Complete OS/Kernel Development Guide

Mastering C Fundamentals for Kernel Programming

Version: 1.0
Target Audience: Competitive exam students & kernel developers
Skill Level: Intermediate (Basic C knowledge assumed)

TABLE OF CONTENTS

A. POINTERS & MEMORY ACCESS

Core Concept

A **pointer** is a variable that stores a memory address. In kernel development, pointers are fundamental because:

PROF

- You directly access hardware registers (memory-mapped I/O)
- You manipulate page tables and memory structures
- You work with device memory directly

A1. What is a Pointer?

```
// Normal variable: stores a value
int x = 42;        // x holds value 42, at some memory address

// Pointer: stores an address
int* ptr = &x;     // ptr holds the address of x
```

Memory visualization:

```
Address     Value
0x1000      42      <- x's value
0x1004      0x1000  <- ptr's value (address of x)
```

A2. The Address-of Operator (&x)

Concept: `&` gives you the address of a variable.

```
int value = 100;
int* address = &value;  // address now holds where value is stored

// In kernel: getting register addresses
uint32_t* vga_buffer = (uint32_t*)0xB8000;  // VGA memory starts at
0xB8000
```

Key: You MUST cast to the right pointer type. `0xB8000` is just a number; `(uint32_t*)0xB8000` means "treat this as a pointer to uint32_t".

A3. Dereferencing (*p)

Concept: `*p` gives you the value AT the address that `p` points to.

```
int x = 42;
int* ptr = &x;

printf("%d\n", *ptr);   // Prints 42 (dereference: follow the pointer)
printf("%p\n", ptr);    // Prints 0x... (the address itself)

*ptr = 100;             // Change the value at that address
printf("%d\n", x);      // x is now 100
```

PROF

In kernel context:

```
// VGA text buffer is at 0xB8000
uint16_t* vga = (uint16_t*)0xB8000;

// Write character 'A' (ASCII 65) with white text on black background
*vga = 0x0F00 | 65;     // 0x0F = white on black, 65 = 'A'

// Read from a device register
uint32_t* timer_reg = (uint32_t*)0xFFFF0000;
uint32_t ticks = *timer_reg;  // Read current tick count
```

A4. Pointer Arithmetic (p+1, p++)

Critical Concept: Pointer arithmetic is smart—it scales by the type size!

```
int arr[] = {10, 20, 30, 40};
int* ptr = arr;  // ptr points to arr[0]

ptr++;            // Move to arr[1] — adds sizeof(int) bytes, NOT 1
byte!
                  // If int is 4 bytes, address increases by 4

printf("%d\n", *ptr);  // Prints 20

ptr = arr + 2;    // Points to arr[2]
printf("%d\n", *ptr);  // Prints 30

// Array indexing is pointer arithmetic
printf("%d\n", arr[2]);  // Same as *(arr + 2)
```

Important for kernel:

```
uint8_t* bytes = (uint8_t*)0x1000;
bytes++;                    // Moves to 0x1001 (adds 1 byte)

uint32_t* words = (uint32_t*)0x1000;
words++;                    // Moves to 0x1004 (adds 4 bytes)

// Pointer difference
uint32_t* start = (uint32_t*)0x1000;
uint32_t* end = (uint32_t*)0x2000;
int count = end - start;    // NOT 0x1000, it's (0x2000-
0x1000)/sizeof(uint32_t) = 0x400
```

PROF

A5. Casting Pointers

Concept: Pointers have types. Casting tells the compiler "treat this address as a pointer to THIS type".

```
// Device memory: 0xB8000 is the VGA text buffer
void* generic = (void*)0xB8000;      // void* = "pointer to anything"
uint16_t* vga = (uint16_t*)0xB8000;  // Now it's "pointer to uint16_t"

// Write to first character (VGA stores 2 bytes per character: char +
color)
*vga = (0x0F << 8) | 'A';  // Color byte | Character byte

// Different cast, different interpretation
uint8_t* bytes = (uint8_t*)0xB8000;
bytes[0] = 'A';            // Write character
bytes[1] = 0x0F;           // Write color attributes
```

Critical: Wrong cast = reading/writing wrong data!

```
uint32_t* as_32 = (uint32_t*)0xB8000;
uint16_t* as_16 = (uint16_t*)0xB8000;

// as_32[0] reads 4 bytes starting at 0xB8000
// as_16[0] reads 2 bytes starting at 0xB8000
// Different interpretations of the same memory!
```

A6. Pointers to Structs

Concept: Access struct members through pointers using `->` (arrow operator).

```
struct CPU_State {
    uint32_t eax;
    uint32_t ebx;
    uint32_t ecx;
};

struct CPU_State state;
struct CPU_State* cpu = &state;

// Two ways to access members:
state.eax = 100;        // Direct access
cpu->eax = 100;         // Pointer access (same as (*cpu).eax)

// In kernel: interrupt handler receives CPU state
void interrupt_handler(struct CPU_State* cpu) {
    if (cpu->eax == SYSCALL_EXIT) {
        // Handle exit system call
    }
}
```

PROF

Memory layout matters:

```
struct IDT_Entry {
    uint16_t offset_lo;
    uint16_t selector;
    uint8_t  reserved;
    uint8_t  flags;
    uint16_t offset_hi;
} __attribute__((packed));

// IDT is an array of entries at 0xFFFF8000
struct IDT_Entry* idt = (struct IDT_Entry*)0xFFFF8000;
idt[0].offset_lo = 0x1234;  // Set interrupt handler address low bits
```

A7. Pointers to Functions (Optional but Important)

Concept: Functions have addresses. You can call them through pointers!

```c
// Regular function
void print_message(const char* msg) {
    // implementation
}

// Pointer to that function
void (*func_ptr)(const char*) = print_message;

// Call through pointer
func_ptr("Hello");  // Same as print_message("Hello")

// In kernel: interrupt table
typedef void (*interrupt_handler_t)(void);

interrupt_handler_t handlers[256];
handlers[0] = divide_by_zero_handler;
handlers[1] = debug_handler;
// ...

// Call the handler
handlers[0]();  // Call whatever is registered for interrupt 0
```

Practice Code: Pointers Deep Dive

PROF

```c
/* pointer_practice.c - Understand every line */

#include <stdio.h>
#include <stdint.h>

// Simulate a device register (pretend it's at 0x3F8)
volatile uint8_t device_register;

int main() {
    printf("=== POINTERS & MEMORY ACCESS ===\n\n");

    // ===== 1. Address-of and Dereferencing =====
    printf("1. ADDRESS-OF & DEREFERENCING\n");
    int x = 42;
    int* ptr = &x;

    printf("  Value of x: %d\n", x);
    printf("  Address of x: %p\n", (void*)&x);
```

```c
    printf("  Value at ptr: %d\n", *ptr);
    printf("  Address stored in ptr: %p\n", (void*)ptr);
    printf("  Size of pointer: %zu bytes\n\n", sizeof(ptr));

    // ===== 2. Pointer Arithmetic =====
    printf("2. POINTER ARITHMETIC\n");
    int arr[] = {10, 20, 30, 40, 50};
    int* p = arr;

    printf("  arr[0]=%d, arr[1]=%d, arr[2]=%d\n", arr[0], arr[1],
arr[2]);
    printf("  Using pointer: p=%d\n", *p);

    p++;  // Move to next int
    printf("  After p++: *p=%d (should be 20)\n", *p);

    p += 2;  // Move 2 ints forward
    printf("  After p+=2: *p=%d (should be 40)\n", *p);

    printf("  Address increase: %p vs %p (diff=%ld bytes)\n",
           (void*)arr, (void*)(arr+1), (intptr_t)(arr+1) -
(intptr_t)arr);
    printf("\n");

    // ===== 3. Casting Pointers =====
    printf("3. POINTER CASTING\n");
    uint32_t value = 0x12345678;
    uint32_t* as_32 = &value;
    uint16_t* as_16 = (uint16_t*)&value;
    uint8_t* as_8 = (uint8_t*)&value;

    printf("  Value: 0x%x\n", value);
    printf("  As uint32_t*: 0x%x\n", *as_32);
    printf("  As uint16_t*: 0x%x (first 16 bits)\n", *as_16);
    printf("  As uint8_t*: 0x%x (first 8 bits)\n", *as_8);
    printf("\n");

    // ===== 4. Pointers to Structs =====
    printf("4. POINTERS TO STRUCTS\n");
    struct Point {
        int x;
        int y;
    } point = {10, 20};

    struct Point* pp = &point;
    printf("  Direct: point.x=%d, point.y=%d\n", point.x, point.y);
    printf("  Via pointer: pp->x=%d, pp->y=%d\n", pp->x, pp->y);
    printf("  Equivalent: (*pp).x=%d\n", (*pp).x);
    printf("\n");

    // ===== 5. Arrays and Pointer Equivalence =====
    printf("5. ARRAYS AND POINTER EQUIVALENCE\n");
    int data[] = {100, 200, 300};
```

```
    printf("  data[1]=%d\n", data[1]);
    printf("  *(data+1)=%d (same thing!)\n", *(data+1));
    printf("  data==%p, &data[0]==%p (same!)\n", (void*)data,
(void*)&data[0]);
    printf("\n");

    // ===== 6. Pointer to Pointer =====
    printf("6. POINTER TO POINTER\n");
    int num = 99;
    int* ptr1 = &num;
    int** ptr2 = &ptr1;

    printf("  num=%d\n", num);
    printf("  *ptr1=%d\n", *ptr1);
    printf("  **ptr2=%d\n", **ptr2);
    printf("  ptr2 points to: %p\n", (void*)*ptr2);
    printf("\n");

    return 0;
}
```

