

Tugas Besar

IF2230 - Sistem Operasi

Milestone 2

"Ho/OS"

**Pembuatan Sistem Operasi x86
Interrupt, Driver, dan Filesystem**

Dipersiapkan oleh
Asisten Lab Sistem Terdistribusi



Guidebook IF2230 - Milestone 2

Dokumen ini akan digunakan sebagai panduan pengerjaan dan referensi tugas besar IF2230 - 2023. Untuk melakukan navigasi dengan cepat, gunakan hyperlink yang ada pada [Table of Contents](#).

Semua bagian pada milestone 2 terdapat pada guidebook ini, bagian yang menggunakan simbol titik (Contohnya • **Interrupt & Exception**) di header menandai **bagian tersebut hanya penjelasan bonus** dan beberapa **fun fact** pada sistem operasi sehingga dapat dilompati jika dirasa tidak perlu.

Minor sidenote: Guidebook ini didesain agar menuntun pengerjaan tugas besar dan memberikan pengetahuan tambahan yang mungkin tidak berhubungan dengan kelas secara langsung. **Jika hanya ingin berfokus kepada pengerjaan tugas besar, gunakan hyperlink daftar isi atau document outline (yang ada dibagian kiri Docs) untuk lompat langsung ke pengerjaan.**



Spesifikasi IF2230 - Milestone 2	:  IF2230 - Milestone 2
Repository kit milestone 2	: Sister20/kit-OS-milestone-2-2023
QnA & FAQ	:  IF2230 - FAQ + QnA Tubes
Form asistensi	: Form Asistensi Tugas Besar IF2230 - Sistem Operasi

Table of Contents

Guidebook IF2230 - Milestone 2.....	2
Table of Contents.....	3
3.1. Interrupt.....	4
• Interrupt & Exception.....	4
• Hardware / Software Interrupt & System Calls.....	5
3.1.1 Interrupt Descriptor Table (IDT).....	6
• PIC & IRQ.....	7
3.1.2. PIC Remapping.....	8
3.1.3. Interrupt Handler / Interrupt Service Routine (ISR).....	10
3.1.4. Load IDT & Testing Interrupt.....	14
• Notable CPU Exception.....	16
■ Double Fault.....	16
■ Triple Fault.....	17
■ General Protection Fault.....	17
■ Breakpoint.....	17
3.2. Keyboard Device Driver.....	18
• Device Driver.....	18
• Desain Driver.....	18
3.2.1. IRQ1 - Keyboard Controller.....	19
3.2.2. Keyboard Driver Interfaces.....	20
3.2.3. Keyboard ISR.....	21
• Keyboard Scancode.....	23
3.3. File System - FAT32 IF2230 edition.....	24
• File System.....	24
• ATA, PATA, & SATA.....	24
• Disk Addressing: LBA & CHS.....	25
• Boot Sector, Bootstrap, MBR, & GPT.....	26
• Block & Cluster.....	26
3.3.1. Disk Driver & Image.....	28
3.3.2. File System: FAT32 - IF2230 edition.....	30
• FAT32 File Size Limit.....	34
3.3.3. File System Initializer.....	35
3.3.4. File System CRUD Operation.....	36
■ Read dan Read Directory.....	36
■ Write.....	36
■ Delete.....	37
• Undelete & Disk Recovery.....	39
• Defragmentation.....	40
• OS & Information Security.....	41
Referensi Tambahan.....	42

3.1. Interrupt

Interrupt merupakan salah satu teknik pada arsitektur komputer modern untuk mendukung multitasking. Hardware Interrupt juga berguna untuk menyediakan CPU sebuah cara untuk menerima komunikasi dengan hardware eksternal.

BIOS Interrupt Calls menyediakan fungsi-fungsi dasar yang dapat diakses menggunakan interrupt. Fungsi-fungsi tersebut menyediakan layanan seperti *basic keyboard I/O*, *disk I/O*, dan lain-lain.

Namun, pada x86 protected mode fitur BIOS interrupt calls tidak dapat diakses. Sehingga fitur-fitur dasar seperti *keyboard I/O* dan *disk I/O* perlu dibuat sendiri. Kedua I/O tersebut akan dibahas pada bagian selanjutnya yang akan membuat *driver* sederhana.

Bagian **3.1. Interrupt** akan fokus kepada implementasi **idt.h** dan **interrupt.h**. Template **interrupt** tersedia pada **kit/interrupt/**

• Interrupt & Exception

Jika pernah memprogram sesuatu, pastinya pernah mengalami exception yang disebabkan invalid memory access seperti program C yang mengalami **Segmentation Fault** pada Linux atau **Null Pointer Exception** pada Java.

Kedua contoh merupakan **Exception**. Exception umumnya didefinisikan sebagai **Interrupt** yang dibuat oleh CPU karena terjadi error, sehingga **Exception dapat disebut subset dari Interrupt**.

Interrupt adalah sebuah sinyal yang diterima CPU menandakan ada suatu event sehingga CPU akan berhenti dan menjalankan event handler sebelum kembali menjalankan apapun yang dikerjakan sebelumnya. Interrupt dapat berasal dari device external maupun dari CPU sendiri, **CPU Exception** adalah contohnya. **Intel x86 merupakan Interrupt-driven Architecture**.

Bagian pertama dari pengerjaan milestone ini adalah penyiapan **Interrupt Descriptor Table (IDT)**. IDT bersama dengan GDT, merupakan struktur data yang akan digunakan CPU untuk keperluan lookup interrupt handler dan proteksi memory. Struktur data **GDT & IDT harus diimplementasikan** pada sistem operasi yang menggunakan **x86 protected mode**.

Jika pernah memprogram event handler pada bahasa high-level seperti JavaScript, ketika membuat **Interrupt Handler** atau **Interrupt Service Routine (ISR)** akan terasa familiar. ISR merupakan “**event handler**” untuk interrupt. Sebuah Interrupt akan memiliki **Vector** (angka interrupt contohnya “**INT \$0x21**” berarti vektor 0x21), sebuah angka yang dapat dianggap sebagai nama dari “**Event**”. Angka ini nantinya juga akan digunakan oleh CPU untuk membaca dan mencari address ISR yang sesuai pada tabel IDT.

Sedikit catatan: Terkadang dalam konteks interrupt, hexadecimal menggunakan **notasi suffix h** seperti **INT 10h**, **INT 21h**, dan seterusnya. Notasi tersebut sama persis dengan **notasi prefix 0x** yang biasanya digunakan, yaitu menunjukkan bahwa integer literal dalam radix hexadecimal.

Dari pengerjaan milestone sebelumnya, sebenarnya inisialisasi sistem operasi belum selesai, karena IDT masih belum di-load menggunakan instruksi **LIDT** (Mirip dengan **LGDT**)

Nantinya pada pembuatan IDT, semua interrupt akan dimatikan terlebih dahulu dengan instruksi **CLI** (clear interrupt flag, IF pada register **\$eflags**). IDT yang terdefinisi akan di-load dengan **LIDT**. Akhirnya interrupt dinyalakan dengan instruksi **STI** (set interrupt flag).

• Hardware / Software Interrupt & System Calls

Semua Interrupt yang akan dibahas pada milestone ini merupakan **Hardware Interrupt** yang berasal dari CPU atau device external. Berbeda dengan Hardware Interrupt, **Software Interrupt** berasal dari software yaitu instruksi **INT**. Karena Interrupt dapat menyebabkan perubahan CPU **\$eflags** terutama **IOPL** yang merupakan **privilege level**, Software Interrupt biasanya digunakan sebagai alat pemanggilan **System Calls**.

System Calls merupakan layanan atau interface yang disediakan oleh sistem operasi untuk melakukan operasi-operasi yang hanya diperbolehkan untuk **Ring 0**. Sistem operasi yang menyiapkan Interrupt Handler dapat melakukan privilege, error, attribute checking, dan pemrosesan lainnya; sebelum kembali ke user program. **Kernel** atau **heart** dari sistem operasi terhubung dengan user program menggunakan “**blood vessel**” sistem operasi: **System Calls**.

IDT yang masih kosong dapat digunakan sebagai Interrupt Handler System Calls, umumnya pada UNIX semua System Calls dapat terletak pada **INT 0x80** dan menggunakan **CPU Register** (sama seperti **CPURegister** yang nantinya ada pada **main_interrupt_handler()**) sebagai parameternya.

Small spoiler alert: milestone selanjutnya user mode, if you know what i mean ;)

Untuk penjelasan lebih detail mengenai Interrupt & Exception terdapat pada:
Intel x86 Software Developer Manual Vol 3a - Chapter 6 - Interrupt & Exception Handling

3.1.1 Interrupt Descriptor Table (IDT)

Sebelum memulai pengerjaan pembuatan Interrupt Descriptor Table, perlu diketahui bahwa IDT dan GDT tidak terlalu berbeda. Sehingga bagian ini akan cenderung **hands-on**.

Dokumentasi lengkap terdapat pada **Intel x86 Software Developer Manual Vol 3a**:

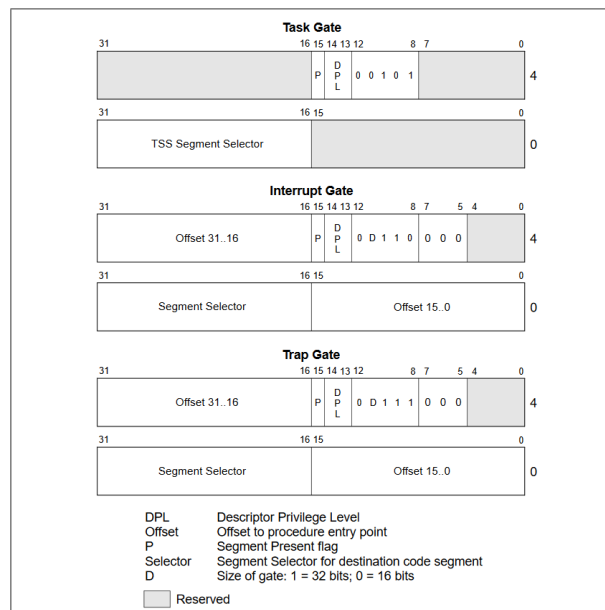
- **Chapter 6.10 - Interrupt Descriptor Table**
- **Chapter 6.11 - IDT Descriptors**

Tugas utama dari bagian ini adalah membuat struktur data yang ada pada IDT, tanda kurung menyatakan struktur data yang mirip pada GDT:

- **IDTGate (SegmentDescriptor)**
- **InterruptDescriptorTable (GlobalDescriptorTable)**
- **IDTR (GDTR)**

Template awal untuk pengerjaan IDT tersedia pada **kit/src/interrupt/idt.h**

Berikut adalah ilustrasi struktur data **IDTGate** yang diambil dari Intel x86 Manual



Intel x86 Manual 3A - Figure 6-2 IDT Gate Descriptors

Milestone ini akan berfokus dengan penggunaan **IDT Gate** bertipe **Interrupt Gate**. **IDTGate** merupakan entry dari **InterruptDescriptorTable**. Sama seperti **GlobalDescriptorTable**, **InterruptDescriptorTable** menyimpan **256** (fixed value dari arsitektur x86) IDT Gate. **IDTR** juga sama dengan **GDTR**.

Setelah membuat deklarasi struktur data IDT, definisikan juga seperti GDT, IDT kosong **interrupt_descriptor_table** dan IDTR yang telah terisi **_idt_idtr** pada **idt.c**.

Nantinya kedua variabel yang didefinisikan akan digunakan untuk menyimpan informasi interrupt handler sistem operasi.

- **PIC & IRQ**

Programmable Interrupt Controller (PIC) merupakan controller chip yang tergolong eksternal pada CPU. PIC bertugas untuk handle dan mengatur semua **Hardware Interrupt** dari device dan mengirimkannya kepada CPU secara **Queue**. **Hardware Interrupt** juga disebut **Interrupt Request** atau **IRQ** dalam konteks sistem operasi.

PIC yang digunakan pada tugas besar ini **8259A PIC**, terhubung ke beberapa device dan chip yang penting seperti **Programmable Interval Timer (IRQ0)**, **Keyboard Controller (IRQ1)**, **CMOS real-time clock (IRQ8)**, dan hardware lainnya. **8259A PIC** merupakan PIC yang sangat sederhana dan didesain waktu **original IBM PC**. **Bagian 3.1.2. PIC Remapping** disebabkan oleh design dari IBM PC yang menyebabkan **x86 CPU exception** yang menggunakan Interrupt 0x00 hingga 0x1F (Dapat dicek pada [Wikipedia/IDT](https://en.wikipedia.org/wiki/Interrupt_descriptor_table)) bertabrakan dengan interrupt yang dibuat oleh **8259A PIC** (contoh konkritnya adalah Keyboard IRQ1, akan menyebabkan **INT \$0x1**).

Selain 8259A PIC, Intel memiliki PIC yang lebih modern bernama **Advanced Programmable Interrupt Controller (APIC)** yang menyediakan fitur-fitur kompleks dan dapat digunakan untuk kebutuhan **Multiprocessor & Multithreading**.

Detail **APIC** terdapat pada **Intel x86 Manual Vol 3A - Chapter 10 - APIC**.

3.1.2. PIC Remapping

Bagian ini didasarkan dari penjelasan bagian opsional sebelumnya: **PIC dan IRQ**. **Bagian 3.1.2. PIC remapping** akan berfokus kepada pembuatan fungsi untuk menggeser interrupt yang dibuat oleh **PIC 8259A** ke interrupt **0x20** dan **0x28**.

