GeekOS Overview

A. Udaya Shankar
shankar@cs.umd.edu

September 15, 2012

Abstract This document gives an overview of the GeekOS distribution. It describes some operations in GeekOS in more detail, in particular, initialization, low-level interrupt handling and context switching, thread creation, and user program spawning.

Contents

1 Introduction

The GeekOS distribution considered here is the one available from:

```
svn co https://svn.cs.umd.edu/repos/geekos/network
```

It contains only source code (in C, x86 assembly (mostly NASM, some AT&T), Makefile, Perl). After executing the makefiles, it will also contain object code and executables that can run on a PC-like hardware platform (x86 processor, memory, IO devices, etc). In this class, the hardware platform is simulated by QEMU.

The directories and files of the GeekOS distribution are listed in appendix A. Briefly:

- Directory `build` has makefiles for starting QEMU with GeekOS and user programs. Its subdirectories, which are initially empty, will hold object and executable modules. In particular, there will be two disk images: diskc, containing a PFAT filesystem with the GeekOS image and user programs; and diskd, initially raw and empty.
- Directories `src/geekos` and `include/geekos` contains the kernel code. Executed by QEMU's processor in kernel mode. You will be adding and modifying significant parts of the files here. You should understand very well what is already there in order to have any hope of gracefully completing the projects.
- Directory `src/user` contains user programs that run on GeekOS. Executed by QEMU's processor in user mode.
- Directory `src/libc` contains C entry functions for system calls. User programs call these functions to obtain OS services. Executed by QEMU's processor in user mode (but switches to kernel mode while executing system calls). Header files are in directory `include/libc`.
- Directory `src/common` has heap manager `bget`, output formatter `fmtout`, string manipulation `string`, and `memmove`. Nothing specific to operating systems here. Header files are in directory `include/libc`.
- Directory `src/tools` contains code for constructing the disk images that is supplied to QEMU. In particular, `buildFat.c` constructs the PFAT file system on diskc.
- Directory `scripts` contains Perl scripts, some of which are used in the makefiles.

Section 2 describes the PC hardware simulated by QEMU. For more details, see the makefiles and documentation at `www.qemu.org`. (Note: the QEMU in linuxlab is an old version.)

Section 3 describes the x86 processor in "real mode". Section 4 describes the x86 processor in "protected mode". For more details, see "IA-32 Intel Architecture Software Developer's Manual, Volume 3" on Intel's website.

Section 5 describes the boot process (`bootsect.asm`, `setup.asm`) and GeekOS initialization (`main.c`).

Section 6 describes the context state of a thread and the low-level steps for context switching and interrupt handling (in `lowlevel.asm`).

Section 7 describes the steps for starting kernel threads and user threads and spawning user programs.

Section 8 identifies "subsystems" of the OS and lists the associated files from the distribution.

2 Qemu

QEMU simulates a PC-like hardware. The QEMU configuration achieved by makefile includes the following. Below addresses are referred to by their hex values or their source code names or both, for example, "0xB800" or "VIDMEM_ADDR" or "0xB800 / VIDMEM_ADDR".

- Processor: Intel 386.
- BIOS: When QEMU is started (corresponding to power up), BIOS loads diskc/sector 0 into memory at offset 0 of memory segment 0x07C0 (mapping to memory address 0x07C00), and the processor starts executing at that address. Thus that disk sector should contain the boot sector.
- Memory: 10 MBytes.
- PIC (programmable interrupt controller, 8259A): receives interrupts from IO devices (keyboard, dma, ide, floppy drive) and funnels them to the processor.

  Ports: 0x20, 0x21, 0xA0, 0xA1 (for loading interrupt vectors?)
- PIT (programmable interval timer): generates interrupts at programmable interval.

  IRQ: 0 (to PIC)

  Ports: 0x40–43
- Keyboard

  IRQ: 1 (to PIC)

  Ports: 0x64 / KB_CMD; 0x60 / KB_DATA.
- VGA (monitor)

  Video memory: 0xB8000–0x100000; 0xB8000 / VIDMEM_ADDR; CRT_ADDR_REG; etc.
- IDE: accomodates up to 4 hard disks.
  Drive 0 (diskc) has a PFAT file system with the GeekOS image and user programs.
  Drive 1 (diskd) is a raw "empty" disk (appears only in later projects).