Compile & Run:

```
gcc -o pointer_practice pointer_practice.c
./pointer_practice
```

B. DATA TYPES FOR KERNEL DEVELOPMENT

Core Concept

PROF

Kernels must be **precise about size**. User-space can be sloppy (`int` might be 32 or 64 bits). Kernels can't afford ambiguity.

B1. Fixed-Size Integer Types

Why fixed sizes matter:

```
// BAD for kernel development
int x = 100;  // Is this 16? 32? 64 bits? Depends on architecture!

// GOOD for kernel development
uint32_t x = 100;  // ALWAYS 32 bits, everywhere
```

Standard fixed-size types (from stdint.h):

```
uint8_t    // 0 to 255
uint16_t   // 0 to 65,535
uint32_t   // 0 to 4,294,967,295
uint64_t   // 0 to very large

int8_t, int16_t, int32_t, int64_t  // Signed versions
```

Kernel use cases:

```
// Page table entries are exactly 32 or 64 bits
uint32_t page_table_entry;

// Memory address in bootloader
uint32_t kernel_load_address = 0x100000;

// Interrupt descriptor table has fixed format
struct IDT_Entry {
    uint16_t offset_lo;   // LOW 16 bits of handler address
    uint16_t selector;    // Code segment selector
    uint8_t  reserved;    // Must be 0
    uint8_t  flags;       // 7=Present, 6-5=DPL, 4-0=Type
    uint16_t offset_hi;   // HIGH 16 bits of handler address
} __attribute__((packed));
```

B2. Why Kernels Avoid Normal int/long

```
// WRONG - Don't know the size
unsigned int count;
long memory_size;

// RIGHT - Explicit about size
uint32_t count;
uint64_t memory_size;
```

PROF

Problem in practice:

```
// On 32-bit system: int = 32 bits
// On 64-bit system: int = 32 or 64 bits (depends on compiler!)
int x = 0xFFFFFFFF;
if (x > 0) printf("x is positive");  // Might or might not print!

// Always safe
uint32_t y = 0xFFFFFFFF;
```

B3. Endianness (Concept)

Concept: How are bytes ordered in memory?

```
Big Endian (Network byte order):
  0x12345678 stored as: [12] [34] [56] [78]  (most significant byte
first)
  Used by: Networks, SPARC, PowerPC

Little Endian (x86/ARM):
  0x12345678 stored as: [78] [56] [34] [12]  (least significant byte
first)
  Used by: x86, x86-64, ARM, most modern CPUs
```

Why it matters:

```c
uint32_t value = 0x12345678;
uint8_t* bytes = (uint8_t*)&value;

// On little-endian (Intel x86):
printf("%02x %02x %02x %02x\n", bytes[0], bytes[1], bytes[2], bytes[3]);
// Output: 78 56 34 12

// On big-endian (network):
// Output would be: 12 34 56 78
```

In kernel:

```c
// Network packet comes in big-endian
struct IPv4_Header {
    uint16_t source_port;      // Big-endian!
};

// On little-endian x86:
uint16_t port = ntohs(source_port);  // ntohs = "network to host short"
```

PROF

Practice Code: Data Types

```c
/* data_types_practice.c */

#include <stdio.h>
#include <stdint.h>
#include <limits.h>
```

```c
int main() {
    printf("=== KERNEL DATA TYPES ===\n\n");

    // ===== 1. Fixed-size types =====
    printf("1. FIXED-SIZE TYPES\n");
    printf("  uint8_t size:  %zu bytes (range: 0-%u)\n",
sizeof(uint8_t), UINT8_MAX);
    printf("  uint16_t size: %zu bytes (range: 0-%u)\n",
sizeof(uint16_t), UINT16_MAX);
    printf("  uint32_t size: %zu bytes (range: 0-%u)\n",
sizeof(uint32_t), UINT32_MAX);
    printf("  uint64_t size: %zu bytes (range: 0-%lu)\n",
sizeof(uint64_t), UINT64_MAX);
    printf("\n");

    // ===== 2. Comparison: int vs uint32_t =====
    printf("2. INT VS UINT32_T (SIZE MISMATCH)\n");
    printf("  int size: %zu bytes\n", sizeof(int));
    printf("  uint32_t size: %zu bytes\n", sizeof(uint32_t));
    printf("  These MIGHT be the same, but in kernel code, use
uint32_t!\n\n");

    // ===== 3. Overflow behavior =====
    printf("3. OVERFLOW BEHAVIOR\n");
    uint8_t small = 255;
    small++;
    printf("  uint8_t: 255 + 1 = %u (wraps to 0)\n", small);

    uint32_t medium = 0xFFFFFFFF;
    medium++;
    printf("  uint32_t: 0xFFFFFFFF + 1 = 0x%x (wraps to 0)\n", medium);
    printf("\n");

    // ===== 4. Endianness Demonstration =====
    printf("4. ENDIANNESS (LITTLE-ENDIAN ON THIS SYSTEM)\n");
    uint32_t value = 0x12345678;
    uint8_t* bytes = (uint8_t*)&value;

    printf("  Value: 0x%x\n", value);
    printf("  Bytes in memory: ");
    for (int i = 0; i < 4; i++) {
        printf("%02x ", bytes[i]);
    }
    printf("\n");
    printf("  (First byte is 0x78 = little-endian)\n\n");

    // ===== 5. Type sizes for kernel structures =====
    printf("5. KERNEL STRUCTURE SIZING\n");
    struct CPU_Registers {
        uint32_t eax;
        uint32_t ebx;
        uint32_t ecx;
        uint32_t edx;
```

PROF

```
    };

    printf("  struct CPU_Registers: %zu bytes\n", sizeof(struct
CPU_Registers));
    printf("  (4 uint32_t = 4*4 = 16 bytes)\n\n");

    // ===== 6. Using exact types for hardware =====
    printf("6. HARDWARE REGISTER SIMULATION\n");
    // Simulate a device that stores color in upper byte, character in
lower byte
    uint16_t vga_char = (0x0F << 8) | 'A';  // White 'A'

    printf("  VGA character: 0x%x\n", vga_char);
    printf("  Upper byte (color): 0x%02x\n", (vga_char >> 8) & 0xFF);
    printf("  Lower byte (char): 0x%02x ('%c')\n", vga_char & 0xFF,
vga_char & 0xFF);
    printf("\n");

    return 0;
}
```

C. STRUCTS & TYPEDEFS

Core Concept

Structs let you group related data. Typedefs let you create shortcuts. Together, they're how kernels organize complex structures (page tables, task descriptors, etc.).

