GeekOS Overview

A. Udaya Shankar
shankar@cs.umd.edu

Jeffrey K. Hollingsworth
hollings@cs.umd.edu

February 25, 2015

Abstract This document gives an overview of the GeekOS distribution (for 412 Spring 2015) and related background on QEMU and x86. 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.

Previous versions: 2/2014 version (multi-core); 9/2013 version (single-core).

Contents

1. Introduction

The GeekOS distribution considered here is for CMSC 412 Spring 2015:

```
svn co https://svn.cs.umd.edu/repos/geekos/spring2015
```

It contains source code (in C, x86 assembly (mostly NASM, some AT&T), Makefile, Perl) and resulting executables for a PC-like hardware platform (multi-core x86 processor, memory, IO devices, etc). In this class, the hardware platform is simulated by QEMU. The project is done in a Linux VM.

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 can be initially empty) holds 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 (`www.qemu.org`).

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".

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

Section **??** describes spin locks and interrupt-disabling functions.

Section 7 describes the context state of a thread and the low-level steps for context switching and interrupt handling (in `lowlevel.asm`).

Section 8 describes the steps for starting kernel threads and user threads and spawning user programs.

Section 9 gives a brief overview of the filesystem (from virtual filesystem to disks). The subsequent sections go into a bit more detail: virtual filesystem (section 10); PFAT filesystem (section 11); fileio (section 12); buffer caches (section 13); block devices (section 14); IDE devices (section 15).

Section 16 identifies "subsystems" of the OS and lists the associated files from the distribution.

Appendix A is a directory listing of the GeekOS distribution.

Appendix B describes the memory after setup and after initialization.

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".

- Two or more 386 cpus in SMP (symmetric multi-processing) configuration.
- PIC (programmable interrupt controller, 8259A): receives interrupts from IO devices (keyboard, dma, ide, floppy drive) and funnels them to the IO APIC.
- APIC (advanced programmable interrupt controller):
  A local APIC per cpu: rcvs interrputs from IOAPIC, and rcvs/sends interrupts to other cpus.
  An IO APIC: routes interrupts from PIC to local APIC(s).
- BIOS:
  At power up, BIOS designates one cpu as "primary" and the others as "secondaries".
  Puts each secondary cpu in halt state and clears its APIC.
  Loads diskc/sector 0 into memory at offset 0 of memory segment 0x07C0 and starts the primary cpu executing at that address (0x07C00).
  Sets up MP configuration table (# cpus, APIC addrs, etc) in specified memory area.
- Memory: 512 MBytes (?)
- PIT (programmable interval timer): generates interrupts at programmable interval.
  IRQ: 0 (to IOAIPIC?).
  Ports: 0x40–43
- Keyboard
  IRQ: to IOAPIC (via 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: to IO APIC (via PIC?)
  Ports: 0x1F6 / IDE_DRIVE_HEAD_REGISTER; IDE_DATA_REGISTER; IDE_SECTOR_COUNT_REGISTER; etc.
- DMA:
  IRQ: to IO APIC (via PIC?)
  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 does the following:

- Set one cpu as "primary" and the others as "secondaries".
- Puts each secondary cpu in halt state and clears its APIC.
- Sets up MP configuration table in specified memory area (FEC00000, FEE00000) indicating the number of cpus, IO APICs (1), buses, etc. Assigns consecutive ids to the the local APICs, starting with 0 for the primary; these ids also serve as cpu ids.
- Loads diskc/sector 0 into memory at offset 0 of memory segment 0x07C0 (BOOTSEG), and starts the primary cpu executing at that address (0x07C00). The makefiles have put src/bootsect.asm (in machine language) in diskc/sector 0.

Thus the primary cpu, aka cpu 0, starts executing src/bootsect.asm starting at memory location BOOTSEG:0. From this point until the kernel is (almost) completely initialized, cpu 0 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 (defined in geekos/bootinfo.h) 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").

Cpu 0 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(0) (defined in geekos/gdt.c):
  - Creates a permanent GDT (static variable s_GDT[0]) for cpu 0.
  - Entry 1 of the GDT points to the kernel code segment and entry 2 to the kernel data segment.
  - Loads the GDT base address and limit into GDTR (of cpu 0).