Beberapa deklarasi untuk PIC remapping sudah tersedia pada **kit/interrupt/interrupt.h**. Berikut adalah implementasi dari **PIC remapping**

interrupt.c

```
void io_wait(void) {
    out(0x80, 0);
}

void pic_ack(uint8_t irq) {
    if (irq >= 8)
        out(PIC2_COMMAND, PIC_ACK);
    out(PIC1_COMMAND, PIC_ACK);
}

void pic_remap(void) {
    uint8_t a1, a2;

    // Save masks
    a1 = in(PIC1_DATA);
    a2 = in(PIC2_DATA);

    // Starts the initialization sequence in cascade mode
    out(PIC1_COMMAND, ICW1_INIT | ICW1_ICW4);
    io_wait();
    out(PIC2_COMMAND, ICW1_INIT | ICW1_ICW4);
    io_wait();
    out(PIC1_DATA, PIC1_OFFSET); // ICW2: Master PIC vector offset
    io_wait();
    out(PIC2_DATA, PIC2_OFFSET); // ICW2: Slave PIC vector offset
    io_wait();
    out(PIC1_DATA, 0b0100);      // ICW3: tell Master PIC that there is a slave PIC at
    IRQ2 (0000 0100)
    io_wait();
    out(PIC2_DATA, 0b0010);      // ICW3: tell Slave PIC its cascade identity (0000
    0010)
    io_wait();

    out(PIC1_DATA, ICW4_8086);
    io_wait();
    out(PIC2_DATA, ICW4_8086);
    io_wait();

    // Restore masks
    out(PIC1_DATA, a1);
    out(PIC2_DATA, a2);
}
```


Fungsi **pic_remap()** yang diberikan menggunakan macro yang telah didefinisikan pada **interrupt.h**. Jika tidak mengubah macro apapun, mestinya fungsi tersebut akan menggeser interrupt **PIC master** ke **0x20** dan **PIC slave** ke **0x28**.

Dalam kata lain, semua interrupt yang dibuat oleh **8259A PIC** akan dimulai dari **0x20** dan **0x28** (IRQ0 yaitu timer akan melakukan interrupt **0x20 + 0 = INT 20h**; IRQ1 keyboard akan melakukan interrupt **0x20 + 1 = INT 21h**; dan seterusnya)

Pemrograman PIC membutuhkan wait untuk menunggu PIC menyelesaikan command yang diberikan. **io_wait()** yang menunggu selama 1-4 microsecond umumnya cukup untuk kebutuhan tersebut. Sedangkan **pic_ack()** digunakan untuk mengirim pesan ACK ke PIC, kenapa hal ini diperlukan? Jika interrupt yang dikirim ke CPU tidak di-ACK oleh CPU, PIC akan mengasumsikan CPU sedang menjalankan interrupt handler sehingga tidak akan mengirimkan interrupt selanjutnya.

Catatan penting : Semua IRQ yang di-generate 8259A PIC harus di-ACK. Jika tidak, seluruh IRQ lain akan tidak dikirim. Contohnya, IRQ Timer & IRQ Keyboard menyala, IRQ Timer masuk tetapi tidak di-ACK, IRQ Keyboard akan dipending juga hingga ACK untuk timer sebelumnya dikirim.

3.1.3. Interrupt Handler / Interrupt Service Routine (ISR)

Interrupt Handler atau **Interrupt Service Routine (ISR)** bertugas untuk memproses interrupt yang diterima CPU. Sesuai yang disebutkan pada spesifikasi milestone 2, diperlukan fungsi **call_generic_handler()** dan **interrupt_handler_i()** yang diimplementasikan pada assembly.

Berikut adalah kode assembly untuk **intsetup.s**

intsetup.s

```
extern main_interrupt_handler
global isr_stub_table

; Generic handler section for interrupt
call_generic_handler:
    ; Before interrupt_handler_n is called (caller of this generic handler section),
    ; stack will have these value that pushed automatically by CPU
    ; [esp + 12] eflags
    ; [esp + 8 ] cs
    ; [esp + 4 ] eip
    ; [esp + 0 ] error code

    ; CPURegister
    push    esp
    push    ebp
    push    edx
    push    ecx
    push    ebx
    push    eax

    ; call the C function
    call    main_interrupt_handler

    ; restore the registers
    pop     eax
    pop     ebx
    pop     ecx
    pop     edx
    pop     ebp
    pop     esp

    ; restore the esp (interrupt number & error code)
    add     esp, 8

    ; return to the code that got interrupted
    ; at this point, stack should be structured like this
    ; [esp], [esp+4], [esp+8]
    ;   eip,   cs,   eflags
    ; improper value will cause invalid return address & register
    sti
    iret
```

```

; Macro for creating interrupt handler that only push interrupt number
%macro no_error_code_interrupt_handler 1
interrupt_handler_%1:
    push    dword 0           ; push 0 as error code
    push    dword %1         ; push the interrupt number
    jmp     call_generic_handler ; jump to the common handler
%endmacro

%macro error_code_interrupt_handler 1
interrupt_handler_%1:
    push    dword %1
    jmp     call_generic_handler
%endmacro

; CPU exception handlers
no_error_code_interrupt_handler 0 ; 0x0 - Division by zero
no_error_code_interrupt_handler 1 ; 0x1 - Debug Exception
no_error_code_interrupt_handler 2 ; 0x2 - NMI, Non-Maskable Interrupt
no_error_code_interrupt_handler 3 ; 0x3 - Breakpoint Exception
no_error_code_interrupt_handler 4 ; 0x4 - INTO Overflow
no_error_code_interrupt_handler 5 ; 0x5 - Out of Bounds
no_error_code_interrupt_handler 6 ; 0x6 - Invalid Opcode
no_error_code_interrupt_handler 7 ; 0x7 - Device Not Available
error_code_interrupt_handler 8 ; 0x8 - Double Fault
no_error_code_interrupt_handler 9 ; 0x9 - Deprecated
error_code_interrupt_handler 10 ; 0xA - Invalid TSS
error_code_interrupt_handler 11 ; 0xB - Segment Not Present
error_code_interrupt_handler 12 ; 0xC - Stack-Segment Fault
error_code_interrupt_handler 13 ; 0xD - General Protection Fault
error_code_interrupt_handler 14 ; 0xE - Page Fault
no_error_code_interrupt_handler 15 ; 0xF - Reserved
no_error_code_interrupt_handler 16 ; 0x10 - x87 Floating-Point Exception
error_code_interrupt_handler 17 ; 0x11 - Alignment Check Exception
no_error_code_interrupt_handler 18 ; 0x12 - Machine Check Exception
no_error_code_interrupt_handler 19 ; 0x13 - SIMD Floating-Point Exception
no_error_code_interrupt_handler 20 ; 0x14 - Virtualization Exception
no_error_code_interrupt_handler 21 ; 0x15 - Control Protection Exception
no_error_code_interrupt_handler 22 ; 0x16 - Reserved
no_error_code_interrupt_handler 23 ; 0x17 - Reserved
no_error_code_interrupt_handler 24 ; 0x18 - Reserved
no_error_code_interrupt_handler 25 ; 0x19 - Reserved
no_error_code_interrupt_handler 26 ; 0x1A - Reserved
no_error_code_interrupt_handler 27 ; 0x1B - Reserved
no_error_code_interrupt_handler 28 ; 0x1C - Hypervisor Injection Exception
no_error_code_interrupt_handler 29 ; 0x1D - VMM Communication Exception
error_code_interrupt_handler 30 ; 0x1E - Security Exception
no_error_code_interrupt_handler 31 ; 0x1F - Reserved

; User defined interrupt handler
; Assuming PIC1 & PIC2 offset is 0x20 and 0x28
; 32 - 0x20 - IRQ0: Programmable Interval Timer
; 33 - 0x21 - IRQ1: Keyboard

```

```

; 34 - 0x22 - IRQ2: PIC Cascade, used internally
; 35 - 0x23 - IRQ3: COM2, if enabled
; 36 - 0x24 - IRQ4: COM1, if enabled
; 37 - 0x25 - IRQ5: LPT2, if enabled
; 38 - 0x26 - IRQ6: Floppy Disk
; 39 - 0x27 - IRQ7: LPT1

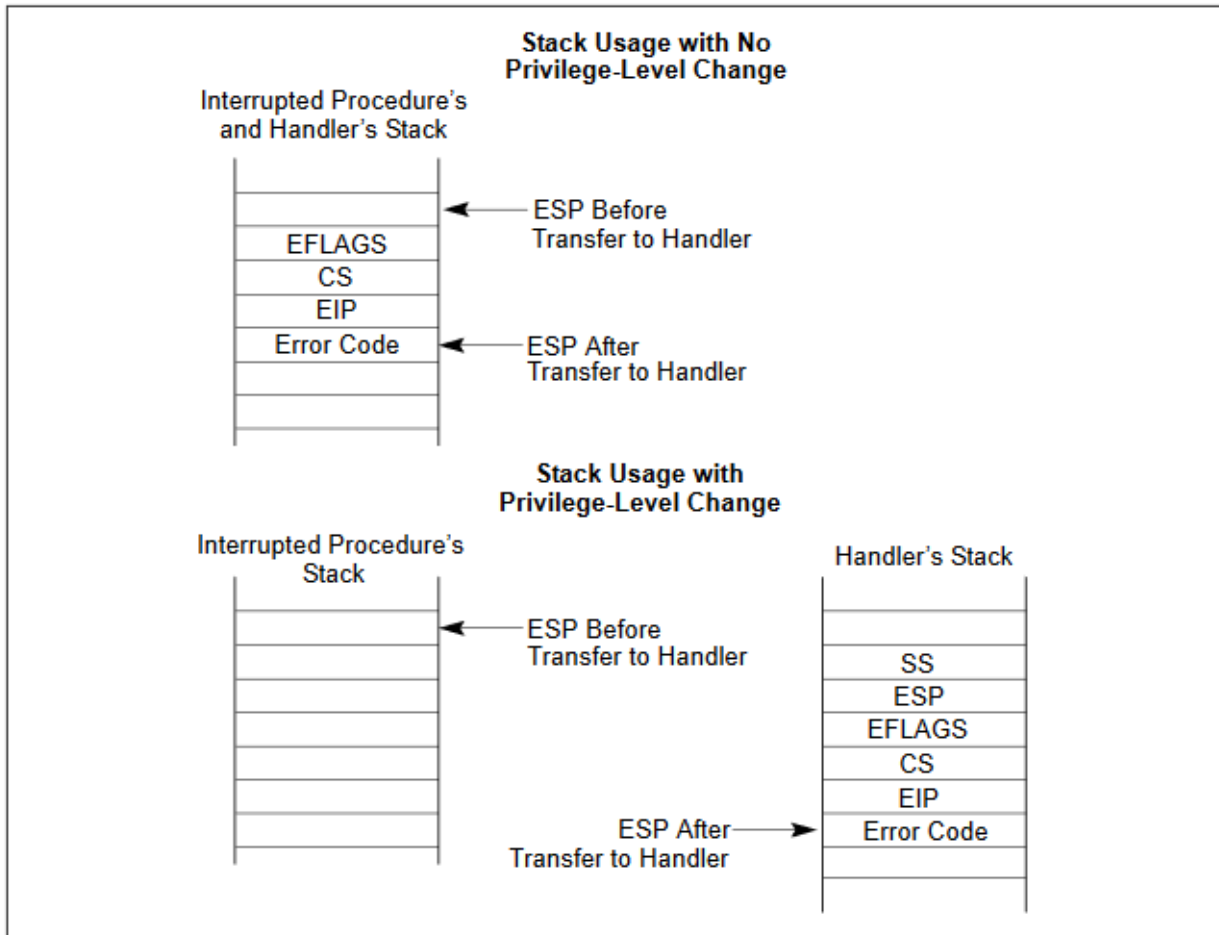
; 40 - 0x28 - IRQ8: CMOS real-time clock
; 41 - 0x29 - IRQ9: Free
; 42 - 0x2A - IRQ10: Free
; 43 - 0x2B - IRQ11: Free
; 44 - 0x2C - IRQ12: PS2 Mouse
; 45 - 0x2D - IRQ13: Coprocessor
; 46 - 0x2E - IRQ14: Primary ATA Hard Disk
; 47 - 0x2F - IRQ15: Secondary ATA Hard Disk
%assign i 32
%rep 32
no_error_code_interrupt_handler i
%assign i i+1
%endrep

; ISR stub table, useful for reducing code repetition
isr_stub_table:
    %assign i 0
    %rep 64
        dd interrupt_handler_%i
    %assign i i+1
    %endrep

```

Kode ini juga akan tersedia pada **kit/interrupt/intsetup.s**

Singkat cerita, **call_generic_handler()** dan **interrupt_handler_i()** bertugas menyiapkan call stack fungsi C **main_interrupt_handler()**. Sedangkan untuk array **isr_stub_table** yang didefinisikan pada assembly akan digunakan pada bagian selanjutnya. Berikut adalah kondisi stack ketika interrupt dipanggil



Intel x86 Manual 3A - **Figure 6-4 Stack Usage on Transfer to Interrupt Routines**

Karena struktur data sudah terlampir pada **interrupt.h**, **main_interrupt_handler()** dapat diimplementasikan pada bahasa C. Tambahkan kode berikut pada **interrupt.c**

interrupt.c

```
void main_interrupt_handler(
    __attribute__((unused)) struct CPURegister cpu,
    uint32_t int_number,
    __attribute__((unused)) struct InterruptStack info
) {
    switch (int_number) {
    }
}
```

3.1.4. Load IDT & Testing Interrupt

Sebelum melakukan load IDT, IDT yang masih kosong perlu disetup dengan **interrupt_handler_i()** yang telah dibuat oleh macro assembly. Macro yang digunakan membentuk **interrupt_handler_i()** akan memasukkan **Interrupt Vector** ke parameter **main_interrupt_handler()**. CPU ketika memanggil ISR tidak menaruh angka interrupt ke stack, sehingga IV perlu di-push secara manual.

