GeekOS 2018

Shankar

June 25, 2018
Goal

- Provide a very compact view of GeekOS
- Provide a framework for
  - understanding the internals of GeekOS
  - posing and answering coding questions in exams
  - describing what is to be done in projects
Hardware and devices (drivers + interrupt handlers)
Booting and kernel initialization
Kernel threads
User processes
Synchronization
Scheduling
Lowlevel.asm
Virtual filesystem
PFAT
Blockdev
Bufcache
- x86 cpus in SMP (symmetric multi-processing) configuration
- apics (interrupt controllers)
  - local apic: recv interrupts from io-apic, send/recv to other cpus
  - io-apic: route interrupts from io devices/timer to local apics
- diskc: kernel image; pfat filesystem with user programs
- emulated by QEMU running on linux (unix) environment
x86 (CPU)

- Has several modes: only “real” and “protected” modes relevant

- Real mode
  - Enters this mode upon power up
  - 16-bit machine (Intel 8086)
  - 20-bit segmented memory address: 1MB
  - 16-bit IO (port) address, 256 interrupts

- Protected mode
  - Enter this mode upon executing a certain instr in real mode
  - 32-bit machine with many more features
  - 4 privilege levels: 0 (kernel mode), 1, 2, 3 (user mode)
  - 32-bit segmented (+ optional paging) memory address: 4GB
  - 16-bit IO (port) address space, 256 interrupts
  - Geekos runs in this mode.
  - Rest of this section deals with protected mode
- **Address space**: 4GB (32-bit address)
- **Segment**: a contiguous chunk of address space
- Address formed from 16-bit **segment selector** and 32-bit offset
- Segment selector indexes into a **segment descriptor table**
  - [which table, index into table, protection level]
  - **global descriptor table** (gdt), **local descriptor table** (ldt)
- Yields a 64-bit **segment descriptor**, which points to a segment
  - [base addr, limit, privilege level, etc]
- If paging is on, the address is divided into [dir, table, offset]
x86: Interrupts

- 256 interrupts: 0–31 hw, rest sw (traps, exceptions, faults, etc)
- Interrupt indexes into a interrupt descriptor table (idt)
- Yields a 64-bit interrupt gate, which points to interrupt handler
  - [seg selector, offset, descriptor privilege level (dpl), etc]
- If interrupt-handler’s privilege-level = cpu’s privilege-level: cpu pushes on its current stack
  - its eflags, cs, eip, and an error code (for some interrupts)
- If interrupt-handler’s privilege-level < cpu’s privilege-level: cpu uses another stack whose location is in a task state segment (tss)
  - pushes its ss and esp // interrupted task’s stack
  - pushes eflags, cs, eip, error code (if present)
- Return-from-interrupt (IRET) undoes the above (in both cases)
x86: Registers

- `eax, ebx, ecx, esi, edi, edx`: “general purpose” (32-bit)
- `esp` (32-bit): stack pointer (in ss segment)
- `ebp` (32-bit): frame pointer (in ss segment)
- `eip` (32-bit): instruction pointer (in cs segment)

Segment registers (16-bit), each holds a segment selector
  - `cs` (code segment), `ss` (stack segment)
  - `ds, es, fs, gs` (data segment)

- `gdtr` (48-bit): addr and size of current gdt
- `idtr` (48-bit): addr and size of current idt
- `ldtr` (16-bit): selector to current ldt (via gdt)
- `tr` (16-bit): selector to current tss (via gdt)

- `eflags` (32-bit): carry, overflow, sign, interrupt enable, etc
- `cr0–cr4` (32-bit): paging enable, page fault, cache enable, etc.
- BIOS stores APICs config info at certain addresses

- Local APIC info starts at 0xFEE00000 (APIC_Addr)
  - offset 0x20 (APIC_ID) stores the apic id (= cpu id) // 0, 1, ...