  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):

  Creates a TSS (static variable s_theTSS[0]) for cpu 0. Zeros the TSS struct, adds the TSS descriptor to cpu 0's GDT, updates TR. [GeekOS uses a single TSS for each cpu, not one per process.]
- Init_Interrupts(0) (defined in geekos/int.c):

  Calls Init_IDT(0) (defined in geekos/idt.c):
  - Creates a permanent IDT (static variable s_IDT[0]) for cpu 0, 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 (g_entryPointTableStart) in geekos/lowlevel.asm. [lowlevel.asm latter gets a pointer to the interrupt handler function (from g_interruptTable in idt.c) and calls the 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 (of cpu 0).
- Init_SMP (defined in geekos/smp.c):

  Identifies (from MP config table) how many secondary cpus there are, and get them running. For each secondary cpu i, it assigns an initial stack page, sends an inter-procesor interrupt (IPI) to wake up cpu i, and another IPI to start cpu i executing at start_secondary_cpu (in setup.asm). (This is the "initial kernel thread" for cpu i.)

  The awakened cpu i reaches setup_2nd_32 (in geekos/setup.asm), where it enters 32-bit mode, and then calls Secondary_Start (in geekos/smp.c), where it does the following:
  - Init_GDT(i)
  - Init_TSS()
  - Init_Interrupts(i)
  - Init_Secondary_VM() // nothing for now
  - Init_Scheduler(i, CPUs[i].stack):
    // creates a thread object for this thread, makes it currently running (g_currentThreads[i])
    // creates runnable Idle thread for cpu i
  - Init_Traps()
  - Init_Local_APIC(i) // initializes cpu i's APIC
  - Init_Timer_Interrupt() // installs interrupt handler
  - Exit() // Idle thread now takes over cpu i?

  Cpu i is left spinning until cpu 0 calls Release_SMP() later in Main.
- Init_VM() (defined in geekos/paging.c); nothing for now
- Init_Scheduler(0, KERN_STACK) (defined in geekos/kthread.c):

  Creates a Kernel_Thread object for the initial kernel thread and makes it the currently executing thread (g_currentThreads[0]). (At this point, the initial kernel thread becomes a true OS thread.)

  Creates Idle thread (runs when there is no other thread to run) for cpu 0 and makes it runnable.

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

  [Note: s_allThreadList is a list with an entry for every thread. s_runQueue is a queue with an entry for every runnable thread. g_currentThreads[i] indicates the currently executing thread on cpu i.]
- 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_Local_APIC(0)` defined in (`geekos/smp.c`) initializes the local interrupt controller for cpu 0.
- `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. (Currently IO APIC directs timer interrupt to cpu 0 only.)
- `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_IDE` (defined in `geekos/ide.c`):

  Reset the IDE controller and drives. Start "IDE request" thread, to wait for requests to IDE. (Interrupt handler?)
- `Init_PFAT` (defined in `geekos/pfat.c`): Registers the PFAT filesystem interface to the virtual file system.
- `Init_GFS2`, `Init_GOSFS`, `Init_CFS`: registers different filesystem interfaces.
- `Init_Alarm`
- `Init_Serial`
- `Release_SMP()` allows all secondary cpus to start running (and they quickly finish their initial threads and start running the Idle thread for that core??).
- `Init_Sound_Devices`
- `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 and waits.
- `Hardware_Shutdown`: after shell terminates.

6. Synchronization

6.1. Spin locks

A spin lock, in assembly language, is an integer that is 0 iff unlocked. It is accessed via the following functions (in lowlevel.asm), where x is a pointer to a lock.

```
Spin_Lock_INTERNAL(x):
  repeat
    busy wait until *x is false
    set eax to 1
    atomically swap eax and *x
  until eax equals 0
  return
```

```
Spin_Unlock_INTERNAL(x):
  set eax to 0
  atomically swap eax and *x
  return
```

In C, a spin lock is an Spin_Lock_t object (lock.h), consisting of an int (the assembly level lock) and two thread pointers (current and previous lockers). The spin lock is accessed via the following functions (in smp.c), where x is a pointer to a spin lock:

- Spin_Lock(x): wrapper to the corresponding assembly function
- Spin_Unlock(x): wrapper to the corresponding assembly function
- Try_Spin_Lock(x): like Spin_Lock(x) but it tries just once and returns 0 if unsuccessful.
- Is_Locked(x):

The GeekOS distro has several spin locks, eg, globalLock, kthreadLock, a lock for every list, etc. In particular, globalLock is also known as the "kernel lock" and "big" global lock. Functions lockKernel(), unlockKernel() and Kernel_Is_Locked() are wrappers for the corresponding operations on globalLock.

6.2. Disabling interrupts globally

Disable_Interrupts() disables interrupts (on the local cpu) *and* acquires the global lock (via lockKernel()). Thus it blocks any other cpu from completing Disable_Interrupts(), and in particular, from executing interrupt handlers.

Enable_Interrupts() releases the global lock and enables interrupts (on the local cpu).

Begin_Int_Atomic() calls Disable_Interrupts() if they are enabled and otherwise acquires the global lock. So it can be called with or without the global lock.