C1. Declaring and Using Structs

PROF

```
// Basic struct
struct Point {
    int x;
    int y;
};

// Usage
struct Point p;
p.x = 10;
p.y = 20;
```

In kernel context:

```
// Interrupt Descriptor Table Entry
struct IDT_Entry {
    uint16_t offset_lo;     // Handler address bits 0-15
```

```
    uint16_t selector;      // Code segment
    uint8_t  reserved;      // Always 0
    uint8_t  flags;         // Interrupt/trap gate, privilege level
    uint16_t offset_hi;     // Handler address bits 16-31
};

// Global IDT
struct IDT_Entry idt[256];  // 256 interrupt handlers
```

C2. Accessing Struct Fields

```
struct Point p;
p.x = 10;           // Direct member access
printf("%d\n", p.x);

struct Point* pp = &p;
pp->x = 20;         // Pointer access
printf("%d\n", pp->x);      // OR (*pp).x
```

C3. Typedef for Shorter Names

Concept: `typedef` creates an alias for a type.

```
// Long form
struct IDT_Entry idt_table[256];

// With typedef
typedef struct {
    uint16_t offset_lo;
    uint16_t selector;
    uint8_t  reserved;
    uint8_t  flags;
    uint16_t offset_hi;
} IDT_Entry;

// Now you can use IDT_Entry instead of struct IDT_Entry
IDT_Entry idt_table[256];
```

PROF

Common kernel pattern:

```
typedef struct {
    uint32_t* frames;       // Bitmap of free frames
    uint32_t frame_count;
} Frame_Allocator;
```

```
Frame_Allocator alloc;  // Much shorter than struct Frame_Allocator
alloc
```

C4. Packed Structs

Concept: By default, C adds padding to align structs. `__attribute__((packed))` removes padding.

```
// Without packed (might have padding)
struct A {
    uint8_t a;      // 1 byte
    uint16_t b;     // 2 bytes (might start at byte 2, leaving gap)
    uint8_t c;      // 1 byte
};
printf("%zu\n", sizeof(A));  // Likely 6, not 4!

// With packed (no padding)
struct B {
    uint8_t a;      // 1 byte
    uint16_t b;     // 2 bytes (immediately after a)
    uint8_t c;      // 1 byte
} __attribute__((packed));
printf("%zu\n", sizeof(B));  // Exactly 4
```

Critical for kernel:

```
// Memory-mapped registers must be exactly the right size
struct Device_Registers {
    uint8_t control;
    uint16_t status;
    uint32_t data;
} __attribute__((packed));  // MUST use packed!

// If it's padded, your register access will be wrong
volatile struct Device_Registers* dev = (void*)0x12345000;
dev->status = 0x1234;  // Write to wrong location if padded!
```

PROF

C5. Bitfields (Optional but Useful)

Concept: Pack multiple small fields into single bits.

```
// Without bitfields (wastes space)
struct Flags {
    uint8_t enabled;        // Uses 8 bits for 1 bit of info
    uint8_t interrupt;      // Uses 8 bits for 1 bit of info
    uint8_t error;          // Uses 8 bits for 1 bit of info
};
```

```c
printf("%zu\n", sizeof(Flags));  // 3 bytes (wasted!)

// With bitfields
struct Flags_Packed {
    uint8_t enabled : 1;    // Only 1 bit
    uint8_t interrupt : 1;  // Only 1 bit
    uint8_t error : 1;      // Only 1 bit
    uint8_t : 5;            // Padding to fill byte
};
printf("%zu\n", sizeof(Flags_Packed));  // 1 byte (tight!)
```

In kernel (CPU flags):

```c
struct CPU_Flags {
    uint8_t carry : 1;      // Bit 0: Carry flag
    uint8_t : 1;            // Bit 1: Reserved
    uint8_t parity : 1;     // Bit 2: Parity flag
    uint8_t : 1;            // Bit 3: Reserved
    uint8_t adjust : 1;     // Bit 4: Adjust flag
    uint8_t : 1;            // Bit 5: Reserved
    uint8_t zero : 1;       // Bit 6: Zero flag
    uint8_t sign : 1;       // Bit 7: Sign flag
};
```

Practice Code: Structs & Typedefs

PROF

```c
/* structs_practice.c */

#include <stdio.h>
#include <stdint.h>
#include <string.h>

// ===== KERNEL STRUCTURES =====

// 1. IDT Entry - EXACT format required
typedef struct {
    uint16_t offset_lo;     // Bits 0-15 of handler address
    uint16_t selector;      // Code segment selector
    uint8_t  reserved;      // Must be 0
    uint8_t  flags;         // Flags: 7=Present, 6-5=DPL, 4-0=Type
    uint16_t offset_hi;     // Bits 16-31 of handler address
} __attribute__((packed)) IDT_Entry;

// 2. Page Table Entry - must be packed
typedef struct {
    uint32_t present : 1;       // Bit 0: Page present in memory
    uint32_t writable : 1;      // Bit 1: Writable
```

```
    uint32_t user : 1;          // Bit 2: User-accessible
    uint32_t write_through : 1; // Bit 3
    uint32_t cache_disable : 1; // Bit 4
    uint32_t accessed : 1;      // Bit 5
    uint32_t dirty : 1;         // Bit 6
    uint32_t pat : 1;           // Bit 7
    uint32_t global : 1;        // Bit 8
    uint32_t : 3;               // Bits 9-11: Available for OS
    uint32_t page_frame : 20;   // Bits 12-31: Physical page address >>
12
} __attribute__((packed)) Page_Table_Entry;

// 3. CPU State structure
typedef struct {
    uint32_t eax;
    uint32_t ebx;
    uint32_t ecx;
    uint32_t edx;
    uint32_t esi;
    uint32_t edi;
    uint32_t ebp;
    uint32_t esp;
    uint32_t eip;
} CPU_State;

int main() {
    printf("=== STRUCTS & TYPEDEFS FOR KERNEL DEV ===\n\n");

    // ===== 1. Basic struct usage =====
    printf("1. BASIC STRUCT\n");
    CPU_State state = {
        .eax = 0x12345678,
        .ebx = 0x87654321,
        .ecx = 0xDEADBEEF,
    };

    printf("  CPU State: eax=0x%x, ebx=0x%x, ecx=0x%x\n",
           state.eax, state.ebx, state.ecx);
    printf("  Size of CPU_State: %zu bytes\n", sizeof(CPU_State));
    printf("\n");

    // ===== 2. Struct pointers =====
    printf("2. STRUCT POINTERS\n");
    CPU_State* cpu_ptr = &state;
    printf("  Via pointer: eax=0x%x\n", cpu_ptr->eax);
    printf("  Via dereference: eax=0x%x\n", (*cpu_ptr).eax);
    printf("\n");

    // ===== 3. Packed vs unpacked =====
    printf("3. PACKED STRUCTS (SIZE MATTERS!)\n");

    struct Unpacked {
        uint8_t a;
```

```
        uint16_t b;
        uint8_t c;
    };

    struct Packed {
        uint8_t a;
        uint16_t b;
        uint8_t c;
    } __attribute__((packed));

    printf("  Without packed: %zu bytes\n", sizeof(struct Unpacked));
    printf("  With packed: %zu bytes\n", sizeof(struct Packed));
    printf("  (Kernel registers MUST use packed!)\n\n");

    // ===== 4. IDT Entry example =====
    printf("4. IDT ENTRY (REAL KERNEL STRUCTURE)\n");
    IDT_Entry entry = {};

    // Set up interrupt handler at 0x08001234
    uint32_t handler_addr = 0x08001234;
    entry.offset_lo = handler_addr & 0xFFFF;      // Low 16 bits
    entry.offset_hi = (handler_addr >> 16) & 0xFFFF; // High 16 bits
    entry.selector = 0x08;                         // Code segment
    entry.reserved = 0;
    entry.flags = 0x8E;                            // Present, interrupt
gate

    printf("  Handler at: 0x%x\n", handler_addr);
    printf("  Stored in IDT: offset_lo=0x%x, offset_hi=0x%x\n",
           entry.offset_lo, entry.offset_hi);
    printf("  IDT_Entry size: %zu bytes (must be 8!)\n",
sizeof(IDT_Entry));
    printf("\n");

    // ===== 5. Page Table Entry with bitfields =====
    printf("5. PAGE TABLE ENTRY (BITFIELDS)\n");
    Page_Table_Entry pte = {};
    pte.present = 1;
    pte.writable = 1;
    pte.page_frame = 0x100;  // Physical address = 0x100000

    printf("  PTE value: 0x%x\n", *(uint32_t*)&pte);
    printf("  Present: %u, Writable: %u, Page: 0x%x\n",
           pte.present, pte.writable, pte.page_frame);
    printf("  PTE size: %zu bytes (must be 4!)\n",
sizeof(Page_Table_Entry));
    printf("\n");

    // ===== 6. Array of structs (like real IDT) =====
    printf("6. ARRAY OF STRUCTS (IDT TABLE)\n");
    IDT_Entry idt[256];
    memset(idt, 0, sizeof(idt));  // Clear all entries
```

PROF

```
    // Set up first 3 interrupt handlers
    for (int i = 0; i < 3; i++) {
        uint32_t handler = 0x08000000 + (i * 0x100);
        idt[i].offset_lo = handler & 0xFFFF;
        idt[i].offset_hi = (handler >> 16) & 0xFFFF;
        idt[i].selector = 0x08;
        idt[i].flags = 0x8E;
    }

    printf("  IDT table size: %zu bytes\n", sizeof(idt));
    printf("  Each entry: %zu bytes\n", sizeof(IDT_Entry));
    printf("  Entries set up: 3\n\n");

    return 0;
}
```

D. STORAGE CLASSES

Core Concept

Storage class keywords control **visibility** (where can this be seen?) and **lifetime** (how long does it exist?).