- Get_CPU_ID(): // return cpu id of caller thread
  - disable interrupts
  - apicid ← read location APIC_Addr + APIC_ID
  - restore interrupts
  - return apicid

- IO APIC info starts at 0xFEC00000 (IO_APIC_Addr)
PIT timer + LAPIC timers

- PIT timer: interrupt TIMER_IRQ (=0)
- Each Local APIC has a timer: interrupt 32
- PIT timer is used only at boot to calibrate the LAPIC timers

Global and static variables

- g_numTicks // global tick counter
- DEFAULT_MAX_TICKS = 4 // default quantum
- g_Quantum = DEFAULT_MAX_TICKS
Timer Interrupt Handler(istate):  // simplified
    id ← Get_CPU_ID()
    ct ← get_current_thread()
    if id is 0:
        ++g_numTicks
    ++ct.numTicks
    if ct.numTicks >= g_Quantum:
        g_needReschedule[id]

Init_Timer():
    Install_IRQ(32, Timer Interrupt Handler)
    enable interrupt 32

Init_Local_APIC(cpuid):
    Install_IRQ(39, Spurious Interrupt Handler)  // SMP
    enable interrupt 39
    set timer timeout value                 // cpu 0 uses PIT to calibrate
VGA screen

- Ports: CRT_* regs (0x3D4, 0x3D5, etc)
  - access via io instr // eg, Out_Byte(port, value)
  - for refresh, scan rate, blanking, cursor control, etc

- Video memory: VIDMEM (0xb8000 – 0x100000)
  - holds characters to display // NUMROWS = 25, NUMCOLS = 80
  - access via read/write instrs // eg, VIDMEM[offset] = keycode

- Var console_state: row, col, esc, numeric arg, etc

- Update_Cursor() based on console state // ports used here only

- Put_Char_Imp(c): place char c at text cursor position

- Init_Screen(): clear screen, set “text cursor” to origin

- Print(*fmt, ...)
Ports
- input reg: `KB_DATA` (0x60)
- control reg: `KB_CMD` (0x64)
- status regs: `KB_OUTPUT_FULL` (0x01), `KB_KEY_RELEASE` (0x80)

Interrupt: `KB_IRQ` (1)

Static variables (for drivers, interrupt handler)
- `s_queue` // queue for incoming keycodes
- `s_keyboardWaitQueue` // threads waiting for kbd inputs
- `s_kbdQueueLock` // spinlock protecting `s_queue`
- `scantables` // map scancode to keycode
- `kbd state` // shift, esc, control, alt, etc
Keyboard

- **Keyboard Interrupt Handler (istate):**
  - if ports indicate byte available:
    - get byte; convert to keycode or update kbdstate
    - add keycode to s_queue // drop if full; spinlock ops
    - wakeup(s_keyboardWaitQueue)

- **Init_Keyboard():**
  - initialize static variables
  - Install_IRQ(KB_IRQ, Keyboard_Interrupt_Handler)
  - enable kbd interrupt

- **Wait_For_Key():**
  - disable intrpt
  - repeat
    - if s_queue has key, get it // spinlock ops
    - else wait(s_keyboardWaitQueue)
  - until got key
  - restore intrpt
IDE

- 16-bit transfer unit
- 2 hard disks
- PIO and DMA modes

Ports
  - IDE_identify regs  // show disk features
  - IDE_drive/cylinder/head/sector regs  // target disk block
  - IDE_command reg  // read/write
  - IDE_data reg  // successive words of io block show up here
  - IDE_status/control/etc regs  // busy, dma, interrupt, etc
IDE: drivers

- Static variables
  - s_ideWaitQueue: ide server thread waits here
  - s_ideRequestQueue: io request queued here

- IDE_Read(drive, blocknum, *buffer):
  convert blocknum to cylinder, head, sector
  update control and command regs
  read 256 words from data reg into buffer

- IDE_Write(...): like IDE_Read except write to data reg

- IDE_Request_Thread():
  forever: req = dequeue from request queue // blocking
  IDE_Read/Write(req) // synchronous, pio

- Init_IDE():
  register drives as block devices
  start kernel thread executing IDE_Request_Thread()
DMA controller (currently not used)

- Registers
  - memory addr
  - byte count
  - control regs (source, destination, transfer unit, etc)

- Usage for ide io
  - cpu sets up ide interface to initiate data transfer
  - cpu sets up dma interface

- Init_DMA()
- Reserve_DMA(chan)
- Setup_DMA(direction, chan, *addr, size)
Outline

Hardware and devices (drivers + interrupt handlers)
Booting and kernel initialization
Kernel threads
User processes
Synchronization
Scheduling
Lowlevel.asm
Virtual filesystem
PFAT
Blockdev
Bufcache
At power up, BIOS configures
- one cpu-lapic as **primary**, with id 0
- other cpu-lapics as **secondaries**, halted, with ids 1, 2, ...
- MP config table in memory
- loads diskc/block 0 (**bootsect.asm**) into memory
- cpu 0 (in real mode) starts executing it