End_Int_Atomic(iflag) is the inverse, enabling interrupts if iflag is true.

6.3. Mutexes

In GeekOS, a mutex is a lock in which blocked threads wait in a queue. A Mutex object (in synch.h) consists of a "state" (int indicating locked or unlocked), an "owner" (thread holding the lock), and a "waitQueue" (queue where blocked threads wait).

Mutex_Lock and Mutex_Unlock (see synch.c) put a blocked thread on a wait queue until the lock is re-

leased. The former must be called with interrupts enabled.

```
Mutex_Lock(x):
  KASSERT(intrpts enabled, I do not hold x)
  Disable_Interrupts()
  while x locked
    wait on x's waitQueue   // calls Schedule
  update x's state and owner
  Enable_Interrupts()
```

```
Mutex_Unlock(x)
  KASSERT(intrpts enabled)
  Disable_Interrupts()
  KASSERT(I hold x)
  update x's state and owner
  wakeup a thread (if any) in x's waitQueue
  Enable_Interrupts()
```

6.4. Conditions

In GeekOS, a `Condition` object consists of a waitQueue (recall that this includes a spin lock). It is accessed via the following functions, where `c` is a condition pointer and `x` is an associated mutex.

```
Cond_Wait(c,x):
  KASSERT(intrpts enabled, I hold x)
  Mutex_Unlock(x)          // no preemption yet
  Disable_Interrupts()
  wait on c's waitQueue   // calls Schedule
  Enable_Interrupts()
  Mutex_Lock(x)
```

```
Cond_Signal(c)
  KASSERT(intrpts enabled,
          I hold assoc mutex)
  Disable_Interrupts()
  wakeup a thread (if any) in c's waitQueue
  Enable_Interrupts()
```

`Cond_Broadcast(c)` is like `Cond_Signal(c)` except that all threads in `c`'s waitQueue are woken up.

7. Context switching

7.1. 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`):
  esp: kernel stack pointer
  numTicks, totalTime, priority
  thread_queue link(s)
  pointer to stack page
  pointer to user context
  ...
- 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 spawning a user program. It contains user-level OS state (LDT, code/data/stack selectors, entry address, etc.).

7.2. Stopping and resuming threads

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

- `Handle_Interrupt`:

  // Thread comes here upon an interrupt (issued by itself or externally).
  // Cpu is using the thread's kernel stack

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

  // Thread comes here due a call (and not due to an interrupt).
  // The call's argument (on the stack) is a pointer to a thread object.
  // The current thread has already been moved to the run/wait queue.

  Constructs the context of the current thread (so it can be resumed later).
  Switches in the thread pointed to by the call's argument.

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.

7.3. Handle_Interrupt

// here on (external or trap) interrupt.
// Cpu using interrupted thread's kernel stack, which has following:
// - user.ss, user.esp (iff thread interrupted in user mode) // placed by hardware
// - eflags, cs, eip // by hardware
// - error code // by hardware or by g_entryPointTable code (in lowlevel.asm)
// - interrupt number // by g_entryPointTable code (in lowlevel.asm)
// [stack top]

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

if this thread is to be switched out // based on g_preemptionDisabled, g_needReschedule of cpu
    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).

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

release kernel lock if new thread has will have interrupts enabled

process signal if present // not present in distribution

pop gp and segment registers

IRET

7.4. Switch_To_Thread(threadptr)

// here on a call from Schedule (and not from an interrupt)
// current thread's object has already been moved to run/wait queue(?)

// switch in the thread pointed to by threadptr (latter on stack)
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

clear APIC interrupt info ??
get kernel lock if interrupted when interrupts enabled
save esp and clear numTicks on current thread struct
// current thread's context is now saved accurately

// switch in threadptr's thread
restore esp to point to threadptr // pass over previous thread's interrupt state
make threadptr's thread current

clear APIC interrupt info ??

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

process signal if present // not present in distribution

release kernel lock if new thread has will have interrupts enabled

pop gp and segment registers

IRET

8. Starting threads and spawning user programs

8.1. 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

8.2. Starting a user thread

`Start_User_Thread(userContext)`

`Create_Thread`:
- get memory for kthread 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

8.3. 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).