  IRQ: ?

  Ports: 0x1F6 / IDE_DRIVE_HEAD_REGISTER; IDE_DATA_REGISTER; IDE_SECTOR_COUNT_REGISTER; etc.
- Floppy drive: Holds a 1.44MB floppy disk.

  IRQ: ?

  Ports: 0x3F0 (FDC_BASE); FDC_STATUS_REG; FDC_DATA_REG; etc.
- DMA:

  Ports: 0x00 (DMA_BASE); DMA_COMMAND_REG; DMA_STATUS_REG; DMA_REQUEST_REG; etc.

3 Intel x86 real mode

The x86 processor can be in one of several modes. Only two of them, “real” mode and “protected” mode, are relevant for GeekOS. The processor starts in real mode upon power-up or reset. Here, it is a 16-bit machine (Intel 8086) with a linear address space of 1MB ($= 2^{20}$), addressed using a combination of a 16-bit segment and a 16-bit offset.

Registers The processor has the following 16-bit registers (assembly names used below):

- Main registers: in each, the 8-bit halves are independently addressable.
  AX: primary accumulator; halves AH (higher) and AL (lower).
  BX: base, accumulator; halves BH and BL
  CX: counter, accumulator; halves CH and CL
  DX: accumulator, other functions; halves DH and DL
- Index registers:
  SI: source index
  DI: destination index
  BP: base pointer
  SP: stack pointer
- Status register:
  Flags: carry, parity, auxiliary, zero, sign, trap, interrupt, direction, overflow
- Segment registers:
  CS: code segment
  DS: data segment
  ES: extra segment
  SS: stack segment
- IP: instruction pointer

Addressing The processor can address 1MB (2^{20} bytes) of memory. A 20-bit memory address is constructed by combining a 16-bit `segment` (from a segment register) and a 16-bit `offset` as follows:

- `16×segment + offset` // equivalently: `(segment « 4) + offset`

The address is usually denoted by `segment:offset`.

Stack, IO, interrupts The hardware stack grows towards lower memory addresses. Push and pop is in terms of 2-byte words. Stack top is pointed to by SS:SP. Stack bottom is pointed to by SS:FFFF.

16-bit IO (port) address space, each referencing an 8-bit IO register. There are 256 interrupts (hardware and software).

4 Intel x86 protected mode

The x86 processor switches from real mode to protected mode upon executing a certain instruction. In protected mode, the processor is a 32-bit machine with many more features, some of which are described next.

The processor can switch between 4 privilege levels: 0–3, in decreasing order of privilege; 0 is kernel mode and 3 is user mode. A task has a separate stack for each level.

The linear address space is 4GB ($= 2^{32}$).

16-bit IO (port) address space, each referencing an 8-bit IO register. There are 256 interrupts (hardware and software).

Segmented memory

The linear address space can be segmented, with an address being formed by combining a 16-bit "segment selector" and a 32-bit "offset". Briefly, the segment selector indexes into a "segment descriptor table" in memory, which yields a 64-bit "segment descriptor" that points to a segment (in memory). There is a "global descriptor table" (GDT) and zero or more "local descriptor tables" (LDTs).

A **segment selector** contains the following:

- 1 bit: indicates GDT or LDT.
- 13 bits: index into GDT or LDT.
- 2 bits: protection level of segment.

A **segment descriptor** contains the following:

- linear base address of a segment: 32 bits
- limit (size) of the segment: 20 bits
- descriptor privilege level (dpl): 2 bits
- type of segment (data, code, system, tss, gate): 4 bits
- present (i.e., in memory): 1 bit
- Various 1-bit attributes

The GDT (global descriptor table) entries point to kernel segments and optionally user segments. GDT entry 0 cannot be used to access memory but it does serve as a "null segment selector". There is a GDTR register in the processor that points to the GDT.

An LDT (local descriptor table) is like the GDT except that it is local to task (its entries point to segments of that task) and entry 0 can be used to access memory. There can be zero or more LDTs in memory. (In GeekOS, each user process gets an LDT.) There is a LDTR register in the processor that points (via the GDT) to the LDT currently being used (if any).