**bootsect.asm**  // executed by cpu 0
- load the kernel image (from diskc) into memory and start executing it (**setup.asm**)  

**setup.asm**  // executed by cpu 0
- get memory size, redirect interrupts (bypass BIOS)
- enter protected mode, set cs to KERNEL_CS
- set ds, es, fs, gs, ss to KERNEL_DS, jump to main.c:Main
Kernel initialization: `Main()`–1 // executed by cpu 0

- blank VGA screen
- init cpu 0’s gdt, gdtr // `s_GDT[0]`; 1: code seg, 2: data seg // `NUM_GDT_ENTRIES = 32`
- organize memory into 4K pages // `g_pageList, s_freeList`
- init kernel heap
- init cpu 0’s tss, tr, gdt[3?] // `s_theTSS[0]`; one tss per cpu
- init cpu 0’s idt, idtr // `s_IDT[0]`
  - syscall entry’s dpl at user level, others at kernel level
  - addresses of interrupt handlers in `g_interruptTable[0]`; set them to dummy interrupt handler
- init SMP: for each secondary cpu i
  - allocate a page for cpu i’s kernel stack (CPUs[i].stack)
  - start cpu i executing `start_secondary_cpu` (in setup.asm) // cpu i does its initialization, then spins until cpu 0 releases it
Kernel initialization: Main()–2 // executed by cpu 0

- init scheduler(0): create threads // with Kernel_Thread objects
  - current thread {Main} // g_currentThreads[0]
  - idle thread {Idle-#0} // s_runQueue
  - reaper thread {Reaper} // s_runQueue

- init traps: 12: stack exception; 13: GPF; 0x90: syscall

- init devices: Local_APIC(0), keyboard, IDE, DMA

- init PFAT: register filesystem PFAT with vfs

- release SMP
  - allow each secondary cpu to exit its initialization; wait for that

- mount root filesystem
  - mount ide0 as PFAT fs at path “/a”

- spawn initial process // shell program

- hardware shutdown
Secondary CPU initialization // executed by CPU i

- **start_secondary_cpu** (in setup.asm)
  - enter protected mode
  - set DS, ES, FS, GS, SS to KERNEL_DS
  - set ESP to CPUs[i].stack // previously assigned by CPU 0
  - jump to Secondary_Start() (in smp.c)

- **Secondary_Start()** (in smp.c)
  - init GDT: point CPU i's GDTR to s_GDT[0] // uses CPU 0's GDT
  - init CPU i's TSS, TR, GDT[3+i?] // s_theTSS[i]
  - init CPU i's IDT (s_IDT[i]), IDTR
  - init scheduler(i): create threads // with Kernel_Thread objects
    - current thread {Main} // g_currentThreads[i]
    - idle thread {Idle-#i} // s_runQueue

- init traps, local APIC
- set flag informing CPU 0 that i is done
- Exit(0), which makes CPU enter scheduler
Hardware and devices (drivers + interrupt handlers)
Booting and kernel initialization
Kernel threads
User processes
Synchronization
Scheduling
Lowlevel.asm
Virtual filesystem
PFAT
Blockdev
Bufcache
Kernel threads: context and queues

- **Context** of a kernel thread:
  - Kernel_Thread struct + stack page

- **struct Kernel_Thread**:
  - esp, *stackPage, *userContext
  - **link for s_allThreadList** // constant
  - **link for current thread queue** // runq, waitq, graveyard
  - numTicks, totalTime, priority, pid, joinq, exitcode, ...
  - pid, owner, joinqueue, exitcode, affinity, ...

- **Thread queues**
  - s_allThreadList // all threads
  - s_runQueue // ready (aka runnable) threads
  - s_graveyardQueue // ended and to be reaped
  - various waitQueues // mutex, condition, devices, etc
  - *g_currentThreads[MAX_CPUS] // running thread
Starting kernel threads

