GeekOS Master Guide

CMSC412: Operating Systems - Spring 2025

Complete Documentation and Reference

Table of Contents

---

1. Introduction and Overview

What is GeekOS?

GeekOS is a teaching operating system for CMSC 412 at the University of Maryland. It's a small, functional OS kernel that demonstrates core OS concepts through a realistic x86 implementation.

Key Statistics:

- ~18,000 lines of C and assembly code
- 44 kernel source files
- Runs on x86 (32-bit protected mode)
- Emulated via QEMU
- Full SMP (multi-core) support

Core Features:

- Multi-core symmetric multiprocessing (SMP)
- Kernel threads and user processes
- Virtual memory with paging
- Multiple filesystem implementations
- Comprehensive device driver support
- Rich synchronization primitives

- Full system call interface

Directory Structure

```
geekos/
├── build/                    # Makefiles, build outputs, disk images
│   ├── diskc                 # IDE0: PFAT with kernel + user programs
│   └── diskd                 # IDE1: Empty disk for projects
├── src/
│   ├── geekos/               # Kernel code (44 .c files, 3 .asm files)
│   ├── user/                 # User programs
│   ├── libc/                 # System call wrappers
│   ├── common/               # Utilities (bget, string, fmtout)
│   └── tools/                # Build tools (buildFat)
├── include/
│   ├── geekos/               # Kernel headers
│   └── libc/                 # User library headers
├── scripts/                  # Build scripts
├── sound/                    # Audio samples
├── Dockerfile                # Container definition
└── geekos_docker.sh          # Container startup script
```

Architecture Layers

```
User Programs (user mode, ring 3)
    ↓ System Calls (INT 0x90)
Kernel (kernel mode, ring 0)
    ├── Process Management (kthread, user, sched)
    ├── Memory Management (mem, paging, malloc)
    ├── File Systems (vfs, pfat, gosfs)
    ├── Synchronization (synch, sem)
    └── Device Drivers (timer, keyboard, ide, screen)
        ↓ Hardware Access (ports, interrupts)
Hardware (QEMU-emulated PC)
    ├── CPU (x86, SMP)
    ├── Memory (10 MB RAM)
    ├── Devices (timer, keyboard, IDE, VGA)
    └── Controllers (APIC, PIC)
```

---

2. Setup Instructions

Docker Installation (Required)

Download Docker:

- Visit https://www.docker.com/
- Download appropriate version for your OS

Platform-Specific Setup:

Windows (WSL):

- Must use WSL-2 (WSL-1 does not work)
- Install Docker Desktop for Windows (not inside WSL)
- Verify: `docker` command should work in WSL
- Enable: Settings → Resources → WSL Integration
- **Critical:** Place geekos folder in Ubuntu filesystem (`~/geekos`), NOT in `/mnt/c/`

Mac (Apple Silicon):

- Docker 4.26+: Settings → General → Uncheck "Use Rosetta for x86/amd64 emulation"
- Docker <4.26: Settings → General → File sharing: gRPC FUSE, Uncheck "Use Virtualization framework"

Linux:

- Standard Docker installation
- Add user to docker group: `sudo usermod -aG docker $USER`

Running GeekOS

```
# 1. Navigate to geekos directory
cd ~/geekos

# 2. Start Docker container (builds image first time)
./geekos_docker.sh

# 3. Inside container, build and run
cd build
make run

# 4. GeekOS boots, shows shell prompt
# Type commands: ls, cat, help

# 5. Exit QEMU: Ctrl-A then X
# 6. Exit container: exit
```

Docker Management

```
# Connect additional terminal
docker exec -it geekos bash

# Remove container (start fresh)
docker container rm geekos

# Remove image (rebuild from scratch)
docker image rm geekos

# View running containers
docker ps
```

---

3. Hardware Architecture

QEMU Emulated Hardware

CPU:

- Intel 386 (32-bit x86)
- Multiple cores for SMP
- Protected mode operation

Memory:

- 10 MB total RAM
- Identity-mapped for kernel (virtual = physical)
- Paged for user processes

Interrupt Controllers:

- **Local APIC** (per CPU): Base `0xFEE00000`
  - Receives interrupts from IO APIC
  - Handles inter-processor interrupts (IPI)
  - Per-CPU timer
- **IO APIC**: Base `0xFEC00000`
  - Routes device interrupts to Local APICs

Timer:

- **PIT** (Programmable Interval Timer): IRQ 0
  - Used only to calibrate LAPIC timers at boot
- **LAPIC Timers**: Interrupt 32
  - One per CPU core
  - Drives scheduling quantum

Keyboard:

- IRQ 1, Interrupt 33
- Ports: 0x60 (data), 0x64 (command)
- Scan code to keycode translation

VGA Display:

- Text mode: 25 rows × 80 columns
- Video memory: 0xB8000
- Each character: 2 bytes (char + attribute)
- Hardware cursor via ports 0x3D4, 0x3D5

IDE Disk Controller:

- Up to 4 drives
- **IDE0 (diskc)**: Boot disk, PFAT filesystem
- **IDE1 (diskd)**: Raw disk for projects
- 16-bit PIO mode (DMA available but unused)
- 256-byte blocks

Serial Port:

- COM1/COM2 standard addresses
- Debug output and serial console (project)

Port I/O

Devices are accessed via x86 port I/O instructions:

```
// Read byte from port
uchar_t In_Byte(ushort_t port);

// Write byte to port
void Out_Byte(ushort_t port, uchar_t value);

// Example: Read keyboard data
uchar_t scanCode = In_Byte(0x60);
```

Interrupt Handling

Interrupt Mapping:

- CPU exceptions: 0-31
- Hardware interrupts: 32+ (IRQ N → Interrupt 32+N)
- System call trap: 144 (0x90)

APIC vs PIC:

- GeekOS uses APIC for SMP support
- PIC used minimally during initialization
- IO APIC distributes interrupts across CPUs

---

4. x86 Processor Architecture

Modes of Operation

Real Mode (16-bit):

- Used briefly during boot
- 1 MB address space (20-bit addresses)
- Address = (segment << 4) + offset
- No memory protection

Protected Mode (32-bit):

- GeekOS runs in this mode
- 4 GB address space (32-bit addresses)
- Memory protection via privilege levels
- Segmentation + optional paging

Privilege Levels

```
Ring 0: Kernel mode (full hardware access)
Ring 1: Unused
Ring 2: Unused
Ring 3: User mode (restricted access)
```

Segmentation

Segment Selector (16-bit):

```
Bits [15:3]: Index into GDT/LDT (13 bits)
Bit  [2]:    Table (0=GDT, 1=LDT)
Bits [1:0]:  Requested Privilege Level (RPL)
```

Segment Descriptor (64-bit):

- Base address: 32 bits (where segment starts)
- Limit: 20 bits (segment size)
- DPL: 2 bits (Descriptor Privilege Level)
- Type: 4 bits (code/data/system)
- Present: 1 bit
- Other flags

Descriptor Tables:

- **GDT (Global Descriptor Table):**
  - System-wide, pointed to by GDTR register
  - Entry 0: Null descriptor
  - Entry 1: Kernel code (KERNEL_CS)
  - Entry 2: Kernel data (KERNEL_DS)
  - Entry 3+: TSS, user segments, LDTs
- **LDT (Local Descriptor Table):**
  - Per-process, pointed to by LDTR register
  - Contains user code/data segments
  - One LDT per user process

