GeekOS 2019
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
Outline

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 n SMP (symmetric multi-processing) configuration
- apics (interrupt controllers)
  - local apic: recv intrpts from io-apc, send/recv to other cpus
  - io-apic: route interpts from io-dev ces/t mer 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 **seg descriptor table**
  - which table, index into table, protection level]
  - **global descriptor table** (gdt), **oca 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, page, 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 (RET) 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)
- **ep** (32-bit): nstruct on pointer (n 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.
Local APICs and IO APIC

- 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 ocat on APIC_Addr + APIC_ID
  - Restoreinterrupts
  - 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 timers used on boot to calculate the LAPIC timers

- Global and static variables
  - g_numTicks // global tick counter
  - DEFAULT_MAX_TICKS = 4 // default quantum
  - g_Quantum = DEFAULT_MAX_TICKS
LAPIC timer

- **Timer_Interrupt_Handler(istate):**  // SMP fed
  
  id ← Get_CPU_ID()
  ct ← get_current_thread()

  for s 0:
    ++g_numTicks
    ++ct.numTicks
  
  if ct.numTicks >= g_Quantum:
    g_needReschedule[id]

- **Init_Timer():**
  Install_IRQ(32, Timer_Interrupt_Handler)
  enable einterrupt 32

- **Init_Local_APIC(cpuid):**
  Install_IRQ(39, Spurious_Interrupt_Handler)  // SMP
  enable einterrupt 39
  
  set timer timeout value  // cpu 0 uses P T to calibrate
VGA screen

- Ports: CRT_* regs (0x3D4, 0x3D5, etc)
  - access v a o nstr // eg, Out_Byte(port, va ue)
  - for refresh, scan rate, b ank ng, cursor contro , etc

- V deo memory: VIDMEM (0xb8000-0x100000)
  - ho ds characters to d sp ay // NUMROWS = 25, NUMCOLS = 80
  - access v a read/wr te nstrs // eg, VIDMEM[offset] = keycode

- Var conso e state: row, co , esc, numer c arg, etc

- Update_Cursor() based on conso e state // ports used here on y

- Put_Char_Imp(c): p ace char c at text cursor pos t on

- Init_Screen(): c ear screen, set "text cursor" to or g n

- Print(*fmt, ...)

- Print(*fmt, ...)

-のが a o nstr

-の a o nstr

-のが a o nstr

-のが a o nstr

-のが a o nstr

-のが a o nstr

-のが a o nstr

-のが a o nstr

-のが a o nstr

-のが a o nstr

-のが a o nstr

-のが a o nstr

-のが a o nstr

-のが a o nstr

-のが a o nstr
Keyboard

- **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)

- **Stat c var ab es (for dr vers, interrupt hand er)**
  - `s_queue`  // queue for ncom ng keycodes
  - `s_keyboardWaitQueue`  // threads wa t ng for
  - `s_kbdQueueLock`  // kbdi n p u t s
  - `scantables`  // sp nock protect ng `s_queue`
  - `kbd state`  // map scancode to keycode
    // sh ft, esc, contro, a t, etc
Keyboard

- Keyboard_Interrupt_Handler(istate):
  - f ports nd cate byte ava abe:
    - get byte; convert to keycode or update kbdstate
  - add keycode to s queue // drop f fu ; sp n ock ops
  - wakeup(s_keyboardWaitQueue)

- Init_Keyboard():
  - n t a ze stat c var ab es
  - Install_IRQ(KB_IRQ, Keyboard_Interrupt_Handler)
  - enab e kbd i nterrupt

- Wait_For_Key():
  - d sab e ntrpt
  - repeat
    - f s_queue has key, get t // sp n ock ops
  - e se wait(s_keyboardWaitQueue)
  - unt got key
  - restore ntrpt
- 16-bit transfer un t
- 2 hard d sks
- P O and DMA modes
- 256-byte b ocks