- **Start_Kernel_Thread**: 
  - **Create_Thread**: get memory for `kthread` context (struct and stack page)  
    - init struct: stackPage, esp, numTicks, pid  
    - add to the all-thread-list
  - **Setup_Kernel_Thread**: configure stack so that upon switching in it executes `Launch_Thread`, then `startfunc`, then `Shutdown_Thread`
    - stack (bottom to top):  
      - `startfunc arg`, `Shutdown_Thread addr`, `startfunc addr`  
      - `0 (eflags)`, `KERNEL_CS (cs)`, `Launch_Thread addr (eip)`  
      - `fake error code`, `intrpt#`, `fake gp regs`  
      - `KERNEL_DS (ds)`, `KERNEL_DS (es)`, `0 (fs)`, `0 (gs)`
  - **Make thread runnable**: add struct to runq
CURRENT_THREAD: // return the thread struct of the caller
- disable interrupts
- ct ← g_currentThreads[GET_CPU_ID]
- restore interrupts
Outline

- Hardware and devices (drivers + interrupt handlers)
- Booting and kernel initialization
- Kernel threads
- User processes
- Synchronization
- Scheduling
- Lowlevel.asm
- Virtual filesystem
- PFAT
- Blockdev
- Bufcache
Context of a user process:
- Kernel_Thread struct + stack page + struct User_Context

struct User_Context:
- name[]
- ldt[2]  // code segment, data segment
- *ldtDescriptor  // segment descriptor
- *memory, size  // memory space for process
- ldtSelector  // index into gdt
- csSelector, dsSelector  // index into ldt
- entryAddr, argBlockAddr, stackPointerAddr
- *pageDir, *file_descriptor_table[]
- refCount, mappedRegions, etc
Spawn user process

- **Spawn(program, cmd, *kthread, background):**
  - read executable file from filesystem           // vfs, pfat
  - unpack elf header and content, extract exeFormat   // elf
  - mem ← malloc(program maxva + argblock size + stack page)
  - copy program segments into mem space
  - malloc usercontext and set its fields:
    - *memory ← mem
    - ldt, ldt selectors/descriptors
    - entry point, argblock, stack bottom, ...
  - *kthread ← Start_User_Thread(userContext)
Start user thread

- Start_User_Thread(uc, detached):  // “uc” is “usercontext”
- Create_Thread:
  malloc kthread struct and stack, init, add to all-thread-list
- Setup_User_Thread:
  point kthread.usercontext to uc
  configure kernel stack as if it was interrupted in user mode
  // stack (bottom to top):
  // uc.ds (user ss), uc.stackaddr (user esp)
  // eflags (intrpt on), uc.cs (cs), uc.entryaddr (eip)
  // errorcode, intrpt#, gp regs except esi  // fake
  // uc.argblockaddr (esi), uc.ds (ds, es, fs, gs)
  // How is termination handled?
- Make thread runnable: add struct to runq
Copying between user and kernel spaces

- **User_To_Kernel**(usercontext, userptr):  // kernel addr of useraddr
  return usercontext.memory + userptr

- **Copy_From_User**(dstInKernel, srcInUser, bufsize):
  ucontext ← CURRENT_THREAD.usercontext
  srcInKernel ← User_To_Kernel(ucontext, srcInUser)
  memcpy(dstInKernel, srcInKernel, bufsize)

- **Copy_To_User**(dstInUser, srcInKernel, bufsize):
  ucontext ← CURRENT_THREAD.usercontext
  dstInKernel ← User_To_Kernel(ucontext, dstInUser)
  memcpy(dstInKernel, srcInKernel, bufsize)
Hardware and devices (drivers + interrupt handlers)
Booting and kernel initialization
Kernel threads
User processes
Synchronization
Scheduling
Lowlevel.asm
Virtual filesystem
PFAT
Blockdev
Bufcache
Interrupt disable/enable: affects only this CPU

Disable Interrupts(): // abbrv: disable intrpt
    __asm__ "cli"

Enable Interrupts(): // abbrv: enable intrpt
    __asm__ "sti"

Begin_Int_Atomic(): // abbrv: disable intrpt
    ion ← true iff interrupts enabled
    if ion:
        Disable Interrupts()
    return ion

End_Int_Atomic(ion): // abbrv: restore intrpt
    if ion:
        Enable Interrupts()
Spinlocks

- Spinlock in assembly: an int that is 0 iff unlocked

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

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

- Spinlock in C: struct {lock, locker, ra, lastlocker}

- Spin_Lock(x): wrapper of assembly fn + update to locker, ra, ...

- Spin_Unlock(x): ..............................................................

  Ensure interrupts disabled before acquiring a spinlock // Why?