Paging

Linear to Physical Translation:

```
Linear Address (32 bits):
  [31:22] Directory Index → Page Directory Entry → Page Table
  [21:12] Table Index     → Page Table Entry     → Physical Page
  [11:0]  Offset          → Byte within page

CR3 register: Physical address of page directory
```

Page Directory/Table Entry (32-bit):

```
[31:12] Physical address (20 bits)
[11:9]  Available for OS
[8]     Global
[7]     Page size (PDE only)
[6]     Dirty (PTE only)
[5]     Accessed
[4]     Cache disabled
[3]     Write-through
[2]     User/Supervisor (0=kernel, 1=user)
[1]     Read/Write
[0]     Present (0 = page fault)
```

Interrupts

IDT (Interrupt Descriptor Table):

- 256 entries (pointed to by IDTR)
- Each entry is an interrupt gate (64 bits)

Interrupt Gate:

```
[63:48] Offset high
[47:32] Segment Selector
[31:16] Offset low
[15:13] DPL (user or kernel only)
[12:8]  Type
```

Interrupt Stack Frame:

```
Same privilege:
  [EFLAGS] [CS] [EIP] [Error Code]

Privilege change (user→kernel):
  [Old SS] [Old ESP] [EFLAGS] [CS] [EIP] [Error Code]
```

Registers

General Purpose (32-bit):

- EAX: Accumulator
- EBX: Base
- ECX: Counter
- EDX: Data
- ESI: Source Index
- EDI: Destination Index
- EBP: Base Pointer (stack frame)
- ESP: Stack Pointer

Segment (16-bit visible + 64-bit hidden):

- CS: Code Segment
- DS, ES, FS, GS: Data Segments
- SS: Stack Segment

Control:

- CR0: Protected mode, paging enable
- CR2: Page fault linear address
- CR3: Page directory base
- CR4: Extended features

System:

- GDTR: Points to GDT (48-bit)
- IDTR: Points to IDT (48-bit)
- LDTR: Points to LDT (16-bit selector)
- TR: Points to TSS (16-bit selector)

- EIP: Instruction Pointer
- EFLAGS: Status flags

---

5. Boot Process

Boot Sequence

```
1. Power On
   ↓
2. BIOS Initialization
   - Configure primary CPU (ID 0)
   - Halt secondary CPUs
   ↓
3. BIOS loads sector 0 of diskc to 0x07C00
   ↓
4. Execute bootsect.asm (real mode)
   - Relocate self to 0x90000
   - Load setup.asm
   - Load kernel image
   ↓
5. Execute setup.asm
   - Determine memory size
   - Create temporary GDT/IDT
   - Enable A20 line
   - Reprogram PIC
   - Enter protected mode
   - Jump to Main()
   ↓
6. Kernel Initialization (Main function)
   ↓
7. Mount root filesystem
   ↓
8. Spawn shell
   ↓
9. System Running
```

bootsect.asm

Purpose: First-stage bootloader

Operations:

1. Relocate from 0x07C00 to 0x90000 (INITSEG)
2. Load setup.asm from disk to 0x90200 (SETUPSEG)
3. Load kernel image from disk to 0x10000 (KERNSEG)
4. Jump to setup.asm

setup.asm

Purpose: Second-stage bootloader, enters protected mode

Operations:

1. Query extended memory size (>1MB)
2. Kill floppy motor
3. Create temporary GDT (kernel code/data segments)
4. Create temporary IDT (null entries)
5. Initialize A20 address line (enable >1MB access)
6. Reprogram PIC (remap interrupts)
7. Set CR0.PE = 1 (enter protected mode)
8. Load segment registers (CS, DS, ES, FS, GS, SS)
9. Push Boot_Info struct on stack
10. Jump to Main() in main.c

---

6. Kernel Initialization

Main() Initialization Sequence

File: `src/geekos/main.c`

Executed by CPU 0 (primary CPU):

```
void Main(struct Boot_Info *bootInfo) {
    // 1. BSS (uninitialized globals)
    Init_BSS();

    // 2. VGA screen
    Init_Screen();

    // 3. Memory management
    Init_Mem(bootInfo);
      // - Init_GDT() - create permanent GDT
      // - Organize RAM into 4KB pages
      // - Init_Heap() - kernel malloc

    // 4. CRC-32 checksums
    Init_CRC32();

    // 5. Task State Segment
    Init_TSS();

    // 6. Interrupts
    Init_Interrupts(0);
      // - Init_IDT() - create permanent IDT (256 entries)
      // - Install dummy handlers

    // 7. SMP (multicore)
    Init_SMP();
      // - Detect CPU count
      // - Start secondary CPUs
      // - Each executes Secondary_Start()

    // 8. Scheduler
    Init_Scheduler(0, KERN_STACK);
```

```
  // - Create initial thread (Main)
  // - Create idle thread
  // - Create reaper thread

// 9. Trap handlers
Init_Traps();
  // - INT 12: Stack exception
  // - INT 13: GPF
  // - INT 0x90: System call

// 10. Local APIC (CPU 0)
Init_Local_APIC(0);

// 11. Timer
Init_Timer();

// 12. Keyboard
Init_Keyboard();

// 13. DMA
Init_DMA();

// 14. IDE disk
Init_IDE();

// 15. Filesystems
Init_PFAT();
Init_GFS2();   // if enabled
Init_GFS3();   // if enabled
Init_GOSFS();
Init_CFS();

// 16. Alarm system
Init_Alarm();

// 17. Serial port
Init_Serial();

// 18. Release secondary CPUs
Release_SMP();

// 19. Mount root filesystem
Mount_Root_Filesystem();
  // Mount IDE0 at /c as PFAT

// 20. Spawn shell
Spawn_Init_Process();
  // Load and execute /c/shell.exe

// 21. Wait for shell to exit
// 22. Shutdown
```

```
    Hardware_Shutdown();
}
```

Secondary CPU Initialization

Each secondary CPU (ID 1, 2, ...) executes:

```
void Secondary_Start(void) {
    Init_GDT();             // Use CPU 0's GDT
    Init_TSS();             // Own TSS
    Init_Interrupts(cpuid); // Own IDT
    Init_Secondary_VM();    // Virtual memory
    Init_Scheduler(cpuid);  // Own scheduler state
    Init_Traps();
    Init_Local_APIC(cpuid); // Own LAPIC
    Init_Timer_Interrupt();

    // Signal CPU 0 that initialization complete
    // Spin until Release_SMP()
    // Then enter scheduler (run idle thread)
}
```

---

7. Thread and Process Management

Thread Types

Kernel Thread:

- Executes only in kernel mode
- Has kernel stack
- No user context

User Thread:

- Can execute in user or kernel mode
- Has kernel stack + user context
- Separate address space

Kernel_Thread Structure

```
struct Kernel_Thread {
    ulong_t esp;                    // Saved stack pointer
    unsigned char *stackPage;       // Kernel stack (4KB)
    struct User_Context *userContext; // NULL for kernel threads

    int priority;
    int totalTime;
    int numTicks;                   // Current quantum
    struct Thread_Queue *runQueue;

    int pid;
```

```
    struct Kernel_Thread *owner;
    struct Thread_Queue joinQueue;
    int exitCode;
    bool alive;
    int refCount;

    const void *tlocalData[MAX_TLOCAL_KEYS];

    // Linked list management
    DEFINE_LINK(Thread_Queue, Kernel_Thread);
    DEFINE_LINK(All_Thread_List, Kernel_Thread);
};
```