Array function pointer **isr_stub_table** hanya menyimpan semua address dari **interrupt_handler_i()**. Hal ini akan mempermudah **initialize_idt()** yang diimplementasikan pada bahasa C. Semua address ISR dapat langsung di-iterasi dari **isr_stub_table** dan dimasukkan ke **IDT.table[i]**.

Lengkapi kode berikut sesuai deskripsi fungsi yang ada pada header pada **idt.c**

idt.c

```
void initialize_idt(void) {
    /* TODO :
    * Iterate all isr_stub_table,
    * Set all IDT entry with set_interrupt_gate()
    * with following values:
    * Vector: i
    * Handler Address: isr_stub_table[i]
    * Segment: GDT_KERNEL_CODE_SEGMENT_SELECTOR
    * Privilege: 0
    */
    __asm__ volatile("lidt %0" : : "m"(_idt_idtr));
    __asm__ volatile("sti");
}

void set_interrupt_gate(uint8_t int_vector, void *handler_address, uint16_t
gdt_seg_selector, uint8_t privilege) {
    struct IDTGate *idt_int_gate = &interrupt_descriptor_table.table[int_vector];
    // TODO : Set handler offset, privilege & segment

    // Target system 32-bit and flag this as valid interrupt gate
    idt_int_gate->r_bit_1 = INTERRUPT_GATE_R_BIT_1;
    idt_int_gate->r_bit_2 = INTERRUPT_GATE_R_BIT_2;
    idt_int_gate->r_bit_3 = INTERRUPT_GATE_R_BIT_3;
    idt_int_gate->gate_32 = 1;
    idt_int_gate->valid_bit = 1;
}
```

Sesuai dengan spesifikasi milestone 2, jika semua struktur data IDT sudah di-setup dengan baik, IDT dapat dites dengan kode berikut pada kernel

kernel.c

```
void kernel_setup(void) {
```

```
enter_protected_mode(&_gdt_gdtr);
pic_remap();
initialize_idt();
framebuffer_clear();
framebuffer_set_cursor(0, 0);
__asm__("int $0x4");
while (TRUE);
}
```

Inline assembly diatas akan memanggil Interrupt 4h atau 0x4. Karena **main_interrupt_handler()** masih kosong selain **switch**, interrupt tersebut mestinya hanya lewat dan return kembali ke fungsi **kernel_setup()**.

Jika masih terdapat error, coba cari penyebabnya menggunakan **Breakpoint** pada **main_interrupt_handler()**. Detail dari error code CPU exception ada pada [OSDev/Exceptions](#).

PIC, by default menyalakan beberapa IRQ dan diketahui bahwa PIT atau timer termasuk IRQ yang aktif. Jika PIC belum diremap, mestinya akan terjadi interrupt 0 hingga 8 yang disebabkan oleh timer. Perhatikan juga jika menduga penyebabnya adalah PIC, semestinya interrupt hanya terjadi sekali karena **pic_ack()** tidak dikirim.

Sekali lagi, karena bagian **3.1. Interrupt** merupakan bagian vital untuk selanjutnya (**IRQ1 - Keyboard Controller**), pastikan bagian ini sudah berjalan dengan baik.

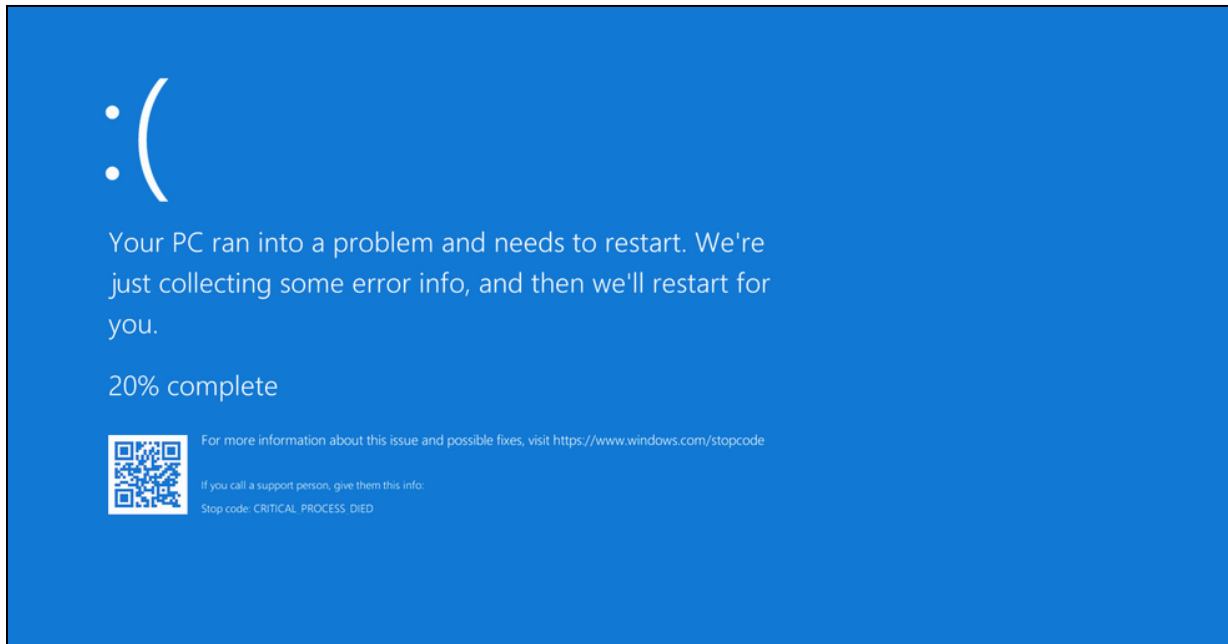
• Notable CPU Exception

Beberapa CPU exception yang ada pada list [OSDev/Exception](#) terdengar familiar. Bahkan jika pernah memainkan **Exception Handling** construct pada bahasa C++, Python, Java, dll, salah satu cara yang paling mudah mengimplementasikan-nya (ketika kita membuat interpreter atau compiler bahasa tersebut misalnya) adalah “mendaftarkan” exception handler ke daftar **CPU Exception Handler** jika OS menyediakan fitur tersebut. Ketika ada **CPU Exception**, Interrupt Handler utama OS akan dapat menentukan fungsi handler custom yang akan dipanggil jika ada.

Namun ada juga beberapa yang terdengar tidak familiar tapi sebenarnya memiliki nama lain yang ada pada sistem operasi modern, perhatikan beberapa exception berikut:

■ Double Fault

Ah **Double Fault**, nama yang asing tetapi sangat familiar dengan pengguna komputer x86 modern. Meskipun jarang mengoprek komputer, layar ini mestinya pernah dilihat oleh pengguna sistem operasi **Windows**



Source: [Wikipedia/BSOD](#) : (- *The Cosmic Horror of Modern PC*

Blue Screen of Death, atau salah satu penyebab utamanya: **Double Fault**, disebabkan ketika kode kernel mengalami exception. Kode driver yang membutuhkan kernel-privilege (Contohnya: **Networking stack & GPU**) yang mengalami exception juga salah satu sumber penyebab **Double Fault**.

Namun terkadang hal ini juga dapat disebabkan oleh interrupt handler kernel yang dipanggil oleh suatu user-level exception, mengalami exception, sehingga sesuai dengan namanya: **Double Fault**.

Bergantung kepada kebijakan sistem operasi masing-masing, terkadang Double Fault mungkin dihandle dan dikembalikan ke flow normal sistem operasi. Untuk Windows, Double Fault akan dilanjutkan ke penulisan log (termasuk identifikasi **stopcode** yang sesuai) dan **Blue Screen of Death**. Technically, BSOD juga mencakup beberapa kegagalan sistem lainnya, tetapi Double Fault merupakan salah satu *culprit* utama.

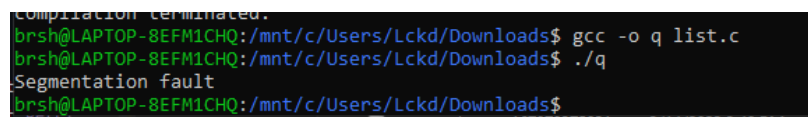
Untuk sistem operasi non-Windows, Blue Screen of Death umumnya ekuivalen dengan **Kernel Panic**.

■ Triple Fault

Kelanjutan dari **Double Fault**, **Triple Fault** terjadi ketika Double Fault exception handler mengalami exception. Berbeda dengan Double Fault yang memiliki CPU exception dan dapat memiliki interrupt handler, **Triple Fault tidak bisa dihandle oleh sistem operasi**, CPU akan secara otomatis berhenti dan melakukan reboot seluruh sistem. Jika terdapat permasalahan kode startup kernel yang menyebabkan **Triple Fault**, komputer akan mengalami **bootloop**.

■ General Protection Fault

Semestinya semua mahasiswa yang mengambil mata kuliah ini (**IF2230 - Sistem Operasi**) pernah mengambil **IF2110 - Algoritma & Struktur Data**. **IF2110** umumnya menggunakan bahasa C, sehingga pastinya pernah mendengar atau merasakan **Segmentation Fault**.



```
Compilation terminated.  
brsh@LAPTOP-8EFM1CHQ:/mnt/c/Users/Lckd/Downloads$ gcc -o q list.c  
brsh@LAPTOP-8EFM1CHQ:/mnt/c/Users/Lckd/Downloads$ ./q  
Segmentation fault  
brsh@LAPTOP-8EFM1CHQ:/mnt/c/Users/Lckd/Downloads$
```

IF2110 - “Ini kenapa sih, ga jelas”

General Protection Fault atau terkadang disingkat **GP Fault**, merupakan nama lain dari **Segmentation Fault** pada sistem operasi **UNIX**. General Protection Fault terjadi ketika sebuah instruksi membaca memori yang tidak diperbolehkan (Kernel sendiri dapat mengalami General Protection Fault pada beberapa kasus khusus), umumnya pada **IF2110**, hal ini terjadi karena mengakses pointer yang belum terinisiasi dengan baik. **Segmentation Fault** pada user program umumnya berakhir dengan terminasi program tersebut, tetapi untuk tingkat kernel umumnya menyebabkan **Double Fault** atau **Triple Fault**.

■ Breakpoint

Jika CPU modern berjalan diatas 1 GHz (a.k.a. 1.000.000.000+ tick per second dan tentunya juga jutaan *instruction cycles* per detik), bagaimana bisa kita menghentikan eksekusi program dengan debugger? Alasannya adalah **Breakpoint Exception**. Breakpoint debugger biasanya bekerja dengan mengganti / mensubstitusi instruksi source code dengan instruksi **INT \$0x3**. Berbeda dengan interrupt yang lain, khusus untuk **INT \$0x3** hanya membutuhkan 1 byte untuk **opcode**. Hal ini memperbolehkan untuk mensubstitusi instruksi apapun dengan opcode tersebut, tanpa menggeser memory yang dapat merusak offset program (terutama pada x64, dimana kode umumnya menggunakan **Position Independent Code**).

3.2. Keyboard Device Driver

Keyboard merupakan device yang umumnya ada pada komputer untuk kebutuhan input. Karena kompleksitas requirement yang ada pada sistem operasi modern, Keyboard yang memiliki banyak sekali standar (Contohnya, Japanese Keyboard layout yang memiliki Kana Mode atau jutaan varian keyboard yang ada pada pasaran & mensupport fitur yang berbeda-beda) harus di-support dengan device driver sendiri-sendiri.

Keyboard device driver yang akan dibuat pada tugas besar ini akan berfokus kepada **Generic US Keyboard Layout** dan **Scancode set 1** yang digunakan QEMU.

• Device Driver

Device driver adalah suatu software yang bertanggung jawab untuk mengabstraksikan interaksi hardware langsung (seperti menggunakan I/O port) dan menyediakan interface yang lebih sederhana ke pengguna. Pada protected mode, terdapat rule of thumb: **BYODD, Bring Your Own Device Driver**. Jika ingin sesuatu, ada kemungkinan besar bahwa tidak ada driver.

Pada OS modern, biasanya untuk menulis device driver sudah disediakan banyak sekali syscall dan API-API bawaan OS untuk mempermudah penulisan device driver dan abstraksi untuk berbagai macam device, seperti yang disebutkan sebelumnya.

Pada OS modern juga biasanya terdapat dua pilihan untuk menulis device driver: **Kernel space** atau **User space** device driver. User space memiliki kelebihan stabilitas karena kegagalan device driver hanya menyebabkan terminasi driver program oleh kernel, dengan kekurangan overhead low-level interface yang harus melewati API OS.

Kernel space memiliki kekurangan yang krusial: program device driver yang kurang bagus akan sangat mungkin menyebabkan **Double Fault** dan kernel tidak dapat melakukan recovery, alhasil **Blue Screen of Death** dan **Reboot**. Namun salah satu alasan menggunakan kernel space adalah overhead yang hampir non-existent dari sisi OS, semua resource dan low-level interface dapat diakses langsung oleh driver. Contoh device driver yang membutuhkan kernel space adalah **GPU & Networking Driver** yang sangat sensitif terhadap latency.

Fun fact: Masih belum terasa low-level? Device driver sendiri berdiri diatas **Device Controller**.

Tugas besar ini hanya bertujuan untuk memberikan gambaran berkomunikasi komputer dengan dunia eksternal, sehingga tidak diperlukan abstraksi dan mekanisme yang kompleks untuk handle berbagai macam device.

• Desain Driver

Kode utama keyboard driver akan terletak pada **keyboard_isr()**. Seperti yang disebutkan pada spesifikasi milestone 2, tugas dari driver yang dibuat hanya mengkonversikan scancode keyboard ke ASCII dan menyimpannya ke buffer. Untuk memudahkan pengerjaan, telah disediakan array **keyboard_scancode_1_to_ascii_map** yang dapat digunakan layaknya map pada **kit/keyboard/keyboard.c**.