D1. extern Keyword

Concept: `extern` says "this variable/function is defined elsewhere, I'm just using it".

```
// file1.c
int global_counter = 0;  // Definition: allocates storage
void increment() {
    global_counter++;
}

// file2.c
extern int global_counter;  // Declaration: says it exists elsewhere
extern void increment();     // Declaration

void print_counter() {
    printf("%d\n", global_counter);  // Can use it here
}
```

In kernel (linker script integration):

```
// kernel.c - entry point
extern uint8_t _kernel_start;   // Defined by linker script
extern uint8_t _kernel_end;
```

```
int main() {
    uint32_t kernel_size = (uint32_t)&_kernel_end -
(uint32_t)&_kernel_start;
}
```

D2. Static at Global Scope (Internal Linkage)

Concept: `static` at global scope makes a variable **private to this file**.

```
// file1.c
static int private_counter = 0;  // Only visible in this file
int public_counter = 0;          // Visible to other files

// file2.c
extern int public_counter;       // OK - can access
extern int private_counter;      // ERROR - doesn't exist here!
```

Why it matters:

```
// memory.c
static uint8_t kernel_heap[4096];  // Private to memory.c
static uint32_t heap_ptr = 0;      // Private allocation pointer

void* malloc(size_t size) {
    void* ptr = kernel_heap + heap_ptr;
    heap_ptr += size;
    return ptr;
}

// Other files can't accidentally use kernel_heap
```

PROF

D3. Static Inside Functions (Local Static)

Concept: `static` inside a function makes the variable **persistent** between calls.

```
void counter() {
    static int count = 0;  // Initialized only once
    count++;
    printf("Call %d\n", count);
}

counter();  // Prints "Call 1"
counter();  // Prints "Call 2" - count persists!
counter();  // Prints "Call 3"
```

In kernel:

```
// Interrupt counter
void handle_interrupt() {
    static uint32_t interrupt_count = 0;
    interrupt_count++;
    // ...
}

// IDT lazily initialized
void* get_idt() {
    static IDT_Entry idt[256] = {0};  // Created once, reused
    return idt;
}
```

D4. Global vs Local Variables

```
// Global - exists entire program lifetime
int global_var = 10;

void function() {
    // Local - exists only during function execution
    int local_var = 20;

    {
        // Block scope - exists only within this block
        int block_var = 30;
    }
    // block_var doesn't exist here
}
```

PROF

D5. Linkage: Internal vs External

```
// EXTERNAL LINKAGE - visible to other files
int global = 10;
void function() { }

// INTERNAL LINKAGE - NOT visible to other files
static int private = 10;
static void private_func() { }

// file-private linkage
extern int external;  // Declares something from elsewhere
```

Practice Code: Storage Classes

```c
/* storage_classes_practice.c */

#include <stdio.h>
#include <stdint.h>

// ===== GLOBAL SCOPE =====

// 1. External linkage (visible to other files)
int global_counter = 0;

// 2. Internal linkage (private to this file)
static int private_counter = 0;

// 3. Function with external linkage
void increment_global() {
    global_counter++;
}

// 4. Function with internal linkage
static void increment_private() {
    private_counter++;
}

// 5. Static inside function (persistent)
void demo_static_function() {
    static int call_count = 0;
    call_count++;
    printf("  Static inside function, call #%d\n", call_count);
}

// ===== SIMULATED KERNEL ALLOCATOR =====

static uint8_t kernel_heap[1024];  // Private heap
static uint32_t heap_position = 0;

void* kernel_malloc(uint32_t size) {
    if (heap_position + size > 1024) return NULL;

    void* ptr = kernel_heap + heap_position;
    heap_position += size;
    return ptr;
}

int main() {
    printf("=== STORAGE CLASSES ===\n\n");

    // ===== 1. Global vs Local =====
    printf("1. GLOBAL vs LOCAL\n");
    printf("  global_counter=%d (global scope, exists always)\n",
```

PROF

```
global_counter);

    int local_var = 100;
    printf("  local_var=%d (local scope, exists until function ends)\n",
local_var);
    printf("\n");

    // ===== 2. Static at global scope =====
    printf("2. STATIC AT GLOBAL SCOPE\n");
    increment_global();
    increment_global();
    printf("  global_counter after 2 increments: %d\n", global_counter);

    increment_private();
    increment_private();
    printf("  private_counter after 2 increments: %d\n",
private_counter);
    printf("  (private_counter is only visible in this file)\n\n");

    // ===== 3. Static inside function =====
    printf("3. STATIC INSIDE FUNCTION (PERSISTENT)\n");
    demo_static_function();
    demo_static_function();
    demo_static_function();
    printf("  (Count persists between calls)\n\n");

    // ===== 4. Simulated kernel memory allocator =====
    printf("4. KERNEL MEMORY ALLOCATOR (STATIC PRIVATE)\n");
    void* ptr1 = kernel_malloc(256);
    void* ptr2 = kernel_malloc(256);

    printf("  Allocated 256 bytes: %p\n", ptr1);
    printf("  Allocated 256 bytes: %p\n", ptr2);
    printf("  Heap position: %u / 1024\n", heap_position);
    printf("  (kernel_heap is private static buffer)\n\n");

    return 0;
}
```

PROF

E. HEADER FILES & COMPILATION STRUCTURE

Core Concept

Large projects split code across multiple files. Headers declare what's available; source files implement it.

E1. Header Files (.h)

Purpose: Tell other files what functions/types exist.

```c
/* math.h */
#ifndef MATH_H
#define MATH_H

int add(int a, int b);
int multiply(int a, int b);

#endif
```

```c
/* math.c - implements what math.h declares */
#include "math.h"

int add(int a, int b) {
    return a + b;
}

int multiply(int a, int b) {
    return a * b;
}
```

```c
/* main.c - uses math functions */
#include "math.h"
#include <stdio.h>

int main() {
    printf("%d\n", add(2, 3));
    return 0;
}
```

PROF

E2. Header Guards / #pragma once

Concept: Prevent including same header multiple times.

```c
/* device.h - BAD (no guard) */
typedef struct {
    uint32_t id;
} Device;  // ERROR if included twice: "Device already defined"

/* device.h - GOOD (header guard) */
#ifndef DEVICE_H
#define DEVICE_H

typedef struct {
    uint32_t id;
} Device;
```

```
#endif
```

Modern alternative:

```
/* device.h - Alternative: #pragma once */
#pragma once

typedef struct {
    uint32_t id;
} Device;
```

E3. Function Declarations vs Definitions

```
// DECLARATION - just says it exists
void print_message(const char* msg);

// DEFINITION - actually implements it
void print_message(const char* msg) {
    printf("%s\n", msg);
}
```

Usage:

```
/* utils.h - declarations */
void print_message(const char* msg);
int calculate(int x, int y);

/* utils.c - definitions */
void print_message(const char* msg) {
    printf("%s\n", msg);
}

int calculate(int x, int y) {
    return x + y;
}

/* main.c - declarations allow calling */
#include "utils.h"

int main() {
    print_message("Hello");  // Declaration in utils.h tells compiler
this is OK
    return 0;
}
```

PROF

E4. Splitting Kernel Code

Real kernel structure:

```
kernel/
├── kernel.c          # Entry point, main kernel loop
├── kernel.h          # Declarations for other files
├── boot.asm          # CPU bootloader
├── memory/
│   ├── paging.c      # Virtual memory implementation
│   ├── paging.h      # Virtual memory interface
│   ├── heap.c        # Memory allocator
│   └── heap.h
├── interrupts/
│   ├── idt.c         # Interrupt descriptor table
│   ├── idt.h
│   ├── handler.c     # Interrupt handlers
│   └── handler.h
├── drivers/
│   ├── uart.c        # Serial port driver
│   ├── uart.h
│   └── vga.c         # Text mode display
└── Makefile          # Compilation rules
```