Ports
- IDE_dent fy regs // show d sk features
- IDE_dr ve/cy nd er/head/sector regs // target d sk b ock
- IDE_command reg // read/wr te
- IDE_data reg // success ve words of o b ock show up here
- IDE_status/contro /etc regs // busy, dma, interrupt, etc
IDE: drivers

- Statistics variables
  - s_ideWaitQueue: server thread waits here
  - s_ideRequestQueue: 0 requests queued here

- IDE_Read(drve, 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, po

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

- Registers
  - memory addr
  - byte count
  - contro regs (source, dest nat on, transfer un t, etc)

- Usage for de o
  - cpu sets up de nterface to n t ate data transfer
  - cpu sets up dma nterface

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

Hardware and devices (drivers + interrupts + interrupt handlers)
Booting and kernel initialization
Kernel threads
User processes
Synchronization
Scheduling
Low level.asm
Virtual filesystem
PFAT
Blockdev
Bufcache
At power up, B OS configures
- one cpu- ap cs as primary, with d 0
- other cpu- ap cs as secondaries, hanged, with ds 1, 2, ...
- MP config table n memory
- oads d skc/b ock 0 (bootsect.asm) nto memory
- cpu 0 ( n rea mode) starts execut ng t

bootsect.asm // executed by cpu 0
- oad the kerne mage (from d skc) nto memory and start execut ng t (setup.asm)

setup.asm // executed by cpu 0
- get memory size, red rectinterrupts (bypass B OS)
- 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

- boot VGA screen
- nt 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
- nt kernel heap
- nt cpu 0's tss, tr, gdt[3?] // s_theTSS[0]; one tss per cpu
- nt cpu 0's dt, dtr // s_IDT[0]
  - syscall entry's dp at user eve, others at kernel eve
  - addresses of interrupt handlers n g_interruptTable[0];
    set them to dummy interrupt handler
- nt 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 (n setup.asm)
    // cpu i does nt a zat on, then sp ns unt cpu 0 re eases t
Kernel initialization: Main() - 2 // executed by cpu0

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

- nt traps: 12: stack except on; 13: GPF; 0x90: sysca

- nt dev ces: Loca_AP C(0), keyboard, DE, DMA

- nt PFAT: Register f esystem PFAT with vfs

- re ease SMP
  - a ow each secondary cpu to ex t ts n t a zat on; wa t for that

- mount root f esystem
  - mount ide0 as PFAT fs at path "/a"

- spawn n t a process // she program

- hardware shutdown
Secondary cpu initialization  // executed by cpu \(i\)

- **start_secondary_cpu** (n 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() (n smp.c)

- **Secondary_Start()** (n smp.c)
  - n t gdt: point cpu \(i\)'s gdtr to s_GDT[0]  // uses cpu 0's gdt
  - n t cpu \(i\)'s tss, tr, gdt 3+i ?]  // s_theTSS[i]
  - n t cpu \(i\)'s dt (s_IDT[i]), dtr
  - n t scheduler(): create threads // with Kernel_Thread objects
    - current thread {Main}  // g_currentThreads[i]
    - d e thread {Idle-\#i}  // s_runQueue
  - n t traps, oca ap c
  - set flag nform ng cpu 0 that \(i\) s done
  - Exit(0), which makes cpu enter scheduler
Outline

Hardware and devices (drivers + interrupt handlers)

Bootng and kernel initialization

Kernel threads

User processes

Synchronization

Scheduling

Low level .asm

Virtual filesystem

PFAT

Blockdev

Buffers
Kernel threads: context and queues

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

- **struct Kerne _Thread:**
  - esp, *stackPage, *userContext
  - nk for s_a ThreadL st // constant
  - nk for current thread queue // runq, wtq, graveyard
    - numTcks, totalTme, prorTy, pd, jomq, exitcode, owner, ...