3.2.1. IRQ1 - Keyboard Controller

Step pertama untuk membuat driver keyboard adalah meminta **Keyboard Controller** untuk melakukan **Hardware Interrupt** setiap kali tombol ditekan. Keyboard Controller terletak pada **IRQ1** untuk **8259A PIC**.

Tambahkan kode berikut pada **interrupt.c** untuk menyalakan IRQ keyboard (mematikan mask untuk interrupt keyboard controller)

interrupt.c

```
void activate_keyboard_interrupt(void) {  
    out(PIC1_DATA, PIC_DISABLE_ALL_MASK ^ (1 << IRQ_KEYBOARD));  
    out(PIC2_DATA, PIC_DISABLE_ALL_MASK);  
}
```

Panggil kode ini pada **kernel_setup()** milik kernel setelah **initialize_idt()** agar menyalakan **IRQ1 - Keyboard** dan mematikan **IRQ0 - Timer**

Arsitektur Intel x86 adalah **Interrupt-driven Architecture**. I/O akan menotify CPU menggunakan **Interrupt**, I/O ini juga termasuk keyboard. Setiap kali tombol keyboard ditekan, keyboard akan mengirimkan Interrupt IRQ1 ke CPU.

Nantinya IRQ1 di handle menggunakan **main_interrupt_handler()** (ada pada bagian Interrupt) yang akan memanggil **keyboard_isr()** setiap kali tombol ditekan.

3.2.2. Keyboard Driver Interfaces

Pada **keyboard.h** yang ada pada **kit/keyboard/keyboard.h**, akan disiapkan beberapa deklarasi fungsi yang perlu diimplementasikan. Bagian ini hanya memanipulasi driver state saja.

Karena terdapat beberapa konstanta yang perlu didefinisikan, file **keyboard.c** akan juga disediakan template dasar dengan beberapa definisi untuk konstanta. File akan tersedia pada **kit/keyboard/keyboard.c**

Definisikan fungsi-fungsi yang ada pada spesifikasi milestone 2 pada **keyboard.c**:

- Satu static variable **keyboard_state** pada **keyboard.c**
- **keyboard_state_activate()** - Menyalakan state input keyboard
- **keyboard_state_deactivate()** - Mematikan state input keyboard
- **get_keyboard_buffer()** - Mengcopy isi **keyboard_buffer** ke pointer
- **is_keyboard_blocking()** - Mengecek state input keyboard

Semua fungsi diatas semestinya hanya membutuhkan manipulasi **keyboard_state** yang didefinisikan dan tidak terlalu kompleks.

Fungsi-fungsi keyboard interface ini tidak ditujukan untuk penggunaan didalam **keyboard_isr()**. Nantinya akan dibuat untuk kepentingan user program yang tidak memiliki akses ke keyboard secara langsung.

Catatan: **Semua yang ada pada template diperbolehkan untuk dimodif jika dirasa kurang.** Tambahkan sendiri state pada driver, fungsi pembantu, interface baru, dll jika menginginkan. Spesifikasi hanya meminta interface tertentu diimplementasikan, detail implementasi diserahkan kembali ke praktikan

3.2.3. Keyboard ISR



Ilustrasi tambahan untuk menggambarkan Interrupt & Keyboard bekerja

<https://www.youtube.com/watch?v=3oR5WYDxNF8>

Keyboard ISR merupakan bagian utama driver yang akan dibuat. Sesuai dengan namanya, driver ini akan menggunakan Interrupt yang dihasilkan keyboard sebagai pemanggil **keyboard_isr()**.

Perlu diperhatikan bahwa sistem pembacaan keyboard yang digunakan oleh QEMU setelah dites adalah **Make-Break + Scancode set 1**.

Apa itu Make-Break? Keyboard akan melakukan **interrupt dua kali untuk semua keypress**:

- Satu interrupt ketika tombol **ditekan (Make)**
- Satu interrupt ketika tombol **dilepas (Break)**

Untuk mempermudah, **abaikan semua interrupt yang terjadi ketika tombol dilepas**.

Tips: Map **keyboard_scancode_1_to_ascii_map** akan memetakan break scancodes ke 0.

Berikut adalah kerangka dasar untuk **keyboard_isr()**

keyboard.c

```
void keyboard_isr(void) {
    uint8_t scancode = in(KEYBOARD_DATA_PORT);
    if (!keyboard_state.keyboard_input_on)
        keyboard_state.buffer_index = 0;
    else {
        char mapped_char = keyboard_scancode_1_to_ascii_map[scancode];
        // TODO : Implement scancode processing
    }
    pic_ack(IRQ_KEYBOARD);
}
```

Berikut adalah desain driver yang ada pada spesifikasi milestone 2:

- Ketika **keyboard_state.keyboard_input_on** bernilai true
 - Driver memproses scancode yang diterima ke ASCII character
 - Driver menyimpan hasil ASCII ke **keyboard_state.keyboard_buffer**
 - Menuliskan karakter ASCII ke layar dan menggeser kursor
 - Jika ASCII backspace ``\b`` dibaca, hapus karakter dari buffer dan layar
 - Berhenti pembacaan input ada ASCII Line Feed / Enter ``\n``
- Behavior wrapping & scrolling dibebaskan

Selain itu, jangan lupa untuk memasang **keyboard_isr()** pada **main_interrupt_handler()**, dengan interrupt vector yang sesuai IRQ dari Keyboard (**PIC1 + IRQ_KEYBOARD**).

Penting: Keyboard akan bekerja secara **non-blocking by default**, karena menggunakan Interrupt & ISR. Mirip dengan cara kerja **Event Handler** seperti pada bahasa **JavaScript**

Semestinya tidak ada loop blocking pada **keyboard_isr()** (setelah memproses 1 scancode, fungsi **keyboard_isr()** akan langsung return).

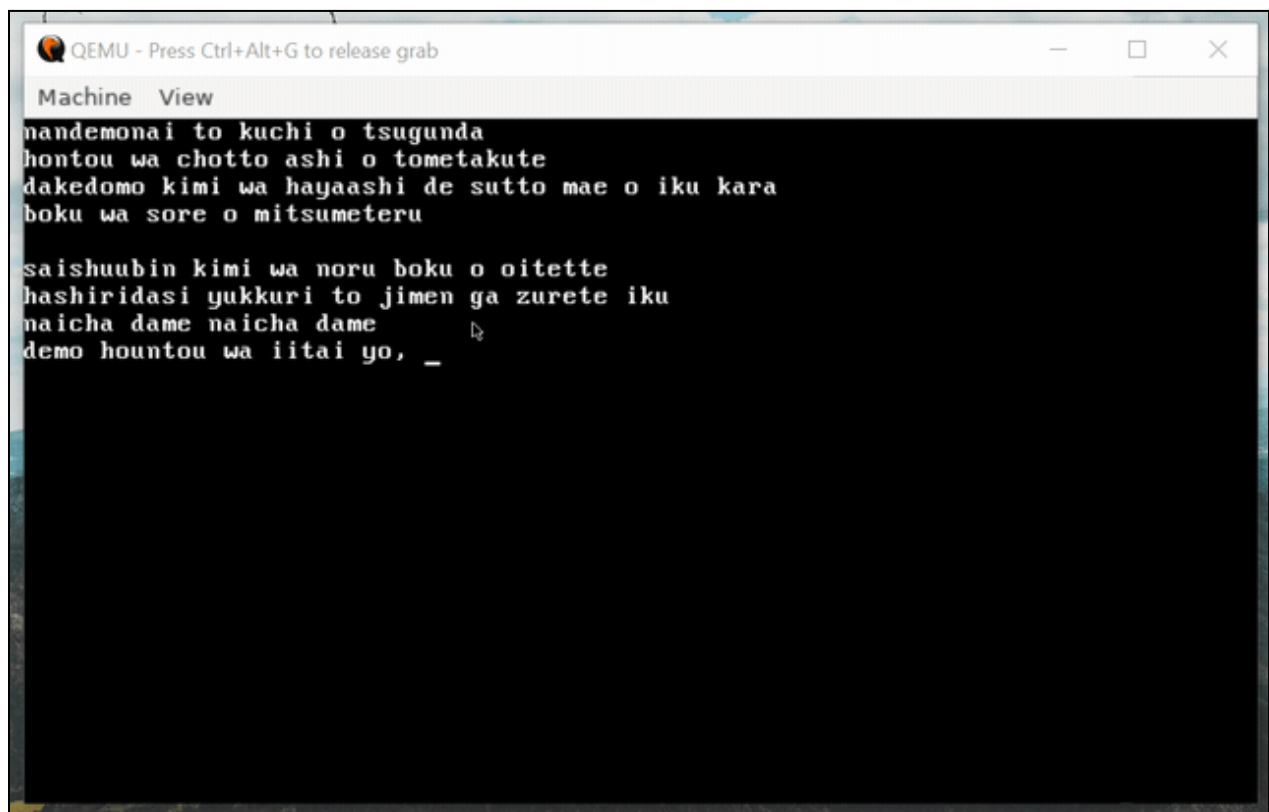
Cek bagian [Tips Keyboard](#) untuk beberapa tips pengerjaan dan trik debugging untuk bagian ini.

Jika sudah diimplementasikan, driver dapat dites dengan menambahkan kode berikut pada kernel

kernel.c

```
void kernel_setup(void) {
    enter_protected_mode(&_gdt_gdtr);
    pic_remap();
    activate_keyboard_interrupt();
    initialize_idt();
    framebuffer_clear();
    framebuffer_set_cursor(0, 0);
    while (TRUE)
        keyboard_state_activate();
}
```

Karena **keyboard_state_activate()** selalu dipanggil, semestinya dapat mengetik bebas dengan Enter dan Backspace seperti gambar berikut



Tips Keyboard

- Bisa dikatakan jika milestone 1 dan Interrupt merupakan pemanasan, bagian keyboard adalah mulainya **kernel development**
- **Tidak wajib untuk menghandle edge case yang ekstrim**
- Hover pada function signature untuk melihat dokumentasi doxygen

```
src > C fat32.c > ...  
6  
7 const uint8_t fs_signature[BLOCK_SIZE] = {  
8     'C', 'o', 'u', 'n', 't', 's', 'e', 'c', 't', 'o', 'r', ' ', 'f', 'a', 't', '3', '2', '  
9     'D', 'e', 's', 'i', 'g', 'n', 'e', 'd', ' ', 'b', 'y', ' ', 'M', 'i', 'c', 'r', 'o', 's', 'o', 'f', 't', '  
10    'L', 'a', 'b', ' ', 'S', 'i', 'm', 'p', 'l', 'e', ' ', 'I', 'n', 't', 'e', 'r', 'f', 'a', 'c', 'e', '  
11    'M', 'a', 'd', 'e', ' ', 'w', 'i', 't', 'h', ' ', '<', '3', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '  
12    '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '2', ' ', '0', ' ', '2', ' ', '3', ' ', '\n', '  
13  
14 void write_clusters(const void *ptr, uint32_t cluster_number, uint8_t cluster_count)  
15 };  
16  
17 Parameters:  
18 ptr - Pointer to source data  
19 cluster_number - Cluster number to write  
20 cluster_count - Cluster count to write, due limitation of write_blocks block_count 255 => max cluster_count = 63  
21 void write_clusters(const void *ptr, uint32_t cluster_number, uint8_t cluster_count) {
```

- Jika mengalami permasalahan atau behavior kernel tidak sesuai dengan ekspektasi kode yang dibuat, **gunakan breakpoint pada debugger**.

The screenshot shows a debugger interface with the following components:

- Source Code:** The main window displays the source code of a C program. The function `main_interrupt_handler` is shown, with line 25 highlighted. The code includes a `switch` statement for `int_number`, with a case for `PIC1_OFFSET + IRQ_KEYBOARD` that calls `keyboard_isr()`.
- VARIABLES:** The left sidebar shows the 'VARIABLES' pane. It contains two variables: `cpu` (type `CPUPRegister`) and `int_number` (type `uint32_t`), both with values of `0x21`. Below this, the 'WATCH' pane shows `$ss: 0x10`, `$cs: 0x8`, and `$eip: 0xc01027ec <main_interrupt_han...`.
- CALL STACK:** The bottom pane shows the 'CALL STACK' pane. It contains two entries: `main_interrupt_handler(struct CPUPRegister, uint32_t)` and `call_generic_handler()`. The first entry is highlighted, and the second entry is shown below it.
- PROBLEMS:** The right sidebar shows the 'PROBLEMS' pane. It contains a list of errors, including `error: 'main_interrupt_handler' was not declared in this scope` and `error: 'main_interrupt_handler' was not declared in this scope`.

- Taruh breakpoint pada lokasi yang strategis seperti pada line yang mengandung behavior yang tidak tereksekusi (misalnya pada bagian keyboard, **switch** pada **main_interrupt_handler()**, seperti gambar diatas)
- Jika breakpoint tidak menghentikan instruksi, mundurkan lagi breakpoint hingga ditemukan posisi bug

- Selain menggunakan breakpoint biasa, gunakan juga **logpoint** dan **watchpoint** untuk mempermudah debugging. Watchpoint akan berhenti ketika ekspresi dievaluasi bernilai true pada line tersebut; operator logika ==, !=, <, dan lain-lain dapat digunakan pada ekspresi. Untuk logpoint, tambahkan { } agar ekspresi dievaluasi nilai variable

The screenshot shows a debugger interface with a 'WATCH' panel on the left. A watchpoint is set on the expression 'int_number == 0x21' at line 23 of the file 'src/interrupt.c'. The main window displays the code for 'main_interrupt_handler', which includes a switch statement for 'int_number'.