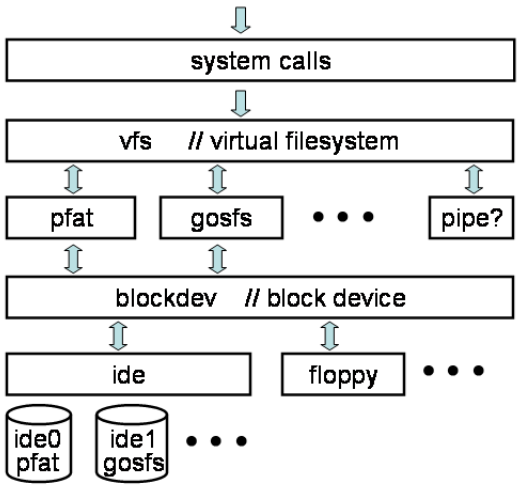
`Start_User_Thread(userContext)`

9. Filesystem overview

GeekOS has a virtual filesystem (VFS) onto which "concrete" filesystems, such as PFAT, GOSFS, GSFS2, and others are "mounted". VFS acts as a wrapper to these mounted filesystems, allowing them to be accessed by users in a uniform manner. Initially VFS consists of just an empty root directory ("/"). At the end of OS initializing, the PFAT filesystem in IDE 0 is mounted onto VFS at position "/c", after which user programs can access the PFAT filesystem as the VFS subdirectory "/c". Similarly, your GOSFS filesystem (in project 5) can be mounted at another point (say "/d") and accessed.

GeekOS also has a virtual block device into which any block-structured storage device (e.g., IDE, floppy) can be registered. The block device acts as a wrapper to these registered storage devices, allowing them to be accessed in a uniform manner. The users in this case are other parts of the kernel (filesystems, paging, etc.).

The figure below illustrates the context. User programs invoke system calls, which invoke functions in `vfs.c`, which invoke functions in `pfat.c` or `gosfs.c`, which invoke functions in `blockdev.c`, which in turn invoke functions in `ide.c`. (In project 5, you implement the functions in `gosfs.c`.)



The pfat functions that are called by vfs have names of the form `PFAT_<function>` (see `pfat.c`). Similarly, the gosfs functions that are called by vfs have names of the form `GOSFS_<function>` (see `gosfs.c`). Note that vfs does not call these functions by name. Rather vfs gets pointers to these functions at run time (when a filesystem type is registered, when a filesystem is mounted, etc.).

10. Vfs

Vfs functions called by "users" (syscalls, main, other systems in the kernel)

Filesystem operations

```
Format(*devname, *fstype)
Mount(*devname, *pathPrefix, *fstype)
Get_Paging_Device(void);
```

Mount-point operations

```
Open(*path, mode, **pFile);
Close(*file);
Stat(*path, *stat);
Sync(void);
```

File operations

```
FStat(*file, *stat);
Read(*file, *buf, len);
Write(*file, *buf, len);
Read_Fully(*path, **pBuffer, *pLen);
```

Directory operations

```
Create_Directory(*path);
Open_Directory(*path, **pDir);
Read_Entry(*file, *entry);
```

Vfs functions called by filesystem implementations (pfat, gosfs)

```
Register_Filesystem(*fstype, *fsOps)
Register_Paging_Device(*pagingDevice);
Allocate_File(*fileOps, filePos, endPos, *fsData, mode, *mountPoint)
```

Vfs static variables

- `s_vfsLock`: lock for ensuring atomicity of vfs operations.
- `s_mountPointList`: list of mountPoints, one for each mounted file system (e.g., pfat on dev1 at `/c`).
- `s_filesystemList`: list of fstypes (filesystem types), one for each registered filesystem (e.g., pfat).
- `s_pagingDevice`: registered paging device.
  Struct with following: fileName, blockDev, start sector, number of sectors

Each mountPoint is a struct containing the following:

- ops: pointers to functions `Open`, `Create_Directory`, `Open_Directory`, `Stat`, `Sync`, `Delete` in filesystem implementation (eg, `pfat.c`). (Note: `SetSetUid`, `SetAcl` omitted here.)
  Each function's args: mountpoint, path in filesystem, mode, pfile.
  Pointers supplied by filesystem when mounted.
- prefix: where filesystem is mounted wrt root (eg, "`/c`").
- blockDev: pointer to block device containing filesystem.
- fsData: filesystem info. Supplied by filesystem when mounted.

Each fstype is a struct containing the following:

- ops: pointers to functions `Format(blockDev)`, `Mount(mountPoint)` in filesystem implementation (eg, in `pfat.c`). pointers supplied by filesystem when registered.
- fsName: name of filesystemType (eg, "pfat").

`File` struct for each opened file or directory containing:

- ops: pointers to functions `FStat`, `Read`, `Write`, `Seek`, `Close`, `Read_Entry` in filesystem implementation (eg, in `pfat.c`).
  Set by filesystem when mounted. Function args include `*file`.
- filePos: current position in file.
- endPos: end position in file (i.e., length of file).
- fsData: for use by filesystem implementation.
- mode: mode of open file (read vs write). Set by Open(), Create_Directory(), Open_Directory().

