stack page
- The context object is used to store register contents when the thread is inactive

# Thread states

- A thread can be in one of the following states:
  1. executing (`g_currentThread`, `kthread.c`, line 27)
  2. waiting to execute (on the run queue)
  3. waiting for an event to occur (on a wait queue)
  4. dead, waiting to be deallocated (on the reaper queue)
- The threads are cooperatively scheduled

# Thread context switch

- File `switch.asm`
- Functions `Switch_To_Thread` and `Restore_Thread`
  - ▷ `Switch_To_Thread` saves context of current thread, activates a new thread
  - ▷ `Restore_Thread` just activates a new thread
- Context switch is really pretty simple, just saving and restoring registers

# Thread context switch (continued)

- Since `Switch_To_Thread` is a function, the thread that called it will push its instruction pointer (return address) onto the stack
  - ▷ Thus, the top of an inactive thread's stack always has the address of the instruction where execution should resume
  - ▷ Explains the calls to `Push()` in `Start_Thread()`

# Start\_Thread

- kthread.c, line 173
- Line 179: Allocate two pages of memory, one for the context object, one for the stack
- Line 197: initialize registers for new thread — stack pointer (esp) is the only one that's important
- Line 212: push argument to thread start function onto thread stack, to pass data to thread
- Line 213: push address of Shutdown\_Thread() function, which the thread start function will 'return' to

## Start\_Thread (continued)

- Line 216: push address of thread start function, which the Launch\_Thread() function will 'return' to
- Line 221: push address of Launch\_Thread() function, which performs initialization needed before the start function is executed
- So, the thread will execute Launch\_Thread() first, then startFunc(), then Shutdown\_Thread()

# Shutdown\_Thread

- `kthread.c`, line 64
- Executed when a thread start function returns
- Line 74: find a new runnable thread
- Line 77: put the thread on the 'graveyard queue', and wake up the reaper thread
- Note that we can't call `Switch_To_Thread()`, since the current thread is going away and we don't want to save its context
- Instead we put the address of the new runnable thread in the `eax` register, and jump to `Restore_Thread`

# Schedule

- `kthread.c`, line 259
- Called when the current thread is being deactivated, and has been placed on a queue for later reactivation
- Finds a new thread to run, and switches to it
- `Schedule()` is the basis of the `Yield()` and `Wait()` synchronization primitives

# Yield, Wait, and Wake\_Up

- `kthread.c`, line 278: `Yield()`
  - ▷ Give up the CPU if another thread is runnable
  - ▷ Otherwise, do nothing
- `kthread.c`, line 336: `Wait()`
  - ▷ Puts the thread on the wait queue
  - ▷ Finds a new thread to run
- `kthread.c`, line 353: `Wake_Up()`
  - ▷ Atomically transfers all threads in a wait queue to the run queue, so they can be scheduled

# Event driven keyboard input

- The keyboard driver uses the thread synchronization primitives to implement an event-driven interface
- keyboard.c, line 200: `Wait_For_Key()`
- Loops checking to see if a key is available in the buffer.
  - ▷ If so, returns it.
  - ▷ If not, waits in the keyboard wait queue for one to become available.

# Outline

- Goals / background
- Implementation
  - ▷ Makefile
  - ▷ boot sector and setup
  - ▷ initialization
  - ▷ operation
- Conclusions

# Conclusions

- Writing programs to run on bare x86 hardware isn't too difficult
- There are lots of good tools and documentation
- I think that learning about hardware and device level programming is critical to understanding how operating systems work (and thus, all other software)
- GeekOS can serve as a template for getting code running on x86 hardware

# Bochs/gcc/nasm as a teaching platform?

- There are many approaches to introductory OS projects
  - ▷ Nachos: simulated MIPS CPU
  - ▷ CMSC 412: DOS, Borland C
- Many intro projects try to hide details of hardware
  - ▷ I.e., write a 'fake' operating system

# Bochs/gcc/nasm as a teaching platform?

- Personally, I prefer the Bochs approach
  - ▷ It's realistic
  - ▷ And, it's easy to restart, and has a built-in debugger
  - ▷ Code that runs on Bochs has a decent chance of running on actual hardware
- I think that it would be possible to provide code to students to take care of some of the low-level details (for example, the boot sector, setting up the IDT and GDT, etc.)
- More investigation is needed

# Future work?

- Timer interrupt
- Real user/kernel distinction (i.e., a process model)
- Paging
- Disk device driver