E5. Makefile Basics

Simple Makefile:

```
CC = gcc
CFLAGS = -ffreestanding -Wall -Wextra

# Rule to compile .c to .o
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

# Rule to link multiple .o files
kernel.bin: boot.o kernel.o memory.o interrupts.o
    ld -m elf_i386 -T linker.ld -o kernel.bin $^

# Default target
all: kernel.bin

clean:
    rm -f *.o kernel.bin
```

Usage:

```
```

PROF

```
make           # Builds kernel.bin
make clean     # Removes built files
```

Practice Code: Header Files & Compilation

```c
/* device.h - PUBLIC INTERFACE */
#ifndef DEVICE_H
#define DEVICE_H

#include <stdint.h>

typedef struct {
    uint32_t id;
    uint32_t port;
    uint8_t status;
} Device;

// Function declarations
void device_init();
void device_write(uint32_t data);
uint32_t device_read();

#endif  // DEVICE_H
```

```c
/* device.c - IMPLEMENTATION */
#include "device.h"

// Static: private to this file
static Device devices[10];
static uint32_t device_count = 0;

void device_init() {
    for (int i = 0; i < 10; i++) {
        devices[i].id = i;
        devices[i].port = 0x1000 + (i * 0x100);
        devices[i].status = 0;
    }
    device_count = 10;
}

void device_write(uint32_t data) {
    if (device_count > 0) {
        devices[0].status = 1;
        // Simulate writing to device
    }
}
```

PROF

```c
uint32_t device_read() {
    if (device_count > 0) {
        return devices[0].id;
    }
    return 0;
}
```

```c
/* main.c - USES DEVICE THROUGH HEADER */
#include "device.h"
#include <stdio.h>

int main() {
    device_init();
    device_write(0x12345678);
    uint32_t data = device_read();
    printf("Read from device: 0x%x\n", data);
    return 0;
}
```

Compile:

```
gcc -c device.c -o device.o
gcc -c main.c -o main.o
gcc device.o main.o -o program
./program
```

F. MEMORY LAYOUT OF A KERNEL PROGRAM

PROF

Core Concept

Your kernel binary has different sections. Understanding this is critical for kernel configuration.

F1. ELF Sections Explained

```
MEMORY LAYOUT:
┌──────────────┐ 0x00000000
│    .text     │  Code (read-only, executable)
├──────────────┤
│    .data     │  Initialized globals (read-write)
├──────────────┤
│    .bss      │  Zero-initialized globals (not in binary!)
├──────────────┤
```

```
|    HEAP      |  Dynamic allocation
├──────────────┤ STACK grows down
|   STACK      |  Function locals, return addresses
└──────────────┘ High addresses
```

F2. .text Section (Code)

```c
/* program.c */
void hello() {
    printf("Hello");
}

int main() {
    hello();
    return 0;
}

// .text contains:
//   - Machine code for hello()
//   - Machine code for main()
//   - String "Hello"
```

Check it:

```
gcc -c program.c -o program.o
objdump -s program.o | head  # Shows sections
```

F3. .data Section (Initialized Globals)

PROF

```c
// Goes in .data
int global_var = 42;
const char* string = "Hello";
static uint32_t config = 0xDEADBEEF;

int main() {
    return 0;
}

// .data contains actual values: 42, "Hello", 0xDEADBEEF
```

F4. .bss Section (Zero-Initialized)

Critical optimization: Uninitialized globals don't need space in binary!

```
// Goes in .bss (NOT in binary file)
int uninitialized_global;           // 4 bytes, value = 0
static uint8_t kernel_stack[4096];  // 4096 bytes, value = 0

// Binary file doesn't contain these bytes!
// Kernel just allocates space at load time
```

Why it matters:

```
binary_with_init.bin:     2 MB (contains all .data bytes)
binary_with_uninit.bin:   50 KB (only metadata about .bss)

Kernel boots smaller! Bootloader reads only 50 KB, allocates 2 MB at
runtime.
```

F5. Stack vs Heap

```
void function() {
    int x = 10;            // Stack - automatic, freed at }
    int* ptr = malloc(100);  // Heap - manual, must free()
}
```

Layout during execution:

```
HIGH ADDRESSES
┌──────────────────┐
│   STACK          │  x stored here, freed when function returns
│   grows ↓        │
├──────────────────┤
│   HEAP           │  malloc'd memory stays until free()
│   grows ↑        │
├──────────────────┤
│   .bss           │  uninitialized globals (kernel_stack, etc.)
├──────────────────┤
│   .data          │  initialized globals (global_var = 42)
├──────────────────┤
│   .text          │  code (function instructions)
LOW ADDRESSES
```

F6. Kernel Boot Memory Layout

When bootloader loads kernel at 0x100000:

```
```

```
Bootloader prepares:
1. Load .text from binary into RAM at 0x100000
2. Load .data from binary into RAM after .text
3. Calculate .bss size, zero that memory
4. Set up initial stack at high address
5. Set initial registers (esp to stack, eip to main)
6. Jump to main()
```

Kernel linker script controls this:

```
SECTIONS
{
    . = 0x100000;  /* Start at 1 MB */

    .text : { *(.text) }     /* All .text sections */
    .data : { *(.data) }     /* All .data sections */
    .bss : { *(.bss) *(COMMON) }  /* All .bss sections */
}
```

Practice Code: Memory Layout Investigation

PROF

```c
/* memory_layout.c */

#include <stdio.h>
#include <stdint.h>

// .text section - code stored here
void function_a() {
    printf("In function_a\n");
}

void function_b() {
    printf("In function_b\n");
}

// .data section - initialized globals stored here
int initialized_global = 42;
const char* global_string = "Hello from .data";
static int static_global = 100;

// .bss section - zero-initialized (NOT in binary)
int uninitialized_global;
static uint8_t large_buffer[10000];

// Stack demonstration
void show_stack_layout() {
```

```c
    int local1 = 1;
    int local2 = 2;
    int local3 = 3;

    printf("Stack variables (going down):\n");
    printf("  local1 at %p (value: %d)\n", (void*)&local1, local1);
    printf("  local2 at %p (value: %d)\n", (void*)&local2, local2);
    printf("  local3 at %p (value: %d)\n", (void*)&local3, local3);
}

int main() {
    printf("=== MEMORY LAYOUT ===\n\n");

    // ===== 1. Code location =====
    printf("1. .TEXT SECTION (Code)\n");
    printf("  function_a at %p\n", (void*)&function_a);
    printf("  function_b at %p\n", (void*)&function_b);
    printf("  main at %p\n", (void*)&main);
    printf("  (All in low memory - .text section)\n\n");

    // ===== 2. Data section =====
    printf("2. .DATA SECTION (Initialized Globals)\n");
    printf("  initialized_global at %p (value: %d)\n",
           (void*)&initialized_global, initialized_global);
    printf("  global_string at %p (value: %s)\n",
           (void*)&global_string, global_string);
    printf("  static_global at %p (value: %d)\n",
           (void*)&static_global, static_global);
    printf("  (All above .text)\n\n");

    // ===== 3. BSS section =====
    printf("3. .BSS SECTION (Uninitialized/Zero-init Globals)\n");
    printf("  uninitialized_global at %p (value: %d)\n",
           (void*)&uninitialized_global, uninitialized_global);
    printf("  large_buffer at %p (size: 10000)\n",
           (void*)&large_buffer);
    printf("  (Above .data, not in binary file!)\n\n");

    // ===== 4. Stack layout =====
    printf("4. STACK (Local Variables)\n");
    show_stack_layout();
    printf("  (Stack grows downward in memory)\n\n");

    // ===== 5. Relative memory positions =====
    printf("5. MEMORY ORDER\n");
    printf("  .text:  %p\n", (void*)&main);
    printf("  .data:  %p\n", (void*)&initialized_global);
    printf("  .bss:   %p\n", (void*)&large_buffer);
    printf("  Stack:  %p\n", (void*)&uninitialized_global);  // Local in
this frame
    printf("\n");
```

PROF

```
    return 0;
}
```

Compile & Run:

```
gcc -o memory_layout memory_layout.c
./memory_layout
```

Inspect the binary:

```
objdump -h memory_layout    # Shows section sizes
size memory_layout          # Shows .text, .data, .bss sizes
nm memory_layout | head     # Shows symbol addresses
```

G. C RESTRICTIONS IN KERNEL DEVELOPMENT

Core Concept

Standard C assumes you have an operating system. Kernels are the OS. You can't use OS features!

