

Book I: Protected Mode x86

IF2130 - Sistem Operasi

Edisi 3

oleh

Asisten Laboratorium Sistem Paralel dan Terdistribusi & 13519214



Overview

Dokumen ini akan digunakan sebagai panduan penggerjaan dan referensi tugas besar IF2130 Sistem Operasi - 2025. Untuk melakukan navigasi dengan cepat, gunakan hyperlink yang ada pada [Table of Contents](#). Selain navigasi menggunakan daftar isi, gunakan juga fitur **Outline** yang terdapat pada bagian kiri Google Docs.

Dokumen ini akan **mengasumsikan pembaca memiliki pemahaman baik** dengan software engineering dasar (Pemrograman, struktur data, mencari dan membaca dokumentasi, dan lain-lain).

Buku ini akan mencakup penjelasan untuk semua spesifikasi IF2130 dan istilah relevan. Bagian utama (**Header Hitam dengan angka**) akan berfokus untuk memandu penggerjaan dan menjelaskan overview dari konsep-konsep penting. Penjelasan lebih detail, fun fact, dan deep dive untuk istilah dan konsep akan ditandai dengan **Header Biru**.

Internal hyperlink akan ditandai dengan [Text Bold](#).

Preface

Percaya atau tidak, sistem operasi pada komputer menyediakan sangat banyak layanan dan fitur yang digunakan sehari-hari baik pengguna awam maupun developer. Namun sayangnya hampir semua literatur di Indonesia menjelaskan OS secara “magical blackbox”; OS menyediakan fitur A, B, C *and that's it.*

Sistem operasi menjadi salah satu subjek paling unik dibandingkan mata kuliah lain. Dimana mata kuliah lain berdiri diatas abstraksi tingkat tinggi, OS akan mempelajari langsung **bagaimana sebenarnya komputer bekerja** dan memanipulasinya hingga memenuhi keinginan.

Jika dipelajari tanpa motivasi, materi OS seperti [Process & Thread](#), [Paging](#), [Scheduling](#) akan terasa hampa. Padahal materi-materi tersebut menjadi konsep dasar dari banyak hal-hal yang tidak akan tercover oleh mata kuliah seperti

- Interaksi [Kernel-Hardware](#) → [Device Driver](#)
- [Memory](#), [Process](#) manipulation, [Hooking](#) → [Memory Manipulation: Game Cheat](#)
- [Kernel-User Privileges](#) & Protection Ring → Malware & Information Security
- [File System](#) & [Memory Management](#) → [Digital Forensic](#)
- Memahami [FPU & AVX](#) → Optimisasi kinerja Grafkom & ML

Buku ini ditulis sepenuhnya dengan *tangan* pada 2025 dan didasari alasan *konyol* untuk membuat pembaca tertarik dan termotivasi untuk mempelajari OS. Dengan membuka blackbox dan menjelaskan inner working dari OS, diharapkan pembaca akan menemukan sebuah *dunia baru*.

Catatan kecil dari penulis

Buku ini merupakan iterasi ke-3. Beberapa perubahan yang ada berdasarkan *feedback* yang diterima setiap akhir tugas besar IF2130. Jika memiliki kritik dan saran untuk panduan ini, gunakan form peer assessment dan K&S pada akhir tugas besar atau mengirim langsung ke brshlckd@gmail.com.

Edisi ke-3 melengkapi buku edisi kedua dengan edisi kedua merupakan hasil tulis ulang awal hingga akhir dari edisi pertama. Dari hasil evaluasi asisten, meskipun guidebook edisi sebelumnya sudah membantu cukup banyak, masih banyak istilah yang tidak dijelaskan sehingga membuat pembaca bingung. Penulisan ulang pada edisi ke-2 mencoba untuk mengurangi istilah yang kurang penting dan menjelaskan lebih dalam setiap istilah yang ada.

Buku ini ditulis sebagai bagian dari personal project. Semua hasil tulisan *di-fact check* oleh penulis sendiri sehingga kemungkinan besar akan memiliki bias dan misinformasi yang tidak terlihat oleh penulis. Gunakan kontak yang sama dengan kritik dan saran untuk melaporkan misinformasi ataupun kesalahan dalam penulisan.

Special thanks to 13520103 for init, 13521095 for Ch. 3 & various sections, Dosen Lab Sister, Sister '19, '20, '21, '22, '23!

*Dedicated to IF2130, Low Level Programming, & OS
Anyway, Good luck, Have Fun!*
Penulis

Table of Contents

Overview.....	1
Preface.....	2
Catatan kecil dari penulis.....	2
Table of Contents.....	4
Chapter Dependencies.....	9
Ch. 0 - Toolchain, Kernel, GDT.....	10
0.1. Repository & Toolchain.....	11
0.1.1. Repository & Manual.....	11
0.1.2. Toolchain & Visual Studio Code.....	12
0.2. Kernel.....	14
0.2.1. C Kernel & Linker.....	15
• Booting Sequence & GRUB.....	17
• Boot Sector, Bootstrap, MBR, GPT.....	18
0.2.2. Image Creation & Automation.....	19
0.2.3. Running OS in QEMU.....	21
• x86 Memory Addressing: Virtual, Linear, Physical.....	22
0.3. Global Descriptor Table.....	23
0.3.1. Data Structure: Segment Descriptor.....	24
0.3.2. GDT & GDTR Definition.....	25
0.3.3. Load GDT.....	27
0.4. - Extras: Computer.....	28
• x86 & Von Neumann Architecture.....	28
• Bitness.....	29
• FPU & AVX.....	30
Ch. 1 - Framebuffer, Interrupt, Driver.....	31
1.0. Short Note: Kernel Development.....	32
• Coding Style & Compiler Warning.....	33
• C: Include Guard.....	34
1.1. Driver Text Framebuffer.....	35
1.1.1. framebuffer_write().....	37
1.1.2. framebuffer_set_cursor().....	37
1.1.3. framebuffer_clear().....	37
1.1.4. Test: Framebuffer.....	38
• Memory-mapped I/O & Port-mapped I/O.....	39
• C: Subscript Operator.....	40
1.2. Interrupt.....	41
• Hardware & Software Interrupt.....	42
• CPU Exception.....	43
• Familiar Exception.....	44
■ Double Fault.....	44

■ Triple Fault.....	44
■ General Protection Fault.....	45
■ Breakpoint.....	45
1.2.1. IRQ Remapping.....	46
● 8259 PIC & IBM PC.....	48
1.2.2. Interrupt Descriptor Table.....	49
1.2.3. Interrupt Service Routine.....	50
● x86: Handling Intra & Inter Privilege Interrupt.....	51
1.2.4. Load IDT & Testing Interrupt.....	52
1.3. Keyboard Driver.....	54
● Device Driver.....	55
1.3.1. IRQ1 - Keyboard Controller.....	56
1.3.2. Keyboard ISR.....	57
1.3.3. Keyboard Interface.....	59
Tips: Keyboard Driver.....	61
● Keyboard Scancode.....	63
Ch. 2 File System: EXT2 - IF2130 Edition.....	64
● File System.....	65
● Memory Hierarchy.....	66
2.1. Disk Driver.....	67
● ATA: PATA & SATA.....	69
2.2. Disk Image.....	70
● Disk Addressing: LBA & CHS.....	72
● EXT2: Block & Inode.....	73
2.3. Volatile & Non-Volatile Memory.....	74
2.4. FS: Design of EXT2 - IF2130 Edition.....	75
2.4.1. Overview & Terminology.....	76
2.4.2. Block Group.....	77
2.4.3. File & Directory.....	79
2.4.3.1 File EXT2.....	80
2.4.3.2 Directory EXT2.....	81
2.4.4. Root Directory & Interaction.....	82
2.4.5. Design & Constraint.....	85
2.5. FS: Initializer.....	86
2.6. FS: CRUD.....	87
2.6.1. Read.....	88
2.6.2. Write.....	89
2.6.3. Delete.....	90
2.6.4. CRUD Implementation & Testing.....	91
Tips: File System.....	93
● Standardization Mess: SI & Binary Prefix.....	94

• Controller, Driver, Scam.....	95
• Fragmentation.....	96
2.7. - Extras: Hardware.....	97
• Direct Memory Access.....	97
• Device Recognition & Hot Plug.....	98
• Digital Forensic.....	99
Ch. 3 - Paging, User Mode, Shell.....	100
3.1. Paging.....	101
3.1.0. Paging: Overview.....	102
• MMU & TLB.....	103
• Page Table Structure.....	104
• Page Table: Address Translation.....	105
3.1.1. Data Structure: Page Table.....	106
3.1.2. Higher Half Kernel.....	107
• Modern OS: Virtual Address Space.....	109
3.1.3. Activate Paging.....	110
3.1.4. Memory Manager.....	111
3.1.4.1. Frame Allocator.....	112
3.1.4.2. Frame Deallocator.....	112
3.1.4.3. Free Memory Check.....	112
• Physical Memory Hole.....	113
Tips: Paging.....	114
Extra: Paging.....	115
Frequent Issue: Paging.....	116
• Memory Paging.....	117
• Windows: Memory & Performance.....	118
3.2. User Mode.....	119
3.2.1. External Program: Inserter.....	120
• Modular & Reusability.....	122
3.2.2. GDT: User & Task State Segment Descriptor.....	123
3.2.2.1. Task State Segment.....	124
3.2.2.2. User Segment Descriptor.....	125
3.2.3. Simple User Program.....	127
• C: Program Entrypoint.....	129
3.2.4. Execute Program.....	130
3.2.5. Launching User Mode.....	132
Tips: User Mode.....	133
Frequent Issue: User Mode.....	134
3.3. Shell.....	135
3.3.1. System Calls.....	136
• Security: User Mode & System Calls.....	137

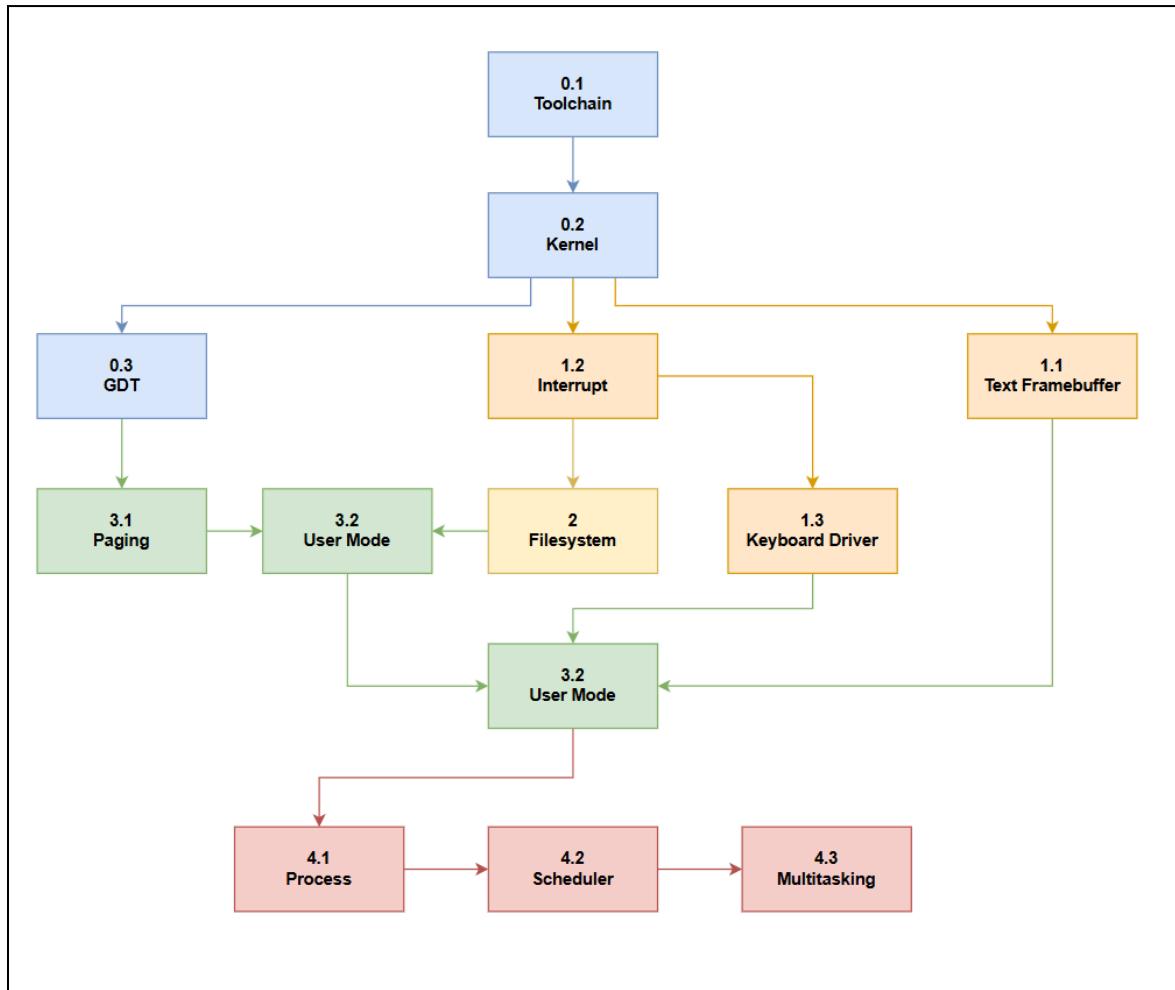
3.3.1.1. Designing System Calls.....	138
3.3.1.2. Inter-Privilege Interrupt Syscall.....	139
3.3.1.3. Calling Syscall.....	140
3.3.2. Command Line Interface.....	141
3.3.2.1. Shell: Debugger.....	142
3.3.2.2. Shell: Specification.....	144
Tips: Shell.....	145
• EXT2, Graph, Tree.....	146
Extra - Ch. 3: Security.....	147
• Code Injection.....	147
• Hooking.....	148
• Memory Manipulation.....	149
Ch. 4 - Process, Scheduler, Multitasking.....	150
4.1. Process.....	151
4.1.0. Correction: Kit Chapter 1.....	152
4.1.1. Multi Virtual Address Space.....	153
4.1.2. Process Control Block.....	154
4.1.2.1. Process Context.....	156
4.1.2.2. Process State.....	157
4.1.2.3. Memory & Metadata.....	158
4.1.3. Process Creation.....	159
4.1.3.1. Virtual Address Space.....	160
4.1.3.2. Load Executable.....	161
4.1.3.3. Context Initialization.....	162
4.1.3.4. Process Metadata & Cleanup.....	163
4.1.4. Process: Init.....	164
Frequent Issue: Process.....	165
4.2. Scheduler.....	166
4.2.1. Task Scheduler.....	167
4.2.1.1. IRQ0 - Timer Interrupt & Scheduler Initialization.....	168
4.2.1.2. Scheduling Algorithm.....	169
4.2.2. Context Switch.....	170
4.2.2.1. CPU Register.....	171
4.2.2.2. Virtual Address Space & Process.....	172
4.2.3. Test: Single Process.....	173
4.3. Multitasking.....	174
4.3.1. Process Entrypoint & Exit.....	175
4.3.2. Process Management & Command.....	176
4.3.3. Clock.....	177
4.3.4. External Application.....	178
4.3.5. Environment Variables.....	179

4.3.6. Seperate Shell Commands.....	180
4.3.7. Grand Finale.....	184
Ch. 4 - Extras: Operating System.....	185
• Kernel.....	185
• Bare Metal & Embedded System.....	186
• Operating System.....	187
Epilogue.....	188
References.....	189
Chapter 0: Toolchain, Kernel, GDT.....	189
Chapter 1: Interrupt, Driver.....	190
Chapter 2: File System.....	191
Chapter 3: Paging, User Mode, Shell.....	192
Chapter 4: Process, Scheduler, Multitasking.....	193

Chapter Dependencies

Panduan ini menyediakan alur pengerjaan yang disarankan sesuai dengan urutan chapter terkecil hingga terbesar. Susunan chapter didasarkan dengan balancing mengikuti difficulty curve tertentu dan dependency antar chapter agar pengerjaan tidak mengalami kesulitan hingga stuck.

Disarankan untuk mengikuti urutan yang ada pada buku ini, tetapi Chapter 1 dan 2 cukup fleksibel sehingga dapat juga dikerjakan secara non-linear. Berikut adalah dependency tree panduan ini



Dependency tree dari setiap chapter

Beberapa bagian pada Chapter 1 dan 2 dapat dikerjakan secara *konkuren*, dan jika situasi juga mendukung dapat dikerjakan secara *paralel*. Sebagian subchapter ([File System](#) & [Shell](#)) juga dapat dibagi menjadi subtask lagi (interfaces berbeda, command shell bermacam-macam).

Semua fitur dan subsistem **Chapter 0, 1, 2, dan 3** akan menjadi prerequisite dan digunakan pada **Chapter 4**. Disarankan untuk mengecek kembali implementasi agar terhindar dari bug dan issue sebelum melanjutkan ke Chapter 4.

Ch. 0 - Toolchain, Kernel, GDT

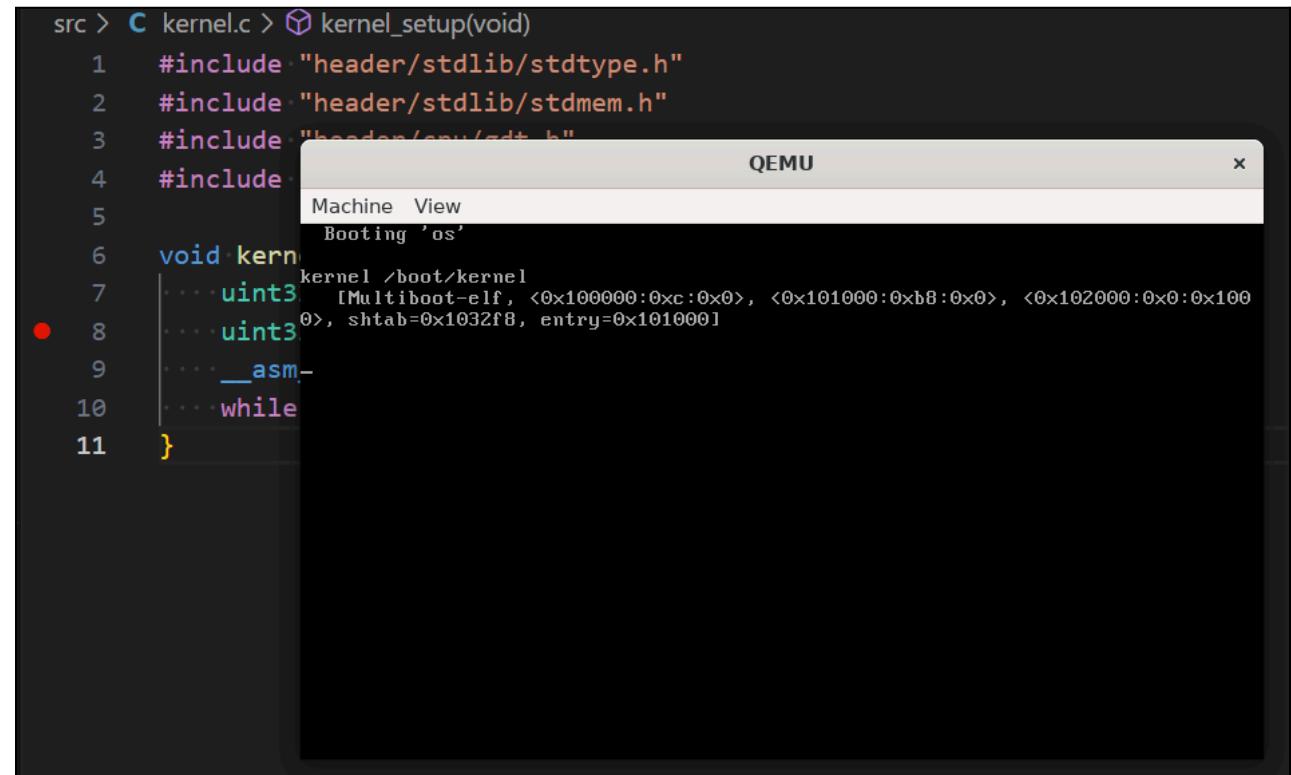
Chapter 0 adalah introduction awal ke kernel development. Sebagian besar chapter ini akan dihabiskan untuk setup [Repository & Toolchain](#), membuat [Kernel](#) dasar yang dapat dicompile, dan [Global Descriptor Table](#) sebagai penutup.