User_Context Structure

```
struct User_Context {
    char name[MAX_PROC_NAME_LEN+1];

    char *memory;                   // Process memory
    ulong_t size;

    struct Segment_Descriptor *ldtDescriptor;
    ushort_t ldtSelector;
    ushort_t csSelector;            // Code segment
    ushort_t dsSelector;            // Data segment
    ulong_t entryAddr;              // Entry point
    ulong_t stackPointerAddr;
    ulong_t argBlockAddr;

    pde_t *pageDir;                 // Page directory
    struct File *file_descriptor_table[USER_MAX_FILES];

    int refCount;
};
```

Thread Queues

```
// All threads in system
struct All_Thread_List s_allThreadList;

// Ready to run
extern struct Thread_Queue s_runQueue;

// Currently executing (one per CPU)
struct Kernel_Thread *g_currentThreads[MAX_CPUS];

// Terminated, awaiting cleanup
struct Thread_Queue s_graveyardQueue;
```

Creating a Kernel Thread

```
struct Kernel_Thread* Start_Kernel_Thread(
    Thread_Start_Func startFunc,
    ulong_t arg,
    int priority,
    bool detached,
    const char *name
) {
    // 1. Allocate thread struct + stack
    struct Kernel_Thread *thread = Create_Thread(priority, detached);

    // 2. Setup stack for execution
    // Stack configured so when switched in:
    //   - Executes Launch_Thread (enables interrupts)
    //   - Calls startFunc(arg)
    //   - When returns, calls Shutdown_Thread
    Setup_Kernel_Thread(thread, startFunc, arg);

    // 3. Add to run queue
    Make_Runnable(thread);

    return thread;
}
```

Creating a User Process

```
int Spawn(const char *program, const char *command,
          struct Kernel_Thread **pThread, bool background) {
    // 1. Load executable from filesystem
    char *exeData;
    Read_Fully(program, &exeData, &exeLen);

    // 2. Parse ELF format
    struct Exe_Format exeFormat;
    Parse_ELF_Executable(exeData, exeLen, &exeFormat);

    // 3. Allocate memory
    ulong_t size = exeFormat.maxva + STACK_SIZE;
    char *memory = Malloc(size);

    // 4. Load program segments
    for each segment:
        memcpy(memory + segment.addr, segment.data, segment.size);

    // 5. Format command-line arguments
    Format_Argument_Block(command, memory + exeFormat.maxva);

    // 6. Create user context
    struct User_Context *uc = Create_User_Context();
```

```
    uc->memory = memory;
    uc->size = size;
    uc->entryAddr = exeFormat.entryAddr;
    // Setup LDT, selectors

    // 7. Start user thread
    *pThread = Start_User_Thread(uc, detached);

    return 0;
}
```

---

8. Context Switching

Context State Storage

When a thread is not running, its processor state is saved on its kernel stack:

```
Kernel Stack Layout (high→low addresses):
  [Previous stack contents]
  ------------------------
  [User SS]        ← Only if was in user mode
  [User ESP]       ← Only if was in user mode
  [EFLAGS]
  [CS]
  [EIP]            ← Return address
  [Error Code]     ← Only for some interrupts
  [Interrupt Number]
  [EAX, EBX, ECX, EDX, ESI, EDI, EBP]
  [DS, ES, FS, GS]
  ------------------------
  ESP points here  ← Saved in thread->esp
```

Context Switch Functions

File: `src/geekos/lowlevel.asm`

Handle_Interrupt

Entry: Hardware interrupt, trap, or exception occurred

```
Handle_Interrupt:
    ; Stack already has: [EFLAGS, CS, EIP, error, int#]

    ; Save complete processor state
    push DS, ES, FS, GS
    push EAX, EBX, ECX, EDX, ESI, EDI, EBP

    ; Call C interrupt handler
    push ESP                    ; Pointer to interrupt state
    call [g_interruptTable + interrupt# * 4]
    pop EAX
```

```
    ; Check if context switch needed
    if NOT g_preemptionDisabled[cpuid] AND g_needReschedule[cpuid]:
        ; Move current thread to run queue
        ; Select next thread from run queue
        ; Update g_currentThreads[cpuid]
        ; Load new ESP

    ; Activate user context if present
    if current->userContext != NULL:
        ; Update LDTR, TSS

    ; Restore registers
    pop EBP, EDI, ESI, EDX, ECX, EBX, EAX
    pop GS, FS, ES, DS

    ; Skip error code and interrupt number
    add ESP, 8

    ; Return from interrupt
    IRET
```

Switch_To_Thread

Entry: Called from Schedule(), current thread yielding voluntarily

```
Switch_To_Thread(threadptr):
    ; Stack: [threadptr, return address]

    ; Transform stack to interrupt state
    pushf                       ; EFLAGS
    push CS
    push [return address]       ; EIP
    push 0                      ; Fake error code
    push 0                      ; Fake interrupt number
    push EAX, EBX, ECX, EDX, ESI, EDI, EBP
    push DS, ES, FS, GS

    ; Save current thread's ESP
    mov [current->esp], ESP

    ; Switch to new thread
    mov current, threadptr
    mov ESP, [threadptr->esp]

    ; Activate user context if present
    ; Process signals if present

    ; Restore registers and IRET
    pop GS, FS, ES, DS
    pop EBP, EDI, ESI, EDX, ECX, EBX, EAX
```

```
    add ESP, 8                  ; Skip error/int#
    IRET
```

Getting Current Thread

```
#define CURRENT_THREAD (g_currentThreads[GET_CPU_ID()])

int GET_CPU_ID(void) {
    ulong_t apicId;

    Disable_Interrupts();
    apicId = *((volatile ulong_t *)(0xFEE00000 + 0x20));
    Enable_Interrupts();

    return apicId >> 24;
}
```

---

9. Synchronization

Spinlocks

Use Case: Short critical sections, interrupt handlers

```
typedef struct {
    volatile int lock;
    int state;
} Spin_Lock_t;

void Spin_Lock(Spin_Lock_t *lock) {
    bool intState = Begin_Int_Atomic();  // Disable interrupts

    while (1) {
        while (lock->lock != 0);  // Spin
        if (Test_And_Set(&lock->lock, 1) == 0)
            break;  // Acquired
    }

    lock->state = intState;
}

void Spin_Unlock(Spin_Lock_t *lock) {
    bool intState = lock->state;
    lock->lock = 0;
    End_Int_Atomic(intState);  // Restore interrupts
}
```

Mutexes

Use Case: Longer critical sections, can block

```
struct Mutex {
    int state;                  // LOCKED/UNLOCKED
    struct Kernel_Thread *owner;
    struct Thread_Queue waitingThreads;
    Spin_Lock_t guard;
};

void Mutex_Lock(struct Mutex *mutex) {
    Disable_Interrupts();
    Spin_Lock(&mutex->guard);

    if (mutex->state == MUTEX_LOCKED) {
        // Block until available
        Enqueue(&mutex->waitingThreads, CURRENT_THREAD);
        Schedule_And_Unlock(&mutex->guard);
    } else {
        mutex->state = MUTEX_LOCKED;
        Spin_Unlock(&mutex->guard);
    }

    mutex->owner = CURRENT_THREAD;
    Enable_Interrupts();
}

void Mutex_Unlock(struct Mutex *mutex) {
    Disable_Interrupts();
    Spin_Lock(&mutex->guard);

    if (!Is_Queue_Empty(&mutex->waitingThreads)) {
        struct Kernel_Thread *next = Dequeue(&mutex->waitingThreads);
        mutex->owner = next;
        Wake_Up_One(next);
    } else {
        mutex->state = MUTEX_UNLOCKED;
        mutex->owner = NULL;
    }

    Spin_Unlock(&mutex->guard);
    Enable_Interrupts();
}
```