- **Thread queues**
  - s_a ThreadL st // a threads
  - s_runQueue // ready (aka runnabe) threads
  - s_graveyardQueue // ended and to be reaped
  - various watQueues // mutex, cond t on, dev ces, etc
  - *g_currentThreads MAX_CPUS] // runn ng thread
Starting kernel threads

- **Start_Kerne_Thread**(startfunc, arg, pr or ty, detached, name):
  - **Create_Thread**: get memory for kthread context (struct and stack page)
    - nt struct: stackPage, esp, numT cks, p d
    - add to the a-thread-st
  - **Setup_Kerne_Thread**: configure stack so that upon swtching n t 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 (e p)
    - // fake error code, ntrpt#, fake gp regs
    - // KERNEL_DS (ds), KERNEL_DS (es), 0 (fs), 0 (gs)
  - **Make thread runnab e**: add struct to runq
CURRENT_THREAD:       // return the thread struct of the current samedead interrupts
  ct ← g_currentThreads[GET_CPU_ID]
  restoreinterrupts
Outline

Hardware and devices (drivers + interrupt handlers)

Bootng and kernel tacula

Kerne threads

User processes

Synchronization

Scheduling

Low eve .asm

Virtual filesystem

PFAT

Blockdev

Bufcache
User process context

- Context of a user process:
  - Kerne_Thread struct + stack page + struct User_Context

- struct User_Context:
  - name ]
  - dt 2] // code segment, data segment
  - * dtDescr ptor // segment descriptor
  - *memory, size // memory space for process
  - dtSe ector //index nto gdt
  - csSe ector, dsSe ector //index nto dt
  - entryAddr, argB ockAddr, stackpointerAddr
  - *pageD r, *f e_desctr ptor_table ]
  - refCount, mappedReg ons, 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 ← maoc(*program maxva + argb ock size + stack page*)
  - copy program segments nto mem space
  - maoc usercontext and set ts fds:
    - *memory ← mem
    - dt, dt sectors/descr ptors
    - entry point, argb ock, stack bottom, ...
  - *kthread ← Start_User_Thread(userContext)
Start user thread

- **Start_User_Thread(uc, detached)**: // "uc" s "usercontext"
  - Create_Thread:
    ma oc kthread struct and stack, n t, add to a -thread- st
  - Setup_User_Thread:
    po nt kthread.usercontext to uc
    configure kerne stack as f t wasinterrupted n user mode
    // stack (bottom to top):
    //    uc.ds (user ss), uc.stackaddr (user esp)
    //    eflags ( ntrpt on), uc.cs (cs), uc.entryaddr (e p)
    //    errorcode, ntrpt#, gp regs except es    // fake
    //    uc.argb ockaddr (es ), uc.ds (ds, es, fs, gs)
    // How s term nat on hand ed?
  - Make thread runnab e: add struct to runq
Copying between user and kernel spaces

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

- **Copy_From_User** (dst nKerne, src nUser, bufsize):
  ucontext ← CURRENT_THREADS.usercontext
  src nKerne ← User_To_Kerne (ucontext, src nUser)
  memcpy(dst nKerne, src nKerne, bufsize)

- **Copy_To_User** (dst nUser, src nKerne, bufsize):
  ucontext ← CURRENT_THREADS.usercontext
  dst nKerne ← User_To_Kerne (ucontext, dst nUser)
  memcpy(dst nKerne, src nKerne, bufsize)
Outline

Hardware and devices (drivers + interrupts + handlers)

Bootng and kernel initialization

Kernel threads

User processes

Synchronization

Scheduling

Low level .asm

Virtual filesystem

PFAT

Blockdev

Bufcache
Interrupt disable/enable: affects only this CPU

Disable_Interrupts(): // abbrv: d sab e ntrpt
    __asm__"cli"

Enable_Interrupts(): // abbrv: enab e ntrpt
    __asm__"sti"

Begin_Int_Atomic(): // abbrv: d sab e ntrpt
    ion ← true iff interrupts enabled
    if ion:
        Disable_Interrupts()
    return ion