Buku ini akan menggunakan **Linux** dan **x64** sebagai dev environment. Panduan telah diuji pada **arsitektur x64, Windows 10 & 11 + WSL2 Ubuntu 20.04/22.04, Ubuntu Desktop 20.04 & Arch Linux**.

Untuk Apple Silicon, akan disediakan alternate toolchain dan workflow yang dapat digunakan untuk melakukan development pada Apple Silicon. Panduan tersebut diuji pada **MacOS Ventura 13.2.1**.

Jika mengalami kendala ketika memasang tools, dapat dicoba untuk mencari solusi terlebih dahulu menggunakan search engine dan dapat menggunakan sheet QnA jika masih mengalami kendala.

Expect pada akhir **Chapter 0** sistem operasi telah dapat dijalankan menggunakan QEMU.



The screenshot shows a terminal window on the left displaying the `kernel.c` source code. The code includes includes for stdlib, stdtype, stdmem, and gdt.h, and defines a `kernel` function. A red dot marks the line `8: uint32_t shtab=0x1032f8, entry=0x101000;`. To the right of the terminal is a QEMU graphical interface window titled "QEMU". The window has a menu bar with "Machine" and "View" options, and a sub-menu "Booting" with "os" selected. The main area of the QEMU window is currently black, indicating the system is booting or has just booted.

```
src > C kernel.c > kernel_setup(void)
1  #include "header/stdlib/stdtype.h"
2  #include "header/stdlib/stdmem.h"
3  #include "header/cpu/gdt.h"
4  #include
5
6  void kernel
7  {
8      uint32_t shtab=0x1032f8, entry=0x101000;
9
10     __asm__ __attribute__((naked))
11     while(1)
12 }
```

0.1. Repository & Toolchain

Kerangka dasar untuk sistem operasi telah disediakan dalam [Repository Template & Manual](#). Buku ini akan memandu penggerjaan sebagian besar untuk tahap [Repository & Toolchain](#) dan [Kernel](#).

0.1.1. Repository & Manual

Untuk memulai penggerjaan, buatlah repository menggunakan link Github Classroom yang disediakan pada [Hub](#). Repository yang telah dibuat akan secara **otomatis menggunakan template milestone 0** sebagai template dasar. Kit milestone yang lain dapat diakses pada directory [.kit/](#).

Repository dengan template dasar akan berbentuk seperti berikut

```
.vscode/
├── launch.json
├── tasks.json
└── settings.json
.kit/
├── ch1/
├── ch2/
├── ch3/
└── ch4/
bin/
└── .gitignore
other/
└── grub1
src/
├── header/
│   ├── kernel-entrypoint.h
│   └── cpu/
│       └── gdt.h
└── stdlib/
    ├── string.h
    └── string.c
└── kernel-entrypoint.s
.gitignore
makefile
README.md
```

Sebelum melanjutkan, **sangat direkomendasikan untuk mendownload kitab Intel x86** sebagai pelengkap buku ini. Link untuk [Kitab Intel x86 dan x64 Volume 3A - Intel® 64 and IA-32 Architectures Developer's Manual Volume 3A System Programming Guide](#) tersedia pada referensi.

0.1.2. Toolchain & Visual Studio Code

Panduan ini akan menggunakan kombinasi **Visual Studio Code + WSL2 Ubuntu 20.04/22.04**. Template awal akan menyediakan konfigurasi dasar untuk vscode yang dapat mempermudah proses build & debugging. Lakukan eksplorasi secara mandiri jika menggunakan text editor / IDE selain vscode.

Sebelum melanjutkan ke pemasangan tool, pastikan **WSL2 Ubuntu 20.04/22.04** telah terpasang dan dapat berjalan. Panduan untuk pemasangan WSL2 ini terdapat pada dokumen berikut [WSL Guide](#)

Untuk pengguna **Apple Silicon (M1, M2, etc)** dapat menggunakan alternate toolchain yang disediakan pada dokumen berikut

 IF2130 - Apple Silicon Toolchain 2025

Berikut adalah tool-tool pada Linux yang akan digunakan untuk development sistem operasi

- **Netwide Assembler** (<https://www.nasm.us/>)
Compiler assembly x86 untuk kode yang membutuhkan instruksi asm langsung
- **GNU C Compiler** (<https://man7.org/linux/man-pages/man1/gcc.1.html>)
Compiler C untuk sistem operasi
- **GNU Linker** (<https://linux.die.net/man/1/ld>)
Linker object code hasil kompilasi
- **QEMU - System i386** (<https://www.qemu.org/docs/master/system/target-i386.html>)
Emulator-VM utama untuk menjalankan sistem operasi
- **GNU Make** (<https://www.gnu.org/software/make/>)
Build tools untuk sistem operasi
- **genisoimage** (<https://linux.die.net/man/1/genisoimage>)
Tool untuk pembuatan *image* sistem operasi
- **GDB** (<https://man7.org/linux/man-pages/man1/gdb.1.html>)
Debugger untuk melakukan dynamic debugging kernel

Pasang semua tool diatas menggunakan command berikut

```
sudo apt update  
sudo apt install -y nasm gcc qemu-system-x86 make genisoimage gdb
```

Selanjutnya adalah memasang **Visual Studio Code**

1. Install **Microsoft Visual Studio Code**

Untuk Windows dapat menggunakan perintah berikut pada terminal cmd

```
winget install microsoft.visualstudiocode
```

Bila anda menggunakan WSL, ikuti [instruksi berikut](#) untuk menginstall VSCode di WSL.

2. Tambahkan ekstensi berikut pada VS Code

- [C/C++ Extension Packs](#) by Microsoft
- [Remote Development](#) dan [WSL](#) (Bila menggunakan WSL)

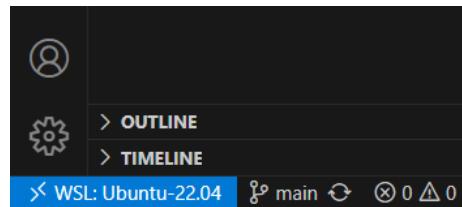
Pemasangan ekstensi dapat menggunakan perintah berikut pada terminal

```
code --install-extension ms-vscode.cpptools-extension-pack  
code --install-extension ms-vscode-remote.remote-wsl
```

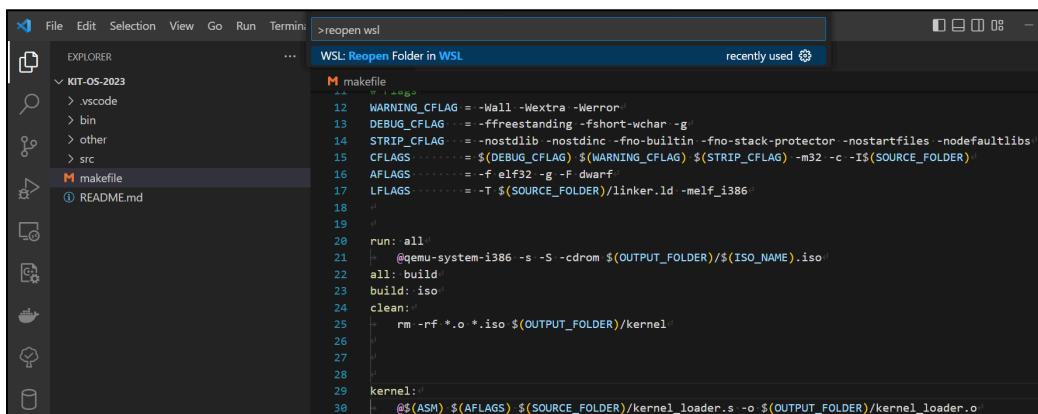
3. Navigasi terminal ke repository dan buka menggunakan perintah `code .` seperti berikut

```
barcode@LAPTOP-1ARP10R0:~/keos$ ls  
bin  makefile  other  README.md  src  
barcode@LAPTOP-1ARP10R0:~/keos$ code .  
barcode@LAPTOP-1ARP10R0:~/keos$ |
```

4. Akan terbuka VS Code yang telah terattach ke WSL. Bagian pojok kiri bawah akan tertulis WSL: <Distro>

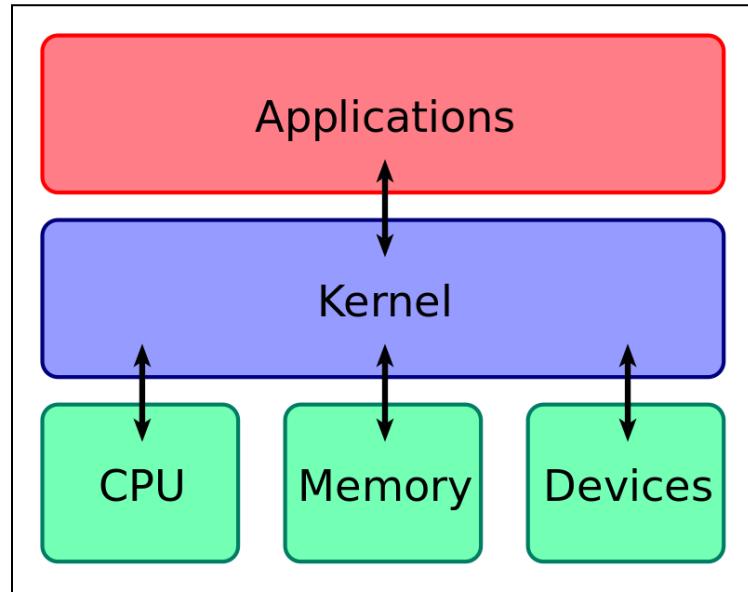


Alternatif: Buka VS Code pada folder, gunakan **Ctrl + Shift + P** dan ketik **reopen wsl**



0.2. Kernel

Kernel adalah bagian sistem operasi yang bertanggung jawab untuk menyediakan layanan seperti [System Calls](#), [Memory Manager](#), [Task Scheduler](#), dan hal lain yang berhubungan dengan hardware. **Kernel merupakan inti dari sistem operasi.**



Kernel, Sumber: [Kernel - Wikipedia](#)

Ilustrasi di atas adalah ilustrasi umum yang menggambarkan posisi kernel pada abstraksi komputer. Berbeda dengan penjelasan umum kernel, buku ini juga akan memandu implementasi: interaksi kernel dengan CPU pada bagian [GDT](#) & [IDT](#), dengan memory pada [Paging](#), dengan devices pada [Keyboard Driver](#), dan memberikan layanan [System Calls](#) ke [User Mode](#).

Template repository telah menyediakan kode-kode awal yang digunakan sebagai dasar sistem operasi. Kode-kode yang disediakan adalah kerangka dasar untuk membuat kernel sistem operasi. Bagian ini akan membuat kode kernel dasar yang dapat dijalankan pada GRUB.

Awal penggeraan dimulai dengan membuat [C Kernel & Linker](#), dilanjutkan dengan [Disk Image Creation & Automation](#), dan diakhiri dengan [Running OS in QEMU](#). Pada akhir bagian ini sistem operasi harusnya sudah dapat dijalankan dengan QEMU.

0.2.1. C Kernel & Linker

Booting Sequence komputer bermula dari rangkaian pemanggilan bootloader GRUB dan berakhir pada diserahkannya eksekusi instruksi kepada kernel. Kode kernel yang akan dieksekusi pertama terdapat pada source file assembly **kernel-entrypoint.s**, yaitu prosedur **loader**.

Prosedur loader akan menyiapkan Call Stack (Stack yang digunakan oleh register **esp** dan **ebp** untuk memanggil fungsi) yang diperlukan oleh bahasa C dan memanggil fungsi C bernama **kernel_setup()**. Kode assembly **kernel-entrypoint.s** sudah siap untuk digunakan dan pengerajan dilanjutkan untuk membuat definisi fungsi **kernel_setup()** dengan bahasa C. Tipe data boolean disediakan oleh bahasa C pada **stdbool.h**.

Buatlah file **kernel.c** yang kurang lebih berisikan kode berikut

```
kernel.c

#include <stdint.h>
#include "header/cpu/gdt.h"
#include "header/kernel-entrypoint.h"

void kernel_setup(void) {
    uint32_t a;
    uint32_t volatile b = 0x0000BABA;
    __asm__ ("mov $0xCAFE0000, %0" : "=r"(a));
    while (true) b += 1;
}
```

Instruksi kernel hasil kompilasi yang berbentuk binary executable wajib di-load atau dimasukkan pada memori **0x100000** ke atas sehingga diperlukan untuk mengatur linker terlebih dahulu.

Memori **0x0** hingga **0xFFFF** digunakan untuk GRUB dan Memory Mapped I/O sehingga tidak dapat digunakan oleh kernel.

Buatlah file baru **linker.ld** di folder **/src/** yang berisikan lokasi & alignment berikut

linker.ld

```
ENTRY(loader)                      /* the name of the entry label */

SECTIONS {
    . = 0x00100000;           /* the code should be loaded at 1 MB */

    .multiboot ALIGN (0x1000) : /* align at 4 KB */
    {
        *(.multiboot)         /* GRUB multiboot header */
    }

    .text ALIGN (0x1000) : /* align at 4 KB */
    {
        *(.text)              /* all text sections from all files */
    }

    .rodata ALIGN (0x1000) : /* align at 4 KB */
    {
        *(.rodata*)           /* all read-only data sections from all files */
    }

    .data ALIGN (0x1000) : /* align at 4 KB */
    {
        *(.data)               /* all data sections from all files */
    }

    .bss ALIGN (0x1000) : /* align at 4 KB */
    {
        *(COMMON)             /* all COMMON sections from all files */
        *(.bss)                /* all bss sections from all files */
    }
}
```

Nantinya kernel akan di compile dan menghasilkan binary executable. Binary executable dan GRUB, dan informasi letak executable akan dimasukkan ISO yang dapat dibaca QEMU. Informasi letak executable akan ditulis pada file **menu.lst** seperti berikut

menu.lst

```
default 0
timeout 0

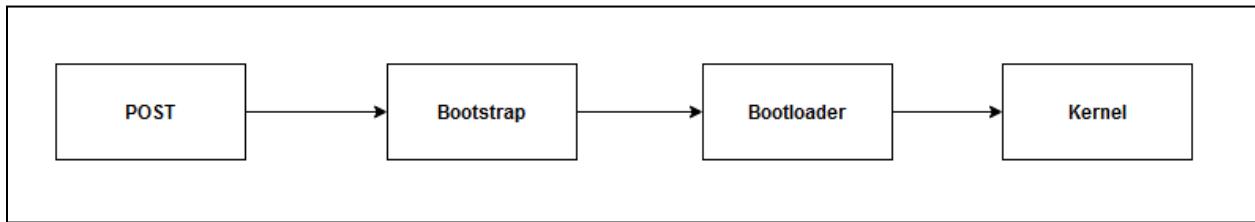
title os
kernel /boot/kernel
```

Konfigurasi diatas akan menambahkan OS yang dibuat dengan nama “**os**” dan binary kernel yang berada pada **/boot/kernel**.

Dokumentasi lebih lengkap dapat dilihat pada [GNU GRUB Manual 0.97](#).

• Booting Sequence & GRUB

Saat komputer pertama kali menyala, komputer akan menjalankan [Power-on self-test \(POST\)](#) dan jika berhasil BIOS akan melanjutkan untuk mengeksekusi program [Bootstrap](#) yang terdapat di [Master Boot Record \(MBR\) / Boot Sector](#). MBR merupakan sektor pertama (Hanya berukuran 512 bytes) pada non-volatile partitioned storage (seperti HDD, removable drive, etc).



Booting sequence yang umum pada komputer

Bootstrap yang berukuran sangat kecil biasanya hanya bertugas untuk membaca **Bootloader** dari non-volatile memory (HDD, SSD, etc) ke volatile memory (RAM) dan mengeksekusi kode bootloader pada RAM.

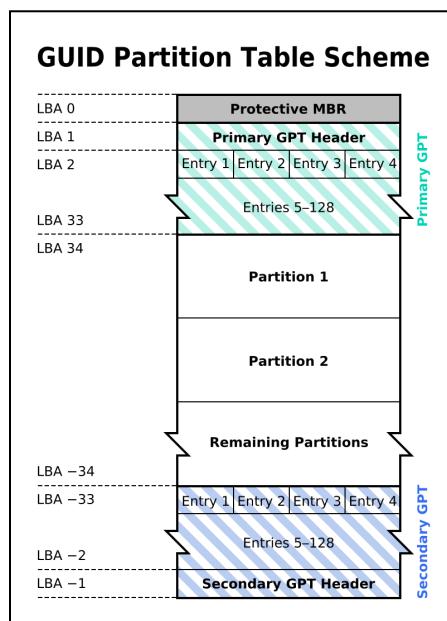
Bootloader memiliki ukuran cukup besar untuk dapat menyediakan beberapa fungsionalitas sederhana seperti selector untuk multiple OS, command line, hardware setup, dan lain-lain. Setelah bootloader selesai melakukan tugasnya, eksekusi sistem akan diserahkan kepada kernel sistem operasi.

Buku ini akan menggunakan GRUB sebagai bootloader, tepatnya GRUB Legacy version 0.95 (eltorito). Bootloader GRUB sudah disediakan pada template **other/grub1**.

- **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 sektor / block pertama dalam secondary storage. Jika boot sector mengalami corrupt kemungkinan besar akan menyebabkan **kegagalan booting** komputer.

Kode yang ada pada boot sector dinamai **Bootstrap**. Bootstrap merupakan kode pertama yang akan dieksekusi oleh CPU sesuai flow yang dideskripsikan pada [Booting Sequence](#). Komputer hanya memastikan 2-byte signature (**0xAA55**) pada akhir block dan mempercayai penuh kode yang dieksekusi. Terdapat tipe malware [Bootkit](#) (varian dari Rootkit) yang memiliki target untuk menginfeksi kode bootstrap pada komputer.



GPT partition, Sumber: [GUID Partition Table - Wikipedia](#)

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

GUID Partitioned Table (GPT) yang berbeda dari [GPT](#) di-ChatGPT merupakan sistem partisi yang lebih modern. Singkatnya GPT bisa dikatakan ekstensi dari MBR. Boot sector pada media dengan GPT akan berisikan MBR yang valid bernama **Protective MBR**. Hal ini mencegah sistem operasi yang tidak mengenal partisi GPT menganggap media penyimpanan kosong. Sector yang ditunjuk protective MBR inilah yang merupakan isi utama dari sistem partisi GPT. GPT header dapat menyimpan informasi yang lebih banyak dibandingkan partisi MBR.