Condition Variables

```
struct Condition {
    struct Thread_Queue waitingThreads;
};

void Cond_Wait(struct Condition *cond, struct Mutex *mutex) {
    Disable_Interrupts();
    Spin_Lock(&mutex->guard);
```

```
    // Add to wait queue
    Enqueue(&cond->waitingThreads, CURRENT_THREAD);

    // Release mutex atomically
    // (unlock mutex, block, reacquire when woken)
    Mutex_Unlock_And_Wait(mutex, &cond->waitingThreads);

    Enable_Interrupts();
    Mutex_Lock(mutex);  // Reacquire
}

void Cond_Signal(struct Condition *cond) {
    Disable_Interrupts();
    if (!Is_Queue_Empty(&cond->waitingThreads)) {
        Wake_Up_One(Dequeue(&cond->waitingThreads));
    }
    Enable_Interrupts();
}

void Cond_Broadcast(struct Condition *cond) {
    Disable_Interrupts();
    while (!Is_Queue_Empty(&cond->waitingThreads)) {
        Wake_Up_One(Dequeue(&cond->waitingThreads));
    }
    Enable_Interrupts();
}
```

Semaphores (Student Project)

```
struct Semaphore {
    int count;
    struct Thread_Queue waitingThreads;
    Spin_Lock_t guard;
};

void P(struct Semaphore *sem) {
    Spin_Lock(&sem->guard);
    sem->count--;
    if (sem->count < 0) {
        Enqueue(&sem->waitingThreads, CURRENT_THREAD);
        Schedule_And_Unlock(&sem->guard);
    } else {
        Spin_Unlock(&sem->guard);
    }
}

void V(struct Semaphore *sem) {
    Spin_Lock(&sem->guard);
    sem->count++;
    if (sem->count <= 0) {
```

```
        Wake_Up_One(Dequeue(&sem->waitingThreads));
    }
    Spin_Unlock(&sem->guard);
}
```

---

10. Scheduling

Scheduler Configuration

```
// Per-CPU state
struct Kernel_Thread *g_currentThreads[MAX_CPUS];
int g_needReschedule[MAX_CPUS];
volatile int g_preemptionDisabled[MAX_CPUS];

// Global run queue
struct Thread_Queue s_runQueue;
Spin_Lock_t run_queue_spinlock;

// Quantum
#define DEFAULT_MAX_TICKS 4
int g_Quantum = DEFAULT_MAX_TICKS;
```

Timer Interrupt Handler

```
void Timer_Interrupt_Handler(struct Interrupt_State *state) {
    int cpuid = GET_CPU_ID();
    struct Kernel_Thread *current = g_currentThreads[cpuid];

    // CPU 0 maintains global tick count
    if (cpuid == 0) {
        g_numTicks++;
    }

    // Update thread's tick count
    current->numTicks++;
    current->totalTime++;

    // Check if quantum expired
    if (current->numTicks >= g_Quantum) {
        g_needReschedule[cpuid] = true;
    }

    // Process alarms
    Check_Alarms();
}
```

Scheduling Decisions

Preemptive (Timer Interrupt):

```
// In Handle_Interrupt (lowlevel.asm):
if (!g_preemptionDisabled[cpuid] && g_needReschedule[cpuid]) {
    // Move current thread to run queue
    Make_Runnable(current);

    // Select next thread
    struct Kernel_Thread *next = Get_Next_Runnable();

    // Switch
    g_currentThreads[cpuid] = next;
    ESP = next->esp;
    // ... activate user context, etc.
}
```

Voluntary:

```
void Yield(void) {
    Disable_Interrupts();
    Make_Runnable(CURRENT_THREAD);
    Schedule();
    Enable_Interrupts();
}

void Schedule(void) {
    g_preemptionDisabled[GET_CPU_ID()] = false;
    struct Kernel_Thread *next = Get_Next_Runnable();
    Switch_To_Thread(next);
}
```

Run Queue Operations

```
void Make_Runnable(struct Kernel_Thread *thread) {
    Spin_Lock(&run_queue_spinlock);
    KASSERT(thread->runQueue == NULL);

    Enqueue(&s_runQueue, thread);
    thread->runQueue = &s_runQueue;

    Spin_Unlock(&run_queue_spinlock);
}

struct Kernel_Thread *Get_Next_Runnable(void) {
    Spin_Lock(&run_queue_spinlock);

    struct Kernel_Thread *next;
    if (Is_Queue_Empty(&s_runQueue)) {
        next = Get_Idle_Thread(GET_CPU_ID());
    } else {
        next = Dequeue(&s_runQueue);
```

```
        next->runQueue = NULL;
        next->numTicks = 0;
    }

    Spin_Unlock(&run_queue_spinlock);
    return next;
}
```

Idle Thread

```
static void Idle(ulong_t arg) {
    while (true) {
        Enable_Interrupts();
        __asm__ __volatile__("hlt");  // Halt until interrupt
        Disable_Interrupts();

        if (!Is_Queue_Empty(&s_runQueue)) {
            Yield();
        }
    }
}
```

---

11. Virtual Memory

Paging Setup

Page Size: 4 KB (4096 bytes)

Address Translation:

```
Virtual Address → Linear Address → Physical Address
     (segmentation)      (paging)
```

Page Table Structure:

- **Page Directory:** 1024 entries (PDEs)
  - CR3 register points to page directory
- **Page Tables:** 1024 entries each (PTEs)
  - One page table per PDE
- **Pages:** 4 KB each

Page Table Entry Flags

```
#define VM_WRITE    0x02   // Writable
#define VM_USER     0x04   // User-accessible
#define VM_NOCACHE  0x10   // Disable caching
#define VM_READ     0x20   // Readable
#define VM_EXEC     0x40   // Executable
#define VM_PRESENT  0x01   // In memory
```

```
#define VM_ACCESSED 0x20   // Been accessed
#define VM_DIRTY    0x40   // Been modified
```

Kernel Address Space

```
0x00000000 - 0x00001000:  Unmapped (NULL protection)
0x00001000 - 0x000A0000:  Low memory (kernel)
0x000A0000 - 0x00100000:  ISA hole (VGA memory)
0x00100000 - 0x00A00000:  Kernel heap
0x00A00000 - 0xF0000000:  Available for user processes
0xFEC00000 - 0xFEE00000:  IO APIC
0xFEE00000 - 0x00000000:  Local APIC
```

User Address Space (per process)

```
0x00000000 - 0x00001000:  Unmapped
0x00001000 - 0x08000000:  Code and data
0x08000000 - 0xC0000000:  Heap (grows up)
0xC0000000 - 0xF0000000:  Stack (grows down from 0xEFFFFFFF)
0xF0000000 - 0x00000000:  Kernel (inaccessible)
```