- mountPoint: mountPoint of filesystem that file is part of. Set by Open(), Create_Directory(), Open_Directory().

Functions in `vfs`

`Register_Filesystem(fstype, fsOps)`:
add fstype to list of fstypes.

`Format(*devname, *fstype)`:
if [fstype is registered and has Format] and [device is registered (in blockdev) and opens]
call fstype.ops.Format(device)

`Mount(*devname, pathPfx, *fstype)`:
if [fstype is registered and has Format] and [device is registered (in blockdev) and opens]
create mountPoint(device, pathPfx), call fstype.ops.Mount(mountpoint), add mountPoint to mountPointList.

`Open(path, mode, pFile)`: // wrapper for mounted filesystem Open
split path into pfx and sfx
get mountPoint at pfx
call mountPoint.ops.Open(mountpoint, sfx, mode, pFile).

`Close`, `Stat`, `FStat`, `Read`, `Write`, `Seek`, `Create_Directory`, `Open_Directory`, `Delete`:
each is a wrapper for the corresponding mounted filesystem operation

`Sync()`: // wrapper for Sync of all mounted filesystems

`ReadFully(path, buffer, pLen)`:
Stat(path)
Open(path)
Read repeatedly until all of stat.size is read

11. Pfat

Pfat static variables

`s_pfatFileOps:` // instance of File_Ops (defined in vfs.h)

- &PFAT_FStat
- &PFAT_Read
- &PFAT_Write
- &PFAT_Seek
- &PFAT_Close
- 0: // (Read_Entry

`s_pfatDirOps:` // instance of File_Ops (defined in vfs.h)

- &PFAT_FStat_Dir
- 0, 0, 0, // Read, Write, Seek
- &PFAT_Close_Dir
- &PFAT_Read_Entry

`s_pfatMountPointOps:` // instance of Filesystem_Ops (defined in vfs.h)

- PFAT_Open
- 0, // Create_Directory()
- PFAT_Open_Directory,
- PFAT_Stat
- PFAT_Sync
- 0 //Delete

`s_pfatFilesystemOps:` // instance of Filesystem_Ops (defined in vfs.h)

- 0 // Format
- &PFAT_FStat_Dir

Pfat structs

`bootSector`:

- magic: filesystem id
- fatOffset: start of FAT
- fatLength: length of FAT (in sectors?)
- rootDirectoryOffset: start of root directory
- rootDirectoryCount: number of items in root directory
- setupStart
- setupSize
- kernelStart
- kernelSize

`directoryEntry`:

- filename: 8 + 4 chars (including null terminator). 12 bytes
- readOnly, hidden, systeFile, volumeLabel, directory: each 1 bit
- time: 2 bytes
- date: 2 bytes
- firstBlock: 4 bytes
- fileSize: 4 bytes

`PFAT_Instance:` // in-memory info of mounted PFAT fs; kept in mountpoint.fsInfo.

- fsinfo: bootsector instance
- *fat: pointer to fat table
- *rootDir: pointer to rootDirEntry
- rootDirEntry:
- lock:
- fileList: PFAT file list

`PFAT_File:` // in-memory info of open PFAT file; kept in file.fsInfo.

- fsinfo: bootsector instance
- entry: directory entry of this file.
- numBlocks: number of blocks of file
- *fileDataCache:
- *validBlockSet: which data blocks of cache are valid

Pfat public functions

All exported when pfat filesystem is mounted.

`PFAT_FStat(*file, *stat):`
copy file.fsData info into stat

`PFAT_Read(*file, *buf, numBytes):`
// this function accesses file.fsData and file.mountPoint.fsData
set numBytes to min(endPos, filePos + numBytes)
traverse FAT (in file.mountpoint.fsData) for blocks of the file
for each block that is not in cache or not clean, read it into cache
copy relevant cache buffers into buf
update filePos
copy file.fsData info into stat

`PFAT_Write(*file, *buf, numBytes):`
return EACCESS // writes not allowed

`PFAT_Seek(*file, pos):`
set file.filePos to pos if in range

`PFAT_Close(*file, pos):`
return 0 // no-op

`PFAT_FStat_Dir(*dir, *stat):`
copy file.mountPoint.fsData.rootDirEntry into stat // only one directory

`PFAT_Close_Dir(*dir):`
return 0 // no-op

`PFAT_Read_Entry(*dir, *entry):`
if dir.filePos $\geq$ dir.endPos
return VFS_NO_MORE_ENTRIES
pfatDirEntry $\leftarrow$ dir.mountPoint.fsData.rootDir[dir.filePos++]
copy pfatDirEntry to entry.name.states