  Restore interrupts after releasing a spinlock
Some spinlock variables

- `globalLock`  // lockKernel(), unlockKernel(); smp.c
- `kthreadLock`  // kthread.c, user.c
- Every `list_t` in `DEFINE_LIST(list_t, node_t)` has a spinlock `lock`
  - Guards the list in list operations (append, remove, etc)
  - eg, Thread_Queue: `s_graveyardQueue.lock, waitQueue.lock`
- `pidLock`  // k.thread.c
- `kbdQueueLock`  // keyboard.c
- `s_free_space_spin_lock`  // paging.c
- `run_queue_spinlock`  // sched.c
- `mutex->guard`  // synch.c
Wait and Wakeup

- **Wait**(waitq):
  disable intrpt, Spin_Lock(waitq.lock)
  add current thread to waitq
  Schedule_And_Unlock(waitq.lock)
  restore intrpt

- **Wake_Up**(waitq):
  disable intrpt, Spin_Lock(waitq.lock)
  move all threads in waitq to runq
  Spin_Unlock(waitq.lock), restore intrpt

- **Wake_Up_One**(waitq):
  if waitq not empty:
    move waitq.front thread to runq
Mitex

- struct Mutex: {state, guard (spinlock), owner, waitq}  // waitQueue

- Mutex_Lock(x)
  disable intrpt
  Spin_Lock(x.guard)
  if x was locked:
    add current thread to x.waitq
    Schedule_And_Unlock(x.guard)
  else:
    Spin_Unlock(x.guard)
    set x.owner to current thread
    restore intrpt

- Mutex_Unlock(x)
  disable intrpt
  Spin_Lock(x.guard)
  if x.waitq not empty:
    set x.owner to waitq.front
    wakeup x.waitq.front
  else:
    set x.state to unlocked
    Spin_Unlock(x.guard)
    restore intrpt

- Mutex_Unlock_And_Schedule(x)
  Mutex_Unlock(x) w/o last two lines
  Schedule_And_Unlock(x.guard)
  restore intrpt
```
struct Condition: {waitq}  // waitQueue

Cond_Wait(cv, x)
disable intrpt, Spin_Lock(x.guard)
add current thread to cv.waitq
Mutex_Unlock(x) w/o first two and last two lines
Schedule_And_Unlock(x.guard)
restore intrpt
Mutex_Lock(x)

Cond_Signal(cv)
disable intrpt
  wakeup cv.waitq.front
restore intrpt

Cond_Broadcast(cv)
disable intrpt
  wakeup cv.waitq
restore intrpt
```
Outline

Hardware and devices (drivers + interrupt handlers)
Booting and kernel initialization
Kernel threads
User processes
Synchronization
Scheduling
Lowlevel.asm
Virtual filesystem
PFAT
Blockdev
Bufcache
Scheduling

- Flags checked at every potential switch:
  - g_preemptionDisabled[MAX_CPUS]
  - g_needReschedule[MAX_CPUS]

- Schedule():
  - // current thread voluntarily giving up cpu,
  - // eg, Wait(), Mutex_Lock(), Cond_Lock(), Yield().
  - // current thread already in runq or a waitq.
  - set g_preemptionDisabled[this cpu] to false
  - runme ← remove a thread from runq
  - Switch_To_Thread(runme)

- Schedule_And_Unlock(x):  // x is a spinlock
  - like Schedule() but unlocks x before Switch_To_Thread(runme)
Outline

Hardware and devices (drivers + interrupt handlers)
Booting and kernel initialization
Kernel threads
User processes
Synchronization
Scheduling
Lowlevel.asm
Virtual filesystem
PFAT
Blockdev
Bufcache
Handling an interrupt

- **Handle_Interrupt()**:

  // Here on intrpt. save regs, [choose new thread], push regs, iret
  // Using current thread’s kernel stack, containing:
  // user.ss/esp (iff user mode), eflags, cs, eip, errorcode, intrpt#

- push cpu’s gp and seg regs // complete interrupt-state
- call C interrupt handler // with ptr to interrupt-state as arg
- if not g_preemptionDisabled and g_needReschedule:
  move current thread to runq
  update current thread’s state wrt esp, numticks
  get a thread from runq and make it current
- activate user context (if any) // update ldtr, s_TSS, ...
- process signal (if any)
- restore gp and seg regs
- iret
Switching a thread

Switch_To_Thread(thrdptr):

// called from Schedule(). interrupts off.
// using current thread’s kernel stack. stack has return addr.
// current thread struct already in runq or a waitq.
// save current thread context, activate thread passed as param.