End_Int_Atomic(ion): // abbrv: restore ntrpt
    if ion:
        Enable_Interrupts()
Spinlocks

- **Spin lock assembly**: an interrupt that is off when unlocked

  ```c
  Spin_Lock_INTERNAL(x):
  repeat
    busy_wait until *x == 0
    set eax to 1
    atomic swap eax and *x
  until eax equals 0
  
  Spin_Unlock_INTERNAL(x):
  set eax to 0
  atomic swap eax and *x
  
  Spin_Lock(x): wrapper of assembly fn + update to ocker, ra, ...
  Spin_Unlock(x): I I I I I I I I //Why?

- **Spin lock C**: struct { ock, ocker, ra, astocker}

- **Ensure interrupts disabled before acquiring a spin lock**
  - Restore interrupts after releasing a spin lock
Some spinlock variables

- g oka Lock  // ockKerne (), un ockKerne (); smp.c
- kthreadLock  // kthread.c, user.c
- Every list_t n DEFINE_LIST(list_t, node_t) has a sp n ock ock
  - Guards the st n st operat ons (append, remove, etc)
  - eg, Thread_Queue: s_graveyardQueue. ock, wa tQueue. ock
- p dLock  //k.thread.c
- kbdQueueLock  //keyboard.c
- s_free_space_sp n ock  // pag ng.c
- run_queue_sp n ock  // sched.c
- mutex->guard  //synch.c
**Wait and Wakeup**

- **Wait\(wa\ tq\):**
  
  ```
  d s a b e n t r p t, S p n_L o c k(wa t q. o c k)
  a d d c u r r e n t t h r e a d t o w a t q
  S c h e d u l e_ A n d_U n o c k(wa t q. o c k)
  r e s t o r e n t r p t
  ```

- **Wake\_Up(wa\ tq):**
  
  ```
  d s a b e n t r p t, S p n_L o c k(wa t q. o c k)
  m o v e a t h r e a d s n w a t q t o r u n q
  S c h e d u l e_U n o c k(wa t q. o c k), r e s t o r e n t r p t
  ```

- **Wake\_Up\_One(wa\ tq):**
  
  ```
  f w a t q n o t e m p t:
  m o v e w a t q.f r o n t t h r e a d t o r u n q
  ```
Mutex

- struct Mutex: {state, guard (sp n ock), owner, wa tq} // watQueue

- Mutex_Lock(x)
  d sab e ntrpt
  Sp n_Lock(x.guard)
  f x.state s ocked:
    add current thread to x.wa tq
  Schedu e_And_Un ock(x.guard)
  e se:
    set x.state to ocked
    Sp n_Un ock(x.guard)
    set x.owner to current thread
  restore ntrpt

- Mutex_Un ock(x)
  d sab e ntrpt
  Sp n_Lock(x.guard)
  f x.wa tq not empty:
    set x.owner to wa tq.front
    wakeup x.watq.front
  e se:
    set x.state to un ocked
    Sp n_Un ock(x.guard)
    restore ntrpt

- Mutex_Un ock_And_Schedu e(x)
  Mutex_Un ock(x) w/o ast two nes
  Schedu e_And_Un ock(x.guard)
  restore ntrpt
Condition

- struct Cond on: \{wa tq\}  // wa tQueue

- Cond_Wa t(cv, x)
  
  d sab e ntrpt, Sp n_Lock(x.guard)
  add current thread to cv.wa tq
  Mutex_Unock(x) w/o frst two and ast two nes
  Schedu e_And_Unock(x.guard)
  restore ntrpt
  Mutex_Lock(x)

- Cond_S gna (cv)
  
  d sab e ntrpt
  wakeup cv.wa tq.front
  restore ntrpt

- Cond_Broadcast(cv)
  
  d sab e ntrpt
  wakeup cv.wa tq
  restore ntrpt
Outline

Hardware and devices (drivers + interrupt handlers)