Page Fault Handler

```
void Page_Fault_Handler(struct Interrupt_State *state) {
    ulong_t faultAddr = Get_CR2();
    ulong_t errorCode = state->errorCode;

    bool present = (errorCode & 0x1);
    bool write = (errorCode & 0x2);
    bool user = (errorCode & 0x4);

    if (user) {
        // User page fault - try to handle
        if (!present) {
            if (Handle_User_Page_Fault(CURRENT_THREAD, faultAddr, write)) {
                return;  // Handled
            }
        }
        // Could not handle - terminate process
        Print("Page fault: addr=0x%x\n", faultAddr);
        Exit(-1);
    } else {
        // Kernel page fault - panic
        Print("KERNEL PAGE FAULT at 0x%x\n", faultAddr);
        Print("Error code: 0x%x\n", errorCode);
        Dump_Interrupt_State(state);
        KASSERT(false);
    }
}
```

Demand Paging

```
bool Handle_User_Page_Fault(struct Kernel_Thread *thread,
                            ulong_t faultAddr, bool write) {
    struct User_Context *context = thread->userContext;

    // Find memory region
    struct Memory_Region *region = Find_Region(context, faultAddr);
    if (!region) return false;

    // Check permissions
    if (write && !region->writable) return false;

    // Allocate physical page
    void *pageFrame = Alloc_Page();
    if (!pageFrame) {
        pageFrame = Swap_Out_Page();
        if (!pageFrame) return false;
    }

    // Load from swap or zero-fill
    ulong_t pageNum = faultAddr >> 12;
    if (Is_Page_In_Swap(context, pageNum)) {
        Swap_In_Page(context, pageNum, pageFrame);
    } else {
        memset(pageFrame, 0, PAGE_SIZE);
    }

    // Map page
    Map_Page(context->pageDir, faultAddr, pageFrame,
             region->writable, true /* user */);

    return true;
}
```

Copy-on-Write

```
int Fork(void) {
    struct Kernel_Thread *parent = CURRENT_THREAD;
    struct User_Context *parentCtx = parent->userContext;

    // Clone user context
    struct User_Context *childCtx = Clone_User_Context(parentCtx);

    // Mark all writable pages as read-only + COW
    for each writable page:
        Mark_Page_Read_Only(parent, page);
        Mark_Page_Read_Only(child, page);
        Set_COW_Flag(parent, page);
        Set_COW_Flag(child, page);
```

```
        Increment_Page_Ref_Count(page);

    // Start child thread
    struct Kernel_Thread *child = Start_User_Thread(childCtx, false);
    return child->pid;  // Parent returns child PID
    // Child returns 0
}

// On write to COW page:
bool Handle_COW_Page_Fault(struct User_Context *ctx, ulong_t addr) {
    void *oldPage = Get_Physical_Page(ctx, addr);

    if (Get_Page_Ref_Count(oldPage) == 1) {
        // Only reference - just make writable
        Set_Page_Writable(ctx, addr);
        return true;
    }

    // Multiple references - copy page
    void *newPage = Alloc_Page();
    memcpy(newPage, oldPage, PAGE_SIZE);
    Map_Page(ctx->pageDir, addr, newPage, true, true);
    Decrement_Page_Ref_Count(oldPage);

    return true;
}
```

---

12. Filesystem Architecture

VFS Layer

Purpose: Uniform interface to multiple filesystem implementations

Architecture:

```
System Calls (Open, Read, Write, Close)
    ↓
VFS (vfs.c)
    ↓
Filesystems (pfat.c, gosfs.c, gfs2.c)
    ↓
Buffer Cache (bufcache.c)
    ↓
Block Device (blockdev.c)
    ↓
Device Drivers (ide.c, floppy.c)
```

VFS Data Structures

```
// Filesystem type
struct Filesystem {
    char name[VFS_MAX_FS_NAME_LEN+1];
    struct Filesystem_Ops *ops;
        // Format(dev)
        // Mount(mountPoint)
};

// Mount point
struct Mount_Point {
    char prefix[VFS_MAX_PATH_LEN+1];  // e.g., "/c"
    struct Block_Device *dev;
    struct Mount_Point_Ops *ops;
        // Open, Create_Directory, Open_Directory
        // Stat, Sync, Delete
    void *fsData;
};

// Open file
struct File {
    struct File_Ops *ops;
        // FStat, Read, Write, Seek, Close, Read_Entry
    struct Mount_Point *mountPoint;
    int mode;
    ulong_t filePos;
    ulong_t endPos;
    void *fsData;
};
```

VFS Operations

```
// Register filesystem type
int Register_Filesystem(const char *name, struct Filesystem_Ops *ops);

// Mount filesystem
int Mount(const char *devname, const char *prefix, const char *fstype);
// Example: Mount("ide0", "c", "pfat");

// Open file
int Open(const char *path, int mode, struct File **pFile);
// Example: Open("/c/file.txt", O_READ, &file);

// Read/Write
int Read(struct File *file, void *buf, ulong_t numBytes);
int Write(struct File *file, void *buf, ulong_t numBytes);

// Other operations
int Close(struct File *file);
int Seek(struct File *file, ulong_t pos);
int Stat(const char *path, struct VFS_File_Stat *stat);
```

```
int Create_Directory(const char *path);
int Delete(const char *path);
int Sync(void);
```

PFAT Filesystem

Characteristics:

- Read-only
- Single-level directory (no subdirectories)
- FAT for block allocation
- Boot disk filesystem

On-Disk Layout:

```
Block 0:        Boot Sector
Blocks 1–N:     File Allocation Table (FAT)
Blocks N+1–M:   Root Directory
Blocks M+1–End: Data Blocks
```

Structures:

```
struct PFAT_Boot_Sector {
    ushort_t magic;
    ulong_t fileAllocationOffset;
    ulong_t fileAllocationLength;
    ulong_t rootDirectoryOffset;
    ulong_t rootDirectoryCount;
    // ... boot loader info ...
};

struct PFAT_Directory_Entry {
    char filename[12];          // 8.3 format
    ushort_t attributes;
    ushort_t time, date;
    ulong_t firstBlock;
    ulong_t fileSize;
    struct VFS_ACL acls[VFS_MAX_ACL_ENTRIES];
};

struct PFAT_Instance {
    struct PFAT_Boot_Sector bootsector;
    uint_t *fat;                // In-memory FAT
    struct PFAT_Directory_Entry *rootDir;  // In-memory root dir
    struct Mutex lock;
    struct PFAT_File_List fileList;
};
```

Buffer Cache

Purpose: Cache disk blocks in memory

```
struct FS_Buffer {
    int fsBlockNum;
    char *data;                  // 4KB page
    int flags;                   // DIRTY, IN_USE, VALID
};

struct FS_Buffer_Cache {
    struct Block_Device *dev;
    int fsBlockSize;
    int numCached;
    struct FS_Buffer_List bufferList;
    struct Mutex lock;
    struct Condition cond;
};

// Operations
struct FS_Buffer_Cache* Create_FS_Buffer_Cache(struct Block_Device *dev, int
blockSize);
int Get_FS_Buffer(struct FS_Buffer_Cache *cache, int blockNum, struct FS_Buffer
**pBuf);
void Release_FS_Buffer(struct FS_Buffer_Cache *cache, struct FS_Buffer *buf);
void Sync_FS_Buffer_Cache(struct FS_Buffer_Cache *cache);
```