Watchpoint pada `main_interrupt_handler()`, eksekusi hanya berhenti ketika `int_number == 0x21`

The screenshot shows a debugger interface with a 'Log Message' panel. A logpoint is set on the expression '{int_number}' at line 23 of the file 'src/interrupt.c'. The main window displays the code for 'main_interrupt_handler'. The 'DEBUG CONSOLE' panel at the bottom shows the log messages generated by the logpoint, displaying the value of 'int_number' as '0x21'.

Logpoint, menuliskan `int_number` ke debug console setiap kali line dieksekusi

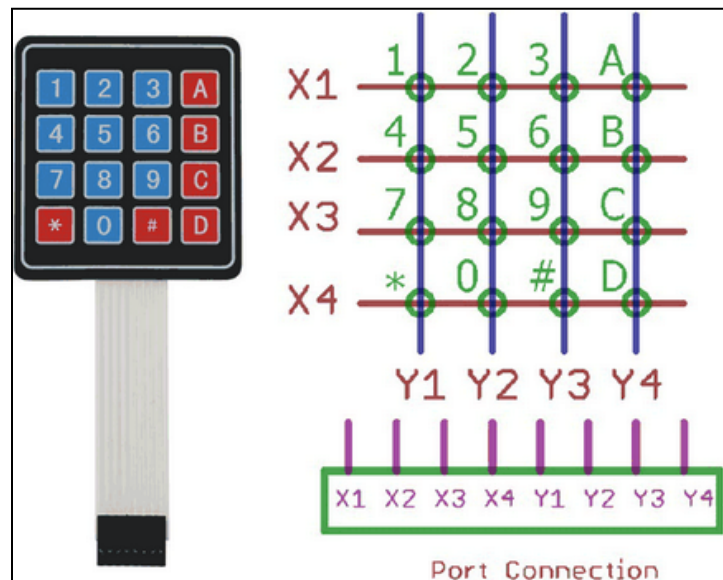
- Mengulangi dari [dokumen Debugger](#), Ingat, debugger yang digunakan adalah gdb, seluruh command gdb dapat diakses pada debug console

• Keyboard Scancode

Kenapa keyboard tidak mengeluarkan ASCII char saja ketika ditekan? Mengapa harus mempersusah dengan menggunakan scancode? Salah satu alasannya adalah abstraksi.

Dengan menggunakan scancode dan mapping oleh device driver, keyboard dapat dalam sembarang bentuk dan dipetakan device driver ke karakter yang sesuai (Kanji, Cyrillic, atau *Makeshift DIY keyboard* misalnya). Selain itu dengan scancode make-break, kita juga bisa membuat driver yang mengetahui & memproses tombol ketika ditekan atau dilepas.

Selain itu, salah satu alasan lain adanya scancode adalah penggunaan desain matrix pada keyboard kuno dan keyboard modern yang relatif murah



Source: kursuselektronikaku.blogspot.com - Arduino matrix keypad

Meskipun gambar diatas merupakan keypad arduino, bukan keyboard, inti cara kerja dari keyboard matrix masih sama, menggunakan **Grid Coordinate** untuk mendeteksi tombol, seperti pada gambar diatas. Kekurangan dari sistem ini adalah jika tombol yang ditekan cukup banyak (umumnya max 4), keyboard umumnya tidak akan bisa mengetahui eksak tombol yang ditekan (**4-key rollover**).

Untuk keyboard modern yang memiliki **N-key rollover** umumnya menggunakan teknik wiring yang berbeda dan mekanisme pembacaan yang lebih kompleks.

Untuk penjelasan tambahan terkait PS/2 Keyboard dapat dicek pada video **Ben Eater**:

▶ So how does a PS/2 keyboard interface work?

Selain itu Ben juga memiliki **Keyboard ISR**, tetapi tidak dengan x86:

▶ Keyboard interface software

3.3. File System - FAT32 IF2230 edition

Sebelum memulai, sangat direkomendasikan untuk menginstall ekstensi VScode: Hex editor. Hex editor VSCode dapat mengecek binary file, terutama disk image nantinya, tanpa blocking (VSCode hexeditor dapat tetap dibuka ketika sistem operasi yang berjalan di QEMU menuliskan sesuatu). **Penting : Buka VSCode pada separate instance agar auto update**

Ingat, bagian ini akan memisahkan antara **Secondary Memory (HDD, SSD, etc)** dan **Primary Memory (RAM)**. Sebagian besar yang direfer pada ilustrasi adalah yang ada pada **Secondary Memory** (Persistent dan disimpan pada **storage.bin**)

File system yang akan dibuat menggunakan ide dari file system FAT32, tetapi terdapat beberapa perbedaan untuk mempermudah pengerjaan.

• File System

Apa itu file system? Kenapa tidak menulis langsung ke disk saja? **File System** merupakan layanan yang umumnya disediakan sistem operasi untuk menyimpan data terstruktur layaknya sebuah **physical file** (Lembar yang biasanya ditaruh didalam **folder coklat muda** pada dunia nyata).

Layanan File System sistem operasi juga menyediakan operasi-operasi umum yang ada seperti **CRUD** untuk memanipulasi file, tanpa memperhatikan bagaimana penyimpanan dalam **Physical Hard Drive, Solid State Drive**, dan secondary memory lainnya. Tanpa file system, data harus dikelola dan diingat oleh pengguna sendiri. Tentunya hal ini sangat tidak praktis, dan jika kita menggunakan suatu sistem pencatatan lokasi data dan sistem melabelinya, hal tersebut identik dengan definisi file system.

Inti dari file system adalah sebuah sistem yang digunakan untuk menyusun dan melabeli data sehingga mempermudah proses information retrieval. Tentunya cara mengorganisir data tidak hanya menggunakan file system, terdapat beberapa struktur lain yang memiliki fokus berbeda seperti **Database** yang diajarkan pada **IF2240 - Basis Data** yang biasanya diambil bersamaan dengan mata kuliah sistem operasi. Namun umumnya, pada tingkat sistem operasi, yang sering digunakan adalah file system, sedangkan struktur yang lain biasanya diimplementasikan diatas file system sistem operasi.

• ATA, PATA, & SATA

Karena pada protected mode tidak ada fitur-fitur dasar untuk disk I/O yang dapat digunakan dengan mudah untuk sistem operasi, disk driver akan menggunakan ATA.

Advanced Technology Attachment atau **ATA** merupakan interface standard yang sering digunakan pada komputer untuk secondary storage. ATA sendiri merupakan evolusi dari interface **Integrated Drive Electronic (IDE)** dan terkadang **Parallel ATA** juga disebut IDE.

Parallel ATA merupakan awal implementasi dari ATA interface. Jika pernah mengoprek komputer lama, semestinya pernah melihat sebuah kabel pita yang cukup lebar seperti pada gambar berikut



Source : [Wikipedia/PATA](https://en.wikipedia.org/wiki/PATA) - *Ancient cable*

Kabel tersebut sangat umum untuk menghubungkan HDD ke motherboard pada era komputer lama hingga sekitar awal **Intel Pentium 4**.

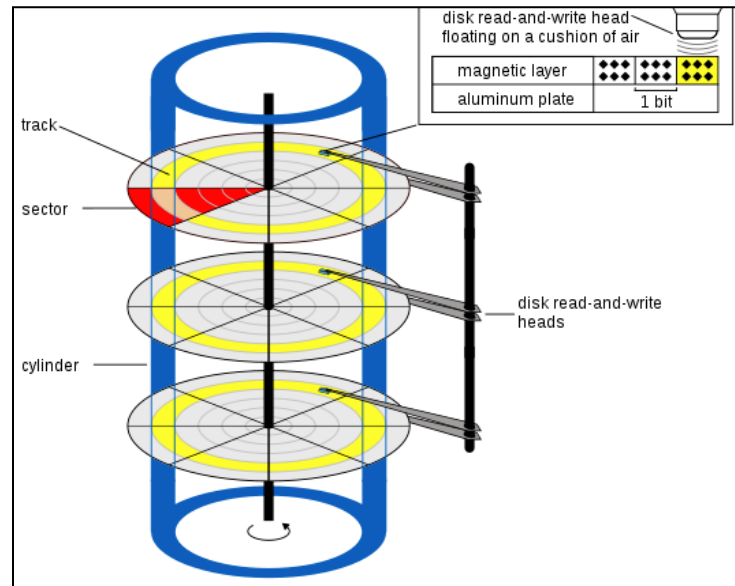
Namun seiring meningkatnya teknologi, serial interface mulai mengejar kecepatan parallel interface dan memiliki kelebihan relatif sederhana untuk implementasi elektronik memperbolehkan serial interface jauh lebih kecil dibandingkan parallel. Pada akhirnya serial interface juga mempengaruhi PATA dan dibuatlah **Serial ATA (SATA)** yang masih relatif umum digunakan sampai sekarang. PATA dan SATA masih menggunakan ATA interface untuk interaksinya, meskipun kedua physical electrical interface (serial / parallel) berbeda.

Perlu dicatat bahwa ATA PIO yang disediakan pada template bersifat **blocking**. Hal ini dikarenakan implementasi versi blocking lebih sederhana. Namun tidak menutup kemungkinan juga jika kalian ingin mengimplementasikan sistem **asynchronous I/O** untuk ATA atau menyediakan **Direct Memory Access** untuk ATA.

• Disk Addressing: LBA & CHS

Pada bagian filesystem, nantinya akan mengenal LBA lebih dekat. **LBA** atau **Logical Block Addressing** merupakan skema addressing untuk disk yang cukup simpel. Unit terkecil yang dapat di-address LBA adalah **Block**. Bergantung dengan disk controller yang digunakan, 1 block mungkin memiliki ukuran dalam byte yang berbeda-beda tetapi salah satu ukuran umum untuk physical block size adalah **512 bytes**. Block ini sebenarnya abstraksi yang disediakan oleh disk controller sehingga driver tidak perlu memikirkan susunan lokasi fisik suatu data, meskipun umumnya juga ditata secara berurutan pada fisik.

CHS atau **Cylinder-Head-Sector** merupakan skema addressing yang didasarkan dari secondary storage berbasis piringan seperti berikut



Sumber : [Wikipedia/CHS](https://en.wikipedia.org/wiki/CHS)

Dengan menggunakan addressing CHS (dan jika controller menyediakan akses langsung dengan CHS), driver dapat mengelola susunan data secara langsung pada media fisik. Hal ini akan memiliki hubungan dengan **fragmentation** yang ada pada file system, yang secara tidak langsung akan terlihat ketika mengimplementasikan FAT32 - IF2230 edition.

Namun pada era modern, dimana **Solid State Drive (SSD)** sudah banyak mengganti **Hard Disk Drive (HDD)** sebagai secondary storage, umumnya CHS hanya digunakan untuk backward compatibility support. Sama seperti block, 1 sector mungkin memiliki ukuran byte yang berbeda-beda antar controller, tetapi ukuran umumnya pada sistem x86 adalah **512 bytes**.

• Boot Sector, Bootstrap, MBR, & GPT

Boot Sector merupakan sektor atau block pada secondary storage yang akan dieksekusi pertama kali ketika komputer dinyalakan. Biasanya boot sector terletak pada indeks 0 atau dalam kata lain, sektor / block pertama dalam secondary storage. Seperti namanya, boot sector, jika mengalami corrupt, akan menyebabkan **kegagalan booting** komputer.

Kode yang ada pada boot sector dinamai **Bootstrap code**. Mekanisme loading komputer sama seperti yang dijelaskan pada [IF2230 - Milestone 1: 3.2. Booting Sequence & GRUB](#). Perhatikan bahwa bootstrap akan dieksekusi otomatis oleh CPU ketika booting, jika terdapat virus yang berhasil menuliskan malware code ke bootstrap code, kode malware tersebut memiliki **privilege penuh** terhadap sistem meskipun terbatas dalam segi jumlah kode.

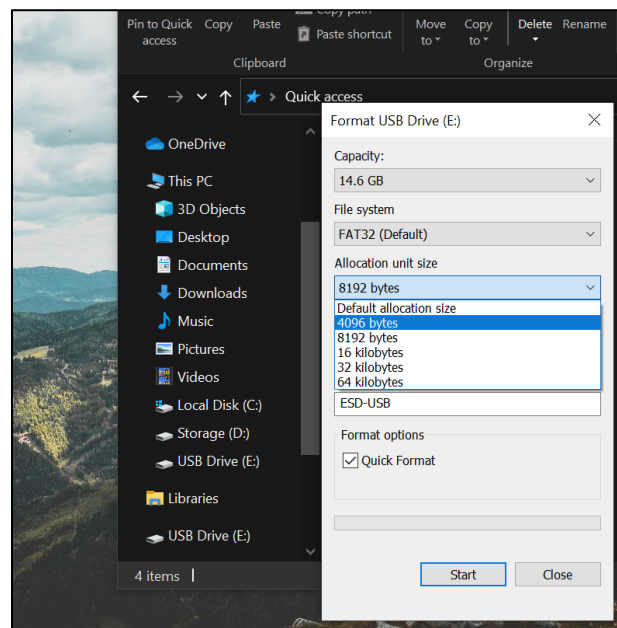
Master Boot Record (MBR) adalah tipe boot sector yang berisikan tidak hanya bootstrap code, tetapi informasi partisi dari secondary storage. MBR, sama seperti boot sector, memiliki keterbatasan ukuran 1 sektor dan keterbatasan sistem addressing yang hanya dapat mencakup hingga **2 TiB**.

GUID Partitioned Table (GPT) yang bukan GPT di-ChatGPT) merupakan sistem partisi yang lebih modern. Singkatnya GPT bisa dikatakan ekstensi dari MBR dengan menyimpan GPT header pointer pada MBR. GPT header inilah menyimpan informasi partisi secondary storage dan mungkin terletak pada lokasi yang lebih besar daripada 1 sektor yang merupakan limitasi MBR.