0.2.2. Image Creation & Automation

Sekarang sistem operasi dapat dikompilasi dan link menjadi satu executable **ELF32**, format ini dapat dibaca dan dijalankan langsung oleh GRUB. Bagian ini akan melakukan otomasi proses kompilasi OS dan membuat image untuk OS.

Berikut adalah command yang akan digunakan untuk melakukan kompilasi sistem operasi

```
gcc -ffreestanding -fshort-wchar -g -nostdlib -fno-builtin -fno-stack-protector  
-nostartfiles -nodefaultlibs -Wall -Wextra -Werror -m32 -c -Isrc src/kernel.c -o  
bin/kernel.o
```

```
nasm -f elf32 -g -F dwarf src/kernel-entrypoint.s -o bin/kernel-entrypoint.o
```

```
ld -T src/linker.ld -melf_i386 bin/kernel.o bin/kernel-entrypoint.o -o bin/kernel
```

Command diatas dapat dijalankan langsung pada root repository jika ingin mencoba kompilasi manual. Hasil kompilasi akan menghasilkan executable ELF32 bernama **kernel** pada folder **bin**.

Untuk menjalankan sistem operasi, kernel yang dibuat perlu masukkan kedalam disk image. Dengan menggunakan tool **genisoimage**, disk image iso dapat dibuat dengan menyusun folder yang akan digunakan sebagai struktur folder iso.

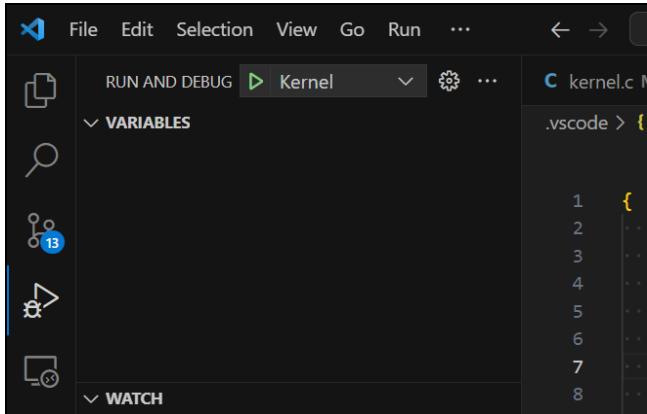
Folder **iso** akan dibuat pada folder **/bin/** dengan struktur seperti berikut

```
iso/  
└ boot/  
    └─ grub/  
        └─ menu.lst  
        └─ grub1  
            ; binary GRUB yang diberikan pada /other/  
            ; kernel executable dengan format ELF32  
        └─ kernel
```

Dengan susunan directory tree tersebut, perintah berikut akan membuat disk image iso dari sistem operasi (asumsi [Current Working Directory](#) adalah **parent** dari **iso** yaitu pada folder **/bin/**)

```
genisoimage -R  
    -b boot/grub/grub1  
    -no-emul-boot  
    -boot-load-size 4  
    -A os  
    -input-charset utf8  
    -quiet  
    -boot-info-table  
    -o OS2025.iso  
    iso
```

Setelah perintah dijalankan, sebuah image bernama **OS2025.iso** akan terbentuk pada folder **/bin/**. Repository template akan menyediakan folder **.vscode** yang memiliki konfigurasi untuk menjalankan command **make build** dan menjalankan QEMU. Konfigurasi ini dinamai "**Kernel**".



Konfigurasi “Kernel” pada panel Run and Debug

Oleh karena itu, proses kompilasi sebelumnya perlu untuk dimasukkan kedalam **makefile** agar dapat mengotomasi proses kompilasi kernel dan disk image. Kompilasi nantinya akan menggunakan hotkey **F5 (Start with debugging)** pada Visual Studio Code.

Berikut adalah potongan **makefile** yang disediakan template

```
makefile

kernel:
    @$(ASM) $(AFLAGS) src/kernel_loader.s -o bin/kernel_loader.o
# TODO: Compile C file with CFLAGS
    @$(LIN) $(LFLAGS) bin/*.o -o $(OUTPUT_FOLDER)/kernel
    @echo Linking object files and generate elf32...
    @rm -f *.o

iso: kernel
    @mkdir -p $(OUTPUT_FOLDER)/iso/boot/grub
    @cp $(OUTPUT_FOLDER)/kernel      $(OUTPUT_FOLDER)/iso/boot/
    @cp other/grub1                $(OUTPUT_FOLDER)/iso/boot/grub/
    @cp $(SOURCE_FOLDER)/menu.lst   $(OUTPUT_FOLDER)/iso/boot/grub/
# TODO: Create ISO image
    @rm -r $(OUTPUT_FOLDER)/iso/
```

Lengkapilah kedua komentar **TODO** pada makefile hingga sistem operasi dapat di-build dengan **make build**. Semua flags untuk compiler, assembler, dan linker telah didefinisikan dan disediakan pada template makefile. Berikut adalah contoh command untuk melakukan kompilasi

```
$(CC) $(CFLAGS) $(SOURCE_FOLDER)/kernel.c -o $(OUTPUT_FOLDER)/kernel.o
```

Sesuaikan **makefile** dengan perubahan yang dilakukan jika memindahkan lokasi file, mengganti nama, dan lain-lain sehingga proses build dapat berjalan.

0.2.3. Running OS in QEMU

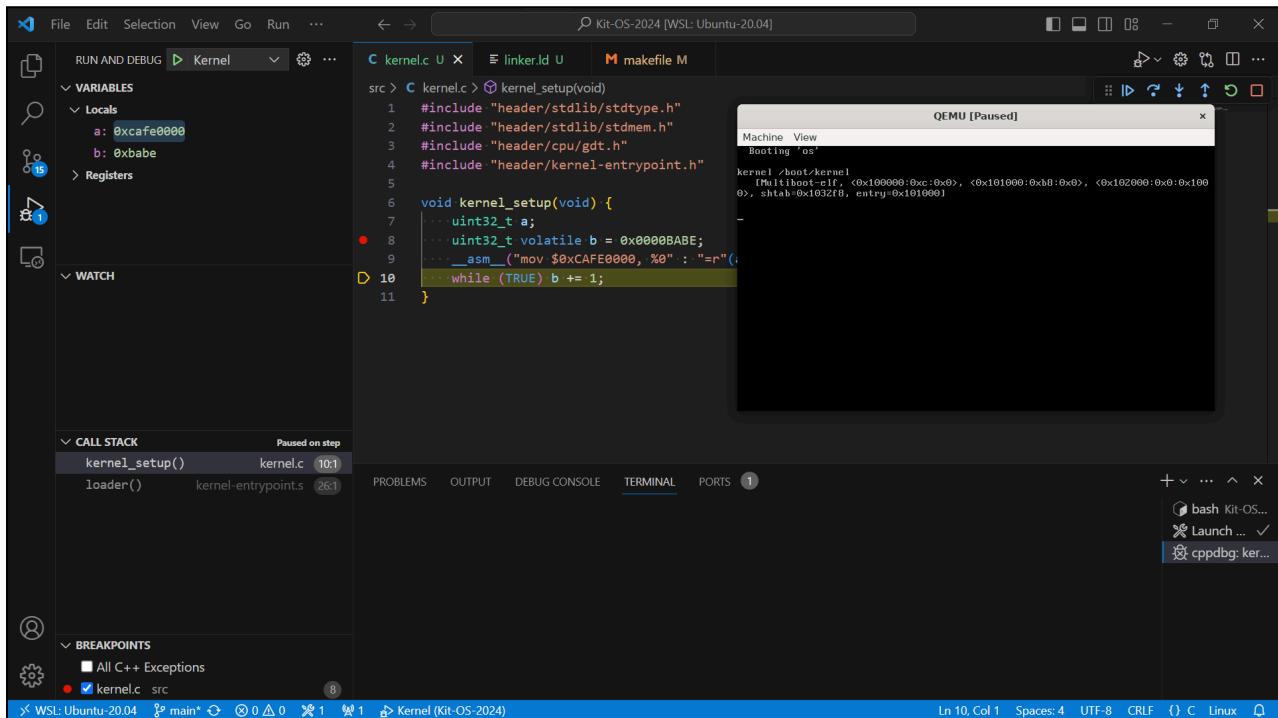
Image dalam format `.iso` yang telah dibuat dapat dijalankan menggunakan QEMU dengan perintah berikut pada folder `/bin/`

```
qemu-system-i386 -s -cdrom OS2025.iso
```

Flag `-s` digunakan untuk membuka gdb server pada `localhost:1234` sedangkan flag `-s` yang ada pada `.vscode/tasks.json` menghentikan eksekusi QEMU hingga debugger telah ter-attach dengan gdb server. Jika ingin OS langsung berjalan tanpa menunggu, hilangkan flag `-s` seperti perintah diatas.

Untuk menjalankan OS menggunakan Visual Studio Code, gunakan hotkey **F5** untuk menjalankan debugger dan **Shift + F5** untuk menutup debugger.

Tekan F5 pada vscode untuk menjalankan sistem operasi. Jika berjalan dengan baik, QEMU akan menampilkan tampilan seperti berikut



Eksekusi QEMU dengan vscode dan gdb debugger yang telah terattach

Jika window QEMU terlihat seperti diatas (dengan cursor blinking jika ada)

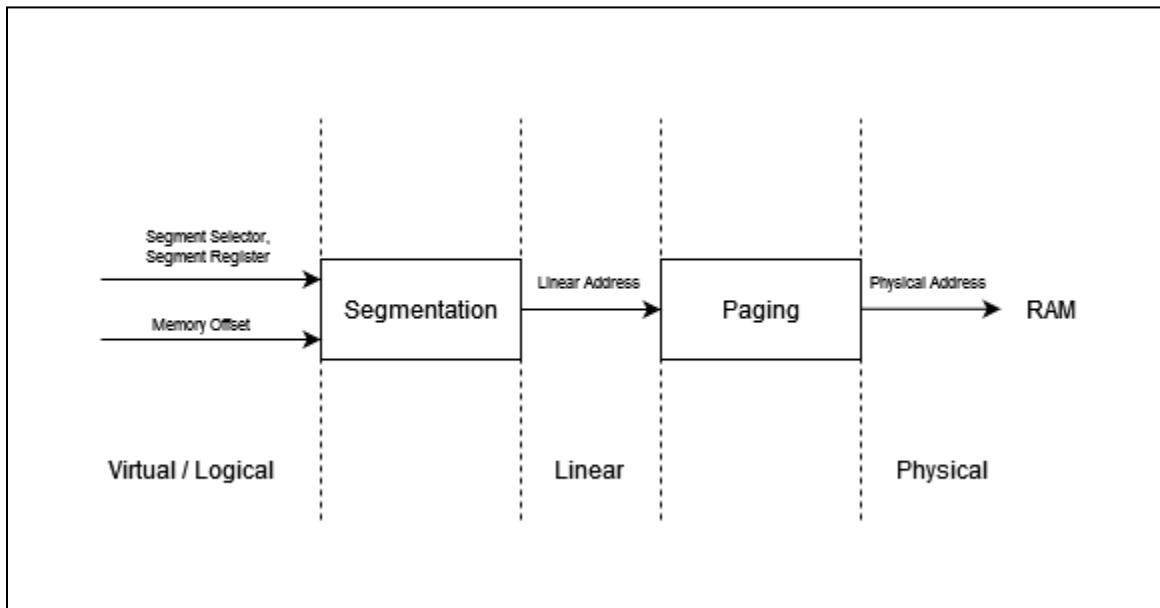
Selamat sistem operasi telah berjalan!

Sidenote 2025: Jika mengalami graphical glitch pada QEMU pada daerah window header, cobalah untuk melakukan update pada Visual Studio Code atau restart vscode.

• **x86 Memory Addressing: Virtual, Linear, Physical**

Pada arsitektur x86 terdapat 3 istilah untuk merefer memori:

- **Virtual / Logical Address**
- **Linear Address**
- **Physical Address**



Ilustrasi istilah memory addressing x86

Ilustrasi diatas memperlihatkan letak setiap address pada proses translasi dari virtual address ke physical. Sistem operasi modern x86 umumnya akan mengabaikan fitur segmentasi memori dan berfokus menggunakan **Paging** sebagai alat manajemen memori. Panduan ini juga akan membuat bagian segmentasi memori GDT secara sederhana.

Dengan mengabaikan fitur segmentasi memori, virtual address dan linear address akan selalu menunjukkan ke memori yang sama. GDT secara efektif hanya digunakan untuk menandai region memory dengan flag.

Paging merupakan konsep penting dari memory management pada sistem modern. Abstraksi virtual-physical memory memperbolehkan kernel untuk melakukan memory management yang lebih kompleks dengan mudah. Fitur **Memory Paging** menggunakan paging untuk menyimpan sementara data dari volatile memory (RAM) ke non-volatile memory seperti HDD dan SSD.

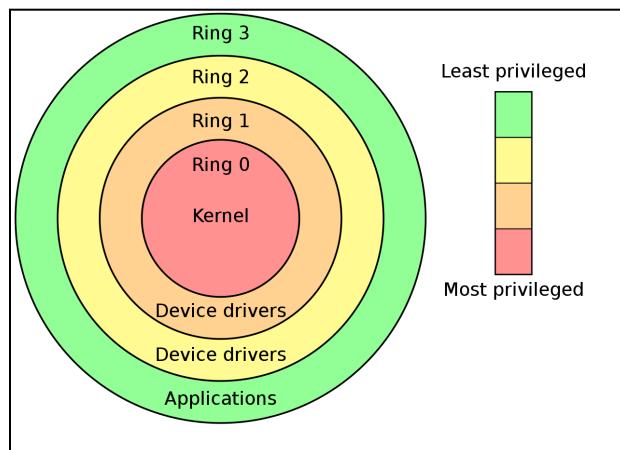
[Chapter 2](#) dari buku ini akan membahas lebih detail dan mengimplementasikan Paging.

0.3. Global Descriptor Table

Global Descriptor Table adalah sebuah tabel yang memiliki entry-entry bernama [Segment Descriptor](#). Setiap segment descriptor akan menyimpan informasi flag access level, tipe segment, base offset, dan lain-lain dari satu region linear address memory. **Protected Mode x86** membutuhkan GDT untuk privilege yang nantinya dibahas pada [Chapter 3 - User Mode](#).

GRUB akan secara otomatis menyalakan fitur protected mode x86 sebelum menyerahkan kontrol kepada kernel sehingga kernel hanya perlu untuk membuat dan load struktur data GDT.

Pada penggeraan ini, GDT hanya digunakan untuk menandai region memory dengan flag dan memenuhi requirement minimum untuk protected mode x86. Struktur [Protection Ring](#) yang akan digunakan adalah 2 ring dengan masing-masing mewakili **Kernel** dan [User Mode](#). Kernel akan menggunakan flag **Descriptor Privilege Level (DPL)** 0 dan User menggunakan DPL 3.



Sumber: [Protection Ring - Wikipedia](#)

Pada [Chapter 0](#), GDT hanya akan memiliki 3 segment descriptor yang diperlukan untuk protected mode: **Null & Kernel Code+Data Descriptor**.

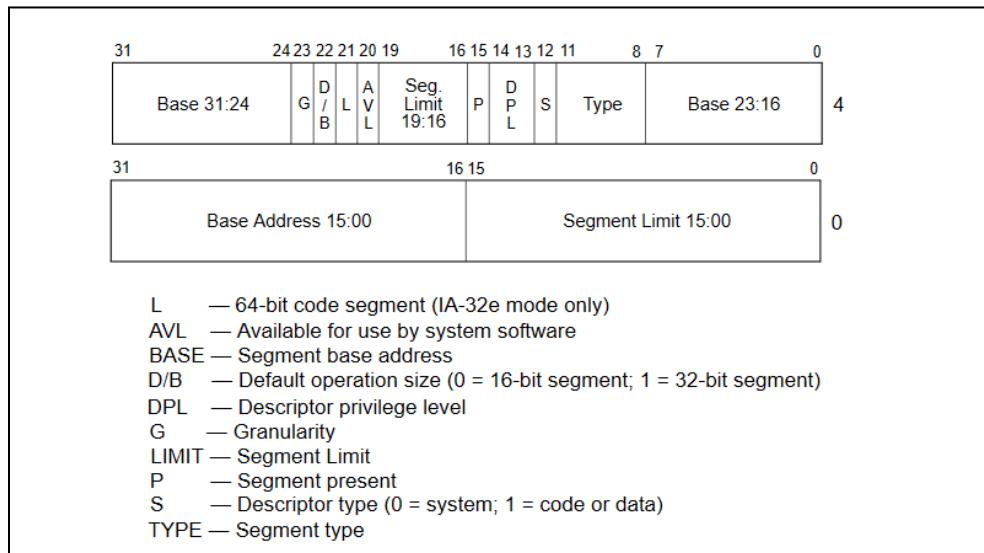
Chapter 2 akan membuat pemisahan kernel-user pada [User Mode](#) yang menambahkan segment descriptor pada GDT untuk [GDT: User & Task State Segment Descriptor](#).

Berikut referensi yang dapat digunakan untuk GDT:

- [Segment Descriptor - OSDev](#)
- [Global Descriptor Table - OSDev](#)
- [GDT Tutorial - OSDev](#)
- [Intel x86 Manual 3A - 3.4.3 Segment Registers - 3.5.1 Segment Descriptor Table](#)

0.3.1. Data Structure: Segment Descriptor

Berikut adalah definisi struktur data **Segment Descriptor** pada **Intel x86 Manual 3A**



Intel Manual Vol 3a - Chapter 3 - Figure 3-8 Segment Descriptors

Definisi & penjelasan setiap flags dan atribut lain struktur data Segment Descriptor dapat dicek lebih detail pada Intel Manual pada bagian **3.4.5 Segment Descriptors**.

Untuk memulai mengerjakan, template awal telah menyediakan definisi sebagian untuk struktur data **SegmentDescriptor**. Lengkapilah definisi struktur data segment descriptor berikut

```
gdt.h

struct SegmentDescriptor {
    // First 32-bit
    uint16_t segment_low;
    uint16_t base_low;

    // Next 16-bit (Bit 32 to 47)
    uint8_t base_mid;
    uint8_t type_bit : 4;
    uint8_t non_system : 1;
    // TODO : Continue SegmentDescriptor definition

} __attribute__((packed));
```

Definisi **uint8_t abc : 5**; memiliki arti variabel abc berukuran **tepat 5 bit** (bukan byte). Syntax ini bernama [Bit Field](#).

0.3.2. GDT & GDTR Definition

Global Descriptor Table dapat diletakkan dimanapun pada **Volatile Memory atau RAM**. Oleh karena itu, CPU harus mengetahui lokasi GDT diletakkan pada memory dan informasi ini disimpan pada register khusus bernama **GDTR**.

Karena GDT pada dasarnya adalah array of **SegmentDescriptor** dan GDTR relatif sederhana, kedua definisi struktur data telah diberikan pada file yang sama dengan segment descriptor.

Untuk menyelesaikan bagian ini adalah membuat definisi 1 GDT dan 1 GDTR untuk menggunakan definisi struktur data yang telah dibuat sebelumnya.

Setiap GDT wajib memiliki 1 entry GDT kosong (**Null Descriptor**) sebagai entri pertama (index ke-0). Pada tugas ini kernel menggunakan 1 entri untuk menandai segmen memori berisikan kode dan 1 entri lain untuk menandai segmen memori berisikan data (seperti variabel dan lain-lain).