- change stack content to an intrpt state by adding:
  - cs, eflags, fake errorcode/intrpt#, gp and seg regs
- make thrdptr (in arg) as current thread
- activate user context (if any) // update ldtr, s_TSS, ...
- process signal (if any)
- clear APIC interrupt info
- restore gp and seg regs
- iret
Outline

Hardware and devices (drivers + interrupt handlers)
Booting and kernel initialization
Kernel threads
User processes
Synchronization
Scheduling
Lowlevel.asm
Virtual filesystem
PFAT
Blockdev
Bufcache
Static variables

- `s_vfsLock`: Mutex, to protect vfs structures
- `s_fileSystemList`: Filesystem struct for every registered fs type
- `s_mountPointList`: Mount_Point struct for every mounted fs

`struct Filesystem`

- `ops`: functions Format and Mount provided by fs type
- `fsname`: name of fs type // eg, “pfat”, “gfs3”

`struct Mount_Point`

- `ops`: mountpoint functions provided by mounted fs
  - eg, Open, Create_Directory, Stat, ...
- `pathpfx`: where fs is mounted // eg, “/”, “/c”
- `dev`: block device containing fs // eg, ide0
- `fsdata`: for use by fs implementation
struct File
- ops: file functions provided by mounted fs
  - eg, FStat, Read, Write, Close, ...
- filepos: current position in the file
- endpos: end position (length of the file)
- fsdata: for use by fs implementation
- mode: mode
- mountpoint: of filesystem that file is part of
VFS functions: Register, Fs.ops wrappers

- **Register_Filesystem(fsname, fsOps):**
  
  ```
  fs ← fill a Filesystem struct
  add fs to s_fileSystemList // protected by Mutex s_vfsLock
  ```

- **Format(devname, fstype):** // Fs.ops
  
  ```
  fs ← s_fileSystemList[fstype]
  Open_Block_Device(devname, dev)
  fs.ops.Format(dev) // return result
  Close_Block_Device(dev)
  ```

- **Mount(devname, pathpfx, fstype):** // Fs.ops
  
  ```
  fs ← s_fileSystemList[fstype]
  Open_Block_Device(devname, *dev)
  mp ← fill a Mount_Point struct
  fs.ops.Mount(mp) // return result
  add mp to mountPointList // protected by Mutex s_vfsLock
  ```
VFS functions: Mp.ops wrappers

- **Open(path, mode, *file):**
  - split path into pathpfx, pathsfx
  - mp ← s_mountPointList[pathpfx]
  - mp.ops.Open(mp, path, mode, file)  
    // return result
  - file.mode, file.mountpoint ← mode, mp

- **Open_Directory(path, *dir):**
  - like Open() but with mp.ops.Open_Directory

- **Create_Directory(path):**
  - split path into pathpfx, pathsfx
  - mp ← s_mountPointList[pathpfx]
  - mp.ops.Create_Directory(mp, pathsfx)  
    // return result

- **Stat(.), Delete(.), ..., Dist_Properties(.):**
  - similar to above

- **Sync():**
  - similar, but do Sync(.) of every mounted fs
VFS functions: F.ops wrappers

- **Close(*file):**
  ```c
  file.ops.Close(file)
  ```
  // return result

- **FStat(*file, *stat):**
  ```c
  file.ops.Fstat(file, stat)
  ```
  // return result

- **Read(*file, *buf, len):**
  ```c
  file.ops.Read(file, buf, len)
  ```
  // return result

- **Write(*file, *buf, len), Seek(*file, pos), Read_Entry(*dir, *entry)**
  similar to above

- **Read_Fully(path, *buf, *len):**
  ```c
  Stat path and allocate buf of stat.size
  Open file; Read^ stat.size; Close file
  ```
Static variable

- **s_pagingDevice**: registered Paging_Device struct

**struct Paging_Device**

- **filename**: name of paging file
- **dev**: block device of paging file
- **startSector**
- **numSectors**