G1. No printf() - Write Your Own

The Problem:

```
printf("Hello");  // Calls libc, which calls syscalls, which need OS!
// In a kernel, there IS no OS yet. You'll crash.
```

PROF

The Solution: Write your own

```
// uart_printf.c - Write to serial port
void uart_putchar(char c) {
    volatile uint8_t* uart = (uint8_t*)0x3F8;  // Serial port address
    *uart = c;  // Write character to device
}

void uart_printf(const char* fmt, ...) {
    // Simple implementation: handle %s, %d, %x
    // (Full printf is 1000+ lines, implement as needed)
}
```

G2. No malloc() - Write Your Own Allocator

The Problem:

```
int* ptr = malloc(100);  // Calls libc, which needs OS memory management
// In a kernel, malloc doesn't exist.
```

The Solution: Simple bump allocator

```
static uint8_t heap[65536];
static uint32_t heap_pos = 0;

void* kernel_malloc(size_t size) {
    if (heap_pos + size > sizeof(heap)) return NULL;
    void* ptr = heap + heap_pos;
    heap_pos += size;
    return ptr;
}

void kernel_free(void* ptr) {
    // Simple allocator doesn't support free
    // (You'd implement that later)
}
```

G3. No stdlib.h, stdio.h, string.h

What you CAN'T use:

```
#include <stdio.h>      // printf, scanf, FILE* - NO
#include <stdlib.h>     // malloc, free - NO
#include <string.h>     // strcpy, strlen - NO
#include <math.h>       // sin, cos - NO
#include <pthread.h>    // Threading - NO
```

PROF

What you CAN use:

```
#include <stdint.h>     // uint32_t, etc - YES
#include <stddef.h>     // NULL, size_t - YES
#include <stdarg.h>     // varargs for printf - YES (use in your printf)
#include <stdnoreturn.h> // noreturn - YES
#include <stdbool.h>    // bool, true, false - YES
#include <limits.h>     // INT_MAX, etc - YES
```

G4. No Exceptions, No Threads, No System Calls

```
// NO EXCEPTIONS
try {
    dangerous_operation();
} catch (...) {
    handle_error();
}
// Kernels handle errors with return codes!

// NO THREADS
pthread_create(&thread, NULL, func, arg);
// Kernels create processes/tasks manually

// NO SYSTEM CALLS
open("file.txt", O_RDONLY);  // Calls syscall which needs... OS!
```

Kernel error handling:

```
// Return error code instead
typedef enum {
    OK = 0,
    ERROR_NO_MEMORY = 1,
    ERROR_INVALID_ARG = 2,
} Error_Code;

Error_Code allocate_page(void** page_ptr) {
    // ...
    if (out_of_memory) {
        return ERROR_NO_MEMORY;
    }
    *page_ptr = allocated;
    return OK;
}
```

PROF

G5. Only Freestanding C

Hosted C (normal programs):

- Has libc, malloc, printf, etc.
- Runs on top of an OS
- `-fno-builtin` not needed (actually conflicts)

Freestanding C (kernels):

- No libc
- No automatic library functions
- You provide EVERYTHING

```
# Hosted (normal) compilation
gcc -O2 program.c -o program

# Freestanding (kernel) compilation
gcc -ffreestanding -O2 program.c -o program
```

Practice Code: Kernel-Safe Utilities

```
/* kernel_utils.h */
#ifndef KERNEL_UTILS_H
#define KERNEL_UTILS_H

#include <stdint.h>
#include <stddef.h>

// Memory functions (you write these)
void kernel_memset(void* ptr, uint8_t value, size_t size);
void kernel_memcpy(void* dest, const void* src, size_t size);
int kernel_memcmp(const void* a, const void* b, size_t size);

// String functions (you write these)
size_t kernel_strlen(const char* str);
void kernel_strcpy(char* dest, const char* src);
int kernel_strcmp(const char* a, const char* b);

// Simple printf for debugging
void kernel_printf(const char* fmt, ...);

#endif
```

PROF

```
/* kernel_utils.c */
#include "kernel_utils.h"
#include <stdarg.h>

// ===== MEMORY FUNCTIONS =====

void kernel_memset(void* ptr, uint8_t value, size_t size) {
    uint8_t* bytes = (uint8_t*)ptr;
    for (size_t i = 0; i < size; i++) {
        bytes[i] = value;
    }
}

void kernel_memcpy(void* dest, const void* src, size_t size) {
    uint8_t* d = (uint8_t*)dest;
    const uint8_t* s = (const uint8_t*)src;
```

```
    for (size_t i = 0; i < size; i++) {
        d[i] = s[i];
    }
}

int kernel_memcmp(const void* a, const void* b, size_t size) {
    const uint8_t* aa = (const uint8_t*)a;
    const uint8_t* bb = (const uint8_t*)b;
    for (size_t i = 0; i < size; i++) {
        if (aa[i] != bb[i]) {
            return aa[i] < bb[i] ? -1 : 1;
        }
    }
    return 0;
}

// ===== STRING FUNCTIONS =====

size_t kernel_strlen(const char* str) {
    size_t len = 0;
    while (str[len] != '\0') {
        len++;
    }
    return len;
}

void kernel_strcpy(char* dest, const char* src) {
    while (*src != '\0') {
        *dest++ = *src++;
    }
    *dest = '\0';
}

int kernel_strcmp(const char* a, const char* b) {
    while (*a && *a == *b) {
        a++;
        b++;
    }
    return (unsigned char)*a - (unsigned char)*b;
}

// ===== SIMPLE PRINTF =====

// Simulated serial port for output
static void putchar_impl(char c) {
    // In real kernel, write to UART/serial
    // For now, we'll rely on standard putchar
    __builtin_putchar(c);  // Compiler builtin
}

void kernel_printf(const char* fmt, ...) {
    va_list args;
    va_start(args, fmt);
```

```
    while (*fmt) {
        if (*fmt == '%' && *(fmt + 1)) {
            fmt++;
            if (*fmt == 'd') {
                // Print integer
                int num = va_arg(args, int);
                if (num < 0) {
                    putchar_impl('-');
                    num = -num;
                }

                // Simple digit printing
                char digits[20];
                int idx = 0;
                if (num == 0) {
                    digits[idx++] = '0';
                } else {
                    while (num) {
                        digits[idx++] = '0' + (num % 10);
                        num /= 10;
                    }
                }
                while (idx--) putchar_impl(digits[idx]);
            } else if (*fmt == 's') {
                // Print string
                const char* str = va_arg(args, const char*);
                while (*str) putchar_impl(*str++);
            } else if (*fmt == 'x') {
                // Print hex
                unsigned int num = va_arg(args, unsigned int);
                const char* hex = "0123456789abcdef";
                char digits[8];
                for (int i = 7; i >= 0; i--) {
                    digits[i] = hex[num & 0xF];
                    num >>= 4;
                }
                for (int i = 0; i < 8; i++) putchar_impl(digits[i]);
            }
            fmt++;
        } else {
            putchar_impl(*fmt);
            fmt++;
        }
    }

    va_end(args);
}
```

PROF

```c
/* test_kernel_utils.c */
#include "kernel_utils.h"
#include <stdio.h>

int main() {
    printf("=== KERNEL UTILITIES TEST ===\n\n");

    // ===== 1. Memory functions =====
    printf("1. MEMORY FUNCTIONS\n");
    uint8_t buffer[20];
    kernel_memset(buffer, 0x42, 10);
    printf("  After memset(buffer, 0x42, 10):\n");
    printf("  buffer[0]=0x%02x (should be 0x42)\n", buffer[0]);
    printf("  buffer[9]=0x%02x (should be 0x42)\n", buffer[9]);
    printf("  buffer[10]=0x%02x (should be 0x00)\n", buffer[10]);
    printf("\n");

    // ===== 2. String functions =====
    printf("2. STRING FUNCTIONS\n");
    char str1[50];
    kernel_strcpy(str1, "Hello, Kernel!");
    printf("  After strcpy: \"%s\"\n", str1);
    printf("  Length: %zu\n", kernel_strlen(str1));
    printf("\n");

    // ===== 3. String comparison =====
    printf("3. STRING COMPARISON\n");
    const char* a = "kernel";
    const char* b = "kernel";
    printf("  strcmp(\"kernel\", \"kernel\") = %d (should be 0)\n",
           kernel_strcmp(a, b));
    printf("\n");

    return 0;
}
```

PROF

H. ESSENTIAL COMPILER KNOWLEDGE

Core Concept

Compiling a kernel is different from compiling a normal program. Understanding the stages is critical.