• Block & Cluster

Catatan: Dari file system ke file system, definisi terminologi cluster, block, dan sector mungkin sedikit berbeda. Penjelasan berikut merupakan terminologi yang ada pada file system FAT32.

Pernah melakukan format? Berikut tampilan format pada Windows 10 untuk flashdisk



Source : Dokumen pribadi

Perhatikan menu Allocation unit size, kenapa semua pilihan pada menu tersebut merupakan kelipatan bilangan bulat dari 4096 bytes?

Menu allocation unit size merupakan ukuran satu cluster dan **Cluster merupakan unit data terkecil pada file system FAT32**. Satu cluster memiliki beberapa block, sehingga ukuran cluster adalah kelipatan bilangan bulat dari ukuran satu block. Sistem komputer modern biasanya menggunakan ukuran **1 block = 4096 bytes** untuk logical block addressing-nya. **Namun pada tugas besar ini, akan digunakan konvensi 1 block = 512 bytes.**

Untuk ukuran cluster size, tugas besar ini akan menggunakan **4 block per cluster**, yang terdefinisi pada konstanta **CLUSTER_BLOCK_COUNT**. Konstanta **CLUSTER_SIZE** berukuran **CLUSTER_BLOCK_COUNT * BLOCK_SIZE = 2048 bytes = 2 KiB**

Tugas besar ini juga akan menggunakan cluster addressing yang lebih straightforward, tidak ada offset. **Cluster 0 dimulai pada block 0, cluster 1 terletak pada block 4, dan seterusnya.**

Kenapa terdapat menu allocation unit? Apa gunanya mengubah ukuran cluster? Hal ini disebabkan trade-off antara **metadata** dan **ukuran slack space**. Jika ukuran cluster sangat besar, file yang sebenarnya ukurannya jauh dibawah cluster seperti 7 bytes text file, harus dialokasikan sebesar 1 cluster pada disk. Sisa ruang tersebut (bernama **Slack Space**, pada contoh ini berukuran 1 cluster - 7 bytes) tidak dapat digunakan file lain.

Jika ukuran cluster sangat kecil, file yang besar akan terfragmentasi menjadi sangat banyak pada **FileAllocationTable** atau menggunakan metadata dalam jumlah yang sangat besar. Bergantung pada sistem manajemen data pada file system dan implementasi interfacenya, penggunaan metadata yang sangat besar dapat membuat kinerja sistem buruk. Selain itu, kemungkinan data terfragmentasi akan sangat besar, kinerja pada storage sekuensial seperti HDD akan sangat buruk.

3.3.1. Disk Driver & Image

Sebelum membuat file system, tentunya diperlukan sebuah cara untuk menuliskan data ke secondary storage. Sayangnya tidak ada fitur yang dapat digunakan langsung oleh kernel, sehingga disk driver perlu diimplementasikan sendiri. Dokumentasi untuk interface disk driver akan ada pada **kit/filesystem/disk.h**. Berikut adalah implementasi untuk **disk.c**

disk.c

```
#include "lib-header/disk.h"
#include "lib-header/portio.h"

static void ATA_busy_wait() {
    while (in(0x1F7) & ATA_STATUS_BSY);
}

static void ATA_DRQ_wait() {
    while (!(in(0x1F7) & ATA_STATUS_RDY));
}

void read_blocks(void *ptr, uint32_t logical_block_address, uint8_t block_count) {
    ATA_busy_wait();
    out(0x1F6, 0xE0 | ((logical_block_address >> 24) & 0xF));
    out(0x1F2, block_count);
    out(0x1F3, (uint8_t) logical_block_address);
    out(0x1F4, (uint8_t) (logical_block_address >> 8));
    out(0x1F5, (uint8_t) (logical_block_address >> 16));
    out(0x1F7, 0x20);

    uint16_t *target = (uint16_t*) ptr;

    for (uint32_t i = 0; i < block_count; i++) {
        ATA_busy_wait();
        ATA_DRQ_wait();
        for (uint32_t j = 0; j < HALF_BLOCK_SIZE; j++)
            target[j] = in16(0x1F0);
        // Note : uint16_t => 2 bytes, HALF_BLOCK_SIZE*2 = BLOCK_SIZE with pointer arithmetic
        target += HALF_BLOCK_SIZE;
    }
}

void write_blocks(const void *ptr, uint32_t logical_block_address, uint8_t block_count) {
    ATA_busy_wait();
    out(0x1F6, 0xE0 | ((logical_block_address >> 24) & 0xF));
    out(0x1F2, block_count);
    out(0x1F3, (uint8_t) logical_block_address);
    out(0x1F4, (uint8_t) (logical_block_address >> 8));
    out(0x1F5, (uint8_t) (logical_block_address >> 16));
    out(0x1F7, 0x30);
}
```



```

for (uint32_t i = 0; i < block_count; i++) {
    ATA_busy_wait();
    ATA_DRQ_wait();
    /* Note : uint16_t => 2 bytes, i is current block number to write
       HALF_BLOCK_SIZE*i = block_offset with pointer arithmetic
    */
    for (uint32_t j = 0; j < HALF_BLOCK_SIZE; j++)
        out16(0x1F0, ((uint16_t*) ptr)[HALF_BLOCK_SIZE*i + j]);
}
}

```

Kedua fungsi diatas bersifat **blocking** untuk operasi disk I/O. Detail dari ATA dapat dicek pada [bagian opsional ATA](#).

Perhatikan bahwa pada kode diatas menggunakan fungsi **in16()** dan **out16()**. Implementasikan kedua port I/O tersebut mirip seperti **in()** dan **out()** tetapi menggunakan versi instruksi in / out 16-bit.

Tips: 1 block = 512 bytes = **0x200 bytes**

Maka logical block address ke 17, memiliki byte offset $0x200 * 17 = 0x200 * 0x11 = 0x2200$

Tips ini akan berguna untuk melompat dengan hexeditor (Keybind umum: **Ctrl + G**)

Sebelum melanjutkan, perlu diperhatikan bahwa sistem operasi yang dibuat **tidak memiliki secondary storage**. Sehingga perlu untuk menambahkan beberapa step tambahan untuk menyambungkan secondary storage ke QEMU ketika dijalankan.

Tambahkan recipe berikut pada **makefile**

makefile

DISK_NAME = storage

disk:

@qemu-img create -f raw \$(OUTPUT_FOLDER)/\$(DISK_NAME).bin 4M

Kode diatas akan membuat sebuah disk image dengan format raw binary berukuran 4 MiB. Patut dicatat bahwa sebaiknya recipe **disk** makefile tidak selalu dijalankan ketika melakukan run. Lebih baik menjalankan recipe tersebut secara manual, karena jika selalu dijalankan, **semua data pada secondary storage akan selalu terhapus** ketika melakukan kompilasi sistem operasi.

Tambahkan parameter seperti berikut pada semua script yang menjalankan QEMU (**makefile** dan **.vscode/tasks**) untuk menghubungkan storage ke QEMU ketika menjalankan

```

qemu-system-i386 -s -S -drive file=storage.bin,format=raw,if=ide,index=0,media=disk
-cdrom os2023.iso

```

3.3.2. File System: FAT32 - IF2230 edition

Untuk memudahkan pengerjaan file system, **semua deklarasi akan disediakan pada template**, sehingga pengerjaan akan dapat langsung berfokus kepada **pemahaman FAT32** dan **CRUD**. Jika membutuhkan definisikan sendiri fungsi-fungsi helper yang telah dideklarasikan, sesuai dengan dokumentasi yang ada pada doxygen tersebut.

Bagian ini akan menjelaskan bagaimana cara kerja file system yang akan diimplementasikan. Jika merasa sudah memahami, dapat melewati bagian ini dan langsung melanjutkan ke implementasi initializer & CRUD operation.

Untuk pengerjaan bagian ini, deklarasi lengkap struktur data (**FAT32FileAllocationTable**, **FAT32DirectoryEntry**, **FAT32DirectoryTable**) dan interface driver akan disediakan pada **kit/filesystem/fat32.h**

Penting: Jika merasa masih belum paham, teruskan membaca hingga akhir sebelum mengulangi lagi. Terdapat beberapa penjelasan untuk “*frequently asked question*” dan ilustrasi tambahan pada bagian-bagian selanjutnya yang mungkin dapat membantu sebelum melakukan *pembacaan second pass*.

Untuk penjelasan file system FAT32 dan operasi CRUD, akan disertakan animasi tambahan untuk membantu menggambarkan ide dari desain file system.

FAT32 merupakan file system yang menggunakan sebuah tabel berisikan **Linked List**. Pointer linked list ini akan dalam berbentuk **Cluster Number**. Tabel FAT32 akan bernama **FAT32FileAllocationTable** pada tugas besar ini.

File akan terletak pada sebuah direktori dan semua merupakan sub-directory dari **directory root**. Khusus untuk FAT32, **Directory** (atau **Folder**) juga merupakan sebuah **File**. Sehingga ada dua hal yang penting pada **FAT32FileAllocationTable**:

- **Directory** - Sebuah file yang berisikan **FAT32DirectoryTable**
- **File** - Data dalam bentuk binary

Penting: Kedua hal diatas tidak langsung dapat dibaca pada **FileAllocationTable**. **FileAllocationTable** tidak menyimpan data dari file langsung, hanya informasi cluster number.

Pada **FileAllocationTable** yang kosong, akan dijamin akan ada 3 entry yang reserved. Indexing pada **FileAllocationTable** mulai dari 0. Cluster 0 dan 1 merupakan cluster reserved yang akan diberikan nilai khusus **CLUSTER_0_VALUE** dan **CLUSTER_1_VALUE**. Cluster ke 2, merupakan **directory root**.

Seperti yang disebutkan pada desain file system, direktori akan terbatas pada 1 **DirectoryTable** sehingga akan langsung terminasi. Root akan bernilai **FAT32_END_OF_FILE**. Berikut ilustrasi untuk **FileAllocationTable**

Logical View

FileAllocationTable				
Offset	0	1	2	3
0	CLUSTER_0_VALUE	CLUSTER_1_VALUE	Root Directory	<Empty>
4

Physical View

FileAllocationTable				
Offset	0	1	2	3
0	0x0FFF FFF0	0x0FFF FFFF	0x0FFF FFFF	0x0000 0000
4

FAT32_FAT_END_OF_FILE bernilai **0x0FFF FFFF**. Sehingga root directory yang merupakan sebuah folder, akan bernilai **0x0FFF FFFF**. Untuk file, linked list dapat menunjuk ke cluster number yang lain.

Mungkin penjelasan diatas masih terasa kurang jelas dan mungkin ada pertanyaan seperti berikut:

- Jika **FileAllocationTable** tidak membawa informasi data dari file, dimana data disimpan?
- Bagaimana cara membaca folder, dimana datanya?

Disinilah indeks yang ada pada **FileAllocationTable** berfungsi ganda, **cluster number fisik** dan **indeks / pointer linked list** pada **FileAllocationTable**. Data dari file dapat terletak pada cluster ke-**cluster_number**, dengan **cluster_number** adalah indeks pada **FileAllocationTable**.

Misalnya, berikut adalah contoh **FileAllocationTable** yang berisi beberapa file dan direktori (contoh sama dengan [tester spesifikasi milestone 2](#))

Logical View

FileAllocationTable				
Offset	0	1	2	3
0	CLUSTER_0_VALUE	CLUSTER_1_VALUE	Root directory	"daijoubu"-0
4	Folder "kano1"	"daijoubu"-1	"daijoubu"-2	"daijoubu"-3
8	"daijoubu"-4-EOF

Physical View

FileAllocationTable				
Offset	0	1	2	3
0	0x0FFF FFF0	0x0FFF FFFF	0x0FFF FFFF	0x0000 0005
4	0x0FFF FFFF	0x0000 0006	0x0000 0007	0x0000 0008
8	0x0FFF FFFF	0x0000 0000	0x0000 0000	0x0000 0000

Warna merah : Reserved cluster

Warna biru : Directory / folder cluster

Warna hijau : File cluster

Terlihat pada file "daijoubu", awal cluster menunjuk cluster number berikutnya, yang menandakan bagian selanjutnya dari file ada pada cluster number ke-5. Karena cluster number ke-5 bukan bernilai EOF (Konstanta **0x0FFF FFFF**), berarti masih ada lanjutan dari file ini, dan menunjuk cluster number ke-6, dan seterusnya hingga pada cluster ke-8, bernilai EOF yang menandai akhir dari file.

Jika lokasi cluster file berisikan sebuah data binary partisi dari file, lokasi cluster directory berisikan **DirectoryTable**. Satu cluster tersebut digunakan untuk keperluan **DirectoryTable**.

Mungkin juga ada pertanyaan lanjutan, seperti:

- Jika nantinya operasi CRUD beroperasi pada **Physical View** (tidak ada penanda bytes tersebut bermakna apa, hanya murni angka binary), bagaimana cara membaca file system?
- Bagaimana dari struktur **FileAllocationTable** diolah sehingga kita dapat membentuk **Directory Tree**?