Bootng and kernel tation

Kernel threads

User processes

Synchronization

Scheduling

Low level .asm

Virtual filesystem

PFAT

Blockdev

Bufcache
Scheduling

- Flags checked at every potential switch:
  - g_preempt_onD_sab_ed MAX_CPUS]
  - g_needReschedule e MAX_CPUS]

- Schedule():
  // current thread waiting up cpu,
  // eg, Wait(), Mutex_Lock(), Cond_Lock(), Yield().
  // current thread a ready on runq or a waitq.
  - set g_preempt_onD_sab_ed this cpu] to false
  - runme ← remove a thread from runq
  - Switch_To_Thread(runme)

- Schedule_And_Unlock(x):
  // x a s a p n lock
  - keep Schedule() but unlocks x before Switch_To_Thread(runme)
Outline

Hardware and devices (drivers + interrupt handlers)

Bootng and kernelization

Kernel threads

User processes

Synchronization

Scheduling

Low level .asm

Virtual filesystem

PFAT

Blockdev

Bufcache
Handling an interrupt

- **Hand e _interrupt():**
  - // Here on ntrpt. save regs, choose newthread, push regs, ret
  - // Using current thread's kernel stack, containing:
  - // user.ss/esp (ff user mode), eflags, cs, eip, errorcode, ntrpt#
  - push cpu's gp and seg regs  // complete interrupt-state
  - ca Cinterrupt handler
  - f not g_preempt onD sab ed and g_preed Reschedu
  - move current thread to runq
  - update current thread's state wrt esp, numt cks
    get a thread from runq and make t current
  - act vate user context (f any)  // update dtr, s_TSS, ...
  - process s gna (f any)
  - restore gp and seg regs
  - ret
Switching a thread

- **Switch** To Thread(thrdptr):

  // ca ed from Schedu e().interrupts off.
  // us ng current thread's kerne stack. stack has return addr.
  // current thread struct a ready n runq or a wa tq.
  // save current thread context, act vate thread passed as param.

- change stack content to an ntrpt state by add ng:
  cs, eflags, fake errorcode/ ntrpt#, gp and seg regs
- make thrdptr ( n arg) as current thread
- act vate user context ( f any) // update dtr, s_TSS, ...
- process s gna ( f any)
- c ear AP Cinterrupt nfo
- restore gp and seg regs
- ret
Outline

Hardware and devices (drivers + interrupt handlers)

Boot and kernel initialization

Kernel threads

User processes

Synchronization

Scheduling

Low level .asm

Virtual filesystem

PFAT

Blockdev

Bucache
VFS: static vars and structs - 1

- Stat c var ab es
  - s_vfsLock: Mutex, to protect vfs structures
  - s_f eSystemL st: F esystem struct for every Registered fs type
  - s_mountPo ntL st: Mount_Po nt struct for every mounted fs

- struct F esystem
  - ops: functions Format and Mount prov ded by fstype
  - fsname: name of fs type // eg, "pfat", "gfs3"

- struct Mount_Po nt
  - ops: mountpo nt funct ons prov ded by mounted fs
    - eg, Open, Create_D rectory, Stat, ...