Paging

Linear or segmented memory modes can be direct (no paging) or paged. If paged, the linear addrress is [dir, table, offset]:

- dir: indexes into page directory, yields base addr of page table
- table: indexes into page table, yields base addr of page
- physical addr = [page base addr, offset]

Interrupts and task switching

An interrupt indexes into an "interrupt descriptor table" (IDT) in memory, which yields a 64-bit "gate" that points to the interrupt handler and indicates its privilege level. There is a IDTR register in the processor that points to the IDT.

If the interrupt handler's privilege level is numerically lower than that of the interrupted task, the processor also switches to another stack. The location of this new stack is available in a "task state segment" (TSS) in memory, which is pointed to by a task register (TR) in the processor.

(The TSS can also be used to automatically store and retrieve the rest of the processor's state upon a task switch. But GeekOS does not exploit this feature: it maintains only one TSS and uses it only for the stack pointer; it saves and loads the rest of the processor state in software.)

An **interrupt gate** contains the following:

- segment selector (for the segment containing the handler code): 16 bits
- offset within segment (pointing to the handler code): 16 bits
- descriptor privilege level (dpl): 2 bits
- type of segment (data, code, system, tss, gate): 4 bits
- present (in memory): 1-bit

When a task is interrupted and the interrupt handler is at the *same privilege level* as the interrupted task: the processor pushes on the current stack the EFLAGS, CS, and EIP registers (i.e., pertaining to the interrupted task) and (for certain interrupts) an error code.

When a task is interrupted and the interrupt handler is at a *numerically lower privilege level*, a stack switch occurs. The SS and ESP for the stack to be used by the handler are obtained from the current TSS. On this new stack, the processor pushes the SS and ESP of the interrupted task and then (as before) the EFLAGS, CS, and EIP registers and error code (if present).

A "return from interrupt" `IRET` instruction undoes the above (including popping the interrupted task's SS and ESP if they are saved on stack).

Processor registers

The processor has the following registers.

- 8 general purpose registers (each 32-bit):
  EAX: accumulator
  EBX, ECX, ESI, EDI: pointers to data segment; counters
  EDX: IO pointer
  ESP: stack pointer (in SS segment)
  EBP: pointer to data on stack (in SS segment)
- 6 segment registers (each has a 16-bit part + "invisible" 64-bit part):
  CS: code segment register
  SS: stack segment register
  DS, ES, FS, GS: data segment registers

  The 16-bit part is a segment selector. The 64-bit invisible part caches the segment descriptor (from GDT or LDT) pointed to by the segment selector.
- GDTR: 48-bit, points to GDT: 32 bits for GDT base addr, 16 bits for GDT size (in bytes).
- IDTR: 48-bit, points to IDT: 32 bits for IDT base addr, 16 bits for IDT size (in bytes).
- LDTR: 16-bit segment selector (+ invisible 64-bit); points (via GDT) to an LDT.
- TR: 16-bit segment selector (+ invisible 64-bit): points (via GDT) to a TSS.
- EIP: 32-bit instruction pointer (used with CS).
- EFLAGS: 32-bit status and control register: carry, overflow, sign, interrupt enable, new task, etc.
- CR0–CR4: 32-bit control registers: paging enable, cache enable, cache write-mode, protected/real mode, page fault, etc.
- Other registers: debug, memory type range, machine check, etc.

5 Booting and kernel initialization

Upon powering up the PC platform, BIOS loads diskc/sector 0 (of 512 bytes) at offset 0 of memory segment 0x07C0 (BOOTSEG) and the processor starts executing in real-mode from that address (0x07C00).