Catatan: GDT menggunakan zero-indexed array (array dimulai dengan index 0)

Berikut adalah tabel yang menggambarkan isi GDT yang akan dibuat

No	Nama	Limit	Base	Type	S	DPL	P	L	D/B	G
0	Null	0	0	0	0	0	0	0	0	0
1	Kernel code segment	0xFFFFF	0	- Code - Not Accessed - Readable - Not Conforming (0xA/0b1010)	1 (Code or Data Segment)	0 (PL0 / Kernel)	1 (Valid Segment)	0	1 (32-bit operand)	1 (4KB)
2	Kernel data segment	0xFFFFF	0	- Data - Not Accessed - Writable - Direction Up (0x2/0b0010)	1	0	1	0	1	1

- Cell berwarna putih menandai data tersebut disimpan pada GDT
- Cell selain putih hanya pembantu untuk pembaca

Nilai **GDTR.address** adalah address dari GDT dan atribut size adalah **ukuran GDT dikurangi 1**. Ukuran GDT dapat diperoleh menggunakan keyword **sizeof**.

Buatlah sebuah file baru untuk mendefinisikan GDT dan GDTR seperti berikut

gdt.c

```
#include "header/cpu/gdt.h"

/***
 * global_descriptor_table, predefined GDT.
 * Initial SegmentDescriptor already set properly according to Intel Manual & OSDev.
 * Table entry : [{Null Descriptor}, {Kernel Code}, {Kernel Data (variable, etc)},
 ...].
 */
struct GlobalDescriptorTable global_descriptor_table = {
    .table = {
        {
            // TODO : Implement
        },
        {
            // TODO : Implement
        },
        {
            // TODO : Implement
        }
    }
};

/***
 * _gdt_gdtr, predefined system GDTR.
 * GDT pointed by this variable is already set to point global_descriptor_table above.
 * From: https://wiki.osdev.org/Global\_Descriptor\_Table, GDTR.size is GDT size minus
1.
 */
struct GDTR _gdt_gdtr = {
    // TODO : Implement, this GDTR will point to global_descriptor_table.
    //           Use sizeof operator
};
```

Catatan penting : Perhatikan urutan definisi struct, pastikan urutan dan alignment tepat 1:1 dengan Intel x86 manual. Kesalahan alignment menyebabkan kegagalan pembacaan

Sesuaikan definisi variabel dengan tabel pada halaman sebelumnya. Perhatikan ukuran maksimum setiap atribut struct. Ukuran bit field berpengaruh langsung dengan batas atas & bawah dari sebuah integer (ex. 4-bit unsigned → batas atas: $2^4 - 1 = 15 = 0xF$).

Tambahkan juga command kompilasi yang sesuai pada makefile untuk **gdt.c**. Meskipun berhasil melakukan compile, tidak berarti definisi struktur data dan definisi variabel sudah benar. Untuk mengetes bagian ini telah berjalan dengan baik atau belum, lanjutkan bagian selanjutnya yaitu [Load GDT](#).

0.3.3. Load GDT

Setelah GDT dan GDTR terdefinisi, informasi pada variabel tersebut perlu di load oleh CPU menggunakan instruksi `lgdt` menggunakan assembly. Prosedur dalam assembly untuk melakukan load GDT dinamai `load_gdt` dan deklarasi terletak pada `kernel-entrypoint.s`.

Prosedur assembly ini sederhananya melakukan hal-hal berikut

1. Melakukan load GDT dengan instruksi `lgdt`
2. Set **bit ke-0** (Protected mode flag) **CR0** (Control Register 0) dengan nilai 1
3. Lakukan **far jump** untuk memasuki **protected mode**
4. Menyesuaikan semua data segment register (ss, ds, es) ke kernel descriptor

Berikut adalah sumber tambahan jika ingin membaca lebih detail terkait GDT & Protected mode:

- [Protected Mode - OSDev](#)
- **Intel x86 Manual 3A - 9.9.1 Switching to Protected mode**

Tambahkan kode seperti berikut pada kernel untuk menyelesaikan bagian ini

```
kernel.c
```

```
void kernel_setup(void) {
    load_gdt(&_gdt_gdtr);
    while (true);
}
```

Jika terdapat kesalahan pada definisi GDT, instruksi far jump dan operasi segment register akan dapat menyebabkan special CPU exception ([Triple Fault](#)) dan melakukan reboot. Hal ini akan menyebabkan **bootloop**.

Jika tidak terjadi apapun sebelum dan sesudah menambahkan pemanggilan prosedur ini, kernel dasar OS 32-bit x86 telah terbuat dan dapat dijalankan. Sehingga dapat disimpulkan:

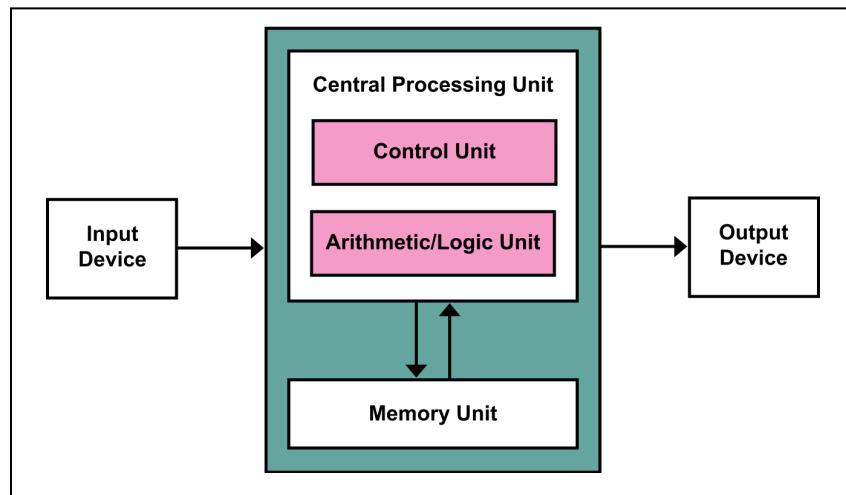
Selamat Chapter 0 telah selesai!

Tentunya kernel ini belum memiliki interaksi dengan user dan fitur yang minimal. Akan tetapi, tenang setiap chapter akan menambahkan fitur pada OS seperti [Driver untuk Keyboard](#), [File System](#), dan lain-lain.

0.4. - Extras: Computer

• x86 & Von Neumann Architecture

Arsitektur [x86](#) didasari dari [von Neumann Architecture](#). Arsitektur von Neumann merupakan model komputasi yang terdiri dari dua komponen utama: **Processing Unit** dan **Memory Unit**.



Arsitektur von Neumann, [Von Neumann Arch - Wikipedia](#)

Processing unit atau CPU bertugas melakukan instruksi komputasi sedangkan memory unit bertugas menyimpan data. Poin penting pada arsitektur von Neumann adalah **instruksi komputasi dianggap sebagai data**. Memory unit menyimpan data instruksi dan data lain.

Apa efek dari instruksi dianggap sebagai data? *Hold that thought, we'll get into it: [Code Injection](#)*

Apa artinya dari semua ini? Ada hubungannya dengan penggeraan?

Ya. Semua informasi instruksi dan data komputasi terletak pada RAM.

Pada konteks kernel development yang langsung menyentuh hardware, essentially hanya ada 4 komponen pada komputer: **CPU, Memory, Input, Output**.

Setiap kali membuat variabel pada bahasa C, call stack pada RAM akan dialokasikan untuk variabel tersebut. Untuk mengeksekusi instruksi, CPU menggunakan [Fetch-Decode-Execute](#) cycle yang mengambil instruksi dari RAM dan mengeksekusi instruksi. Semua kode C yang dibuat ketika penggeraan akan dikompilasi menjadi binary data dan dimasukkan kedalam RAM oleh GRUB layaknya data lain.

Dalam kata lain, semua hal yang dilakukan pada panduan ini adalah **manipulasi dan interaksi antara RAM dan CPU**. Kode yang ditulis adalah perintah kepada CPU untuk memanipulasi RAM.

• Bitness

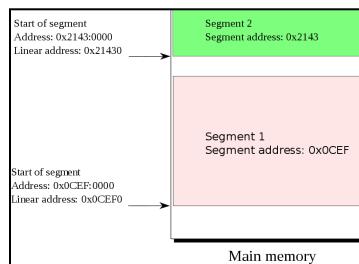
Ketika buku ini ditulis, diskusi terkait “bitness” dari CPU sudah tidak ada lagi. Sekitar pada tahun 2000 hingga 2015 (termasuk di Indonesia), banyak komputer x86 yang migrasi dari 32-bit ke 64-bit. Hal ini terlihat dengan beberapa versi OS Windows yang menyediakan 32-bit dan 64-bit pada XP, Vista, 7, dan beberapa generasi OS selanjutnya.

Secara umum bagi masyarakat yang tidak menggunakan komputer secara intensif, perubahan ini tidak terlalu terasa selain meningkatkan memory limit dari 32-bit yang hanya sekitar **4 GB** ke x86-64 yang mendukung addressing “52 bits” untuk **4 PB** physical memory.

Jadi, apa itu yang dimaksud “bit” pada suatu komponen komputer? Sayangnya jawaban untuk pertanyaan ini bukan jawaban hitam & putih. CPU dan komponen komputer merupakan benda yang sangat kompleks. Banyak bagian dari komponen-komponen tersebut yang dapat didiskusikan terkait berapa “bitness”-nya. [Virtual Memory](#), [Physical Memory](#), [Address Bus](#), [Data Bus](#), [Instruction](#), [Register Width](#), dan banyak hal lainnya yang dapat direfer sebagai “bitness”.

Namun umumnya yang direfer pada sebagian besar diskusi terkait “bitness” adalah [Address Bus](#) dan [Register CPU](#). Kedua ini berdampak langsung kepada seberapa banyak memori yang dapat diakses CPU, seperti yang terlihat pada antara x86 dengan protected mode yang “32-bit” dan ekstensi x86-64 yang “64-bit”. Patut dicatat bahwa jika salah satu bagian adalah “64-bit”, tidak dijamin bagian lain dari komponen juga 64-bit. Hal ini terlihat pada [x86 Real Mode 16-bit](#) dan x86-64 yang sekarang implementasi CPU pada pasar hanya mendukung 52 bits physical memory.

Adakah hubungannya dengan tugas ini? Ada, bagian selanjutnya adalah pembuatan [Global Descriptor Table](#) yang mengurus segmentasi memori. Pada x86 Real Mode 16-bit hanya memiliki register width 16-bit tetapi memori 20-bit. Bagaimana caranya mengakses memori pada address 20-bit jika register CPU hanya memuat 16-bit? x86 menggunakan register tambahan bernama [Segment Register](#)



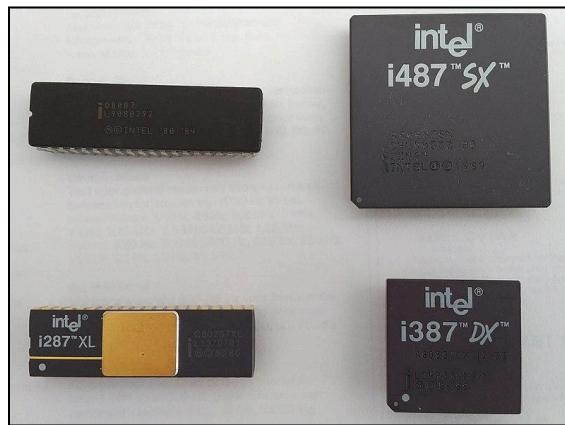
Sumber: [x86 Real Mode Memory Segmentation Addressing - Wikipedia](#)

Berbeda dengan 16-bit real mode yang menggunakan segment register untuk meningkatkan addressing memory, pada 32-bit protected mode segment register hanya digunakan sebagai GDT selector. Struktur data GDT biasanya tidak dimodifikasi sama sekali setelah inisiasi awal oleh sistem operasi modern.

• FPU & AVX

Sebagian besar kalkulasi CPU bekerja pada bilangan bulat (\mathbb{Z}). Memory address, program counter, general purpose register, banyak sekali bagian dari CPU yang didesain hanya untuk melakukan kalkulasi pada bilangan bulat tanpa menggunakan desimal. Faktanya memang cukup banyak hal yang dapat dicapai hanya dengan menggunakan bilangan bulat pada CPU dan bagian utama buku ini tidak akan menyentuh kalkulasi pada bilangan real (\mathbb{R}) sama sekali.

Selain tidak adanya kebutuhan untuk operasi floating point, kernel OS wajib menyiapkan [Floating Point Unit \(FPU\)](#) terlebih dahulu sebelum menggunakannya. Alasan mengapa FPU harus dinyalakan secara terpisah dan manual oleh kernel adalah salah satunya historis. Berbeda dengan CPU modern yang memiliki integrated FPU, tidak semua CPU lama memiliki FPU.



Intel x87 math coprocessor, Sumber: [FPU - Wikipedia](#)

CPU modern juga memiliki fitur yang lebih *fancy* untuk meningkatkan kinerja kalkulasi floating point dengan [Streaming SIMD Extension \(SSE\)](#) dan [Advanced Vector Extensions \(AVX\)](#) yang merupakan [SIMD](#). SIMD merupakan metode paralelisasi yang menggunakan satu instruksi untuk melakukan operasi ke banyak data sekaligus. Operasi matriks merupakan salah satu contoh komputasi yang mendapatkan peningkatan kinerja secara masif dengan paralelisasi SIMD.

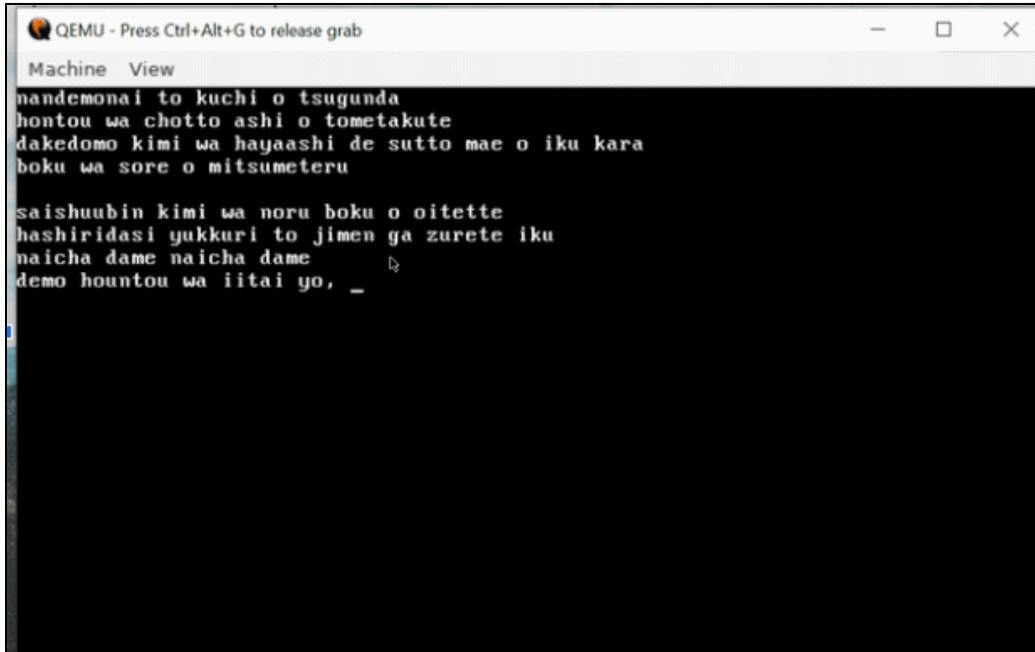
Fitur SSE dan AVX menyediakan register dan instruksi tambahan sendiri. Sama seperti FPU, kernel wajib untuk menyiapkan dan menyalakan fitur SSE dan AVX sebelum aplikasi dapat menggunakan fitur paralelisasi ini. Meskipun dapat mempercepat komputasi pada kasus tertentu, SSE dan AVX jarang dibutuhkan oleh general program. Fitur-fitur ini biasanya digunakan oleh developer library seperti [Numpy](#) dan [TensorFlow](#) untuk mengoptimasi kinerja library sehingga pengguna library tidak perlu untuk menyentuh secara langsung SSE dan AVX.

Meskipun diperlukan step tambahan untuk menggunakan AVX / SSE, komponen ini jauh lebih mudah untuk speedup kalkulasi floating point dibandingkan metode lain seperti **Integrated** atau **Dedicated GPU**.

Ch. 1 - Framebuffer, Interrupt, Driver

Chapter 1 akan membuat sistem [Interrupt](#) yang diperlukan untuk membuat [Keyboard Driver](#) dan [File System](#). Sebelum itu, agar ada hasil pekerjaan yang konkret, Chapter 1 akan diawali dengan membuat [Text Framebuffer](#) terlebih dahulu.

Pada akhir Chapter 1 sistem operasi akan dapat menerima dan menampilkan keyboard input.



Hasil dari implementasi yang dilakukan pada Chapter 1, yaitu mekanisme input berupa [Keyboard Driver](#) dan [Interrupt](#), serta mekanisme output berupa [Framebuffer](#), nantinya akan digunakan untuk membuat CLI [Shell](#) pada [Chapter 2](#).

1.0. Short Note: Kernel Development

Chapter ini adalah awal dari **Kernel Development**. Berbeda dengan development pada tingkat abstraksi atas, disini menerapkan prinsip **do-it-yourself** dengan ekstrim. Fitur yang biasanya taken for granted harus dibuat sendiri. Kernel developer akan dianggap **memahami secara mendalam** tentang kode yang ditulis.

Pada abstraksi tingkat atas developer harus mengikuti batasan dan peraturan yang dibuat dari OS. Sebaliknya, pada low-level **kita yang membuat peraturan sendiri**. Terserah kepada developer OS apakah suatu behavior dari OS ini sudah sesuai keinginan developer atau tidak.

- Ada division by zero diabaikan saja? Boleh
- OS sepenuhnya berjalan diatas RAM? Bisa
- Ada program user yang menghapus semua disk, diperbolehkan oleh kernel? Bebas

Tidak ada yang salah, *yang ada adalah tidak sesuai keinginan*.

Kebebasan ini menjadi pedang bermata dua. Dalam satu sisi, kernel developer dapat melakukan banyak hal dan membuat peraturan sendiri. Dalam sisi lain, kernel development memiliki [hand-holding](#) yang sangat minimum.

Kernel memiliki akses penuh pada hardware sistem. Seluruh memory, register, kendali CPU, dan komponen-komponen lain sepenuhnya dapat dikontrol oleh kernel. Jika pada Alpro sering bertemu dan membenci [Segfault](#), disini kode kernel dapat menuliskan apapun secara bebas pada sembarang memory ;)

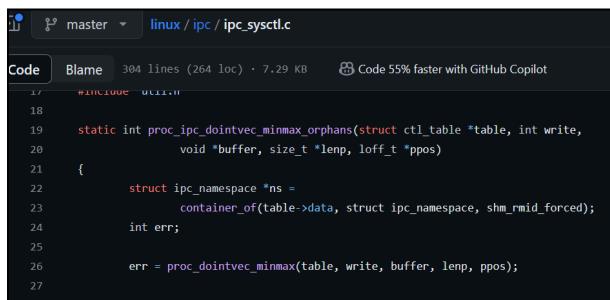


"With great power comes great responsibility"

• Coding Style & Compiler Warning

Project besar yang sudah mature seperti **Linux** dan perusahaan seperti **Google** memiliki **coding style** dan internal tools untuk melakukan style checking secara otomatis. Banyak penulis kode pemula tidak memiliki coding style yang baik dan konsisten. Tidak jarang juga alasan untuk mengabaikan coding style adalah spasi dan formatting tidak berdampak kepada logic code. Kode tidak hanya dibuat untuk menginstruksikan mesin, **tetapi juga ditulis untuk bacaan manusia.**

Memang tidak salah, kode akan dibaca parser compiler dan diubah ke representasi machine code yang sama jika hanya ada perbedaan coding style. **Tapi ingat, kita bukan mesin.** Manusia cenderung menyukai hal-hal yang rapi dan secara tidak sadar akan terpengaruhi dengan objek atau lingkungan yang ada. Ambil [standard Linus Torvalds](#), Indentasi: **Tabs are 8 characters.**



```
#include <linux/ipc.h>
static int proc_ipc_dointvec_minmax_orphans(struct ctl_table *table, int write,
                                             void *buffer, size_t *lenp, loff_t *ppos)
{
    struct ipc_namespace *ns =
        container_of(table->data, struct ipc_namespace, shm_rmid_forced);
    int err;
    err = proc_dointvec_minmax(table, write, buffer, lenp, ppos);
}
```

Sure enough, Linux source code, [linux/ipc/ipc_sysctl.c](#)

Mungkin sekilas coding style tersebut terdengar gila. Beberapa pembaca mungkin sudah membenci 4 char tab dan menggunakan 2 char tab, lupakan 8 char. Salah satu justifikasi dari Linus adalah **nested conditional & loop terlihat dengan jelas pada 8 char** akan membuat penulis kode setidaknya merasa ada sesuatu yang salah dan memperbaiki kode.