  - pathpfx: where fs s mounted // eg, "/", "/c"
  - dev: b ock dev ce conta n ng fs // eg, ide0
  - fsdata: for use by fs mp ementat on
VFS: static vars and structs - 2

- **struct Fe**
  - **ops**: file functions provided by mounted fs
    - eg, FStat, Read, Write, Close, ...
  - **fepos**: current position on the file
  - **endpos**: end position (length of the file)
  - **fsdata**: for use by fs management
  - **mode**: mode
  - **mountpoint**: of filesystem that file is part of
VFS functions: Register, Fs.ops wrappers

- **Register_Filesystem(fsname, fsOps):**
  
  ```
  fs ← f a F esystem struct
  add fs to s_f eSystemL st // protected by Mutex s_vfsLock
  ```

- **Format(devname, fstype):** //Fs.ops
  
  ```
  fs ← s_f eSystemL st fstype]
  Open_B ock Dev ce(devname, dev)
  fs.ops.Format(dev) // return result
  C ose_B ock Dev ce(dev)
  ```

- **Mount(devname, pathpfx, fstype):** //Fs.ops
  
  ```
  fs ← s_f eSystemL st fstype]
  Open_B ock Dev ce(devname, *dev)
  mp ← f a Mount_Po nt struct
  fs.ops.Mount(mp) // return result
  add mp to mountPo ntL st // protected by Mutex s_vfsLock
  ```
VFS functions: Mp.ops wrappers

- **Open(path, mode, *file)**:
  
  ```
  sp t path nto pathpfx, pathsfx mp
  ← s_mountPo ntL st pathpfx]
  mp.ops.Open(mp, path, mode, f e)  // return result
  f e.mode, f e.mountpo nt ← mode, mp
  ```

- **Open_Directory(path, *dir)**:
  
  ```
  ke Open() but with mp.ops.Open_D rectory
  ```

- **Create_Directory(path)**:
  
  ```
  sp t path nto pathpfx, pathsfx mp
  ← s_mountPo ntL st pathpfx]
  mp.ops.Create_D rectory(mp, pathsfx)  // return result
  ```

- **Stat(.), Delete(.), ..., Dist_Properties(.)**
  
  s m ar to above

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

- **Close(***file*)**:  
  
  f e.ops.C ose(f e)  
  // return result

- **FStat(***file, **stat*)**:  
  
  f e.ops.Fstat(f e, stat)  
  // return result

- **Read(***file, **buf, **len*)**:  
  
  f e.ops.Read(f e, buf, en)  
  // return result

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

- **Read_Fully**(path, **buf, **len*):  
  
  Stat path and a ocate buf of stat.size  
  Open f e; Read* stat.size; C ose f e
VFS: paging device

- Stat c var ab e
  - s_pagingDev ce: Registered Paging Device struct

- struct Paging Device
  - f ename: name of paging device
  - dev: block device of paging device
  - startSector
  - numSectors

- Register_Paging_Device(pagingDevice):
  setter for s_pagingDev ce

- Get_Paging_Device():
  getter for s_pagingDev ce
Outline

Hardware and devices (drivers + interrupts + handlers)

Bootng and kernel initialization

Kernel threads

User processes

Synchronization

Scheduling

Low level.asm

Virtual filesystem

PFAT

Blockdev

Bucache
PFAT structs - 1