The makefiles have put src/bootsect.asm (in machine language) in that sector. Thus when the processor powers up, src/bootsect.asm is in memory starting at location BOOTSEG:0 and the processor starts executing the machine instruction at that location. From this point until the kernel is completely initialized, the processor is the only active "thread" in the system. It does the following:

- bootsect.asm: from BeginText to after_move:
  Moves the 512 bytes at BOOTSEG:0 to INITSEG:0 and jumps to INITSEG:0.
- bootsect.asm: from after_move to load_kernel:
  Loads the diskc sector containing setup.asm to memory SETUPSEG:0.
- bootsect.asm: from load_kernel to ReadSector):
  Loads the diskc sectors containing the OS kernel image into memory starting at KERNSEG:0. Then jumps to location SETUPSEG:0 and starts executing setup.asm.
- setup.asm: from BeginSetup to setup_32):
  Determines the size of extended memory available, kills the floppy motor (which is not used henceforth), points GDTR and IDTR to temporary GDT and IDT tables (in setup.asm), initializes A20 address line, initializes the PIC (to bypass BIOS), enters protected mode, and jumps to setup_32 (setting the processor's CS register to KERNEL_CS).
- setup.asm: from setup_32 to just before .returnAddr:
  Sets data and stack segment registers (DS, ES, FS, GS, SS) to KERNEL_DS, pushes on the stack a Boot_Info struct and a pointer to the struct, then jumps to KERNEL_CS:ENTRY_POINT (which points to function Main in geekos/main.c).

The memory now looks as shown in appendix B (under the column titled "At end of setup").

The processor now starts executing Main, which initializes the OS. There is still only one "thread" executing. We refer to it as the "initial kernel thread". In executing Main, this thread initializes the OS kernel and enters itself in the OS data structures, thus becoming a true thread.

- Init_BSS (defined in geekos/mem.c:
  Zeros the BSS (global variables area) of the kernel image.
- Init_Screen (defined in geekos/screen.c:
  Blanks the VGA screen and initializes its hardware cursor.
- Init_Mem (defined in geekos/mem.c):
  Calls Init_GDT (defined in geekos/gdt.c):
  - Creates the (permanent) GDT (static variable s_GDT).
  - Entry 1 points to the kernel code segment and entry 2 to the kernel data segment.
  - Loads the GDT base address and limit into GDTR.

  Treats memory as a sequence of 4KB pages. Creates (in kernel memory) a list of Page structs corresponding to the memory pages, each storing the attributes of its page (kernel, available for users, allocated, etc). Global variable g_pageList points to the list. Also creates a list of the available pages (s_freeList).

  Calls Init_Heap (defined in geekos/malloc.c) to initialize the kernel heap. (Malloc itself is implemented by bget.)
- Init_CRC32 (skipped).
- Init_TSS (defined in geekos/tss.c):

GeekOS uses a single TSS (static variable s_theTSS). Zeros the TSS struct, adds the TSS descriptor to GDT, updates LDTR.

- Init_Interrupts (defined in geekos/int.c):

  Calls Init_IDT (defined in geekos/idt.c):

  - Creates the (permanent) IDT (static variable s_IDT) with 256 interrupt gate entries (one for every exception and interrupt). The first 32 entries are for exceptions and traps. The remaining entries are for external interrupts, i.e., external interrupt j is mapped to entry 32+j.
  - Each IDT entry points to an entry point in geekos/lowlevel.asm. [The latter gets a pointer to the appropriate interrupt handler function (from g_interruptTable in idt.c) and calls handler with the appropriate Interrupt_State argument.]
  - Each IDT entry is at kernel privilege level, except for the syscall trap, which is at user privilege level.

  Installs a pointer to a dummy interrupt handler function in every g_interruptTable entry (in idt.c).

  Loads the IDT base address and limit into IDTR.

- Init_Scheduler (defined in geekos/kthread.c:

  Creates a Kernel_Thread object for the initial kernel thread and indicates that as the currently executing thread (g_currentThread). (At this point, the initial kernel thread becomes a true OS thread.)

  Creates an idle thread (runs when there is no other thread to run) and makes it runnable.

  Creates a reaper thread (responsible for cleaning up terminated threads) and makes it runnable.

  Initializes some queues of pointers to Kernel_Thread objects: s_allThreadList is a list with an entry for every thread; s_runQueue is a queue with an entry for every runnable thread; and g_currentThread indicates the currently executing thread.