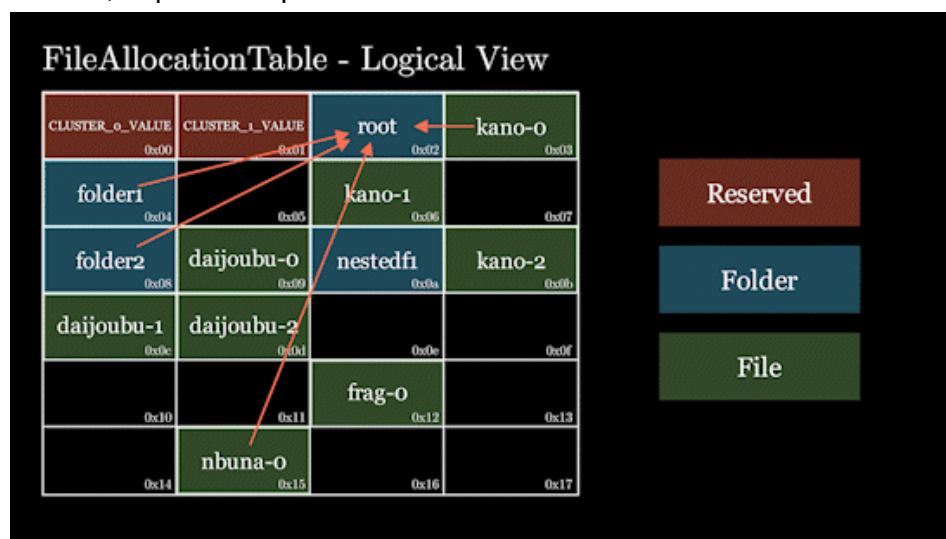
Disini isi dari **DirectoryTable** menjawab pertanyaan tersebut, dengan menggunakan contoh yang sama dengan sebelumnya, berikut adalah isi dari **folder root directory** yang dijamin selalu ada

Logical View

DirectoryTable - Root Directory	
Index	Entry (Struktur data : DirectoryEntry)
0	Parent folder: root (Special first entry, yang dijelaskan pada desain fs)
1	File : "kano"
2	Folder : "folder1"
3	...

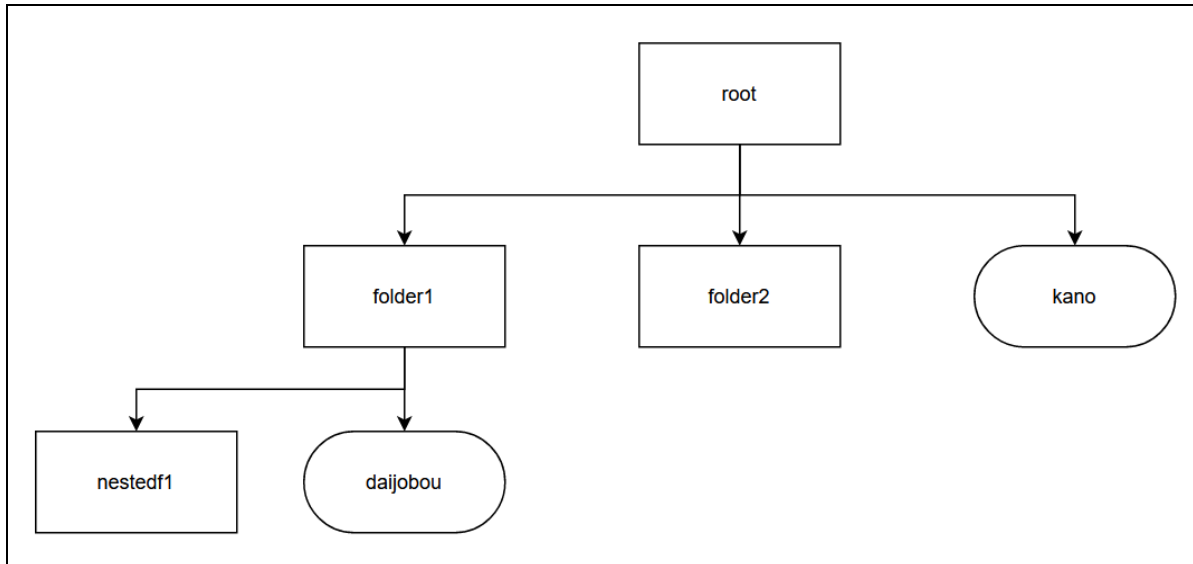
Karena root directory dijamin ada dan tidak bisa dihapus, **Directory Tree** sistem akan dapat dibentuk dari algoritma traversal rekursif dari **root directory** ke **subdirectory**. **DirectoryEntry** pada **DirectoryTable** dapat dianggap pointer ke node lain.

Pada desain file system wajib yang disebutkan pada spesifikasi milestone 2, **folder dijamin tidak memiliki lanjutan cluster**, untuk mempermudah implementasi. Untuk animasi tambahan yang mungkin membantu, dapat dicek pada link berikut:



FAT32 IF2230 edition - <https://www.youtube.com/watch?v=DjfiGvb1iT8>

Note : Animasi mengalami sedikit *miss*, struktur folder animasi berbeda dengan contoh, cek video desc



Beberapa direktori dan subdirektori yang ada pada animasi (kotak: Folder, rounded: file)

Hmm kok familiar ya, apakah ini mirip mata kuliah sebelah, sebuah ~~gra~~

Ingat subdirektori juga memiliki DirectoryTable sendiri. DirectoryTable ini nantinya memiliki cara yang sama seperti file untuk membacanya, contohnya **folder1** yang memiliki **cluster_number** 0x4 pada **DirectoryTable** root (sama seperti animasi), dapat diakses dengan membaca **cluster_number** 0x4 dan memasukkannya ke **DirectoryTable** Kosong.

Untuk desain file system **FAT32 - IF2230 edition** telah dideskripsikan pada [spesifikasi milestone 2](#), cek lagi spesifikasi tersebut ketika mengimplementasikan.

Namun singkatnya, semua batasan pada desain file system **FAT32 - IF2230 edition** ditujukan untuk mempermudah pekerjaan praktikan. Beberapa alasan pilihan desain akan terlihat pada milestone selanjutnya.

Jika masih ada pertanyaan yang diperlukan klarifikasi, gunakan QnA dan asistensi.

- **FAT32 File Size Limit**

Pernah mencoba melakukan copy file ke flash disk atau hard drive lama dan mengalami error karena batasan ukuran file 4 GiB? Hal ini disebabkan oleh attribut file size **FAT32** merupakan **uint32_t**, sehingga hanya dapat menampung $2^{32} - 1 = 4.194.304 - 1 = 4.194.303$ bytes. Meskipun sistem linked list yang digunakan pada **FileAllocationTable** mestinya dapat mendukung sembarang ukuran file, keterbatasan metadata tersebut membatasi ukuran sebuah file.

3.3.3. File System Initializer

Dari file kit **fat32.h** yang disediakan, semestinya sudah dideklarasikan interface-interface dan expected behavior dari fungsi tersebut. Untuk memulai pembuatan file system, diperlukan inisiasi file system terlebih dahulu.

Berikut adalah spesifikasi wajib yang ada pada milestone 2:

- **FAT32DriverState**
- **initialize_filesystem_fat32()**
- **is_empty_storage()**
- **create_fat32()**

Template dasar untuk **fat32.c** (termasuk definisi konstanta **fs_signature**) dan **fat32.h** akan tersedia pada **kit/filesystem/**

Untuk **FAT32DriverState**, template akan menyediakan beberapa atribut yang dirasa penting digunakan. Namun jika merasa membutuhkan state tambahan untuk melakukan keep-track file system, sama seperti bagian sebelumnya, diperbolehkan untuk menambahkan dan memodifikasi sesuai dengan kebutuhan

Fungsi **initialize_filesystem_fat32()** akan melakukan **create_fat32()** ketika **is_empty_storage()** bernilai true. Jika file system ternyata tidak kosong, baca **FileAllocationTable** & state lainnya dan masukkan state ke **FAT32DriverState**.

Fungsi **is_empty_storage()** akan melakukan komparasi (gunakan **memcmp()**) antara boot sector dengan **fs_signature**. Bernilai **true** jika komparasi **fs_signature** dan **boot sector** (tentunya bukan konstanta **BOOT_SECTOR** seperti **memcmp(fs_sign, BOOT_SECTOR)**) menghasilkan inequality.

Fungsi **create_fat32()** akan membuat **FileAllocationTable** kosong dan root **DirectoryTable**, sama seperti yang dideskripsikan pada bagian sebelumnya.

Fungsi-fungsi opsional akan disediakan deklarasinya, tetapi tidak wajib untuk diimplementasikan.

3.3.4. File System CRUD Operation

Sebelum melanjutkan untuk mengimplementasikan CRUD file system, sangat direkomendasikan untuk mengikuti anjuran spesifikasi milestone 2: **Mengganti KERNEL_STACK_SIZE ke 2 MiB.**

Berikut adalah contoh untuk membaca sebuah **DirectoryTable**

```
// Read directory at parent_cluster_number
read_clusters(&fat32driver_state.dir_table_buf, request.parent_cluster_number, 1);
```

Berikut link untuk animasi **read** dan **write** yang mungkin membantu:

FileAllocationTable - Logical View				Read	
CLUSTER_o_VALUE 0x00	CLUSTER_i_VALUE 0x01	root 0x02	kano-o 0x03	buf: <Pointer>	
folder1 0x04		kano-1 0x06		name: kano	
folder2 0x08	daijoubu-o 0x09	nestedfi 0x0a	kano-2 0x0b	ext: <None>	
daijoubu-1 0x0c	daijoubu-2 0x0d			parent_cluster_number: 0x02	
				buffer_size: 10000	
				DirectoryTable - Root (cluster 0x02)	
				Cluster	Name
				0x2	root
				0x4	folder1
				0x8	folder2
				0x3	kano
				0x15	nbuna
					Filesize
					0
					0
					5123
					1337

FAT32 IF2230 edition - Read & Write - <https://www.youtube.com/watch?v=2CM1tsyqbyY>

Note : Animasi mengalami sedikit *miss*, struktur folder animasi berbeda dengan contoh guidebook, cek video desc

■ Read dan Read Directory

Berikut merupakan deskripsi dari interface **read**:

Read akan mencari file yang terletak pada folder parent **request.parent_cluster_number**.

DirectoryTable akan di-iterasi hingga ditemukan **request.name** dan **request.ext** yang sama.

Jika **request.buffer_size** kurang dari **entry.filesize**, kembalikan error. Jika flag subdirectory pada **entry.attribute** menyala (bit subdirectory bernilai 1), kembalikan error.

Jika kondisi sudah valid, lakukan pembacaan hingga semua cluster terbaca dan dimasukkan ke **request.buf**.

Untuk **read_directory** memiliki cara kerja yang sama, tetapi hanya menerima target yaitu sebuah folder. **request.ext** tidak digunakan pada **read_directory**. **read_directory** akan membaca **DirectoryTable** yang ada pada disk dan memasukkannya pada **request.buf**.

■ Write

Berikut merupakan deskripsi dari interface **write**:

Behavior untuk write adalah **first empty location found** pada **FileAllocationTable**. Request wajib memberikan **request.parent_cluster_number** yang valid (cluster tersebut berisikan directory).

Jika directory valid, cek apakah **request.name** dan **request.ext** sudah ada atau belum (Tips: Hati-hati dengan komparasi **Null Terminated String**).

Jika **request.buffer_size** bernilai 0, buatlah sub-direktori pada folder parent **request.parent_cluster_number** dengan nama **request.name**.

Jika **request.buffer_size** lebih dari 0, write akan menuliskan data yang ditunjuk oleh **request.buf** hingga jumlah cluster yang ditulis = **ceil(request.buffer_size / CLUSTER_SIZE)**.

Parent folder **DirectoryTable** akan ditambahkan **DirectoryEntry** baru dengan nama yang sesuai dari **entry.user_attribute = UATTR_NOT_EMPTY**. Sesuaikan flags pada **entry.attribute**.

Pastikan juga mengupdate cache yang ada pada memory pada **FAT32DriverState**.

Style penulisan data ke storage dibebaskan, beberapa contoh:

- Mark first, write later (Defer penulisan, simpan informasi cluster kosong dan tulis diakhir)
- Single iteration & direct cluster writing (Seperti pada animasi)

■ Delete

Berikut merupakan deskripsi dari interface **delete**:

Delete akan menghapus sebuah file jika dan hanya jika **request.name**, **request.ext**, dan **request.parent_cluster_number** menunjuk kepada **DirectoryEntry** file yang valid (hasil pencarian tidak kosong). **DirectoryEntry** pada folder parent **request.parent_cluster_number** akan dihapus tanpa menggeser entry yang lain. Hapus semua linked list clusters yang ada pada **FileAllocationTable**. Data file fisik yang ada pada cluster tidak wajib untuk dihapus (di-overwrite dengan byte 0x00 misalnya).

Entry khusus (indeks ke 0, pointer ke parent folder, dengan nama direktori sekarang) yang disebutkan pada desain file system, **tidak dapat dihapus**. Abaikan request yang ditemukan pada indeks ke 0 pada **DirectoryTable**.

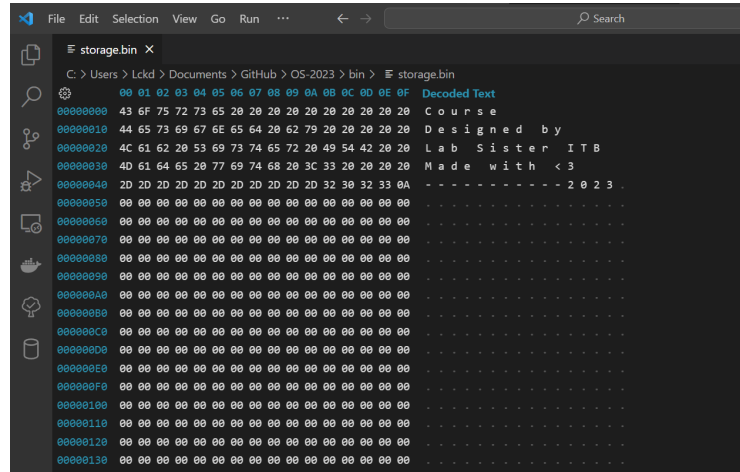
Untuk folder, delete akan menghapus **DirectoryTable** hanya dan hanya jika tepat memiliki 1 entry (entry khusus) dan bukan **root directory**. Jika masih ada entry lain, kembalikan error.