```
struct PFAT_Instance: // kept n vfs.Mount_Po nt.fsdata
  bootsector fsInfo
  nt *fat
  directoryEntry *rootDr
  directoryEntry rootDrEntry
  Mutex ock // protects f eL st
  PFAT_Fe_L st f eL st

struct PFAT_F e: // kept n vfs.F e.fsdata
  directoryEntry *entry
  ulong numBocks
  char *f eDataCache
  Mutex ock // guards concurrent access
```
PFAT structs - 2

- struct bootSector:  // kept n vfs.Mount_Po nt.fsdata
  - mag c
  - feA ocat onOffset/Length  // FAT b ocks
  - rootD rectoryOffset/Count  // rootd r b ocks
  - setupStart/size  // secondary oader b ocks
  - kerne Start/size  // kerne mage b ocks

- struct d rectoryEntry:
  - readOn y, h dden, systemF e, d rectory, ...  // 1-bit flags
  - t me, date
  - frstB ock, f esize
  - ac s
PFAT functions - 1

- **PFAT_Mount(mp):**

  ```
  pf ← Ma oc PFAT_Instance
  pf.fs nfo ← read bootsector from mp.dev block0
  pf.fat ← Ma oc FATsize // ava n pf.fs nfo
  pf.fat ← read mp.dev fat blocks // I I I I
  pf.rootD r ← Ma oc rootd r size // I I I I
  pf.rootD r ← read mp.dev rootdir blocks // I I I I
  pf.rootD rEntry ← fake_rootd r_entry
  n t a ze pf. ock, pf.f e st, pf.f e st. ock
  PFAT_Register_Pag ng_F e(mp, pf)
  mp.ops ← {PFAT_Open, PFAT_Open_Dir}
  mp.fsdata ← pf
  ```
PFAT functions - 2

- PFAT_Register_Paging_Fe(mp, pf):

  quota pagefe s a ready Registered or mp pf has no pagefe e
  pfe ← d rEntry of PAGEFILE_FILENAME  n mp pf
  pdev ← Ma oc Pag ng_Dev ce
         // vfs
  pdev.f eName ← mp.pathpfx / PAGEFILE_FILENAME
  pdev.dev ← mp.dev
  pdev.startSector ← pfe.frstB ock
  pdev.numSectors ← pfe.f esize/SECTOR_SIZE
  Register_Pag ng_Dev ce(pdev)       // vfs
PFAT functions - 3

- **Get_PFAT_Fe** (pf, d rentry):
  
  f pf.f e st has a PFAT_Fe obj for d rentry: return t e se add a new obj for d rentry to pf.f e st, return t

- **PFAT_Open** (mp, path, mode, *f e):
  
  pf ← mp.fsdata
  qu t f mode attempts to create f e or f path not n pf
  pfatf e ← Get_PFAT_Fe (pf, d rentry of path)
  *f e ← vfs.F e for pfatf e with ops
    PFAT_FStat/Read/Write/Seek/Close

- **PFAT_Open_D rectory** (mp, path, mode, *d r):
  
  be ow assumes pathis I/I
  pf ← mp.fsdata
  *d r ← vfs.F e obj for pf.rootD r with ops
    PFAT_FStat_Dir/Close_Dir/Read_Entry
PFAT functions - 4

- **PFAT_Read**(f e, buf, nbytes):
  
  \[ \begin{align*}
  \text{pfatf} e & \leftarrow f e.\text{fsdata} \\
  \text{pf} & \leftarrow f e.\text{mp.fsdata} \\
  \text{Mutex._Lock}(\text{pfatf} e.\text{ock}) \\
  \text{nbytes} & \leftarrow m n(\text{endpos, f epos + nbytes}) \\
  \text{traverse \ FAT( n f e.mp.fsdata) for the b ocks of the f e:} \\
  & \quad \text{for each b ock not n cache, read t nto cache, then to buf} \\
  & \quad \text{update f epos} \\
  \text{Mutex._Unock}(\text{pfatf} e.\text{ock}) \\
  \text{return nbytes}
  \end{align*} \]

- **PFAT_Write**(f e, buf, nbytes):
  
  \[ \text{ke PFAT_Read but on y n sector-un ts and with n f e} \]

- **n t_PFAT()**:
  
  \[ \text{Register_F esystem(IpfatI, PFAT_Mount)} \]
Outline

Hardware and devices (drivers + interrupts + hardware)

Bootng and kernel initialization

Kernel threads

User processes

Synchronization

Scheduling

Low level .asm

Virtual filesystem

PFAT

Blockdev

Bufcache
Blockdev structs - 1

- struct **Block_Request**:
  - dev
  - type
    - **// BLOCK_READ, BLOCK_WRITE**
  - blocknum
  - state
    - **// PENDING, COMPLETED, ERROR**
  - errorcode
  - satisfied
    - **// Cond t on (withis_blockdevRequestLock)**

- struct **Block_Device**:
  - name
  - ops
    - **// Open(dev), Close(dev), Get_Num_Blocks(dev)**
  - u nt
  - nUse
    - **// c osed or open**
  - watqueue
    - **// for request ng thread**
  - reqqueue
    - **// for requests to this device**
Blockdev static vars

- Mutex `s_blockdevLock`: protects `blockdev ce st`
- Mutex `s_blockdevRequestLock`: for a requests
- Cond t on `s_blockdevRequestCond`
- `s_dev ceL st`: st of a Registered `blockdev ces`
Blockdev functions - 1

- **Register Block Dev ce** (name, ops, unt, dr verdata, wa tq, reqq):
  
  ```
  dev ← name, ops, unt, ..., reqq, nUse = false