Tapi psikologi manusia bukanlah hal yang reliable, ada orang yang *picky* dengan formatting, ada yang menerima saja hingga horizontal scroll harus digeser sebanyak tiga layar. Jika compiler dapat membaca kode, lebih reliable, dan lebih konsisten daripada manusia, **mengapa tidak membuat compiler untuk mengecek code style juga?**

Apa hubungannya dengan pengerjaan ini?

Sangat direkomendasikan untuk menggunakan code style yang seragam dan mudah dibaca. Selain itu, **hampir semua flag warning (wall wextra) gcc dinyalakan dan di eskalasi menjadi error** (werror) pada template.

Warning yang dibuat pada flag ini terkadang terkesan tidak penting untuk logic kode (misleading indentation, unused variables & parameter, etc), tetapi sangat penting untuk menulis kode sebersih mungkin dan seminimum mungkin pada kernel development. **Sangat tidak disarankan untuk mematikan flag-flag gcc pada pengerjaan**, pahami dan perbaiki satu per satu error.

● C: Include Guard

Bagi pembaca yang sebelumnya belum pernah melakukan development menggunakan C atau bahasa lain yang menggunakan C preprocessor kemungkinan tidak pernah menemui [Include Guard](#).

```
1  ifndef _PORTIO_H
2  define _PORTIO_H
3
4  include <stdint.h>
5  include <stdbool.h>
6
7  /**
8   * Send data to the given I/O port
9   *
10  * @param port The I/O port to send the data to
11  * @param data The data to send to the I/O port
12  */
13 void out(uint16_t port, uint8_t data);
14
15 /**
16  * Read data from the given I/O port
17  *
18  * @param port The I/O port to request the data
19  * @return Recieved data from the corresponding I/O port
20  */
21 uint8_t in(uint16_t port);
22
23 endif
```

Include guard in portio.h

Include guard pada gambar adalah `#ifndef _PORTIO_H`, `#define _PORTIO_H`, dan `#endif`.

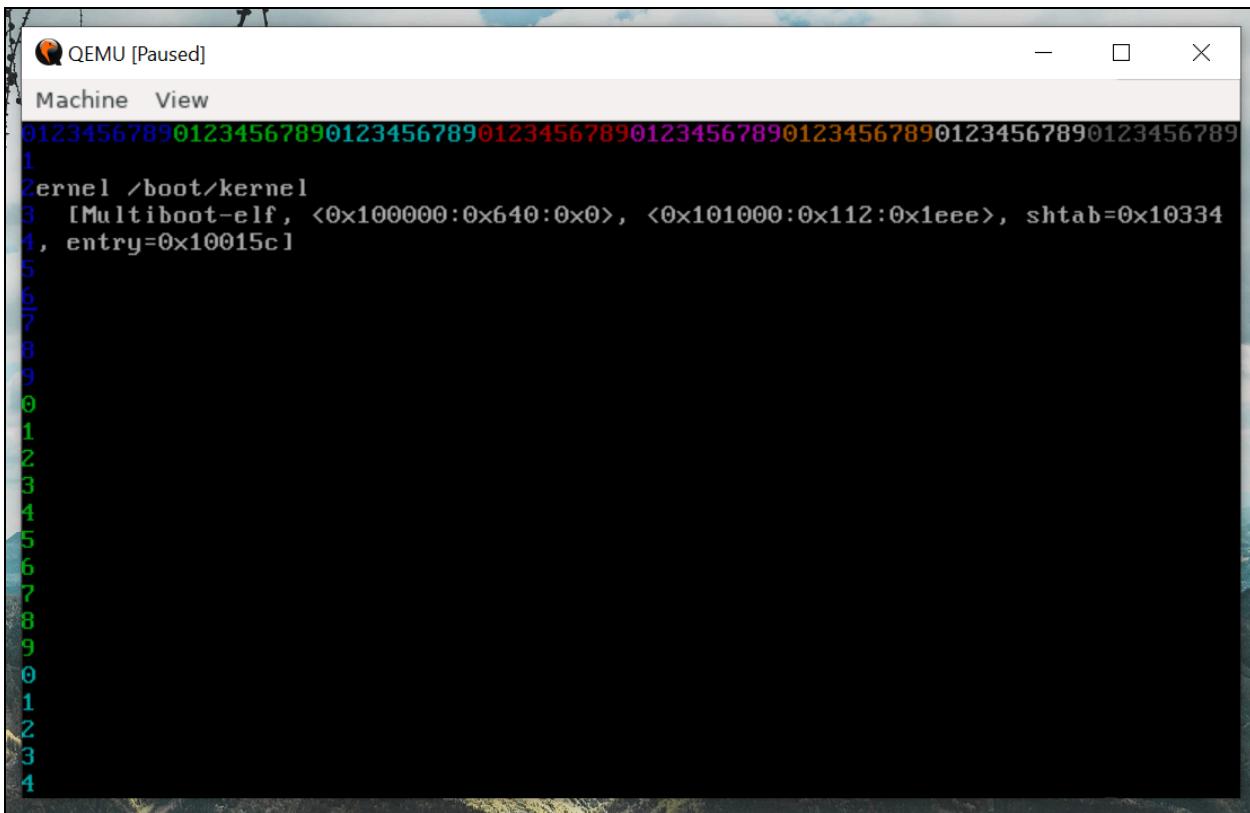
Konstruksi ini selalu digunakan pada project yang menggunakan C dan C preprocessor seperti **Linux** dan header files yang disediakan pada kit dan template. Namun uniknya jarang “tutorial C” yang mengajarkan hal ini.

Kegunaan utama dari include guard adalah mencegah file di-copy-paste berulang kali (dari `#include`) pada proses kompilasi. Header file yang membawa definisi struct akan menyebabkan error jika struct didefinisikan ulang secara tidak sengaja dengan operasi `#include` berulang kali.

Jika pada pengajaran panduan ini membutuhkan file source code baru, jangan lupa untuk menambahkan include guard pada header file agar tidak terjadi error.

1.1. Driver Text Framebuffer

Text Framebuffer merupakan output sederhana dan memiliki resolusi 80x25 karakter ASCII. Bagian ini akan menjadi pemanasan kernel development dan mudah untuk diuji hasilnya.



Ilustrasi text framebuffer dengan resolusi 80x25

Kernel memiliki salah satu tugas untuk mengontrol komponen hardware. Tentunya kernel perlu memiliki suatu mekanisme untuk berkomunikasi dengan hardware untuk mengontrol perilakunya. Pada arsitektur x86 tersedia dua mekanisme untuk berkomunikasi: [Memory-Mapped I/O](#) dan [Port-Mapped I/O](#). Singkatnya komunikasi memory-mapped I/O dapat secara langsung membaca dan mengedit memory pada lokasi tertentu sedangkan port I/O menggunakan instruksi assembly CPU yang khusus untuk mengirim data melewati port.

Text framebuffer menggunakan keduanya untuk hal yang berbeda. Memory-mapped I/O digunakan untuk menyimpan informasi karakter pada layar dan port I/O digunakan untuk mengontrol kursor.

Implementasikan tiga interface yang dideklarasikan berikut

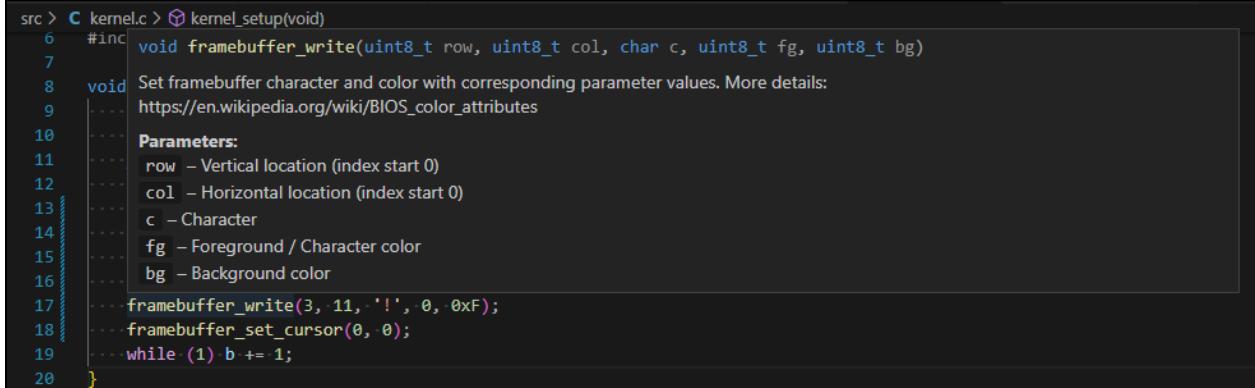
```
framebuffer.h
```

```
void framebuffer_write(uint8_t row, uint8_t col, char c, uint8_t fg, uint8_t bg);
void framebuffer_set_cursor(uint8_t r, uint8_t c);
void framebuffer_clear(void);
```

Kerangka dasar untuk mengerjakan bagian ini terdapat pada kit **ch1/1 - Framebuffer/**. Pada folder tersebut akan disediakan implementasi kode C untuk **Port I/O** dan header framebuffer.

Susun file-file kerangka dasar tersebut pada folder yang sesuai agar mempermudah navigasi source code.

Sama seperti file yang ada pada template, semua deklarasi fungsi akan disertai dokumentasi deskripsi fungsi, parameter, atribut dan hal lain menggunakan [doxygen](#) annotation



```
src > C kernel.c > kernel_setup(void)
6 #inc void framebuffer_write(uint8_t row, uint8_t col, char c, uint8_t fg, uint8_t bg)
7
8 void Set framebuffer character and color with corresponding parameter values. More details:
9     https://en.wikipedia.org/wiki/BIOS_color_attributes
10
11     Parameters:
12         row - Vertical location (index start 0)
13         col - Horizontal location (index start 0)
14         c - Character
15         fg - Foreground / Character color
16         bg - Background color
17
18     framebuffer_write(3, 11, '!', 0, 0xF);
19     framebuffer_set_cursor(0, 0);
20     while (1) b += 1;
21 }
```

Doxygen docs pada Intellisense vscode (Hover pada fungsi untuk melihat)

Ketiga interface sederhana ini akan digunakan terus hingga akhir. Implementasi interface seharusnya tidak memerlukan banyak kode untuk memenuhi behavior yang diinginkan. Perlu diingat bahwa driver ini akan **sepenuhnya berfokus pada manipulasi text framebuffer**. Fitur seperti text scrolling, keyboard input, shell CLI interface akan diimplementasikan menggunakan driver ini.

Driver framebuffer yang dibuat hanya akan menerima **ASCII**. Asumsikan semua parameter karakter dalam encoding ASCII untuk mempermudah pengerajan.

1.1.1. `framebuffer_write()`

Implementasi akan menggunakan **memory-mapped I/O** untuk menuliskan karakter ke layar. Offset memory untuk text framebuffer terletak pada macro `FRAMEBUFFER_MEMORY_OFFSET` (`0xB8000`) pada file `framebuffer.h`.

Setiap karakter akan menggunakan 2 bytes memory; memory genap (misalnya `0xB8002`) adalah karakter ASCII yang ditampilkan dan memory ganjil (misal `0xB003`) membawa informasi warna background dan text. Gunakan [C: Subscript Operator](#) untuk menuliskan data ke lokasi memory tertentu seperti berikut

```
FRAMEBUFFER_MEMORY_OFFSET[10] = 'a';
FRAMEBUFFER_MEMORY_OFFSET[11] = 0x20 | 0x0F;
```

Beberapa detail dan informasi tambahan terkait manipulasi memory-mapped text framebuffer dapat dicek pada [TextUI - OSDev](#).

1.1.2. `framebuffer_set_cursor()`

Implementasi akan menggunakan **port I/O** untuk mengatur lokasi text cursor. Port I/O menggunakan instruksi assembly `in` / `out` untuk menerima / mengirim data melalui bus I/O x86. Kedua instruksi ini telah disediakan wrappernya dalam bahasa C yang siap digunakan pada `portio.c` dengan nama fungsi C yang sama.

Detail perintah untuk mengendalikan text cursor menggunakan port I/O dapat dicek pada [Text Mode Cursor - OSDev](#).

1.1.3. `framebuffer_clear()`

Implementasi dari interface ini dapat menggunakan berbagai cara. Diperbolehkan untuk menggunakan cara apapun selama memenuhi expected behavior yang ditulis pada deklarasi fungsi.

Fungsi `memset()` yang disediakan pada `string.c` dapat digunakan untuk memanipulasi memori.

Semua interface dapat diimplementasikan hanya menggunakan bahasa C dengan kit yang telah disediakan. Diperbolehkan untuk menggunakan assembly jika dibutuhkan.

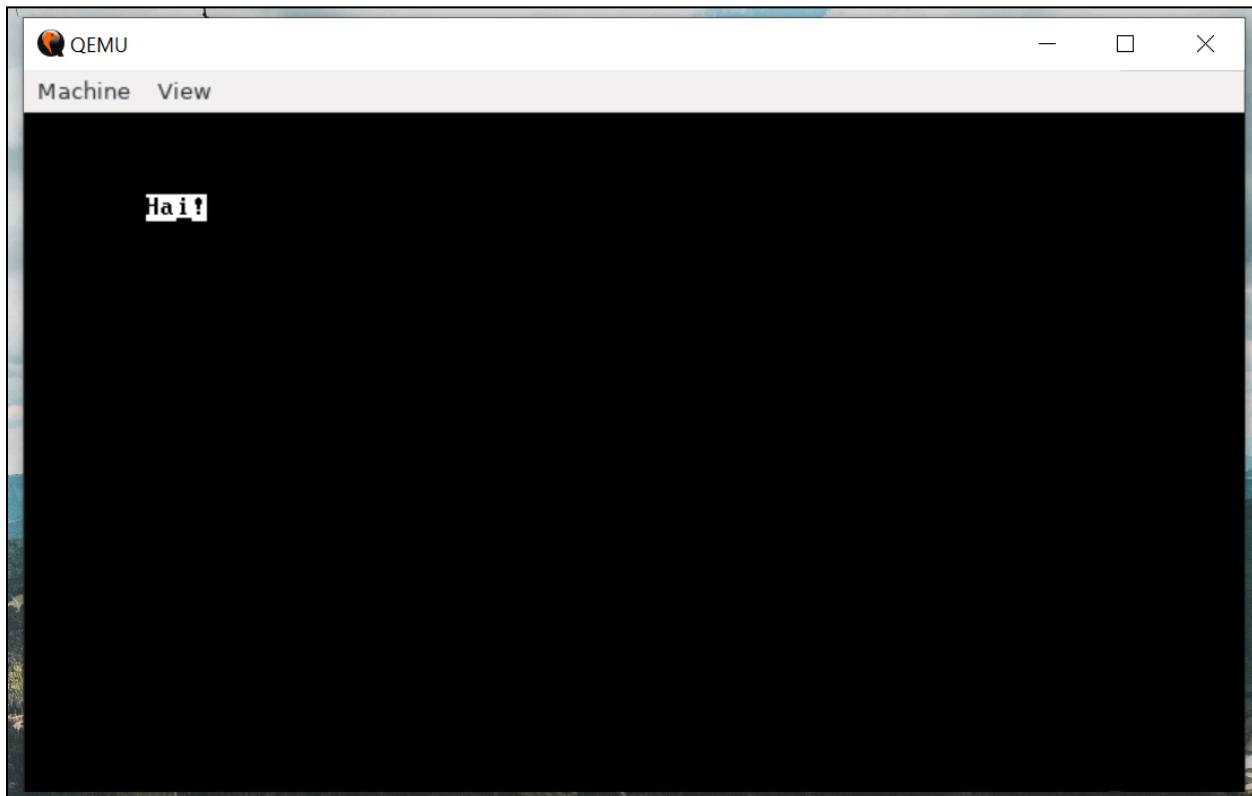
1.1.4. Test: Framebuffer

Tambahkan kode berikut untuk menguji hasil implementasi

```
kernel.c
```

```
void kernel_setup(void) {
    framebuffer_clear();
    framebuffer_write(3, 8, 'H', 0, 0xF);
    framebuffer_write(3, 9, 'a', 0, 0xF);
    framebuffer_write(3, 10, 'i', 0, 0xF);
    framebuffer_write(3, 11, '!', 0, 0xF);
    framebuffer_set_cursor(3, 10);
    while (true);
}
```

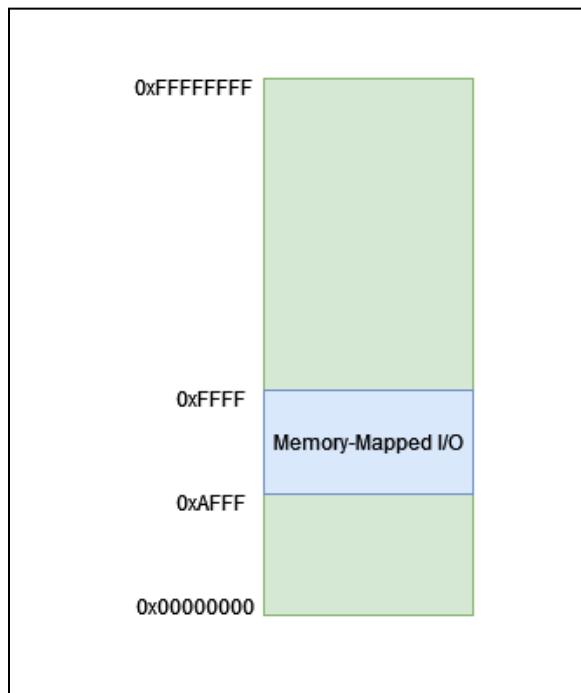
Build sistem operasi (gunakan **F5 & Shift + F5** pada vscode) dan jika implementasi bekerja dengan baik akan menampilkan seperti berikut



Tampilan setelah clear dan write “Hai!” dengan inverted background-foreground

- **Memory-mapped I/O & Port-mapped I/O**

Kernel memiliki tugas untuk mengendalikan hardware dan hal ini memerlukan metode komunikasi untuk mengirimkan data. Pada arsitektur x86 terdapat dua metode komunikasi utama untuk CPU mengirimkan data ke komponen hardware lain yaitu [Memory-mapped I/O](#) dan [Port-mapped I/O](#).