Block Device Layer

```
struct Block_Device {
    char name[BLOCKDEV_MAX_NAME_LEN+1];
    struct Block_Device_Ops *ops;
        // Open, Close, Get_Num_Blocks
    int unit;
    bool inUse;
    void *driverData;
    struct Thread_Queue *waitQueue;
    struct Request_Queue *requestQueue;
};

struct Block_Request {
    struct Block_Device *dev;
    int type;                    // BLOCK_READ, BLOCK_WRITE
    int blockNum;
    void *buf;
    volatile int state;          // PENDING, COMPLETED, ERROR
    volatile int errorCode;
    struct Thread_Queue waitQueue;
};

// Operations
int Register_Block_Device(const char *name, ...);
int Open_Block_Device(const char *name, struct Block_Device **pDev);
```

```
int Block_Read(struct Block_Device *dev, int blockNum, void *buf);
int Block_Write(struct Block_Device *dev, int blockNum, void *buf);
```

---

13. Device Drivers

Timer Driver

```
// Initialization
void Init_Timer(void) {
    Install_IRQ(TIMER_IRQ, Timer_Interrupt_Handler);
    Enable_IRQ(TIMER_IRQ);
}

// Interrupt handler
void Timer_Interrupt_Handler(struct Interrupt_State *state) {
    int cpuid = GET_CPU_ID();
    struct Kernel_Thread *current = g_currentThreads[cpuid];

    if (cpuid == 0) g_numTicks++;
    current->numTicks++;
    current->totalTime++;

    if (current->numTicks >= g_Quantum) {
        g_needReschedule[cpuid] = true;
    }

    Check_Alarms();
}
```

Keyboard Driver

```
// Static data
static struct Key_Queue s_queue;
static struct Thread_Queue s_keyboardWaitQueue;
static Spin_Lock_t s_kbdQueueLock;
static struct Keyboard_State s_kbdState;

// Initialization
void Init_Keyboard(void) {
    Init_Key_Queue(&s_queue);
    Init_Thread_Queue(&s_keyboardWaitQueue);
    Spin_Lock_Init(&s_kbdQueueLock);
    Init_Scan_Code_Tables();
    Install_IRQ(KEYBOARD_IRQ, Keyboard_Interrupt_Handler);
    Enable_IRQ(KEYBOARD_IRQ);
}

// Interrupt handler
void Keyboard_Interrupt_Handler(struct Interrupt_State *state) {
```

```
    uchar_t status = In_Byte(KB_CMD);

    if (status & KB_OUTPUT_FULL) {
        uchar_t scanCode = In_Byte(KB_DATA);
        bool release = (scanCode & KB_KEY_RELEASE);
        scanCode &= ~KB_KEY_RELEASE;

        if (Is_Modifier_Key(scanCode)) {
            Update_Keyboard_State(&s_kbdState, scanCode, release);
        } else if (!release) {
            keycode_t keyCode = Translate_Scan_Code(scanCode, &s_kbdState);

            Spin_Lock(&s_kbdQueueLock);
            if (!Is_Key_Queue_Full(&s_queue)) {
                Enqueue_Key(&s_queue, keyCode);
                Wake_Up(&s_keyboardWaitQueue);
            }
            Spin_Unlock(&s_kbdQueueLock);
        }
    }
}

// Wait for key
keycode_t Wait_For_Key(void) {
    keycode_t key;

    Disable_Interrupts();
    while (true) {
        Spin_Lock(&s_kbdQueueLock);
        if (!Is_Key_Queue_Empty(&s_queue)) {
            key = Dequeue_Key(&s_queue);
            Spin_Unlock(&s_kbdQueueLock);
            break;
        }
        Spin_Unlock(&s_kbdQueueLock);
        Wait(&s_keyboardWaitQueue);
    }
    Enable_Interrupts();

    return key;
}
```

VGA Screen Driver

```
#define VIDMEM_ADDR 0xB8000
#define NUMROWS 25
#define NUMCOLS 80

static ushort_t *s_screenBase = (ushort_t *)VIDMEM_ADDR;
static struct Console_State s_consoleState;
```

```
void Init_Screen(void) {
    Clear_Screen();
    s_consoleState.row = 0;
    s_consoleState.col = 0;
    s_consoleState.attribute = ATTRIB(BLACK, GRAY);
    Update_Cursor();
}

void Put_Char(int ch) {
    switch (ch) {
    case '\n':
        s_consoleState.row++;
        s_consoleState.col = 0;
        break;
    case '\r':
        s_consoleState.col = 0;
        break;
    case '\b':
        if (s_consoleState.col > 0) s_consoleState.col--;
        break;
    default:
        if (ch >= ' ' && ch <= '~') {
            int offset = s_consoleState.row * NUMCOLS + s_consoleState.col;
            s_screenBase[offset] = (s_consoleState.attribute << 8) | ch;
            s_consoleState.col++;
        }
    }

    if (s_consoleState.col >= NUMCOLS) {
        s_consoleState.col = 0;
        s_consoleState.row++;
    }

    if (s_consoleState.row >= NUMROWS) {
        Scroll_Screen();
        s_consoleState.row = NUMROWS - 1;
    }

    Update_Cursor();
}

void Update_Cursor(void) {
    int pos = s_consoleState.row * NUMCOLS + s_consoleState.col;
    Out_Byte(CRT_ADDR_REG, 14);
    Out_Byte(CRT_DATA_REG, (pos >> 8) & 0xFF);
    Out_Byte(CRT_ADDR_REG, 15);
    Out_Byte(CRT_DATA_REG, pos & 0xFF);
}
```

IDE Driver

```
// Driver thread
static void IDE_Request_Thread(ulong_t arg) {
    while (true) {
        struct Block_Request *req =
            Dequeue_Request(&s_ideRequestQueue, &s_ideWaitQueue);

        if (req->type == BLOCK_READ) {
            IDE_Read(req->dev->unit, req->blockNum, req->buf);
        } else {
            IDE_Write(req->dev->unit, req->blockNum, req->buf);
        }

        Notify_Request_Completion(req, COMPLETED, 0);
    }
}

// PIO read
static void IDE_Read(int drive, int blockNum, void *buf) {
    int cylinder, head, sector;
    LBA_To_CHS(drive, blockNum, &cylinder, &head, &sector);

    // Wait for drive ready
    while (In_Byte(IDE_STATUS_REG) & IDE_BSY);

    // Select drive and head
    Out_Byte(IDE_DRIVE_HEAD_REG, 0xA0 | (drive << 4) | head);

    // Set cylinder and sector
    Out_Byte(IDE_CYLINDER_LOW_REG, cylinder & 0xFF);
    Out_Byte(IDE_CYLINDER_HIGH_REG, (cylinder >> 8) & 0xFF);
    Out_Byte(IDE_SECTOR_NUMBER_REG, sector);
    Out_Byte(IDE_SECTOR_COUNT_REG, 1);

    // Issue read command
    Out_Byte(IDE_COMMAND_REG, IDE_READ_SECTORS_CMD);

    // Wait for data ready
    while (!(In_Byte(IDE_STATUS_REG) & IDE_DRQ));

    // Read 512 bytes (256 words)
    ushort_t *wordBuf = (ushort_t *)buf;
    for (int i = 0; i < 256; i++) {
        wordBuf[i] = In_Word(IDE_DATA_REG);
    }
}
```

---

14. System Calls

System Call Interface

Mechanism: INT 0x90 (interrupt 144)

Calling Convention:

- EAX: System call number
- EBX, ECX, EDX, ESI, EDI: Arguments
- EAX: Return value

System Call Numbers

```
#define SYS_NULL            0
#define SYS_EXIT            1
#define SYS_PRINTSTRING     2
#define SYS_GETKEY          3
#define SYS_SETATTR         4
#define SYS_SPAWN           7
#define SYS_WAIT            8
#define SYS_GETPID          9
#define SYS_FORK            12
#define SYS_OPEN            20
#define SYS_CLOSE           21
#define SYS_READ            22
#define SYS_WRITE           23
#define SYS_SEEK            24
#define SYS_STAT            25
// ... many more ...
```