  Mutex_Lock(s_blockdevLock)
  add dev to s_dev ceL st
  Mutex_Un_lock(s_blockdevLock)
  ```

- **Open Block Dev ce** (name, *dev):
  
  ```
  Mutex_Lock(s_blockdevLock)
  fnd dev n s_dev ceL st
  dev.ops.Open(dev)
  Mutex_Un_lock(s_blockdevLock)
  ```

- **Close Block Dev ce** (name, *dev):
  
  ```
  ke Open_B ock_Dev ce but us ng 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_n(t(sat sfed)))
  Mutex_Lock(s_blockdevRequestLock) // post req
  add req to dev.requestQueue
  Cond_Broadcast(s_blockdevRequestCond) // awaken server
  while req.state = PENDING: // wait for req to be served
    Cond_Wait(req.sat sfed, s_blockdevRequestLock)
  Mutex_Lock(s_blockdevRequestLock)
  Mutex_Unlock(s_blockdevLock)
  // and return req.errorcode
  ```

- **Block_Write**(dev, blocknum, buf):
  ke Block_Read
Blockdev functions - 3

- **Dequeuing Request(reqqueue):**  // executed by device driver thread
  
  Mutex_Lock(s_blockdevRequestLock)

  while reqqueue is empty:
    Cond_Wait(s_blockdevRequestCond, s_blockdevRequestLock)

  get req from reqqueue

  Mutex_Lock(s_blockdevRequestLock)

  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 + interrupts +\)
Bootng and kernel\nta\nKerne threads
User processes
Synchronization
Scheduling
Low level .asm
Virtual filesystem
PFAT
Blockdev
Bufcache
Bufcache structs

- struct `FS_Buffer`:
  - `fsb ocknum` // buffer for one fs block
  - `data` // of the fs block data (fnuse)
  - `flags` // 4K page allocated separately
  - `locknum` // dirty, nuse

- struct `FS_Buffer_Cache`:
  - `dev` // block device
  - `fsb ocksize` // size of fs block
  - `numCached` // current number of buffers
  - `bufferL st` // Cond t on: wa t ng for a buffer
  - `mutex`
Bufcache functions - 1

- Create_FS_Buffer_Cache(dev, fsb ocksize):
  
  ```
  cache ← Mac(dev, fsb ocksize, numCached = 0, 
  Cear(bufferL st), n t(mutex), n t(cond))
  ```

- Sync_FS_Buffer_Cache(cache):
  
  ```
  Mutex_Lock(cache.mutex)
  for every buf n cache.bufferL st:
    f buf s d rty, wr te buf.data to d sk and set buf c ean
  Mutex_Un lock(cache.mutex)
  ```

- Destroy_FS_Buffer_Cache(buf):
  
  ```
  Mutex_Lock(cache.mutex)
  for every buf n cache.bufferL st: sync and free mem c
  ear cache.bufferL st
  Mutex_Un lock(cache.mutex)
  free cache
  ```
Bufcache functions - 2

- Get_FS_Buffer(cache, fsb ocknum, *buf):

  Mutex_Lock(cache.mutex)
  if there s a buffer with fsb ocknum n cache.bufferL st:
    buf ← buffer, awa t(cache.cond) not nuse, set nuse, return 0
  f cache.numCached at max mt, a buffers nuse: return ENOMEM
  f cache.numCached < max mt:
    a ocate memory for buf and buf.data       // never fa s?
    add buf to cache.bufferL st front
  e se:
    buf ← ru not- nuse buffer n cache.bufferL st
    sync buf, move buf to bufferL st front
  set buf s fe ds, read d sk b ocks nto buf.data
  Mutex_Un ock(cache.mutex)
  return 0