- Init_Traps (defined in geekos/trap.c):

  Installs interrupt handlers for interrupts 12, 13 and 0x90 (syscall) (in g_interruptTable). The handler for interrupt 12 (stack exception) terminates the current thread. The handler for interrupt 13 (general protection failure) terminates the current thread. The handler for interrupt 0x90 calls the syscall handler function.

- Init_Timer (defined in geekos/timer.c): Initializes the timer. Installs interrupt handler for timer interrupt (IRQ 0, corresponding to IDT entry 32). Enables timer interrupt.

- Init_Keyboard (defined in geekos/keyboard.c):

  Initializes the keyboard state. Installs interrupt handler for keyboard interrupt (IRQ 1, corresponding to IDT entry 33). Enables keyboard interrupt.

- Init_DMA (defined in geekos/dma.c):

  Resets the DMA controller.

- Init_Floppy (defined in geekos/floppy.c): (skipped)

- Init_IDE (defined in geekos/ide.c):

  Reset the IDE controller and drives. Start "IDE request" thread, to wait for requests to IDE. (Why no interrupt handler?)

- Init_PFAT (defined in geekos/pfat.c): Registers the PFAT file system interface to the virtual file system.

- Init_GFS2, Init_GOSFS, $\cdots$, Init_RIP: (skipped)

- Mount_Root_Filesystem: mounts the root drive (diskc) as a PFAT file system to the virtual file system (in vfs.c) at root prefix "/".

- Spawn_Init_Process: starts the user shell program.

6 Context switching

Context state

The context state of a thread is stored in three structures, all reachable from the first:

- A `Kernel_Thread` struct (defined in `geekos/kthread.h`). This contains the kernel stack pointer, various kernel-related state (refcount, pid, etc.), and pointers to stack and user context (see below).
- A stack page. This is the kernel stack of the thread. When the thread is not executing, the *processor* state of the thread is stored here as follows:

  userSS, userESP [present only if thread was stopped in user mode] // stack interior
  eflags,
  eip (= return address),
  cs (= code segment selector),
  error code, interrupt number,
  gp and seg registers // stack top

  Thus the thread can be resumed simply by popping the gp and seg processor registers, clearing the error code and interrupt number, and executing "return from interrupt" (`IRET`).
- A `User_Context` struct (defined in `geekos/user.h`). This is present only if the thread is a user thread, i.e., started by spawing a user program. It contains user-level OS state (LDT, code/data/stack selectors, entry address, etc.).

Stopping and resuming threads

The context switching code appears in the following two functions (both in file `lowlevel.asm`):

- `Handle_Interrupt`:

  Assumes that the current thread got here via an interrupt (external, trap, or exception).

  Constructs the interrupt state of the current thread, calls the C interrupt handler, and finally either resumes the current thread or switches it out and switches in a thread from the run queue.
- `Switch_To_Thread`:

  Assumes the following (verify each and check if it matters):
  - the current thread got here via a call (not an interrupt) with a thread pointer arg on stack;
  - the current thread has already been moved to the run/wait queue;
  - if the current thread has a user context then it is exiting (Is this important?).

  Constructs the context of the current thread and switches in the thread pointer's threaad.

In both functions, the context switching code makes use of the kernel stack of the current thread (i.e., the one to be switched out). Think about what can go wrong if this is not done properly.

`Handle_Interrupt`

```
// here on (external or trap) interrupt; stack as follows:
//    [userSS, userESP] (if user mode was interrupted),            [stack interior
//    eflags, cs, eip, error code, intrpt num                      [stack top]

// save interrupt state of current thread on stack and call C handler
push gp and seg registers                 // completes interrupt state on stack
push esp                                  // pointer to interrupt state
call C interrupt handler                  // get address from g_interruptTable in int.c

if current thread is to be switched out   // according to g_preemptionDisable, g_needReschedule
    // NOTE: using previous thread's kernel stack
    move current thread to run queue;
    get a thread from run queue and make current;
    set esp to its kernel stack (avail in thread's context).

process signal if present                 // not present in distribution

activate user context if thread has one   // update LDTR, s_TSS.esp0, s_TSS.ss0, etc.

pop gp and segment registers

IRET
```