Implementasikan ketiga interface CRUD tersebut untuk file system. **Gunakan fitur-fitur debugger dan hexeditor (mengecek storage) untuk membuat interface tersebut**. Untuk atribut-atribut seperti date and time, dibebaskan behaviornya dan implementasi serius dijadikan bonus (**Bonus: CMOS IRQ**).

Sekali lagi, tugas besar ini tidak didesain untuk pengerjaan satu orang saja. Gunakan fakta bahwa kalian memiliki teman / anggota kelompok yang semestinya dapat diajak bekerja sama. Jika perlu, lakukan *pair-programming* secara offline ketika mengerjakan tugas besar ini.

Agar adil, anggota yang tidak berkontribusi mungkin dapat dinolkan, bergantung pada peer assessment teman dan evaluasi asisten.

Tips File System



Visual Studio Code - Hexeditor - Success writing into boot sector

- Gunakan Ctrl+G untuk melakukan jump ke offset pada hexeditor, contohnya **0x800** (cluster 1) akan terisi **FileAllocationTable**, **0x800*2 = 0x1000** (cluster 2) terisi **root directory**
- Pada tugas besar ini, 1 block = 512 bytes = **0x200 bytes** => 1 cluster = **0x800 bytes**, gunakan hexadecimal untuk mempermudah jump pada hexeditor
- Tips: Gunakan **do-while** construct ketika menuliskan ke storage
- Berikut adalah jumlah kode pada implementasi dari asisten:
 - Reference implementation tidak handle banyak edge case, hanya memenuhi spesifikasi saja
 - Gunakan informasi berikut untuk mengestimasi apakah kode yang dibuat terlalu kompleks atau terlalu sederhana. **Jumlah line of code tidak ada hubungannya dengan penilaian**

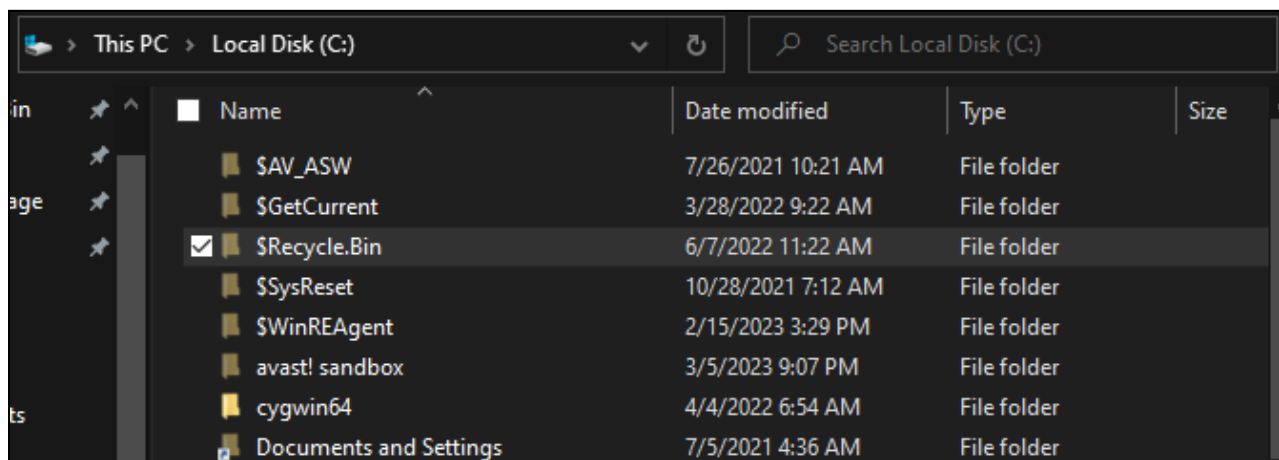
Function & Interfaces	Reference implementation (excluding comment & formatting)
CRUD Helper function (dirtable_linear_search(), etc)	42 line of code
read()	18 line of code
read_directory()	12 line of code
write()	31 line of code
delete()	22 line of code

• Undelete & Disk Recovery

Perhatikan pada interface delete yang dideskripsikan pada file system FAT32 - IF2230 edition, **overwrite tidak diwajibkan untuk ada pada interface tersebut**, tetapi diperbolehkan jika ingin menambahkan pada pengerjaan tugas besar ini. Pada sistem operasi modern, malah *by default* dapat diasumsikan penghapusan sebuah file berarti **hanya menghapus file metadata yang ada file system**. Karena operasi overwrite akan memakan waktu dan bersifat linear dengan ukuran file.

Hal ini menyebabkan data dari file sebenarnya masih ada, jika belum ter-overwrite dengan data baru. Berasal dari situlah, beberapa software menyediakan layanan **Undelete**. Jika data masih belum di-overwrite, kemungkinan besar data dari file yang telah dihapus metadata-nya masih dapat dilakukan recovery.

Perlu dicatat bahwa sistem seperti Recycle Bin tidak menghapus langsung, hanya memindahkan file dan direktori ke direktori khusus: **Recycle Bin**. Pada Windows, setiap partisi akan memiliki direktori Recycle Bin yang berbeda dan merupakan **Hidden System Files**. Folder tersebut dapat dilihat pada lokasi root partition



Disk Recovery mengembangkan ide undelete, data-data pada physical drive mungkin hanya kehilangan metadata atau disk controller mengalami kerusakan. Sehingga layanan disk recovery mungkin menyediakan layanan software untuk melakukan prediksi metadata file system untuk proses recovery data atau yang lebih advanced, melakukan “operasi” fisik physical drive, entah desoldering NAND flash chip SSD atau piringan HDD ke wadah yang baru.

Terkait dengan dua diatas, hal tersebut menjadi dasar **Data Shredding**. Karena penghapusan sebuah data pada file system perlu melakukan penghapusan yang lebih involved dari sekedar operasi penghapusan yang biasanya dilakukan pada API file system.

Detail tentang keamanan memory (baik primary atau secondary) biasanya dibahas pada: **Information Security - Digital Forensics** atau pada STEI ITB tersedia course (**STI**) **II4033 - Forensik Digital** dan (**IF**) **IF4033 - Keamanan dan Penjaminan Informasi** yang juga biasanya menyinggung digital forensics.

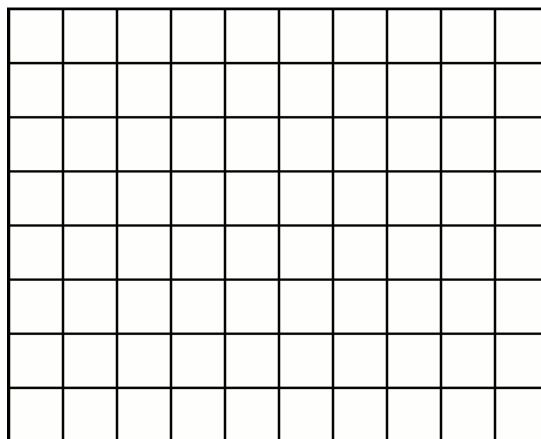
• Defragmentation

Pernah mendengar defragmentation? Jika sebelumnya mungkin belum tergambar langsung, sekarang dengan mengimplementasikan CRUD file system FAT32, mestinya akan tergambar lebih konkrit.

Fragmentation adalah kondisi dimana sebuah file yang terpisah menjadi beberapa block yang tidak kontigu. Seperti yang diperlihatkan pada [contoh tester milestone 2](#), file “daijoubu” ter-fragmentasi antara part ke 0 dan part ke 1 hingga terakhir.

Dengan penggunaan file system yang terus menerus pada sistem operasi, lama kelamaan, fragmentasi pada secondary storage akan terakumulasi dan menyebabkan sequential read yang buruk. **Disk scheduling** yang dipilih dan digunakan untuk sistem operasi juga akan memiliki kinerja yang bervariasi, setiap strategi mungkin memiliki *quirk* yang berbeda dengan fragmentasi.

Sebenarnya pada **read()** dapat dioptimisasi dengan pembacaan cluster secara kontigu jika diketahui lokasi cluster selanjutnya tersusun rapi dan berurutan pada cluster tersebut (spesifikasi tugas besar ini tidak meminta hal tersebut). Namun, dengan adanya fragmentasi, pembacaan file dipaksa harus secara satu-per-satu cluster dan mengecek **FileAllocationTable** terus menerus yang dapat menjadi overhead. Operasi non-sequential (alias dalam terminologi modern: **Random Access**) ini sangat berpengaruh untuk disk-based storage tetapi pengaruhnya tidak terlalu besar pada NAND flash-based storage.



Source : [Wikipedia/Defragmentation](https://en.wikipedia.org/wiki/Defragmentation)

Defragmentation adalah proses dimana menyusun data-data yang ada pada storage menjadi kontigu dan rapi. Hal ini biasanya dilakukan secara otomatis dan periodik pada sistem operasi modern untuk disk-based storage (seperti HDD). Untuk NAND flash-based storage, karena kinerja random access pada teknologi tersebut sangat tinggi dibandingkan disk-based dan adanya wear setiap kali operasi writing dijalankan, NAND flash-storage tidak dilakukan defragmentasi. Benefit dari sequential data lebih kecil jika dibandingkan wear yang disebabkan dari proses defragmentasi dalam konteks NAND flash-storage.

Disk scheduling dan **SSD wear leveling** biasanya dijelaskan lebih lanjut dikelas Sistem Operasi.

• OS & Information Security

Pada sistem komputer modern, umumnya sistem operasi merupakan landasan dasar dari information security. Akses penuh sistem akan di-reserve untuk **Kernel** dan **Root User**. Tentunya, akses ini harus dibatasi untuk user program yang berjalan pada privilege biasa.

Namun seperti program pada umumnya, sistem operasi juga merupakan sebuah program khusus yang ditulis oleh manusia, sehingga seiring meningkatnya kompleksitas sistem operasi, jumlah **Bug** akan juga meningkat. Dari bug-bug tersebut, beberapa mungkin dapat di-**Exploit** oleh pihak yang tidak bertanggung jawab dan menjadi sebuah **Vulnerability** pada sistem.

Dari semua bagian opsional pada guidebook ini, telah ditulis beberapa hal terkait keamanan informasi seperti Digital Forensic dan Boot Sector malware. Karena OS mencakup resource management dan privilege limitation, sistem operasi umumnya berperan langsung dalam information security dalam dunia nyata.

Keamanan pada user program seperti **Stack Canary** dan **Address Space Layout Randomization**, dikelola dan disediakan oleh sistem operasi. Semua interface yang disediakan oleh sistem operasi juga akan memiliki potensi bug yang dapat menjadi sebuah vulnerability. Semakin kompleks dan banyak fitur sistem operasi, semakin banyak juga **Attack Surface** yang disediakan dari interface tersebut.

Sistem operasi yang dibuat pada tugas besar ini lebih cenderung berfokus kepada bagaimana sistem operasi modern melakukan management sebuah resource, dan pada milestone 2 ini: Disk management. Tentunya semua step yang dijelaskan pada tugas besar ini hanyalah dasar dan tidak memperhatikan konsiderasi desain seperti keamanan sistem.

Untuk file system, selain memang didesain untuk mengenalkan dan mengajari cara kerja file system sistem operasi, tugas ini juga bisa dianggap sebagai latihan untuk mengimplementasikan high-level design ke implementation detail dengan code. Tentunya hal ini skill yang diperlukan untuk software engineer, entah low-level seperti OS, web dev (mentranslasikan desain “figma” ke kode FE atau desain infrastruktur BE), dan pembuatan software in general.

Namun, tim asisten berharap, dengan bagian-bagian opsional yang ditambahkan pada guidebook dapat memberikan gambaran seberapa kompleks sistem operasi modern yang kita gunakan sehari-hari dan **taken for granted**. Selain itu beberapa contoh konkrit yang diperlihatkan juga diharapkan *sparking curiosity* untuk mempelajari lebih dalam.

Referensi Tambahan

Interrupt

1. <https://wiki.osdev.org/PIC>
2. https://wiki.osdev.org/Inline_Assembly/Examples

Keyboard Driver

1. https://wiki.osdev.org/PS/2_Keyboard
2. <https://www.win.tue.nl/~aeb/linux/kbd/scancodes-10.html#scancode sets>
3. <https://www.win.tue.nl/~aeb/linux/kbd/scancodes-1.html>
4. https://www-ug.eecg.toronto.edu/msl/nios_devices/datasheets/PS2%20Keyboard%20Protocol.htm
5. <https://www.asciitable.com/>
6. https://users.utcluj.ro/~baruch/sie/labor/PS2/Scan_Codes_Set_1.htm

File System

1. <https://wiki.osdev.org/FAT>
2. <https://www.tavi.co.uk/phobos/fat.html>
3. <https://www.win.tue.nl/~aeb/linux/fs/fat/fat-1.html>
4. [FSU - FAT32 Locating files & dirs](#) - Florida State University, FAT32 project
5. <http://www.osdever.net/documents/fatgen103.pdf> - Microsoft official FAT32 specification
6. https://en.wikipedia.org/wiki/File_Allocation_Table
7. <https://learnitonweb.com/2020/05/22/12-developing-an-operating-system-tutorial-episode-6-ata-pio-driver-osdev/>
8. https://wiki.osdev.org/ATA_read/write_sectors
9. <https://github.com/ApplePy/osdev/blob/master/FAT.c>
10. [wikipedia/Design of the FAT file system#Directory_entry](#) - Directory entry details
11. [Harvard - FAT32 File structure](#)
12. <https://wiki.osdev.org/User:Requimrar/FAT32>

Extras

1. **Ben, Ben, Ben, Ben, and Grant!**
[youtube/@BenEater](#) - Why writing code in IDE when you can design ISA with paper
[youtube/@3blue1brown](#) - Manim 10¹⁰/10, Grant fanboy here
2. No typedef, melestarikan pedoman Linus. Here, kitab C coding style dari Linus:
<https://www.kernel.org/doc/Documentation/process/coding-style.rst>