Ilustrasi memory-mapped I/O (MMIO) yang menggunakan 0xAFFF - 0xFFFF

Ilustrasi diatas menggambarkan sebagian memory digunakan untuk memory-mapped I/O (MMIO). Tentunya salah satu kekurangan dari MMIO adalah menggunakan address space yang sama dengan memory utama. Jika device membutuhkan address space yang besar, address kosong yang dapat digunakan pada memory utama akan tersisa sedikit.

Port-mapped I/O (PMIO) x86 menggunakan instruksi `in` dan `out` untuk mengirimkan data kepada komponen hardware lain. Setiap instruksi hanya dapat mengambil dan mengirim data berukuran 1, 2, dan 4 bytes. Arsitektur ekstensi x86-64 tidak menambahkan instruksi PMIO tambahan untuk 8 bytes. Tidak seperti MMIO yang menggunakan address space pada memory utama, PMIO tidak menggunakan memory utama untuk mengirimkan data.

Sistem x86-64 modern umumnya menggunakan memory-mapped I/O untuk sebagian besar komunikasi dengan komponen I/O. Sistem modern mengasumsikan memory utama berukuran besar sehingga drawback memory-mapped I/O tidak terlalu berpengaruh banyak. Arsitektur x86 juga menyediakan [Physical Address Extension](#) (PAE) untuk protected mode 32-bit yang memperbesar physical address space untuk digunakan MMIO dan [Paging](#). [Port I/O](#) pada sistem x86-64 modern biasanya digunakan untuk melakukan komunikasi dengan komponen tua atau pengaturan awal saja sehingga secara efektif merupakan fitur legacy x86.

● C: Subscript Operator

Ah, C array subscript operator, operator yang akan dijelaskan secara sekilas pada programming course dan tidak akan pernah dibahas lagi. Tapi tidak untuk buku ini, bagian ini akan menjelaskan behavior subscript operator sedikit lebih detail dan menunjukkan kode *terkutuk*.

Biasanya operator ini dijelaskan ketika bagian awal pengenalan array untuk mengakses data pada array. Berikut adalah contoh untuk mengganti nilai pada array dengan operator subscript

```
int arr[64];
arr[42] = 1337;
```

Namun bagaimana operator tersebut bekerja?

Standar C dan C++ (asumsi tidak ada [Operator Overloading](#)) secara eksplisit mendasarkan operator subscript ekuivalen dengan operasi pointer arithmetic berikut

```
arr[42]      = 1337;
*((arr)+(42)) = 1337;
```

Perlu diingat juga bahwa pointer arithmetic akan secara implisit mengalikan konstanta dengan ukuran dari variabel yang di-point. Dalam kasus diatas, **42** akan dikali dengan ukuran satu **int** atau **sizeof(int)**.

Apakah ada perbedaan jika operator ini digunakan pada array dan pointer?

Operasi pointer arithmetic diatas memiliki behavior sama jika digunakan pada array atau pointer. Hal ini terlihat pada kode contoh framebuffer yang diberikan

```
FRAMEBUFFER_MEMORY_OFFSET[10] = 'a';
(uint8_t*) 0xC00B8000)[10] = 'a';
*((uint8_t*) (0xC00B8000 + 10)) = 'a';
```

Mengingat sifat komutatif operasi penjumlahan pada bilangan bulat, kode *terkutuk* dibawah ini akan memiliki behavior yang sama

```
int arr[64];
arr[42] = 1234;
42[arr] = 1234;
```

Definisi dari operator ini ekuivalen pada standar C dan C++:

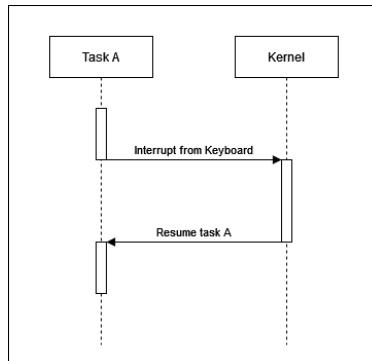
en.cppreference.com/w/cpp/language/operator_member_access#Built-in_subscript_operator

Resource berikut memiliki beberapa FAQ tambahan array-pointer: c-faq.com/aryptr/index.html

1.2. Interrupt

Interrupt adalah mekanisme utama pada komputer modern untuk melakukan **multitasking**. Perlu diingat bahwa **CPU pada intinya hanya dapat melakukan 1 hal pada satu waktu***. Jika CPU mengerjakan task A, task lain akan diam dan tidak berubah.

*Dalam konteks penggerjaan ini, CPU hanya memiliki 1 unit eksekusi.



Ilustrasi CPU menerima interrupt dari keyboard

Ilustrasi di atas menggambarkan alur eksekusi CPU ketika ada keyboard interrupt ketika tombol ditekan. **Interrupt merupakan semacam notifikasi ditujukan ke CPU bahwa terdapat sebuah event yang memerlukan perhatian**. CPU akan berhenti dari pekerjaan sekarang dan merespon event tersebut sebelum kembali melanjutkan pekerjaannya.

Tujuan utama **Chapter 1** adalah membuat input device menggunakan keyboard, sebuah komponen eksternal dari CPU sehingga memerlukan interrupt untuk menginformasikan situasi ke CPU.

Bagaimana CPU merespon suatu interrupt akan sepenuhnya dikendalikan oleh kernel. Kernel bertanggung jawab untuk menyiapkan struktur data untuk keperluan interrupt dan mendefinisikan behavior yang terjadi ketika ada interrupt.

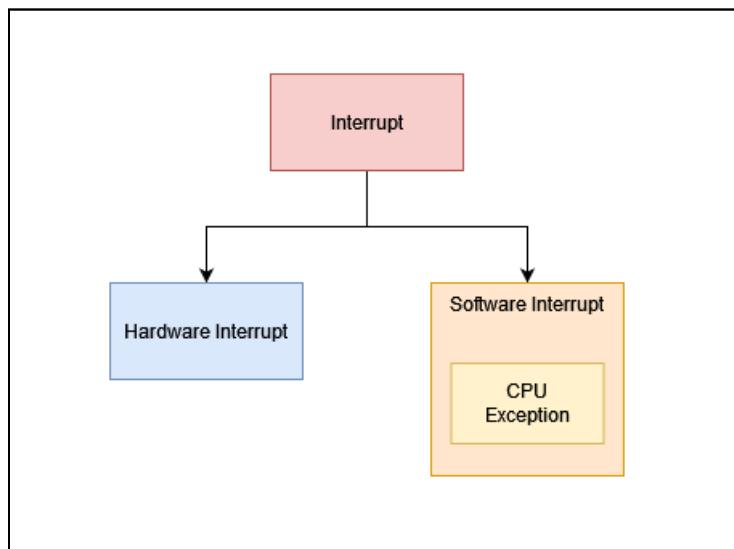
Agar komponen luar CPU mengirimkan interrupt, kernel wajib untuk melakukan setup IRQ Remapping pada Programmable Interrupt Controller. Selain itu, perlu didefinisikan Interrupt Descriptor Table yang menyimpan address Interrupt Service Routine. Address ini akan dipanggil ketika ada interrupt yang datang ke CPU. Sama seperti GDT, CPU juga perlu mengetahui lokasi IDT pada memory sebelum melakukan Load IDT.

Hal diatas akan dibahas satu-per-satu pada bagian-bagian selanjutnya.

- **Hardware & Software Interrupt**

Seperti yang dijelaskan pada bagian sebelumnya, Interrupt merupakan salah satu cara untuk menginterupsi eksekusi CPU dan melakukan hal lain. CPU modern dapat melakukan eksekusi instruksi dengan sangat cepat sehingga dengan adanya interrupt akan memberikan ilusi untuk tetap mengerjakan tugasnya dan merespon dunia luar.

Interrupt dipisah menjadi dua: **Hardware Interrupt / IRQ** dan **Software Interrupt**.



Hardware Interrupt menjadi fokus pada **Chapter 1**. Interrupt tipe ini berasal dari komponen eksternal yang mengirim informasi event ke CPU. Hardware eksternal seperti keyboard akan mengirimkan interrupt setiap kali tombol ditekan sehingga CPU harus meresponnya sesuai dengan interrupt handler yang didefinisikan kernel. Dalam konteks kernel development, Hardware Interrupt juga disebut **Interrupt Request (IRQ)**.

Software Interrupt nantinya akan dibahas lebih lanjut pada Chapter 3 dalam bentuk **System Calls**. Software interrupt merupakan interrupt yang berasal dari instruksi khusus seperti **INT** pada x86 atau **Exception**. Karena interrupt umumnya juga mengganti privilege dari CPU, software interrupt biasanya digunakan untuk mengimplementasikan fitur seperti **System Calls** yang membutuhkan pergantian privilege.

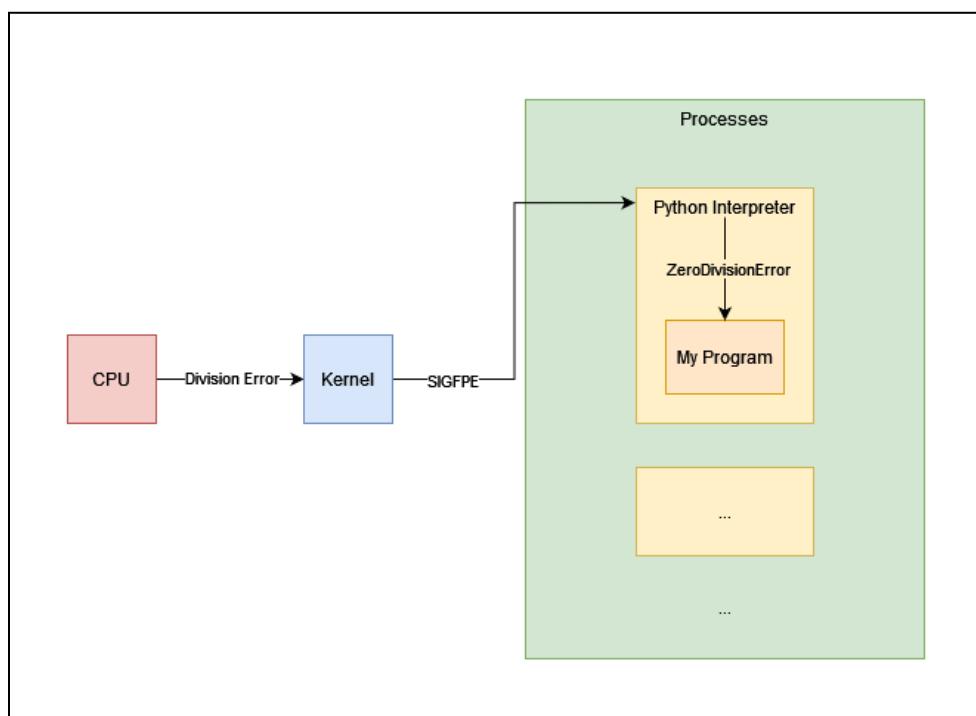
Catatan tambahan: Pada konteks Interrupt, terdapat notasi tambahan untuk radix / basis 16 (Hexadecimal). Banyak bahasa pemrograman yang menggunakan **prefix 0x** untuk hexadecimal, tetapi pada konteks Interrupt terdapat **suffix h**. Contohnya adalah **0x11 = 11h = 17**

• CPU Exception

Exception yang dimaksud pada konteks kernel development adalah **CPU Exception** yang merupakan hardware counterpart dari language exception yang ada pada bahasa seperti C++, Java, Python. CPU exception sepenuhnya akan dihandle oleh kernel sesuai dengan handler yang ada pada **IDT**. Mungkin CPU exception terdengar seperti hal baru, tetapi sebenarnya terdapat beberapa [Familiar Exception](#).

Karena CPU exception yang tidak dihandle akan menyebabkan full system reboot, CPU exception ini akan selalu dihandle OS dan biasanya ditransformasi menjadi pesan yang dapat dihandle proses user. **OS Exception** memiliki berbagai macam implementasi seperti [Signal dari Kernel pada UNIX-like](#) atau [Structured Exception Handling milik Windows](#).

Language Exception yang ada pada C++, Java, Python biasanya diimplementasikan diatas OS exception. Exception seperti Python ZeroDivisionError merupakan hasil **CPU Exception Division Error** yang menjadi **OS Exception SIGFPE** pada UNIX-like dan diteruskan ke interpreter Python menjadi **ZeroDivisionError**. Beberapa language exception dapat sepenuhnya diimplementasikan tanpa menggunakan exception pada layer bawah.



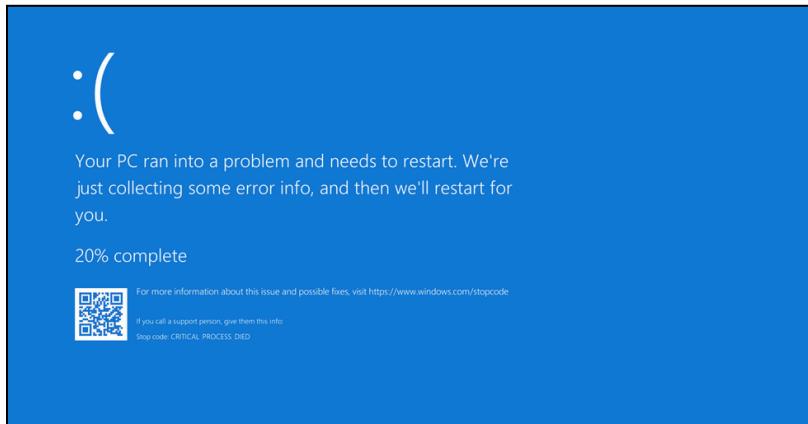
Ilustrasi exception flow dari ZeroDivisionError Python di UNIX-like system

● Familiar Exception

Beberapa CPU exception yang ada pada list [OSDev/Exception](#) akan terdengar familiar. Namun ada juga yang mungkin terdengar tidak familiar tetapi sebenarnya memiliki nama lain pada sistem operasi modern.

■ Double Fault

Ah **Double Fault**, nama yang asing tetapi sangat familiar dengan pengguna komputer x86. Pengguna **Windows** mestinya pernah melihat layar horor ini



The Cosmic Horror of Modern PC - [BSOD - Wikipedia](#)

Blue Screen of Death disebabkan oleh CPU exception **Double Fault**, terjadi ketika interrupt handler milik kernel mengalami exception. **Device Driver** yang ditulis pada kernel-mode (Contohnya: Networking stack & GPU) yang mengalami exception juga salah satu sumber penyebab **Double Fault** (Exception ketika menghandle IRQ dari hardware). Namun terkadang double fault juga dapat disebabkan oleh user-level exception pada beberapa kasus tertentu.

Bergantung kepada kebijakan sistem operasi masing-masing, double fault mungkin dapat dihandle dan dikembalikan ke flow normal sistem operasi. Untuk Windows, double fault akan dilanjutkan ke penulisan log (termasuk identifikasi **stopcode** yang sesuai, contohnya CRITICAL_PROCESS_DIED pada ilustrasi diatas) dan **Blue Screen of Death**.

Untuk sistem operasi non-Windows (Linux, Mac OS), Blue Screen of Death umumnya ekuivalen dengan **Kernel Panic**.

■ Triple Fault

Kelanjutan dari double fault, **Triple Fault** terjadi ketika double fault interrupt handler juga mengalami exception. Berbeda dengan double fault yang masih dapat memiliki interrupt handler, **Triple Fault tidak bisa dihandle oleh kernel**, CPU akan secara otomatis berhenti dan melakukan reboot seluruh sistem. Jika terdapat permasalahan kode startup kernel yang menyebabkan **Triple Fault**, komputer akan mengalami **bootloop**. Implementasi GDT, IDT, Paging, **Context Switch** merupakan step yang sangat rentan untuk mengalami triple fault.

■ General Protection Fault

Flashback IF2110 - Algoritma & Struktur Data: Segmentation Fault.

```
Compilat...n terminated.
brsh@LAPTOP-8E9F1CHQ:/mnt/c/Users/Lckd/Downloads$ gcc -o q list.c
brsh@LAPTOP-8E9F1CHQ:/mnt/c/Users/Lckd/Downloads$ ./q
Segmentation fault
brsh@LAPTOP-8E9F1CHQ:/mnt/c/Users/Lckd/Downloads$
```

IF2110 strukdat experience: "*Ini kenapa sih, ga jelas*"

General Protection Fault atau **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 GP fault pada beberapa kasus khusus).

Umumnya pada **IF2110**, hal ini terjadi karena mengakses pointer yang belum terinisiasi. **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, bagaimana bisa kita menghentikan eksekusi program dengan debugger? Jawabannya adalah **Breakpoint Exception**. Debugger biasanya bekerja dengan mensubstitusi instruksi source code pada breakpoint dengan instruksi **INT \$0x3**.

Berbeda dengan interrupt lain yang membutuhkan lebih dari 1 byte **opcode**, khusus untuk **INT \$0x3** hanya membutuhkan 1 byte untuk **opcode**. Hal ini memperbolehkan untuk mensubstitusi instruksi apapun dengan opcode tersebut, tanpa menggeser instruksi yang dapat merusak offset program (terutama pada x64 dengan **System V ABI**, dimana machine code executable akan menggunakan [Position Independent Code](#)).

1.2.1. IRQ Remapping

Programmable Interrupt Controller (PIC) merupakan controller chip eksternal CPU yang mengontrol semua Interrupt Request (IRQ) dari komponen eksternal. Sebelum melanjutkan membuat keyboard driver dan file system, IRQ dari keyboard dan disk tersebut harus sudah bekerja. Bagian ini akan mengatur PIC sehingga IRQ dapat diterima oleh CPU.

Karena alasan legacy tertentu (IBM PC), bagian ini akan menambahkan offset untuk semua IRQ yang dikirimkan PIC, termasuk IRQ keyboard dan disk. Meskipun alasan dari IRQ remapping adalah legacy, bagian ini tetap wajib diimplementasikan agar CPU exception tidak mengirimkan interrupt dengan nomor yang sama dengan hardware interrupt.

Deklarasi prosedur-prosedur yang berhubungan dengan PIC remapping tersedia pada **interrupt.h**. Bagian ini akan mengimplementasikan fungsi `pic_remap()`, `io_wait()`, dan `pic_ack()`.

Fungsi `pic_remap()` akan menggunakan macro yang telah didefinisikan pada **interrupt.h**. Fungsi ini akan menggeser interrupt **PIC master / IRQ1** ke **0x20** dan **PIC slave / IRQ2** ke **0x28**.

Semua interrupt yang dibuat oleh **8259A PIC akan dimulai dari 0x20 dan 0x28** (IRQ0 yaitu timer akan melakukan interrupt **0x20 + 0 = INT 20h** (notasi suffix); IRQ1 keyboard akan melakukan interrupt **0x20 + 1 = INT 21h**; dan seterusnya).