`Switch_To_Thread(threadptr)`

```
// switch out current thread (it has already been moved to run/wait queue?)
// switch in the thread pointed to by threadptr (latter on stack)
// here on a call from Schedule (and not from an interrupt).
// current thread has no user context or is exiting. (Correct?)
// Stack: threadptr   (= arg to Switch_To_Thread)                  [stack interior]
//        return addr in Schedule (= eip)                          [stack top]

change current thread's stack to following (so it can be switched in later):
    threadptr,                                                     // stack interior
    eflags,
    return addr in Schedule (= eip),
    fake error code, fake intrpt num,
    gp and seg registers                                           // stack top
save esp and clear numTicks on current thread struct
// current thread's context is now saved accurately

// switch in threadptr's thread
// NOTE: using previous thread's kernel stack
restore esp to point to threadptr         // pass over previous thread's interrupt state
make threadptr's thread current

process signal if present                 // not present in distribution

activate user context if thread has one   // update LDTR, s_TSS.esp0, s_TSS.ss0, etc.

pop gp and segment registers

IRET
```

7 Starting threads and spawning user programs

Starting a kernel thread

`Start_Kernel_Thread(startFunc, arg, priority)`

`Create_Thread`:
- get memory for kthread struct and for stack;
- initialize kthread fields: stackPage, esp, numTicks, pid, etc.

`Setup_Kernel_Thread`:
- configure kthread's stack so that when this kthread is switched in (in `lowlevel.asm`), it executes `Launch_Thread`, then `startFunc(arg)`, then `Shutdown_Thread`.

  Stack bottom:
  `startFunc arg`, `Shutdown_Thread addr`, `startFunc addr`,
  `eflags` (with intrpts off), `KERNEL_CS` (CS), `Launch_Thread addr` (EIP),
  fake error code, fake intrpt number
  fake gp registers, fake seg registers
  Stack top

Add to runQ

Starting a user thread

`Start_User_Thread(userContext)`

`Create_Thread`:
- get memory for kthreadd object and stack; initialize (as with kernel thread)

`Setup_User_Thread`:
- point `kthrd.userContext` to `userContext`
- fix up (kernel) stack as above except:
  first push userSS and userESP (avail from `usercontext`)
  have interrupts on in eflags

Add to runQ

Spawning a user program

`Spawn(programPathname, command, userContext)`

Load user prog:
- get file from file system (`vfs.c`, `pfat.c`),
  unpack into elf header and content, extract exeFormat (`elf.c`).
- get max virtual address of program and argBlockSize (from exeFormat),
  acquire memory 1 for program segment, arg block and user stack,
  load program segment into memory 1,
  format argblock in memory 1,
  acquire memory 2 for usercontext and initialize fields (size, ldt, entry point).

`Start_User_Thread(userContext)`

8 OS subsystems

Each subsection below identifies a "subsystem" of the OS and lists the associated files.

8.1 Utilities

The following files provide non-OS-specific functionality, such as debug macros, output formatting, strings, generic lists, linking maps, etc.

- `libc/bget.h`, `common/bget.c`, `geekos/bget.h`: heap structure.
- `malloc`: memory manager; wrapper for `bget`.
- `geekos/bitset.h|c`: bitset structure.
- `libc/fmtout.h`, `common/fmtout.c`, `geekos/fmtout.h`: output formatting.
- `geekos/ktypes.h`: aliases to integer and char types, min/max functions, etc.
- `geekos/kassert.h`: debugging macros (`KASSERT`, `TODO`, `PAUSE`, etc).
- `common/libuser.h`: includes user libray (`conio.h`, `sema.h`, `sched.h`, `fileio.h`).
- `geekos/list.h`: generic list structure.
- `common/memmove.h`: standard "memory move" function.
- `geekos/range.h`: checking memory range containership.
- `libc/string.h`, `common/string.c`, `geekos/string.h`: string manipulation.
- `geekos/symbol.h|asm`: symbol mangling macros (for linking C and asm).

8.2 Memory system

Physical memory managment: divides physical memory into 4KB pages, keeps track of the pages (kernel, user, free, kernel heap, etc.), gives out memory when needed (e.g., for process creation, data structures, etc.), gets back memory when released.
Files: `geekos/malloc.*`, `geekos/mem.*`.