User-Side Wrappers

```
// File: src/libc/process.c
int Spawn_Program(const char *program, const char *command, int background) {
    int result;
    __asm__ __volatile__ (
        "int $0x90"
        : "=a" (result)
        : "a" (SYS_SPAWN), "b" (program), "c" (command), "d" (background)
    );
    return result;
}

void Exit(int exitCode) {
    __asm__ __volatile__ (
        "int $0x90"
        : : "a" (SYS_EXIT), "b" (exitCode)
    );
}

int Get_PID(void) {
    int pid;
    __asm__ __volatile__ (
        "int $0x90"
```

```
        : "=a" (pid)
        : "a" (SYS_GETPID)
    );
    return pid;
}

// File: src/libc/fileio.c
int Open(const char *path, int mode) {
    int result;
    __asm__ __volatile__ (
        "int $0x90"
        : "=a" (result)
        : "a" (SYS_OPEN), "b" (path), "c" (mode)
    );
    return result;
}

int Read(int fd, void *buf, int numBytes) {
    int result;
    __asm__ __volatile__ (
        "int $0x90"
        : "=a" (result)
        : "a" (SYS_READ), "b" (fd), "c" (buf), "d" (numBytes)
    );
    return result;
}
```

Kernel-Side Handler

```
// File: src/geekos/trap.c
static void Syscall_Handler(struct Interrupt_State *state) {
    int syscallNum = state->eax;

    switch (syscallNum) {
    case SYS_EXIT:
        Sys_Exit((int)state->ebx);
        break;
    case SYS_PRINTSTRING:
        state->eax = Sys_PrintString((char *)state->ebx);
        break;
    case SYS_GETKEY:
        state->eax = Sys_GetKey();
        break;
    case SYS_SPAWN:
        state->eax = Sys_Spawn((char *)state->ebx,
                               (char *)state->ecx,
                               (int)state->edx);
        break;
    case SYS_WAIT:
        state->eax = Sys_Wait((int)state->ebx);
        break;
```

```
    case SYS_GETPID:
        state->eax = Sys_GetPID();
        break;
    case SYS_FORK:
        state->eax = Sys_Fork();
        break;
    case SYS_OPEN:
        state->eax = Sys_Open((char *)state->ebx, (int)state->ecx);
        break;
    case SYS_READ:
        state->eax = Sys_Read((int)state->ebx,
                              (void *)state->ecx,
                              (int)state->edx);
        break;
    default:
        Print("Unknown syscall: %d\n", syscallNum);
        state->eax = EINVAL;
        break;
    }
}
```

Example System Call Implementations

```
// File: src/geekos/syscall.c
static int Sys_Exit(int exitCode) {
    Exit(exitCode);
    return 0;  // Never reached
}

static int Sys_PrintString(char *str) {
    char kbuf[1024];
    if (Copy_From_User(kbuf, str, sizeof(kbuf)) != 0) {
        return EINVAL;
    }
    Print("%s", kbuf);
    return 0;
}

static int Sys_GetKey(void) {
    return (int)Wait_For_Key();
}

static int Sys_GetPID(void) {
    return CURRENT_THREAD->pid;
}

static int Sys_Fork(void) {
    return Fork();
}

static int Sys_Open(char *path, int mode) {
```

```
    char pathBuf[VFS_MAX_PATH_LEN];
    if (Copy_From_User(pathBuf, path, sizeof(pathBuf)) != 0) {
        return EINVAL;
    }

    struct File *file;
    int rc = Open(pathBuf, mode, &file);
    if (rc != 0) return rc;

    struct User_Context *context = CURRENT_THREAD->userContext;
    int fd = Allocate_FD(context, file);
    return fd;
}

static int Sys_Read(int fd, void *buf, int numBytes) {
    struct User_Context *context = CURRENT_THREAD->userContext;
    struct File *file = Get_File(context, fd);
    if (!file) return EBADF;

    char *kbuf = Malloc(numBytes);
    if (!kbuf) return ENOMEM;

    int result = Read(file, kbuf, numBytes);
    if (result > 0) {
        Copy_To_User(buf, kbuf, result);
    }

    Free(kbuf);
    return result;
}
```

---

15. Student Projects

Project Configuration

File: `include/geekos/projects.h`

All projects are disabled by default (`false`). Students enable projects by changing to `true` :

```
#define PROJECT_LIMIT_SYSCALLS          false
#define PROJECT_LIMIT_ACTIVE_PROCESSES  false
#define PROJECT_BACKGROUND_JOBS         false
#define PROJECT_SIGNALS                 false
#define PROJECT_SEMAPHORES              false
#define PROJECT_SCHEDULING              false
#define PROJECT_VIRTUAL_MEMORY_A        false
#define PROJECT_VIRTUAL_MEMORY_B        false
#define PROJECT_GOSFS                   false
#define PROJECT_GFS2                    false
#define PROJECT_GFS3                    false
#define PROJECT_CFS                     false
```

```
#define PROJECT_SOUND                   false
#define PROJECT_SERIAL                  false
#define PROJECT_PIPE                    false
#define PROJECT_FORK                    false
```

TODO_P Macro

```
#define TODO_P(proj, message) do { \
    if(proj) { \
        TODO(__FILE__ ": " message); \
    } else if(PROJECT_VERBOSITY) { \
        Print("Invoked function of project: " #proj ": " message "\n"); \
    } \
} while(0)
```

Usage Example:

```
void Init_VM(struct Boot_Info *bootInfo) {
    TODO_P(PROJECT_VIRTUAL_MEMORY_A,
           "initialize virtual memory page tables.");
}
```

Typical Project Sequence

Project 0: Process Limits

- Limit number of active processes
- System call restrictions
- Resource management

Project 1: Background Jobs

- Background process execution
- Job control
- Shell enhancements

Project 2: Signals

- Signal delivery mechanism
- User-space signal handlers
- Signal masks

Project 3: Semaphores & Scheduling

- Implement semaphores (P/V operations)
- Custom scheduling policies
- Priority scheduling, lottery scheduling

Project 4: Virtual Memory

- Part A: Page tables, page fault handling
- Part B: Demand paging, page replacement, swapping

Project 5: Filesystem

- Implement GOSFS (or GFS2/GFS3/CFS)
- Directory hierarchies
- Inodes and block allocation
- Read/write operations

Additional Projects:

- Sound driver
- Serial console
- Pipes
- Fork() implementation
- Network stack (advanced)

Key Files for Projects

Semaphores: `src/geekos/sem.c` **Scheduling:** `src/geekos/sched.c` **Paging:** `src/geekos/paging.c` , `src/geekos/uservm.c` **Filesystem:** `src/geekos/gosfs.c` **Signals:** `src/geekos/signal.c` **System Calls:** `src/geekos/syscall.c`

---

16. Memory Maps

Physical Memory at Boot

```
Address         Contents
-------         --------
0x00000000      BIOS data, IVT
0x00001000      Available memory
0x00007C00      Boot sector (loaded by BIOS)
0x00010000      Kernel image (loaded by bootsect.asm)
                  .text (code)
                  .rodata
                  .data
                  .bss
                kernEnd
0x00090000      Boot sector (relocated, INITSEG)
0x00090200      setup.asm (SETUPSEG)
0x00090400      Setup stack
0x000A0000      ISA hole start
0x000B8000      VGA text memory
0x00100000      ISA hole end
                Initial thread structure
0x00101000      Initial kernel stack
                Kernel heap
0x00111000      Kernel heap end
                Available pages
...
0xFEC00000      IO APIC
0xFEE00000      Local APIC
```