`PFAT_Open(*mountPoint, *path, mode, **pFile):`
if (mode is not O_READ) or (path is not a file entry in mountPoint.fsData)
return errorcode
get a pfatFile object for path // already cached?, else cache, etc.
create vfs file object (with pfatFileOps, pfatFile, etc) and return in **pFile

```
PFAT_Open_Directory(*mountPoint, *path, mode, **pDir):
    if (path is not "/")
        return errorcode
    create vfs File object and set object's
        ops to s_pfatDirOps
        filePos to 0
        endPos to mountPoint.fsData.fsinfo.rootDirectoryCount
        fsData to 0
    return pointer to File object via **pDir

PFAT_Stat(*mountPoint, *path, *stat):
    get pfatfile object for path from mountPoint.fsData
    copy info from pfatfile object into stat

PFAT_Sync(*mountPoint):
    return 0      // no-op; read-only fs

PFAT_Register_Paging_File(*mountPoint, *pfatInstance):
    return if paging device already registered (Get_Paging_Device())
    get dirEntry for file with PAGEFILE_FILENAME ("pagefile.bin"?)
    create pagedev and set its
        fileName to mountPoint.pathPfx + PAGEFILE_FILENAME
        dev to mountPoint.dev
        startSector to dirEntry.firstBlock
        numSectors to dirEntry.fileSize / SECTOR_SIZE
    Register_Paging_Device(pagedev)

PFAT_Mount(*mountPoint):
    allocate pfatInstance and bootsect
    read mountpoint.dev's sector 0 into bootsect      // using Block_Read
    copy bootsect's bootsector into pfatInstance.fsinfo
    return if fsinfo's magic number, FAT offset/size, root dir offset/size not ok
    allocate pfatInstance.fat      // in-memory FAT; fsinfo.fatLength gives size in sectors
    read FAT from dev into pfatInstance.fat      // using Block_Read
    allocate pfatInstance.rootDir      // in-memory rootDir; fsinfo.rootDirCount gives size in entries
    read root directory from dev into pfatInstance.rootDir      // using Block_Read
    set psfatInstance.rootDirEntry's fields (read-only/directory/fileSize)    // fake root directory entry
    initialize pfatInstance.lock
    clear pfatInstance.fileList
    PFAT_Register_Paging_File(mountPoint, pfatInstance)    // if present and unregistered
    set mountPoint.ops to s_pfatMountPointOps
    set mountPoint.fsData to pfatInstance

Init_PFAT(void):
    call vfs's Register_Filesystem("pfat", &s_pfatFilesystemOps)
```

12. Geekos/fileio.h

```
VFS_MAX_PATH_LEN 1023     // /d/d1/f1
VFS_MAX_FS_NAME_LEN 15    // "pfat", "gosfs", ...
VFS_MAX_ACL_ENTRIES 10   // max ACL entries per directory entry
SECTOR_SIZE 512    // sector size for all block devices
BLOCKDEV_MAX_NAME_LEN 15   // "ide0", "ide1", ...

// File permission flags for Open() VFS function
O_CREATE (create if file does not exist), O_READ, O_WRITE, O_EXCL, O_OWNER

struct VFS_ACL_Entry:
   uid, permission

struct VFS_File_Stat: // dir entry metadata; filled by vfs.Stat(), vfs.FStat()
   size, isDirectory, isSetuid, VFS_ACL_Entry acls[...]

struct VFS_Dir_Entry:  // dir entry structure; filled by vfs.Read_Entry()
   name[1024], VFS_File_Stat stats

struct VFS_Mount_Request:  // request to mount a filesystem
    devname (e.g., "ide1"), prefix (mount point), fstype (e.g., "gosfs")
```

To be completed .....

13. Bufcache

Comes between vfs/pfat/gosfs and blockdev.

No static variable. No local thread.

Bufcache structs

`FS_Buffer`: // holds one fsblock

- fsBlockNum: filesystem block number
- *data: in-memory data of block. May be out of sync with disk.
- flags: state of buffer (dirty, pending, ...)

`FS_Buffer_Cache`:

- *blockdev: associated device
- fsBlockSize: size of filesystem blocks
- numCached: current number of buffers (cached blocks)
- fsBufferList: list of buffers
- lock: lock for synchronization
- cond: condition variable for waiting for a buffer

Bufcache public functions

All called by vfs/pfat/gosfs.