Segmented memory management: implements segmentation over physical memory; creates segment selectors and descriptors, maintains GDT.
Files: `geekos/segment.*`, `geekos/gdt.*`.

8.3 Process management

Kernel process managment: kernel thread state; thread queues; creation, deletion and switching of kernel threads; thread signalling and synchronization.
Files: `geekos/kthread.*`, `geekos/tss.*`, `geekos/lowlevel.asm` (function `Switch_To_Thread`).

User process management: augmenting kernel threads with user context and user process creation, deletion, switching. `libc/process.*`, `geekos/user.*`, `geekos/userseg.c`, `geekos/tss.*`, `geekos/lowlevel.asm` (function `Switch_To_Thread`).

User program loading: loading a user executable (obtained from diskc) into memory.
Files: `geekos/elf.*`, `geekos/argblock.*`,

8.4 Interrupt system

This comprises the mapping from interrupt entry points (in IDT) to interrupt handlers and the mapping from interrupt handlers back to resuming the interrupted processes. Covers both external (hardware) interrupts and internal interrupts (exceptions, traps).
Files in `geekos`: `idt`, `int`, `irq`, `trap`, `lowlevel.asm` (function `Handle_Interrupt`, table `g_entryPointTable`).

8.5 Syscall system

Syscalls are all instances of a trap 0x90; i.e., `trap.c` forwards it to the appropriate syscall handler.
Files in `geekos`: `trap` (function `Syscall_Handler`), `syscall`.

8.6 Device drivers

This comprises the functions for I/O on hardware devices and the interrupt handlers for handling interrupts issued by these devices.

Files in `geekos`: `timer`, `screen`, `keyboard`, `floppy`, `ide`, `dma`, `io`.

8.7 Console

The console is the user-level "device" consisting of keyboard and screen.

Files: `include/libc/conio.h`, `src/libc/conio.c`, `geekos/syscall` (handlers for syscalls in `conio`).

8.8 File system

This comprises the virtual file system, the user interface to the virtual file system, the concrete file systems (pfat, gsfs2, gosfs) that can be mounted on the virtual file system, and the block device interface to the hardware disk devices.

OS side (all in `geekos`): `vfs`, `pfat`, `gosfs`, `gsfs2`, `blockdev`, `bufcache`, `syscall` (`fileio` syscall handlers).

User side: `libc/fileio.*`.