**Register_Paging_Device(pagingdevice)**:

setter for **s_pagingDevice**

**Get_Paging_Device()**:

getter for **s_pagingDevice**
Outline

Hardware and devices (drivers + interrupt handlers)
Booting and kernel initialization
Kernel threads
User processes
Synchronization
Scheduling
Lowlevel.asm
Virtual filesystem
PFAT
Blockdev
Bufcache
struct PFAT_Instance: // kept in vfs.Mount_Point.fsdata
    bootsector fsinfo
    int *fat
    directoryEntry *rootDir
    directoryEntry rootDirEntry
    Mutex lock // protects fileList
    PFAT_File_List fileList

struct PFAT_File: // kept in vfs.File.fsdata
    directoryEntry *entry
    ulong numBlocks
    char *fileDataCache
    Mutex lock // guards concurrent access
PFAT structs – 2

- **struct bootSector:**  // kept in vfs.Mount_Point.fsdata
  - magic
  - fileAllocationOffset/Length  // FAT blocks
  - rootDirectoryOffset/Count  // rootdir blocks
  - setupStart/Size  // secondary loader blocks
  - kernelStart/Size  // kernel image blocks

- **struct directoryEntry:**
  - readOnly, hidden, systemFile, directory, ...  // 1-bit flags
  - time, date
  - firstBlock, fileSize
  - acls
PFAT functions – 1

- PFAT_Mount(mp):
  
  pfi ← Malloc PFAT_Instance
  pfi.fsinfo ← read bootsector from mp.dev block 0
  pfi.fat ← Malloc FATsize // avail in pfi.fsinfo
  pfi.fat ← read mp.dev fat blocks // " " " "
  pfi.rootDir ← Malloc rootdir size // " " " "
  pfi.rootDir ← read mp.dev rootdir blocks // " " " "
  pfi.rootDirEntry ← fake_rootdir_entry

  initialize pfi.lock, pfi.filelist, pfi.filelist.lock

  PFAT_Register_Paging_File(mp, pfi)

  mp.ops ← {PFAT_Open, PFAT_Open_Dir}

  mp.fsdata ← pfi
PFAT_Register_Paging_File(mp, pfi):
quit if a pagefile is already registered or mp pfi has no pagefile
pfe ← dirEntry of PAGEFILE_FILENAME in mp.pfi
pdev ← Malloc Paging_Device
pdev.fileName ← mp.pathpfx / PAGEFILE_FILENAME
pdev.dev ← mp.dev
pdev.startSector ← pfe.firstBlock
pdev.numSectors ← pfe.fileSize/SECTOR_SIZE
Register_Paging_Device(pdev)
PFAT functions – 3

- **Get_PFAT_File**(pfi, direntry):
  
  if pfi.filelist has a PFAT_File obj for direntry: return it
  else add a new obj for direntry to pfi.filelist, return it

- **PFAT_Open**(mp, path, mode, *file):

  pfi ← mp.fsdata
  quit if mode attempts to create file or if path not in pfi
  pfatfile ← Get_PFAT_File(pfi, direntry of path)
  *file ← vfs.File for pfatfile with ops
    - PFAT_FStat/Read/Write/Seek/Close

- **PFAT_Open_Directory**(mp, path, mode, *dir):

  below assumes path is "/"
  pfi ← mp.fsdata
  *dir ← vfs.File obj for pfi.rootDir with ops
    - PFAT_FStat_Dir/Close_Dir/Read_Entry
PFAT functions – 4

- **PFAT_Read(file, buf, nbytes):**
  
  ```c
  pfatfile ← file.fsdata
  pfi ← file.mp.fsdata
  Mutex_Lock(pfatfile.lock)
  nbytes ← min(endpos, filepos + nbytes)
  traverse FAT (in file.mp.fsdata) for the blocks of the file:
     for each block not in cache, read it into cache, then to buf
     update filepos
  Mutex_Unlock(pfatfile.lock)
  return nbytes
  ```

- **PFAT_Write(file, buf, nbytes):**
  like PFAT_Read but only in sector-units and within file

- **Init_PFAT():**
  
  ```c
  Register_Filesystem("pfat", PFAT_Mount)
  ```
Hardware and devices (drivers + interrupt handlers)
Booting and kernel initialization
Kernel threads
User processes
Synchronization
Scheduling
Lowlevel.asm
Virtual filesystem
PFAT
Blockdev
Bufcache
- **struct Block_Request:**
  - dev
  - type (// BLOCK_READ, BLOCK_WRITE)
  - blocknum
  - state (// PENDING, COMPLETED, ERROR)
  - errorcode
  - satisfied (// Condition (with s_blockdevRequestLock))