H1. What -ffreestanding Means

Hosted environment (normal programs):

```
gcc program.c -o program
# Compiler assumes libc exists, syscalls work, malloc is available
```

Freestanding environment (kernels):

```
gcc -ffreestanding program.c -o program
# Compiler does NOT assume libc, malloc, or any library support
```

Concretely:

```
// Without -ffreestanding, compiler might "optimize" this:
// (because it assumes strlen is available)
if (strlen(str) == 0) printf("empty");

// With -ffreestanding, compiler won't use libc strlen
// You must provide it yourself!
```

H2. What a Freestanding Environment Is

Assumptions in freestanding:

- ✅ You have CPU and memory
- ✅ You can read/write memory directly
- ✅ You can execute code
- ✕ No OS system calls
- ✕ No libc functions
- ✕ No stdio, stdlib, threads, etc.
- ✕ No standard library

PROF

What's available:

- stdint.h - Fixed-size integer types
- stddef.h - NULL, size_t, offsetof
- stdarg.h - Variable arguments (for printf)
- stdnoreturn.h - noreturn attribute
- limits.h - INT_MAX, etc.
- stdbool.h - bool, true, false
- stdatomic.h - Atomic operations

H3. Compilation Stages

Kernel compilation is multi-stage:

```

```

```
# Stage 1: Compile C to object files
gcc -ffreestanding -c boot.c -o boot.o
gcc -ffreestanding -c kernel.c -o kernel.o
gcc -ffreestanding -c memory.c -o memory.o

# Stage 2: Assemble assembly files
as -o boot_asm.o boot.s

# Stage 3: Link object files into kernel binary
ld -T linker.ld boot_asm.o boot.o kernel.o memory.o -o kernel.elf

# Stage 4: Convert to binary (optional)
objcopy -O binary kernel.elf kernel.bin
```

H4. How .o Files Combine (Linking)

Concept: Each .o file is incomplete. Linking combines them.

```
/* kernel.c */
int main() {
    set_up_memory();  // Defined in memory.c
    return 0;
}

/* memory.c */
void set_up_memory() {
    // implementation
}
```

Compilation:

PROF

```
gcc -ffreestanding -c kernel.c -o kernel.o   # References set_up_memory
gcc -ffreestanding -c memory.c -o memory.o   # Defines set_up_memory
```

Linking:

```
ld kernel.o memory.o -o kernel.elf
# Linker finds "set_up_memory" defined in memory.o
# Replaces reference in kernel.o with actual address
# Creates complete executable
```

What the linker script does:

```
```

```
/* linker.ld - controls how pieces fit together */
SECTIONS
{
    . = 0x100000;           /* Start at 1 MB in memory */

    .text : {               /* All code here */
        *(.text)
    }

    .data : {               /* All initialized data here */
        *(.data)
    }

    .bss : {                /* All uninitialized data here */
        *(.bss)
    }
}
```

Practice Code: Multi-File Compilation

```c
/* boot.c - Entry point */
#include <stdint.h>

extern void kernel_main();  // Defined in kernel.c

void _start() {
    // Boot code (normally in assembly, but demo in C)
    // Set up stack, clear BSS, then call main

    kernel_main();

    // Infinite loop - kernel should never return
    while (1);
}
```

PROF

```c
/* kernel.c - Main kernel */
#include <stdint.h>

extern void memory_init();  // Defined in memory.c

void kernel_main() {
    // Initialize kernel
    memory_init();

    // Kernel main loop
    while (1) {
```

```
        // Process events
    }
}
```

```
/* memory.c - Memory management */
#include <stdint.h>

static uint8_t kernel_heap[65536];
static uint32_t heap_pos = 0;

void memory_init() {
    heap_pos = 0;
    // Initialize memory structures
}

void* kernel_malloc(uint32_t size) {
    if (heap_pos + size > sizeof(kernel_heap)) {
        return NULL;
    }
    void* ptr = kernel_heap + heap_pos;
    heap_pos += size;
    return ptr;
}
```

```
# Makefile - Controls compilation

CC = gcc
CFLAGS = -ffreestanding -Wall -Wextra -O2

# Compile C to object files
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

# Link object files to kernel
kernel.elf: boot.o kernel.o memory.o
    ld -T linker.ld -o kernel.elf $^

# Build everything
all: kernel.elf

# Clean up
clean:
    rm -f *.o kernel.elf

.PHONY: all clean
```

PROF

Build:

```
make          # Creates kernel.elf
make clean    # Removes built files
```

BONUS: ADVANCED TOPICS

B1. Inline Assembly in C

Concept: Sometimes you need CPU instructions that C can't express.

```
// Disable interrupts (clear IF flag in EFLAGS)
asm volatile("cli");

// Enable interrupts
asm volatile("sti");

// Read CPU register
uint32_t cr0;
asm volatile("mov %%cr0, %0" : "=r" (cr0));

// Write CPU register
uint32_t new_cr0 = 0x80000011;
asm volatile("mov %0, %%cr0" : : "r" (new_cr0));
```

Syntax breakdown:

```
asm volatile(
    "instruction"           // The actual assembly
    : "=r"(output)          // Output: register (r)
    : "r"(input),           // Input: register
      "i"(immediate)        // Input: immediate value
    : "memory"              // Clobber: memory affected
);
```

PROF

B2. Volatile Keyword for Hardware Access

Concept: `volatile` tells compiler "this value can change unexpectedly - don't optimize it away".

```
// WRONG - Compiler might optimize this to nothing!
uint32_t* timer = (uint32_t*)0xFFFF0000;
uint32_t value1 = *timer;
uint32_t value2 = *timer;
// Compiler sees: "Same read twice, second one unnecessary"
```

```
// Optimizes to: read once, use value twice

// CORRECT - volatile tells compiler to actually read twice
volatile uint32_t* timer = (volatile uint32_t*)0xFFFF0000;
uint32_t value1 = *timer;
uint32_t value2 = *timer;
// Compiler actually reads the register twice (values might differ!)
```

Why it matters:

```
// Device register changes with each read
volatile uint32_t* status_reg = (volatile uint32_t*)0x1000;

// Loop until flag is set by device
while (*status_reg == 0) {
    // Compiler MUST check register each iteration
}
```

B3. Memory Alignment

Concept: Some hardware requires data at specific memory boundaries.

```
// Unaligned (bad for some devices)
uint32_t value;  // Might be at 0x1001, 0x1002, 0x1003, or 0x1004

// Aligned (good for hardware)
uint32_t __attribute__((aligned(4))) value;  // ALWAYS at 0x1000,
0x1004, etc.

// Array alignment
uint8_t __attribute__((aligned(16))) buffer[256];  // Start at 16-byte
boundary
```

PROF

In structs:

```
struct Device_Registers {
    uint32_t control;
    uint32_t status;
    uint32_t data;
} __attribute__((aligned(16), packed));
```

Practice Code: Advanced Topics

```c
/* advanced_practice.c */

#include <stdint.h>
#include <stdio.h>

// Simulate device register
volatile static uint32_t device_register = 0;

void demonstrate_volatile() {
    printf("=== VOLATILE DEMONSTRATION ===\n");

    // This WILL read twice (because of volatile)
    volatile uint32_t* reg = (volatile uint32_t*)&device_register;

    printf("Reading volatile register...\n");
    printf("  First read: 0x%x\n", *reg);
    printf("  Second read: 0x%x\n", *reg);
    printf("  (Reads can differ even without modification)\n\n");
}

void demonstrate_alignment() {
    printf("=== MEMORY ALIGNMENT ===\n");

    struct Unaligned {
        uint8_t a;
        uint32_t b;
        uint8_t c;
    };

    struct Aligned {
        uint8_t a;
        uint32_t b;
        uint8_t c;
    } __attribute__((aligned(16)));

    printf("Unaligned struct: %zu bytes\n", sizeof(struct Unaligned));
    printf("Aligned struct: %zu bytes\n", sizeof(struct Aligned));
    printf("\n");
}

void demonstrate_bit_operations() {
    printf("=== BIT OPERATIONS (USEFUL FOR FLAGS) ===\n");

    uint32_t flags = 0;

    // Set bit 0
    flags |= (1 << 0);
    printf("After setting bit 0: 0x%x\n", flags);

    // Set bit 5
    flags |= (1 << 5);
    printf("After setting bit 5: 0x%x\n", flags);
```

```c
    // Check if bit 0 is set
    if (flags & (1 << 0)) {
        printf("Bit 0 is SET\n");
    }

    // Clear bit 0
    flags &= ~(1 << 0);
    printf("After clearing bit 0: 0x%x\n", flags);

    printf("\n");
}

int main() {
    demonstrate_volatile();
    demonstrate_alignment();
    demonstrate_bit_operations();
    return 0;
}
```

COMPLETE PRACTICE PROJECT

Building a Minimal Bootloader + Kernel

This comprehensive project ties everything together.