Detail lebih lanjut terkait PIC dan IRQ tersedia pada bagian [8259A & IBM PC](#)

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) {
    // 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, 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();

    // Disable all interrupts
    out(PIC1_DATA, PIC_DISABLE_ALL_MASK);
    out(PIC2_DATA, PIC_DISABLE_ALL_MASK);
}
```

- **8259 PIC & IBM PC**

PIC yang digunakan pada bagian ini adalah **8259 PIC** yang didesain pada era **IBM PC**. Alasan utama dari IRQ remapping adalah nomor interrupt yang dikirimkan oleh PIC secara default dalam interval **[0x0, 0xF]** bertabrakan dengan x86 CPU exception yang menggunakan interval **[0x00, 0x1F]**. Tentunya hal ini akan membuat IRQ & exception handling lebih kompleks jika IRQ menggunakan nomor interrupt yang sama.

INT_NUM	Short Description PM [clarification needed]
0x00	Division by zero
0x01	Single-step interrupt (see trap flag)
0x02	NMI
0x03	Breakpoint (which benefits from the shorter 0xCC encoding of INT 3)
0x04	Overflow
0x05	Bound Range Exceeded
0x06	Invalid Opcode
0x07	Coprocessor not available
0x08	Double Fault
0x09	Coprocessor Segment Overrun (386 or earlier only)
0x0A	Invalid Task State Segment
0x0B	Segment not present
0x0C	Stack Segment Fault
0x0D	General Protection Fault
0x0E	Page Fault
0x0F	reserved
0x10	x87 Floating Point Exception
0x11	Alignment Check
0x12	Machine Check
0x13	SIMD Floating-Point Exception
0x14	Virtualization Exception
0x15	Control Protection Exception (only available with CET)

x86 CPU Exception, Sumber: [IDT - Wikipedia](#)

Paragraf ini akan membahas kode pemrograman PIC yang diberikan pada bagian sebelumnya. Pemrograman PIC perlu memastikan setiap command yang dikirim ke PIC telah diproses. `io_wait()` akan menunggu 1-4 microsecond yang biasanya cukup untuk keperluan ini. IRQ yang dikirimkan oleh PIC **harus direspon dengan pesan ACK** yang dikirim pada `pic_ack()` untuk memberitahu CPU telah selesai memproses IRQ.

Catatan penting: Semua IRQ yang di-generate 8259 PIC harus di-ACK. Jika tidak, semua IRQ setelah IRQ yang diterima akan ditahan. Contohnya, jika IRQ Timer & IRQ Keyboard menyala, IRQ Timer masuk tetapi tidak di-ACK, IRQ Keyboard akan dipending hingga ACK untuk IRQ timer dikirim. Hal ini menjadi kesalahan yang cukup sering ditemui oleh pembaca ketika membuat [**Keyboard Driver**](#).

Selain 8259A PIC, x86 memiliki PIC yang lebih modern yaitu **Advanced Programmable Interrupt Controller (APIC)** yang menyediakan fitur-fitur lebih kompleks dan dapat digunakan untuk kebutuhan **Multiprocessor & Multithreading**.

Informasi lebih detail **APIC** terdapat pada **Intel x86 Manual Vol 3A - Chapter 10 - APIC**.

1.2.2. Interrupt Descriptor Table

Interrupt Descriptor Table (IDT) tidak jauh berbeda dengan GDT yang telah diimplementasikan. Kit akan menyediakan kerangka dasar pada **ch1/2 - Interrupt**

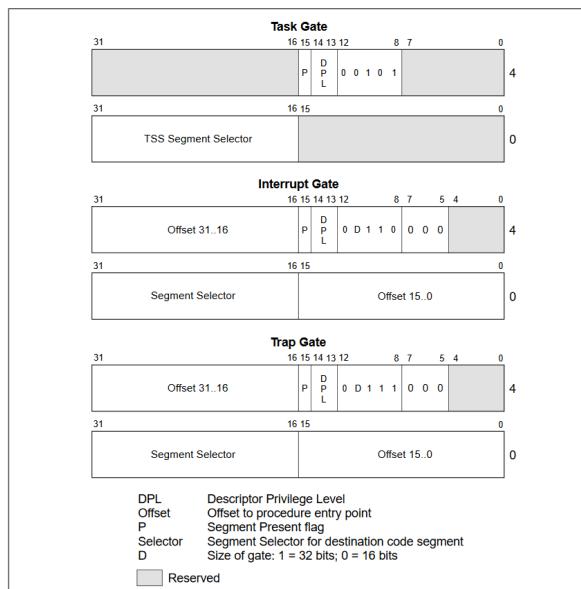
Tugas utama dari bagian ini adalah membuat struktur data yang ada pada IDT

Struktur Data	IDT	GDT Counterpart
Table Entry	IDTGate	SegmentDescriptor
Table	InterruptDescriptorTable	GlobalDescriptorTable
Register	IDTR	GDTR

IDTGate merupakan entry dari **InterruptDescriptorTable**. **IDT Gate** yang akan digunakan bertipe **Interrupt Gate**. Sama seperti **GlobalDescriptorTable**, **InterruptDescriptorTable** menyimpan sebanyak **256** entry IDT Gate (fixed value dari arsitektur x86).

Setelah membuat definisi struktur data IDT, definisikan IDT kosong `interrupt_descriptor_table` dan IDTR yang telah terisi `_idt_idtr` pada `idt.c`. Nantinya kedua variabel ini akan digunakan untuk menyimpan informasi interrupt handler sistem operasi.

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



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

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

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

1.2.3. Interrupt Service Routine

Interrupt Service Routine (ISR) atau **Interrupt Handler** akan dipanggil ketika ada CPU menerima interrupt. Interrupt akan memiliki nomor yang disebut **Interrupt Vector** untuk sebagai identifier sumber interrupt. Ketika CPU menerima interrupt, CPU akan menyiapkan dan [Calling ISR](#) sesuai dengan nomor interrupt yang diterima.

Bagian ini mengimplementasikan array `isr_stub_table` tiga prosedur `call_generic_handler()`, `interrupt_handler_i()`, dan `main_interrupt_handler()`. Singkatnya, hal-hal tersebut menyiapkan kerangka interrupt handler dalam assembly untuk memanggil fungsi C yang lebih mudah diimplementasikan.

Sebagian besar implementasi dari bagian ini akan menggunakan assembly. Kode assembly telah disediakan pada `intsetup.s`.

Struktur data tambahan untuk interrupt sudah terlampir pada `interrupt.h`. Tambahkan kode berikut pada `interrupt.c`

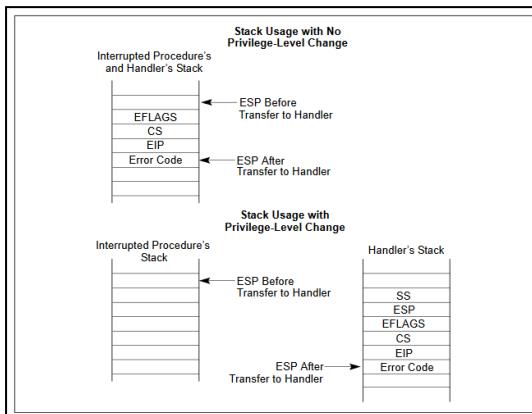
```
interrupt.c
```

```
void main_interrupt_handler(struct InterruptFrame frame) {
    switch (frame.int_number) {
        // ...
    }
}
```

• x86: Handling Intra & Inter Privilege Interrupt

Interrupt dapat dikategorikan menjadi dua berdasarkan pergantian privilege. **Intra Privilege Interrupt** adalah interrupt yang terjadi pada tingkat privilege yang sama (kernel ke kernel). Sedangkan **Inter Privilege** terjadi pergantian privilege (user ke kernel).

Ketika CPU mendapatkan interrupt, CPU akan berhenti dari tugas aktifnya, membaca IDT untuk mencari ISR yang sesuai dengan interrupt vector, menyiapkan call stack, dan memanggil ISR. CPU akan menaruh beberapa informasi tambahan ke call stack yang dapat digunakan ISR.



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

Pada ilustrasi diatas terlihat banyak register dan informasi yang disimpan pada call stack. Prosedur-prosedur assembly yang diberikan pada bagian [Interrupt Service Routine](#) sebenarnya hanya menyiapkan call stack dengan informasi tambahan sebelum memanggil fungsi ISR dalam bahasa C yang lebih mudah untuk diimplementasikan.

Selain itu, jika *perceptive*, terdapat beberapa perbedaan antara ilustrasi diatas dan informasi yang ada pada [InterruptFrame](#). Call stack pada ilustrasi diatas tidak memiliki informasi nomor interrupt atau **Interrupt Vector** sehingga kode asm yang diberikan memasukan interrupt secara manual sebelum memanggil implementasi ISR dalam C.

Struktur data [InterruptFrame](#) juga tidak memberikan akses ke register `ss` dan `esp` untuk inter-privilege interrupt karena tidak ada kebutuhan pada panduan ini. Jika benar-benar memiliki alasan tertentu untuk menggunakan data ini, implementasikan sendiri menggunakan assembly. Gunakan xor antara `cs` dan interrupt stack `cs` untuk membedakan intra/inter-privilege.

Setiap ISR mungkin hanya membutuhkan sebagian kecil atau tidak sama sekali informasi-informasi ini. Agar warning gcc tidak menyebabkan error, tambahkan `--attribute__((unused))` untuk parameter yang tidak digunakan. **Jangan menggunakan atribut ini kecuali terdapat alasan yang kuat** (seperti requirement dari hardware).

1.2.4. Load IDT & Testing Interrupt

Pada tahap ini, IDT telah didefinisikan tetapi masih belum terisi. Namun `isr_stub_table` telah terisi dengan address kerangka ISR yang telah di isi dengan kode assembly. Address-address perlu dimasukkan kedalam `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
    // Use &-bitmask, bitshift, and casting for offset

    // 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;
}
```

Address ISR yang berukuran 32-bit akan disimpan pada struktur data `IDTGate` menjadi dua atribut 16-bit. Gunakan operasi [and-bitmask](#) dan bitshift untuk bagian ini.

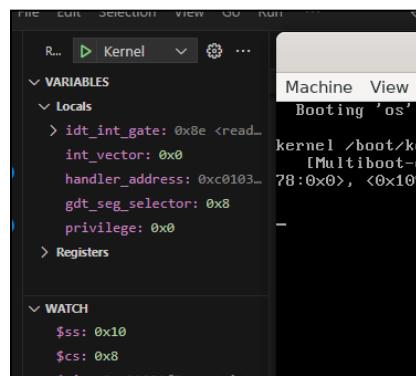
Uji implementasi IRQ remap, IDT, dan ISR dengan kode berikut

kernel.c

```
void kernel_setup(void) {
    load_gdt(&_gdt_gdtr);
    pic_remap();
    initialize_idt();
    framebuffer_clear();
    framebuffer_set_cursor(0, 0);
    __asm__("int $0x4");
    while (true);
}
```

Inline assembly diatas akan menyebabkan Interrupt 4h atau 0x4. Karena implementasi `main_interrupt_handler()` tidak melakukan apapun, interrupt hanya akan memanggil `main_interrupt_handler()` dan melakukan return ke fungsi `kernel_setup()`. Setelah itu sistem operasi seharusnya tidak melakukan apapun.

Jika masih terdapat permasalahan, cobalah untuk mencari penyebabnya menggunakan **Breakpoint** pada `main_interrupt_handler()`. Cek state variabel pada bagian kiri vscode. Detail dari error code CPU exception ada pada [OSDev/Exceptions](#).



Contoh breakpoint pada `set_interrupt_gate()`

Issue yang sering dialami:

- PIC by default menyalakan beberapa IRQ salah satunya adalah **IRQ0: Programmable Interrupt Timer**. Jika melihat `int_number` 0x0 atau 0x20 pada breakpoint `main_interrupt_handler()`, kemungkinan besar IRQ0 belum dimatikan.
- Jika melihat ada `int_number` dibawah 0x20, ada kemungkinan IRQ remap belum diimplementasikan dengan baik.

Bagian **1.2. Interrupt** menjadi landasan untuk semua fitur yang akan diimplementasikan ([IRQ1 - Keyboard Controller](#), [System Calls](#)), pastikan bagian ini sudah berjalan dengan baik.

1.3. Keyboard Driver

Keyboard tentunya adalah hal yang sangat umum dilihat dan disentuh oleh pembaca panduan ini. Pernahkah bertanya bagaimana tombol fisik yang ditekan diolah menjadi huruf pada layar?



Keyboard, Sumber: [Keyboard - Wikipedia](#)

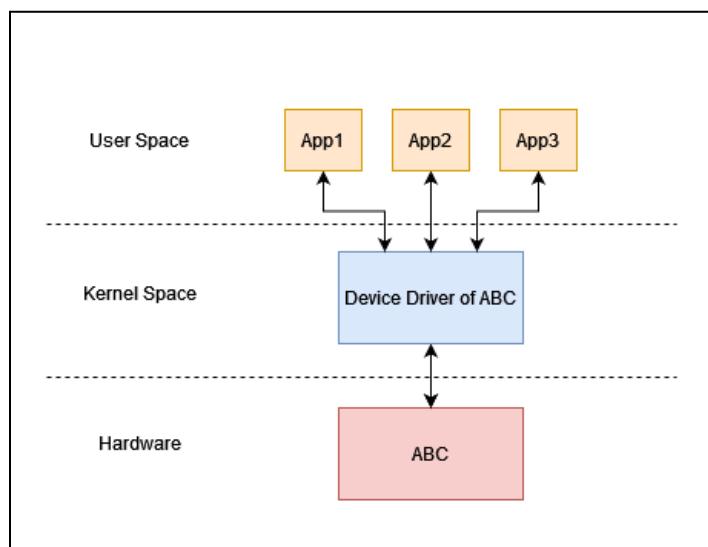
Bagian ini akan menjawab bagaimana komputer memproses tombol keyboard dalam tingkat software. Terdapat hal baru seperti Scancode yang akan ditemui ketika mengerjakan bagian ini.

Bagian ini akan mengimplementasikan Device Driver untuk menyediakan layanan input dari keyboard. Bagian ini akan bermula dari IRQ1 - Keyboard Controller, Keyboard Driver Interface, hingga implementasi Keyboard ISR.

Keyboard driver yang akan dibuat akan berfokus kepada **Generic US Keyboard Layout** dan **Scancode set 1** yang digunakan QEMU.

● Device Driver

Device Driver adalah software yang bertanggung jawab untuk mengabstraksikan interaksi hardware langsung (seperti konfigurasi menggunakan port I/O, menuliskan data ke memory-mapped I/O) dan menyediakan interface yang dapat digunakan kernel dan user program.



Ilustrasi device driver yang hidup pada kernel space

Pada layer abstraksi device driver adalah layer paling rendah pada sisi software yang langsung berkomunikasi dengan hardware. Device driver pada sisi software yang berjalan pada sistem operasi ini akan berkomunikasi dengan layer teratas hardware [Device Controller](#) untuk melakukan operasinya.

OS modern biasanya menyediakan pilihan untuk device driver: **Kernel space** atau **User space**. Windows menyediakan framework untuk menuliskan device driver bernama [Windows Driver Frameworks \(WDF\)](#) yang menyediakan penulisan driver pada kernel (KMDF) dan user space (UMDF). Kernel dan user space akan dibahas lebih mendalam pada pembuatan [User Mode](#).

User space driver memiliki kelebihan stabilitas karena kegagalan device driver hanya menyebabkan terminasi driver program oleh kernel. Kekurangannya adalah overhead low-level interface yang disebabkan harus melewati API OS.

Kernel space driver memiliki satu kekurangan kekurangan yang penting: kode device driver yang menyebabkan exception akan menghasilkan [Double Fault](#). Permasalahan seperti **Blue Screen of Death** setelah melakukan update GPU driver adalah salah satu contoh nyata dari ini. Kelebihan dari kernel space tidak adanya overhead dan dapat langsung mengakses hardware.

1.3.1. IRQ1 - Keyboard Controller

Keyboard controller menggunakan **IRQ1** untuk mengirim notifikasi ke CPU bahwa ada tombol yang ditekan. Bagian IRQ Remapping sebelumnya mematikan semua IRQ sehingga perlu untuk menyalakan IRQ1 dengan mengirimkan nilai mask yang sesuai ke PIC.

Tambahkan kode berikut pada **interrupt.c** untuk menyalakan IRQ keyboard

```
interrupt.c

void activate_keyboard_interrupt(void) {
    out(PIC1_DATA, in(PIC1_DATA) & ~(1 << IRQ_KEYBOARD));
}
```

Panggil kode ini pada **kernel_setup()** milik kernel setelah **initialize_idt()**. Kode diatas akan menyalakan IRQ1 milik keyboard controller. Bit yang bernilai 0 menandai IRQ tersebut diperbolehkan diterima ke CPU.

Setiap kali tombol keyboard ditekan, keyboard akan mengirimkan **IRQ1** ke CPU. Interrupt akan masuk ke **main_interrupt_handler()** dan memanggil **keyboard_isr()**. Terdapat beberapa *quirk* ketika mengimplementasikan ISR yang akan dijelaskan lebih lanjut pada bagian selanjutnya.

1.3.2. Keyboard ISR

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

Konfigurasi yang digunakan oleh QEMU setelah ditest adalah **Generic US Keyboard Layout** dan **Scancode set 1** yang dapat dicek lebih detail pada [OSDev/PS2-Keyboard](#).

Selain itu keyboard QEMU menggunakan mekanisme **Make-Break** untuk menandai tombol yang ditekan. Apa itu Make-Break? Keyboard akan mengirimkan interrupt **tepat dua kali** untuk semua keypress:

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

Akan disediakan map `keyboard_scancode_1_to_ascii_map` dasar yang memetakan semua **Make** scancode ke masing-masing karakter ASCII yang sesuai. Lengkapi dan modifikasi sendiri jika menggunakan skema pemetaan lain untuk make-break.

Tips: **Break scancode** sebenarnya sama persis seperti **Make scancode** dengan bit ke-8 dinyalakan (atau ditambah 128). Contohnya:

Make scancode 'A' = 0x1E = 0b00011110

Break scancode 'A' = 0x9E = 0b10011110

Semua IRQ1 perlu direspon dengan instruksi in ke Keyboard. Hal ini disebabkan oleh *quirk* keyboard controller yang memiliki internal buffer. IRQ1 akan dikirimkan keyboard controller ketika ada tombol ditekan dan menandai scancode siap dibaca oleh CPU. Jika CPU tidak membaca scancode ini, lama-kelamaan internal buffer akan penuh dan **keyboard controller berhenti membaca input hingga buffer dikosongkan**.

Panduan ini hanya meminta keyboard driver mengembalikan data `char` pada `get_keyboard_buffer()`. Jika ingin mengimplementasikan fitur tambahan seperti shift dan ctrl modifier, diperbolehkan untuk memodifikasi behavior ini.

Berikut adalah behavior `keyboard_isr()` yang akan diimplementasikan:

- Keyboard ISR akan dipanggil `main_interrupt_handler()` ketika diterima IRQ1
- **ISR memanggil minimal 1x instruksi port I/O in untuk semua IRQ1**
- ISR mendapatkan scancode dari port KEYBOARD_DATA_PORT
- Ketika `keyboard_state.keyboard_input_on` bernilai true
 - ISR memproses scancode yang diterima ke ASCII character
 - Map `keyboard_scancode_1_to_ascii_map` digunakan untuk memetakan scancode
 - ISR menyimpan karakter ASCII ke `keyboard_state.keyboard_buffer`
- **Dilarang untuk membuat loop di dalam ISR.** Keyboard ISR hanya memproses 1 tombol dan melakukan return
- ISR wajib melakukan `pic_ack()` ke IRQ1 ketika state pembacaan off maupun on
- Dengan behavior-behavior diatas, ISR akan bersifat **non-blocking**

Implementasikan behavior `keyboard_isr()` yang telah dideskripsikan. Jika ingin mengetes hasil implementasi, lanjutkan implementasi interface `keyboard_state_activate()` yang ada pada bagian selanjutnya. Cek bagian [Tips: Keyboard Driver](#) untuk beberapa tips penggerjaan dan trik debugging untuk bagian ini.