- **struct Block_Device:**
  - name
  - ops (// Open(dev), Close(dev), Get_Num_Blocks(dev))
  - unit
  - inUse (// closed or open)
  - waitqueue (// for requesting thread)
  - reqqueue (// for requests to this device)
- Mutex \texttt{s\_blockdevLock}: protects block device list
- Mutex \texttt{s\_blockdevRequestLock}: for all requests
- Condition \texttt{s\_blockdevRequestCond}
- \texttt{s\_deviceList}: list of all registered block devices
- **Register Block Device**(name, ops, unit, driverdata, waitq, reqq):
  
  ```
  dev ← [name, ops, unit, ..., reqq, inUse = false]
  Mutex_Lock(s_blockdevLock)
  add dev to s_deviceList
  Mutex_Unlock(s_blockdevLock)
  ```

- **Open Block Device**(name, *dev):
  
  ```
  Mutex_Lock(s_blockdevLock)
  find dev in s_deviceList
  dev.ops.Open(dev)
  Mutex_Unlock(s_blockdevLock)
  ```

- **Close Block Device**(name, *dev):
  
  ```
  like Open Block Device but using dev.ops.Close(dev)
  ```
Blockdev functions – 2

**Block_Read**(dev, blocknum, buf):

- Mutex_LOCK(s_blockdevLock)
- req ← Block_Request(dev, BLOCK_READ, blocknum, buf, PENDING, Cond_Init(satisfied))
- Mutex_LOCK(s_blockdevRequestLock) // post req
- add req to dev.requestQueue
- Cond_Broadcast(s_blockdevRequestCond) // awaken server
- while req.state is PENDING: // wait for req to be served
  - Cond_WAIT(req.satisfied, s_blockdevRequestLock)
- Mutex_LOCK(s_blockdevRequestLock)
- Mutex_UNLOCK(s_blockdevLock)
- // and return req.errorcode

**Block_Write**(dev, blocknum, buf):

- like Block_Read
• Dequeue_Request(reqqueue):  // executed by device driver thread
  Mutex_Lock(s_blockdevRequestRequestLock)
  while reqqueue is empty:
    Cond_Wait(s_blockdevRequestRequestCond, s_blockdevRequestRequestLock)
  get req from reqqueue
  Mutex_Lock(s_blockdevRequestRequestLock)
  return req

• // executed by device driver thread or interrupt handler
  Notify_Request_Completion(req, state, errorcode):
    req.state ← state
    req.errorcode ← errorcode
    Cond_Signal(req.satisfied)
Outline

- Hardware and devices (drivers + interrupt handlers)
- Booting and kernel initialization
- Kernel threads
- User processes
- Synchronization
- Scheduling
- Lowlevel.asm
- Virtual filesystem
- PFAT
- Blockdev
- Bufcache
Bufcache structs

- struct FS_Buffer:
  - fsblocknum
  - data
  - flags

- struct FS_Buffer_Cache:
  - dev
  - fsblocksize
  - numCached
  - bufferList
  - mutex
  - cond
Bufcache functions – 1

- Create_FS_Buffer_Cache(dev, fsblocksize):
  
  cache ← Malloc(dev, fsblocksize, numCached = 0, 
  Clear(bufferList), Init(mutex), Init(cond))

- Sync_FS_Buffer_Cache(cache):
  
  Mutex_Lock(cache.mutex)
  for every buf in cache.bufferList:
    if buf is dirty, write buf.data to disk and set buf clean
  Mutex_Unlock(cache.mutex)

- Destroy_FS_Buffer_Cache(buf):
  
  Mutex_Lock(cache.mutex)
  for every buf in cache.bufferList: sync and free mem
clear cache.bufferList
  Mutex_Unlock(cache.mutex)
  free cache
*Get_FS_Buffer*(cache, fsblocknum, *buf*):

- MutexLock(cache.mutex)
- if there is a buffer with fsblocknum in cache.bufferList:
  - buf ← buffer, await(cache.cond) not inuse, set inuse, return 0
- if cache.numCached at maxlimit, all buffers inuse: return ENOMEM
- if cache.numCached < maxlimit:
  - allocate memory for buf and buf.data // never fails?
  - add buf to cache.bufferList front
- else:
  - buf ← lru not-inuse buffer in cache.bufferList
  - sync buf, move buf to bufferList front
- set buf's fields, read disk blocks into buf.data
- MutexUnlock(cache.mutex)
- return 0