Project Structure

```
minimal-kernel/
├── boot.s              # Bootloader in assembly
├── kernel.c            # Main kernel entry
├── memory.c            # Memory management
├── io.c                # Input/output routines
├── io.h                # I/O declarations
├── kernel.h            # Kernel declarations
├── linker.ld           # Linker script
├── Makefile            # Build system
└── README.md           # Documentation
```

PROF

boot.s - Assembly Entry Point

```
# boot.s - Minimal bootloader entry

.global _start
.align 4
```

```
_start:
    # Set up stack
    mov $0x90000, %esp

    # Clear BSS section
    xor %eax, %eax
    lea __bss_start, %edi
    lea __bss_end, %ecx
    sub %edi, %ecx
    cld
    rep stosb

    # Call kernel main
    call kernel_main

    # Infinite loop
    cli
    hlt
    jmp .
```

linker.ld - Memory Layout

```
ENTRY(_start)

SECTIONS
{
    . = 0x100000;

    .text : { *(.text*) }
    .rodata : { *(.rodata*) }

    . = ALIGN(4096);
    .data : { *(.data*) }

    . = ALIGN(4096);
    __bss_start = .;
    .bss : { *(.bss*) }
    __bss_end = .;
}
```

PROF

io.h - I/O Interface

```
#ifndef IO_H
#define IO_H

#include <stdint.h>
#include <stddef.h>
```

```
void io_putchar(char c);
void io_puts(const char* str);
void io_printf(const char* fmt, ...);

#endif
```

io.c - I/O Implementation

```
#include "io.h"
#include <stdarg.h>

#define VGA_BUFFER 0xB8000
#define VGA_COLS 80
#define VGA_ROWS 25
#define COLOR 0x0F

static uint16_t* vga = (uint16_t*)VGA_BUFFER;
static uint16_t cursor_pos = 0;

void io_putchar(char c) {
    if (c == '\n') {
        cursor_pos += VGA_COLS - (cursor_pos % VGA_COLS);
    } else if (cursor_pos < VGA_ROWS * VGA_COLS) {
        vga[cursor_pos] = (COLOR << 8) | c;
        cursor_pos++;
    }

    // Wrap around
    if (cursor_pos >= VGA_ROWS * VGA_COLS) {
        cursor_pos = 0;
    }
}

void io_puts(const char* str) {
    while (*str) {
        io_putchar(*str++);
    }
}

void io_printf(const char* fmt, ...) {
    va_list args;
    va_start(args, fmt);

    while (*fmt) {
        if (*fmt == '%' && *(fmt + 1)) {
            fmt++;
            if (*fmt == 'd') {
                int num = va_arg(args, int);
                char buffer[20];
```

```
                int len = 0;
                if (num == 0) {
                    buffer[len++] = '0';
                } else {
                    if (num < 0) {
                        io_putchar('-');
                        num = -num;
                    }
                    while (num) {
                        buffer[len++] = '0' + (num % 10);
                        num /= 10;
                    }
                    while (len--) io_putchar(buffer[len]);
                    fmt++;
                    continue;
                }
                for (int i = 0; i < len; i++) io_putchar(buffer[i]);
            } else if (*fmt == 's') {
                const char* str = va_arg(args, const char*);
                io_puts(str);
            }
            fmt++;
        } else {
            io_putchar(*fmt++);
        }
    }

    va_end(args);
}
```

kernel.h - Kernel Interface

```
#ifndef KERNEL_H
#define KERNEL_H

#include <stdint.h>

void kernel_main();
void memory_init();

#endif
```

kernel.c - Main Kernel

```
#include "kernel.h"
#include "io.h"

void kernel_main() {
```

```
    // Clear screen
    io_puts("=== Minimal Kernel ===\n");
    io_printf("Kernel loaded at 0x100000\n");

    // Initialize subsystems
    io_puts("Initializing memory...\n");
    memory_init();

    io_puts("Kernel ready!\n");

    // Kernel main loop
    while (1) {
        // Do kernel work
    }
}
```

memory.c - Memory Management

```
#include "kernel.h"
#include "io.h"

#define KERNEL_HEAP_SIZE 65536
static uint8_t kernel_heap[KERNEL_HEAP_SIZE];
static uint32_t heap_pos = 0;

void memory_init() {
    io_printf("  Heap at 0x%x, size 0x%x\n",
              (uint32_t)kernel_heap, KERNEL_HEAP_SIZE);
    heap_pos = 0;
}

void* kernel_malloc(uint32_t size) {
    if (heap_pos + size > KERNEL_HEAP_SIZE) {
        io_puts("ERROR: Out of memory\n");
        return NULL;
    }

    void* ptr = kernel_heap + heap_pos;
    heap_pos += size;
    return ptr;
}
```

PROF

Makefile - Build System

```
CC = i686-elf-gcc
AS = i686-elf-as
LD = i686-elf-ld
OBJCOPY = i686-elf-objcopy
```

```
CFLAGS = -ffreestanding -Wall -Wextra -O2
ASFLAGS = --32

# Target
kernel.elf: boot.o kernel.o memory.o io.o
    $(LD) -T linker.ld -o kernel.elf $^

kernel.bin: kernel.elf
    $(OBJCOPY) -O binary kernel.elf kernel.bin

# Compilation rules
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

%.o: %.s
    $(AS) $(ASFLAGS) $< -o $@

# Build everything
all: kernel.bin

# Clean
clean:
    rm -f *.o *.elf *.bin

# Disassemble (for debugging)
disasm: kernel.elf
    i686-elf-objdump -d kernel.elf

.PHONY: all clean disasm
```

Build Instructions

PROF

```
# For x86_64 systems building 32-bit kernel:
# You need: i686-elf-gcc, i686-elf-binutils

# Build
make clean
make kernel.bin

# View structure
file kernel.elf
objdump -h kernel.elf
nm kernel.elf

# Disassemble
make disasm
```

Summary Table: Quick Reference

Topic	Key Concept	Kernel Usage
Pointers	Address, dereference, arithmetic	Memory-mapped I/O, hardware registers
Fixed-size types	uint32_t, uint64_t	Precise register/memory layout
Structs	Group data, packed, typedefs	CPU state, page tables, device registers
Storage classes	extern, static, scope	Module organization, private data
Headers	.h files, guards, compilation	Code organization, Makefiles
Memory layout	.text, .data, .bss, stack	Bootloader setup, linker script
C restrictions	No libc, no malloc, freestanding	Only use what exists in kernel
Compiler	-ffreestanding, linking, stages	Multi-file kernel builds
Advanced	inline asm, volatile, alignment	CPU instructions, hardware access

Exam Tips & Tricks

Common Exam Questions

*1. What's the difference between char a and char a?**

- `char a` = single character
- `char* a` = pointer to character(s)

2. What does `volatile` do?

- Tells compiler to not optimize away reads/writes
- Critical for hardware registers that change unexpectedly

PROF

3. Why do kernels use uint32_t instead of int?

- int size varies (16/32/64 bits depending on arch)
- uint32_t is always 32 bits - predictable

4. What's in .bss but not in binary?

- Uninitialized globals don't need bytes in binary
- Linker just allocates space, no actual data stored

5. What does -ffreestanding do?

- Tells compiler to not assume libc exists
- Don't use standard library functions

Quick Concept Recap

```c
// Pointers
int x;
int* ptr = &x;     // &x = address of x
*ptr = 100;        // *ptr = value at address

// Structs
struct Point { int x, y; } __attribute__((packed));

// Static
static int private;  // File-private at global scope
static int count;    // Persistent at function scope

// Memory layout
// .text = code, .data = initialized globals, .bss = zeros
// stack = locals, heap = malloc'd

// Fixed sizes
uint32_t = always 32 bits (not int which varies)
```

END OF GUIDE

This comprehensive guide covers all core concepts needed for OS/kernel development. Start with pointers and data types, master structs and storage classes, understand compilation and memory layout, then move to restrictions and compiler knowledge. The practice project ties it all together!

Next Steps:

1. Compile and run all practice code
2. Modify examples to understand deeply
3. Build the minimal kernel project
4. Study past year papers with these concepts
5. Write your own utilities (printf, malloc, etc.)

PROF

Good luck with your exam prep!