`*Create_FS_Buffer_Cache(*blockdev, fsBlockSize)`:
malloc cache struct and set fields

`Sync_FS_Buffer_Cache(*cache)`:
lock cache, write out all dirty buffers, release lock

`Destroy_FS_Buffer_Cache(*cache)`:
synch and release all buffers and cache struct

`Get_FS_Buffer(*cache, fsBlockNum, **pBuf)`:
sets **pBuf to buffer fsBlockNum

14. Blockdev

Blockdev static variables

- s_blockdevLock: lock for ensuring atomicity of blockdev operations.
- s_deviceList: list of blockDevices, one for each registered block device (e.g., ide, floppy).

Each blockDevice is a struct of the following:

- name: name of block device.
- *ops: struct of pointers to functions Open, Close, Get_Num_Blocks in device driver (e.g., ide.c). Set when device is registered.
  Each function's arg: *dev.
- unit: device drive number
- inUse: opened?
- *driverData:
- *waitQueue: pointer to wait queue in device driver (e.g., ide). Set when device is registered. A server thread in device waits here.
- *requestQueue: pointer to request queue in device driver (e.g., ide). Set when device is registered. BlockRequests are queued here.

Each blockRequest is a struct of the following:

- *dev: block device.
- type: request type
- blockNum: number of block (to read, write, seek, ...)
- *buf: buffer for request
- requestState: pending, completed, error. (Volatile)
- errorCode: (Volatile)
- waitQueue: requesting thread waits here (until awakened by a server thread in device driver)

Blockdev functions

`Register_Block_Device(devname, devOps, unit, driverData, waitQueue, requestQueue):`
// called by device driver (e.g., ide.c)
malloc blockDevice struct, assign its fields, add to deviceList.

`Notify_Request_Completion(req,state,errorCode):` // called by device driver (e.g., ide.c)
wakeup(req.waitQueue),

`*Dequeue_Request(reqQueue,waitQueue):` // called by device driver (e.g., ide.c)
wait for non-empty requestQueue;
remove request from requestQueue and return it

`Block_Read(dev,blockNum,buf):` // called by user (e.g., vfs/pfat, vfs/gosfs)
create blockRequest, add to req.dev.requestQueue
wakeup(req.dev.waitQueue)
wait at req.waitQueue until req no longer pending

`Block_Write(dev,blockNum,buf):` // just like `Block_Read(.)`

`Open_Block_Device(devname, **pDev)` // // wrapper for dev.Open(); called by user (e.g., vfs/pfat)
lookup devname in blockDeviceList, and call device's Open(), set pDev to device

`Close_Block_Device(dev)` // wrapper for dev.Close(); called by user (e.g., vfs/pfat)
call dev.Close()

`Get_Num_Blocks(dev)` // wrapper for dev.get_Num_Blocks(); called by user (e.g., vfs/pfat)

15. Ide

Ide static variables

- `numDrives`: number of drives (4)
- `drives[i]`: holds drive i's configuration (heads, cylinders, sectors/track, bytes/sector).
- `s_ideWaitQueue`: ide wait queue. Exported when drive is registered.
- `s_ideRequestQueue`: ide request queue. Exported when drive is registered.
- `s_ideDeviceOps`: pointers to functions `IDE_Open`, `IDE_Close`, `IDE_Get_Num_Blocks`; all have arg `blockDev`.

Ide functions

`IDE_Open(*blockDev)`: // exported when drive is registered
null-op if `blockDev.inUse` false, else crashes.

`IDE_Close(*blockDev)`: // exported when drive is registered
null-op if `blockDev.inUse` true, else crashes.

`IDE_Get_Num_Blocks(*blockDev)`: // exported when drive is registered
returns number of (disk) blocks in disk.

`IDE_Read(driveNum, blockNum, *buffer)`:
reads disk block, doing busy waiting

`IDE_Write(driveNum, blockNum, *buffer)`:
writes disk block; busy waiting.

`IDE_Request_Thread(unused arg)`:
waits at ide wait queue (via blockdev.waitqueue) until awakened
ide request queue
dequeues request from ide request queue (via blockdev.requestqueue)
calls `IDE_Read` or `IDE_Write`
calls `blockdev.Notify_Request_Completion`.

`Init_IDE()`:
reset IDE controller and turn off interrupts
for each drive:
read drive configuration and whether ATA or ATAPI
register drive with blockdev
start kernel thread on IDE_Request_Thread(unused arg)

16. OS subsystems

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

16.1. 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 library (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).

16.2. Memory system

Physical memory management: 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.*.

16.3. Process management

Kernel process management: 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.*,

16.4. 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).

16.5. 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`.

16.6. 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`.

16.7. 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`).