Kernel Virtual Address Space

```
0x00000000 – 0x00001000    Unmapped (NULL protection)
0x00001000 – 0x000A0000    Low memory (identity mapped)
0x000A0000 – 0x00100000    ISA hole (VGA at 0xB8000)
0x00100000 – 0x00A00000    Kernel heap, data structures
0x00A00000 – 0xF0000000    Available for user processes
0xFEC00000 – 0xFEE00000    IO APIC (identity mapped)
0xFEE00000 – 0x00000000    Local APIC (identity mapped)
```

User Process Virtual Address Space

```
0x00000000 – 0x00001000    Unmapped (NULL protection)
0x00001000 – 0x08000000    Code and data segments
0x08000000 – 0xC0000000    Heap (grows upward)
0xC0000000 – 0xEFFFFFFF    Stack (grows downward)
0xF0000000 – 0x00000000    Kernel (inaccessible from user mode)
```

---

17. Key Data Structures

Thread Queue (Generic List)

```
struct Thread_Queue {
    struct Kernel_Thread *head;
    struct Kernel_Thread *tail;
    Spin_Lock_t lock;
};

// Operations
void Enqueue(struct Thread_Queue *queue, struct Kernel_Thread *thread);
struct Kernel_Thread *Dequeue(struct Thread_Queue *queue);
bool Is_Queue_Empty(struct Thread_Queue *queue);
```

Interrupt State

```
struct Interrupt_State {
    uint_t gs, fs, es, ds;
    uint_t edi, esi, ebp, esp, ebx, edx, ecx, eax;
    uint_t intNum;
    uint_t errorCode;
    uint_t eip;
    uint_t cs;
    uint_t eflags;
    // If privilege change:
    uint_t userEsp;
    uint_t userSs;
};
```

VFS File Stat

```
struct VFS_File_Stat {
    int size;
    bool isDirectory;
    bool isSetuid;
    struct VFS_ACL acls[VFS_MAX_ACL_ENTRIES];
};
```

VFS Directory Entry

```
struct VFS_Dir_Entry {
    char name[VFS_MAX_NAME_LEN+1];
    struct VFS_File_Stat stats;
};
```

---

18. Important Functions

Memory Management

```
void *Malloc(ulong_t size);                    // Allocate from kernel heap
void Free(void *ptr);                          // Free kernel memory
void *Alloc_Page(void);                        // Allocate 4KB page
void Free_Page(void *page);                    // Free page
```

Thread Management

```
struct Kernel_Thread *Get_Current(void);       // Current thread
void Make_Runnable(struct Kernel_Thread *thread);
void Yield(void);                              // Voluntary yield
void Exit(int exitCode);                       // Terminate current thread
int Join(struct Kernel_Thread *thread);        // Wait for thread
```

Synchronization

```
void Disable_Interrupts(void);
void Enable_Interrupts(void);
bool Begin_Int_Atomic(void);                   // Returns old state
void End_Int_Atomic(bool state);               // Restore state

void Spin_Lock(Spin_Lock_t *lock);
void Spin_Unlock(Spin_Lock_t *lock);

void Mutex_Init(struct Mutex *mutex);
void Mutex_Lock(struct Mutex *mutex);
void Mutex_Unlock(struct Mutex *mutex);

void Cond_Init(struct Condition *cond);
```

```
void Cond_Wait(struct Condition *cond, struct Mutex *mutex);
void Cond_Signal(struct Condition *cond);
void Cond_Broadcast(struct Condition *cond);
```

User/Kernel Data Transfer

```
int Copy_From_User(void *destInKernel, void *srcInUser, ulong_t numBytes);
int Copy_To_User(void *destInUser, void *srcInKernel, ulong_t numBytes);
void *User_To_Kernel(struct User_Context *context, void *userPtr);
```

Debugging

```
void Print(const char *fmt, ...);              // Kernel printf
void KASSERT(bool condition);                  // Assertion
void Dump_Interrupt_State(struct Interrupt_State *state);
void Dump_Thread_Queue(struct Thread_Queue *queue);
```

---

19. Debugging Tips

Using QEMU Monitor

```
# Start with monitor
make run

# In QEMU window:
Ctrl-Alt-2    # Switch to QEMU monitor
Ctrl-Alt-1    # Switch back to console
```

Monitor Commands:

```
info registers       # Show CPU registers
info mem             # Show page tables
info tlb             # Show TLB
x /10wx 0xaddr       # Examine memory (hex)
xp /10wx 0xaddr      # Examine physical memory
```

Using GDB

```
# Terminal 1: Start QEMU with GDB stub
make debug

# Terminal 2: Start GDB
gdb build/geekos.exe
(gdb) target remote localhost:1234
(gdb) break Main
(gdb) continue
```

Useful GDB Commands:

```
break function_name  # Set breakpoint
break *0xaddress     # Break at address
step                 # Step into
next                 # Step over
continue             # Continue execution
print variable       # Print variable
print/x $eax         # Print register in hex
backtrace            # Show call stack
info threads         # Show threads
x /10wx 0xaddress    # Examine memory
```

Common Debugging Scenarios

Page Fault:

- Check CR2 register (fault address)
- Check error code (present, write, user bits)
- Verify page tables with `info mem`

Triple Fault:

- Usually bad IDT or GDT
- Check that IDTR/GDTR are set correctly
- Verify interrupt handlers are valid

Hang/Deadlock:

- Check with QEMU monitor: `info registers`
- Look at EIP - where is it stuck?
- Check spinlock ownership
- Look for circular wait on mutexes

System Call Not Working:

- Verify INT 0x90 is in IDT with DPL=3
- Check system call number in EAX
- Verify arguments in EBX, ECX, EDX
- Check Copy_From_User/Copy_To_User

Print Debugging

```
// Add debug prints
Print("DEBUG: function entered, arg=%d\n", arg);

// Conditional debug
#define DEBUG_SCHEDULER 0
#if DEBUG_SCHEDULER
Print("Scheduler: switching from %d to %d\n", old->pid, new->pid);
#endif
```

```
// Dump interrupt state
Dump_Interrupt_State(state);
```

Assertions

```
// Use liberally
KASSERT(ptr != NULL);
KASSERT(mutex->owner == CURRENT_THREAD);
KASSERT(thread->runQueue == NULL);
```

Conclusion

GeekOS is a comprehensive teaching operating system that demonstrates fundamental OS concepts through a realistic x86 implementation. This guide provides a complete reference for understanding and extending the system.

Key Takeaways:

- GeekOS runs in x86 protected mode with full SMP support
- Modular architecture with clear separation of concerns
- Well-defined extension points for student projects
- Real hardware interaction (emulated by QEMU)
- Production-quality synchronization and scheduling
- Layered filesystem architecture
- Complete system call interface

Best Practices for Students:

- Read and understand existing code before modifying
- Use assertions liberally
- Test incrementally
- Use GDB for complex debugging
- Ask for help when stuck (office hours, Piazza)

Additional Resources:

- Intel x86 Architecture Manuals
- QEMU Documentation
- GDB Documentation
- Operating Systems textbooks (Silberschatz, Tanenbaum, etc.)

End of GeekOS Master Guide