A GeekOS distribution listing

```
root directory:

  build:
    common: <initially empty>
    geekos: <initially empty>
    libc: <initially empty>
    tools: <initially empty>
    user: <initially empty>
    Makefile
    Makefile.common
    Makefile.darwin
    Makefile.linux
    Makefile.linuxlab
    Makefile.linux.x86_64
    Makefile.submitserver

  include:
    libc:
    geekos:

  src:
    common:
    libc:
    tools:
    user:
    geekos:

  scripts:
    dobuildlib
    eipToFunction
    findaddr
    generrs
    kerninfo
    mkcdisk
    mkuprog
    numsecs
    pad
    pcat
    pw
    random_port
    scan
    zerofile
```

```
include/libc:
   bget.h       *
   conio.h      *
   cyclone:
   fileio.h     *
   fmtout.h     *
   ip.h
   libuser.h    *
   net.h
   process.h    *
   sched.h
   sema.h
   signal.h
   socket.h
   string.h     *
```

```
src/common:
  bget.c        *
  fmtout.c      *
  memmove.c     *
  string.c      *
```

```
src/libc:
   compat.c
   conio.c      *
   entry.c
   fileio.c     *
   libuser.h    *
   lowlevel.s
   net.c
   process.c    *
   sched.c
   sema.c
   signal.c
   socket.c
   user.c       *
```

```
src/tools:
   buildFat.c   *
   fake-blockdev.c
   gfs2f.c
   Makefile
```

```
src/user:
   arp.c
   b.c
   cat.c
   c.c
   cp.c
   echoclnt.c
   echoserv.c
   ethrecv.c
   ethsend.c
   ethsendx.c
   gfs2f.c
   gfs2test.c
   ifconfig.c
   ipsend.c
   long.c
   ls.c
   mkdir.c
   mount.c
   null.c
   p5test.c
   rec.c
   recvbyte.c
   route.c
   sched1.c
   sched2.c
   sched3.c
   schedtest.c
   sem-p1.c
   sem-p2.c
   sem-p3.c
   sem-ping.c
   sem-pong.c
   semtest1.c
   semtest2.c
   semtest.c
   sendbyte.c
   setacl.c
   setuid.c
   shell.c
   sync.c
   touch.c
   type.c
   whoami.c
   workload.c
   write.c
```

```
include/geekos:
  alarm.h
  argblock.h                   *
  bget.h <= /libc/bget.h>      *
  bitset.h                     *
  blockdev.h                   *
  bootinfo.h                   *
  bufcache.h                   *
  crc32.h
  defs.h
  dma.h                        *
  elf.h                        *
  errno.h
  fileio.h                     *
  floppy.h                     *
  fmtout.h <= /libc/fmtout.h>        *

  gdt.h                        *
  gfs2.h                       *
  gosfs.h                      *
  ide.h                        *
  idt.h                        *
  int.h                        *
  io.h                         *
  irq.h                        *
  kassert.h                    *
  keyboard.h                   *
  kthread.h                    *
  ktypes.h
  list.h                       *
  malloc.h                     *
  mem.h                        *
  net:
  paging.h
  pfat.h                       *
  projects.h
  range.h                      *
  screen.h                     *
  segment.h                    *
  sem.h
  signal.h
  string.h <= /libc/string.h>  *
  symbol.h
  synch.h
  syscall.h                    *
  sys_net.h
  timer.h                      *
  trap.h                       *
  tss.h                        *
  user.h                       *
  vfs.h                        *
```

```
src/geekos:
  alarm.c
  argblock.c                   *
  bitset.c                     *
  blockdev.c                   *
  bootsect.asm                 *
  bufcache.c                   *
  crc32.c
  defs.asm                     *
  depend.mak
  destroyThread
  dma.c                        *
  elf.c                        *
  fd_boot.asm
  floppy.c                     *
  gdt.c                        *
  gfs2.c                       *
  gosfs.c                      *
  ide.c                        *
  idt.c                        *
  int.c                        *
  io.c                         *
  irq.c                        *
  keyboard.c                   *
  kthread.c                    *
  lowlevel.asm                 *
  main.c                       *
  malloc.c                     *
  mem.c                        *
  net:
  paging.c
  pfat.c                       *
  README.txt
  screen.c                     *
  segment.c                    *
  sem.c
  setup.asm                    *
  signal.c
  symbol.asm
  synch.c
  syscall.c                    *
  timer.c                      *
  trap.c                       *
  tss.c                        *
  user.c                       *
  userseg.c                    *
  uservm.c
  util.asm
  vfs.c                        *
```

B Memory organization after setup and after Main

Address	Name(s) in source code	At end of setup	At end of Main
000000		start BIOS code/data (and PIC interrupt vectors)	
001000	PAGE_SIZE	end BIOS code/data	start available pages
007C00	BOOTSEG:0	bootsect loaded here by BIOS	
010000	KERNSEG:0 KERNEL_START_ADDR KERNEL_THREAD_OBJ	start kernel image	end available pages start kernel image
	BSS_START		kernel global structures initialized
	BSS_END		
	kernEnd	end kernel image	end kernel image start available pages
090000	INITSEG:0	bootsect reloaded here	
090200	SETUPSEG:0	setup loaded here	
090400	MEMMAPSEG:0	setup stack (grows towards 0)	
0A0000	ISA_HOLE_START	start ISA hole (hardware use)	end available pages
0B8000	VIDSEG:0	start video memory	
100000	ISA_HOLE_END KERN_THREAD_OBJ	end ISA hole start initial kernel thread object	start initial kernel thread object
101000	HIGHMEM_START KERN_STACK	initial kernel thread stack start of kernel heap	initial kernel thread stack start of kernel heap
111000	pageListEnd = HIGHMEM_START + KERNEL_HEAP_SIZE	end of kernel heap	end of kernel heap start available pages
	endOfMem		end available pages