16.8. 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 (spring 2015)

Top level

```
build  COPYING  include  LICENSE-klibc  scripts  sound  src
```

/scripts, /sound

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

./sound:
applause.wav  curve.wav  force1.wav  README  space.wav  untie.wav
```

/build

```
./build:
cleanSymLinks.py  common                depend.mak             diskc.img        diskd.img
geekos            libc                  Makefile               Makefile.common  Makefile.darwin
Makefile.linux    Makefile.linux.x86_64 Makefile.submitserver  mux.rb           output.log
pagefile.bin      routing.sh            routing.txt            sounds           tcp.sh
tools             user

./build/common:

./build/geekos:
hdbootsect  kernel.bin  kernel.exe  kernel.syms  net  setup.bin  sound

./build/geekos/net:

./build/geekos/sound:

./build/libc:
errno.c

./build/sounds:
login.wav  pianokey.wav  startup.wav

./build/tools:
builtFat.exe  gfs2f

./build/user:
arp.exe       b.exe         blkpipe.exe   cat.exe       c.exe         cp.exe        echoclnt.exe
echoserv.exe  ethrecv.exe   ethsend.exe   ethsendx.exe  execr1.exe    execr2.exe    execr3.exe
forkexec.exe  fork-p1.exe   forkpipe.exe  gfs2f.exe     gfs2test.exe  gfs2tst2.exe  halt.exe
ifconfig.exe  ipsend.exe    kill.exe      login.exe     long.exe      ls.exe        mkdir.exe
mlc.exe       mount.exe     multimlc.exe  nsp5test.exe  nullderf.exe  null.exe      p5test.exe
pipe-p1.exe   pipe-p2.exe   ps.exe        rec.exe       recvbyte.exe  rm.exe        route.exe
sched1.exe    sched2.exe    sched3.exe    schedtest.exe schedtst.exe  sem-p1.exe    sem-p2.exe
sem-p3.exe    sem-ping.exe  sem-pong.exe  semtest1.exe  semtest2.exe  semtest.exe   sendbyte.exe
setacl.exe    setuid.exe    shell.exe     spin.exe      sum.exe       sync.exe      time.exe
touch.exe     type.exe      whoami.exe    workload.exe  write.exe
```

/include

```
geekos  libc

./include/geekos:
alarm.h   apic.h      argblock.h  bget.h      bitset.h   blockdev.h  bootinfo.h  bufcache.h
cfs.h     cfsmodes.h  crc32.h     defs.h      dma.h      elf.h       errno.h     fileio.h
floppy.h  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      lock.h
malloc.h  mem.h       net         paging.h    pfat.h     pipe.h      projects.h  range.h
screen.h  segment.h   sem.h       serial.h    signal.h   smp.h       sound.h     string.h
symbol.h  synch.h     syscall.h   sys_net.h   timer.h    trap.h      tss.h       user.h
vfs.h

./include/geekos/net:
arp.h  ethernet.h  ipdefs.h  ip.h  ne2000.h  netbuf.h  net.h  port.h  rip.h  routing.h  socket.h
tcp.h  udp.h

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

./include/libc/cyclone:
cyclib.cys
```

/src

```
./src:
common  geekos  libc  tools  user

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

./src/geekos:
alarm.c     argblock.c  bitset.c   blockdev.c   bootsect.asm  bufcache.c  cfs.c      crc32.c
defs.asm    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      pipe.c     README.txt
screen.c    segment.c   sem.c      serial.c     setup.asm     signal.c    smp.c      sound
symbol.asm  synch.c     syscall.c  timer.c      trap.c        tss.c       user.c     userseg.c
uservm.c    util.asm    vfs.c

./src/geekos/net:
arp.c  ethernet.c  ip.c  ne2000.c  netbuf.c  net.c  rip.c  routing.c  socket.c  sys_net.c  tcp.c
udp.c

./src/geekos/sound:
docs  sound.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  spin.c

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

./src/user:
arp.c       b.c         blkpipe.c   cat.c       c.c         cp.c         echoclnt.c  echoserv.c
ethrecv.c   ethsend.c   ethsendx.c  execr1.c    execr2.c    execr3.c     forkexec.c  fork-p1.c
forkpipe.c  gfs2f.c     gfs2test.c  gfs2tst2.c  halt.c      ifconfig.c   ipsend.c    kill.c
login.c     long.c      ls.c        mkdir.c     mlc.c       mount.c      multimlc.c  nsp5test.c
null.c      nullderf.c  p5test.c    pipe-p1.c   pipe-p2.c   ps.c         rec.c       recvbyte.c
rm.c        route.c     sched1.c    sched2.c    sched3.c    schedtest.c  schedtst.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     spin.c      sum.c       sync.c       time.c      touch.c
type.c      whoami.c    workload.c  write.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
FEC00000	`IO_APIC_Addr`		Start of IO APIC region
FEE00000	`APIC_Addr`		Start of local APIC region