Kit untuk keyboard driver tersedia pada [ch1/3 - Keyboard/](#)

Berikut adalah kerangka dasar untuk `keyboard_isr()`

```
keyboard.c

void keyboard_isr(void) {
    uint8_t scancode = in(KEYBOARD_DATA_PORT);
    // TODO : Implement scancode processing
}
```

1.3.3. Keyboard Interface

Pada **keyboard.h** terdapat beberapa deklarasi untuk interface keyboard driver. Interface-interface ini akan digunakan komponen lain untuk mengontrol keyboard.

Definisikan interface yang telah didefinisikan pada **keyboard.h**

- Satu **static** variable **keyboard_state** pada **keyboard.c**
- **keyboard_state_activate()**
Menyalakan pembacaan input keyboard
- **keyboard_state_deactivate()**
Mematikan pembacaan input keyboard
- **get_keyboard_buffer()**
Mengcopy dan **mengosongkan** isi **keyboard_buffer** ke pointer

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

Diperbolehkan untuk melakukan modifikasi kit. Tambahkan sendiri state pada driver, fungsi pembantu, interface baru, dll jika dibutuhkan.

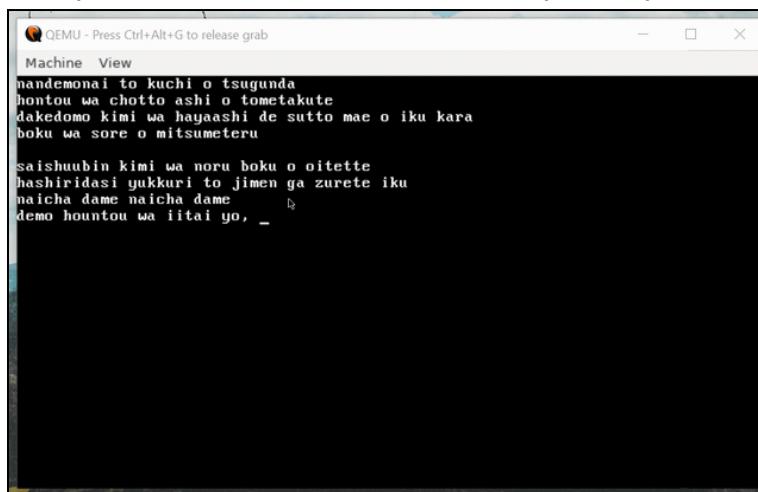
Driver dapat dites dengan menambahkan kode berikut pada kernel

kernel.c

```
void kernel_setup(void) {
    load_gdt(&_gdt_gdtr);
    pic_remap();
    initialize_idt();
    activate_keyboard_interrupt();
    framebuffer_clear();
    framebuffer_set_cursor(0, 0);

    int row = 0, col = 0;
    keyboard_state_activate();
    while (true) {
        char c;
        get_keyboard_buffer(&c);
        if (c) {
            framebuffer_write(row, col, c, 0xF, 0);
            if (col >= FRAMEBUFFER_WIDTH) {
                ++row;
                col = 0;
            } else {
                ++col;
            }
            framebuffer_set_cursor(row, col);
        }
    }
}
```

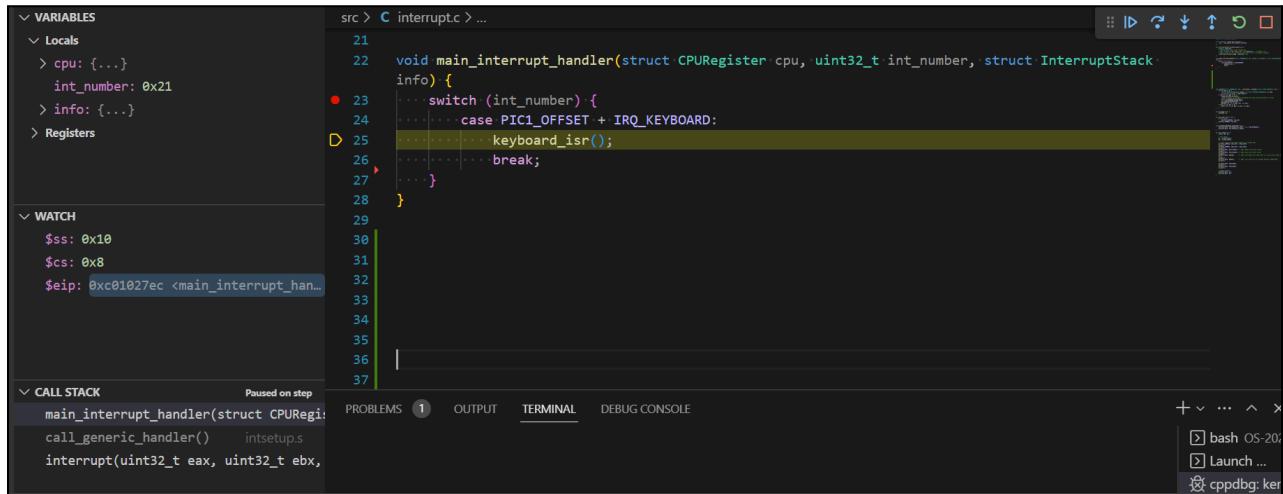
Kode diatas akan mengaktifkan pembacaan keyboard driver. Sistem operasi sekarang harusnya dapat menerima input keyboard dari user dan menampilkannya ke layar.



Keyboard input dengan pemrosesan tambahan

Tips: Keyboard Driver

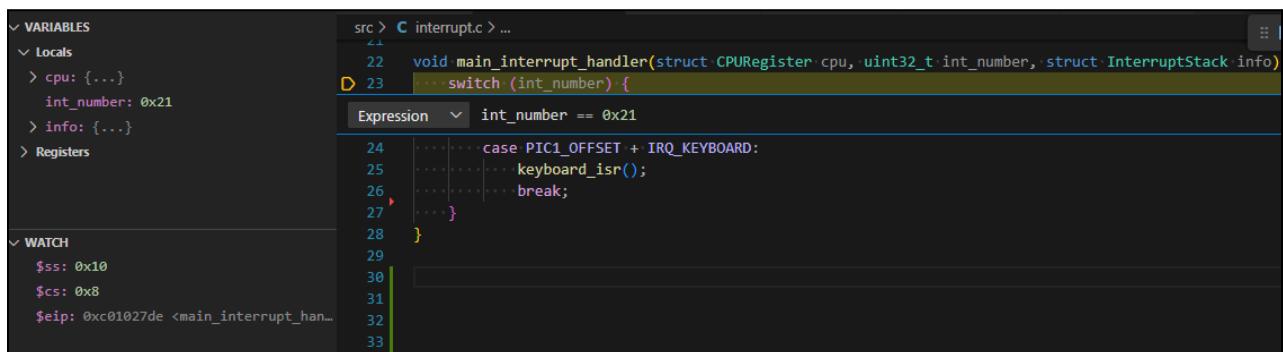
- Jika mengalami permasalahan atau behavior kernel tidak sesuai dengan ekspektasi kode yang dibuat, **gunakan breakpoint pada debugger**.



The screenshot shows a debugger interface with the following details:

- VARIABLES:** Shows local variables `cpu`, `int_number`, and `info`, and registers.
- WATCH:** Shows watchpoints `$ss: 0x10`, `$cs: 0x8`, and `$eip: 0xc01027ec <main_interrupt_han...`.
- CALL STACK:** Shows the stack trace: `main_interrupt_handler(struct CPURegi...)`, `call_generic_handler()`, `intsetup.s`, and `interrupt(uint32_t eax, uint32_t ebx,`. It indicates the code is "Paused on step".
- Code View:** The code for `main_interrupt_handler` is shown, with a red dot at line 23 indicating a breakpoint. The line contains a `switch` statement. The line number 23 is highlighted in yellow.
- Bottom Bar:** Includes tabs for PROBLEMS, OUTPUT, TERMINAL, and DEBUG CONSOLE, along with icons for bash OS-20, Launch ..., and cppdbg: ker.

- **Catatan: Breakpoint akan membuat Break scancode tidak teregister.** Gunakan **logpoint** jika membutuhkan mengecek Break scancode
- Taruh breakpoint pada lokasi yang strategis seperti pada line didekat behavior yang tidak tereksekusi (misalnya menaruh breakpoint pada `switch 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 bahasa C.



The screenshot shows a debugger interface with the following details:

- VARIABLES:** Shows local variables `cpu`, `int_number`, and `info`, and registers.
- WATCH:** Shows watchpoints `$ss: 0x10`, `$cs: 0x8`, and `$eip: 0xc01027de <main_interrupt_han...`.
- Code View:** The code for `main_interrupt_handler` is shown, with a red dot at line 23 indicating a watchpoint. The line contains a `switch` statement. The line number 23 is highlighted in yellow.
- Bottom Bar:** Includes tabs for PROBLEMS, OUTPUT, TERMINAL, and DEBUG CONSOLE, along with icons for bash OS-20, Launch ..., and cppdbg: ker.

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

- Logpoint akan membantu melakukan logging, terutama **Break** scancode yang sulit dicek dengan breakpoint biasa. Tambahkan {} agar ekspresi dievaluasi oleh debugger

The screenshot shows a code editor with a log message and a stack trace. The code is as follows:

```

21
22 void main_interrupt_handler(struct CPURegister cpu, uint32_t int_number, struct InterruptStack info) {
◆ 23     switch (int_number) {
24         case PIC1_OFFSET + IRQ_KEYBOARD:
25             keyboard_isr();
26             break;
27     }
28 }
29
30

```

The line 23 is highlighted with a red diamond, indicating it is a log point. The log message shows:

Log Message ▾ {int_number}

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE Filter (e.g. text, exclude)

23 switch (int_number) {
0x21

Breakpoint 1, main_interrupt_handler (cpu=..., int_number=0x21, info=...) at src/interrupt.c:23
23 switch (int_number) {
0x21

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

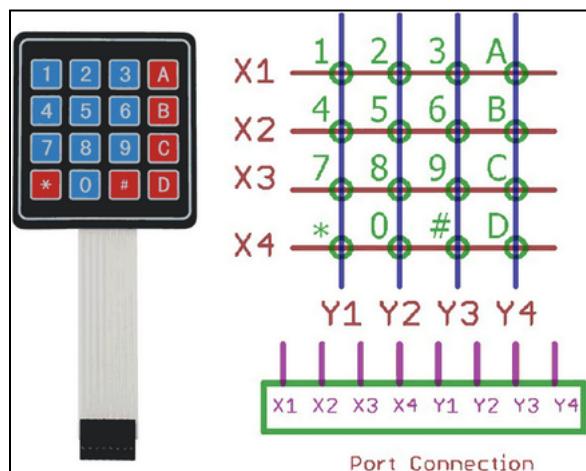
• Keyboard Scancode

Kenapa keyboard tidak mengeluarkan ASCII char saja ketika ditekan?

Mengapa harus mempersulit dengan menggunakan scancode?

Sayangnya karakter yang digunakan pada seluruh dunia tidak hanya alfabet dan bentuk keyboard bermacam-macam. Dengan menggunakan scancode dan mapping driver, keyboard dapat memiliki bentuk yang beragam dan kernel dapat menyediakan metode input untuk karakter lain (Kanji, Cyrillic, Hangul, dll).

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



Arduino matrix keypad, Source: kursuselektronikaku.blogspot.com

Meskipun gambar diatas merupakan keypad arduino, inti cara kerja dari keyboard matrix masih sama. Keyboard matrix menggunakan **Grid Coordinate** untuk mendeteksi tombol, seperti pada gambar diatas. Scancode merepresentasikan koordinat pada grid tersebut.

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**

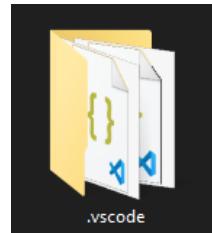
➡ So how does a PS/2 keyboard interface work?

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

➡ Keyboard interface software

Ch. 2 File System: EXT2 - IF2130 Edition

File System adalah sebuah cara untuk mengorganisir data pada komputer. Terdapat banyak macam file system, tetapi panduan ini berfokus ke salah satu file system yaitu **EXT2**.



Sebuah File

Sebelum memulai bagian ini, sangat direkomendasikan untuk menginstall ekstensi vscode: **Hex Editor**. Hex editor vscode dapat mengecek binary file, terutama membaca disk image yang berguna pada bagian ini. Jika tidak menggunakan vscode, gunakan alat lain seperti **HxD** pada Windows atau **hexedit** pada Linux.

Bagian ini akan mengimplementasikan [**File System**](#). File system ini akan menjadi mekanisme utama dari sistem operasi untuk menyimpan data secara persisten. Sangat disarankan untuk mengingat kembali [**Memory Hierarchy**](#) yang telah dijelaskan pada [**IF2130 Organisasi dan Arsitektur Komputer**](#).

File system yang akan dibuat didasarkan pada file system [**EXT2**](#). Desain utama dan sebagian besar akan tetap sama seperti EXT2, tetapi terdapat beberapa simplifikasi untuk mempermudah penggerjaan.

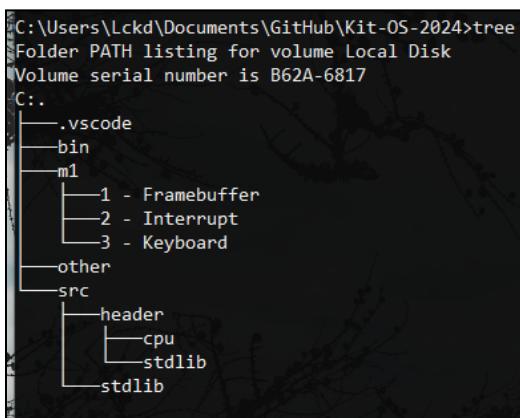
Pengerjaan dimulai dengan membuat [**Disk Driver**](#) untuk menuliskan data ke disk, membuat dan memasang [**Disk Image**](#), penjelasan tentang [**Volatile & Non-Volatile Memory**](#) dan desain [**File System: EXT2 - IF2130 edition**](#).

Setelah penjelasan tentang desain file system yang akan dibuat, **Chapter 2** diakhiri dengan mengimplementasikan interface [**FS: Initializer**](#) dan [**FS: CRUD**](#).

• File System

File System merupakan layanan yang disediakan sistem operasi untuk menyimpan data terstruktur layaknya sebuah **physical file** (Lembar yang biasanya ditaruh didalam **folder coklat muda** pada dunia nyata).

Pada tingkat hardware, **tidak ada penanda apapun** di dalam media penyimpanan selain **angka CHS atau Block Index**. Hal ini mungkin terasa sangat asing jika hanya pernah menyentuh tingkat atas dimana organisasi file system sudah menjadi hal yang taken for granted. Tanpa file system, secondary memory akan menjadi sebuah media penyimpanan kontigu tanpa penanda dimana data disimpan.



```
C:\Users\Lckd\Documents\GitHub\Kit-OS-2024>tree
Folder PATH listing for volume Local Disk
Volume serial number is B62A-6817
C:.
├── .vscode
├── bin
└── m1
    ├── 1 - Framebuffer
    ├── 2 - Interrupt
    └── 3 - Keyboard
├── other
└── src
    ├── header
    │   ├── cpu
    │   └── stdlib
    └── stdlib
```

Contoh hierarchical file system Windows 10 NTFS

Inti dari file system adalah sebuah **sistem yang digunakan untuk menyusun dan melabeli data sehingga mempermudah proses information retrieval**. Setiap kesatuan data akan memiliki label dan metadata sehingga membentuk **File**. Data dalam bentuk file akan lebih mudah diorganisir oleh pengguna dibandingkan mengatur sendiri data pada secondary storage.

File system akan menyediakan interface umum seperti **CRUD** untuk melakukan manipulasi file. Dengan interface CRUD yang disediakan file system, pengguna tidak perlu memperhatikan bagaimana implementasinya pada HDD atau SSD.

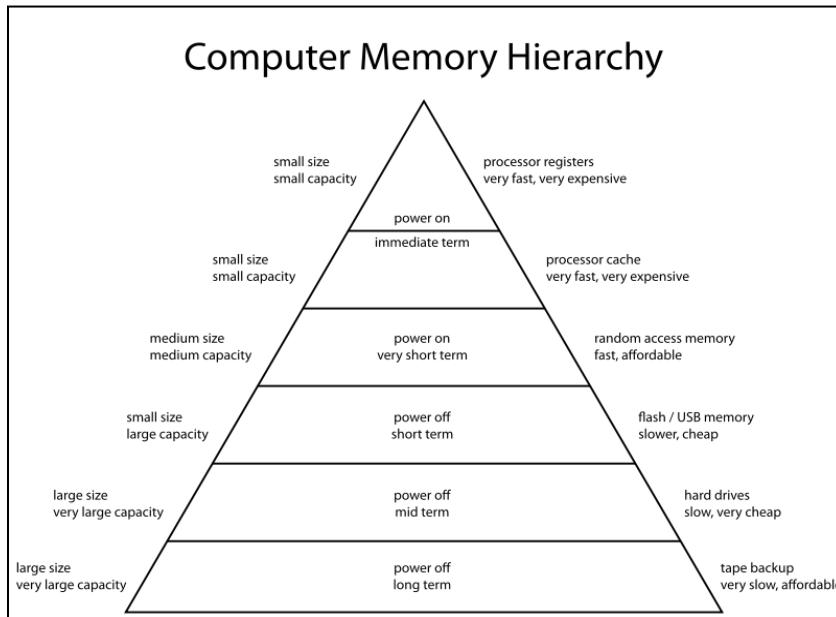
Tentunya cara mengorganisir data pada komputer tidak hanya file system, terdapat beberapa struktur lain yang memiliki fokus berbeda seperti **Database** yang akan diajarkan pada **IF2240 - Basis Data**. Namun umumnya, pada tingkat sistem operasi, yang sering digunakan adalah file system, sedangkan sistem penyimpanan data yang lain biasanya diimplementasikan diatas file system sistem operasi.

● Memory Hierarchy

Orkom, oh orkom. Flashback ke IF2130 Organisasi dan Arsitektur Komputer.

Bagian ini akan menjadi short review materi [Memory Hierarchy](#) yang ada pada IF2130.

Pada komputer terdapat beberapa tipe memory yang biasanya disusun menjadi memory hierarchy.



Memory hierarchy pyramid, [Memory Hierarchy - Wikipedia](#)

Pada ilustrasi diatas terlihat semakin atas piramid, semakin cepat memory, semakin singkat jangka penyimpanannya. Dalam konteks implementasi file system, piramida diatas dapat dibagi menjadi 2 bagian yang sangat penting: [Volatile & Non-Volatile Memory](#).

Memory seperti RAM termasuk [Volatile Memory](#) yang akan hilang ketika listrik dimatikan. Flash drive, HDD, SSD termasuk [Non-Volatile Memory](#) yang akan tetap menyimpan data meskipun tidak ada listrik.

Penyimpanan CPU Register dan Cache adalah penyimpanan yang paling cepat diakses oleh CPU yang disebabkan oleh letak yang sangat dekat dengan CPU. Berbeda dengan penyimpanan lainnya, cache bersifat **transparan** karena didesain hanya untuk mempercepat I/O.

2.1. Disk Driver

Sebelum membuat file system, diperlukan sebuah cara untuk menuliskan data ke non-volatile storage. Bagian ini akan mengimplementasikan [ATA](#) disk driver yang menyediakan interface untuk membaca dan menuliskan data dalam unit [Block](#). Interface yang disediakan akan menggunakan [Logical Block Addressing](#) sebagai skema adresing.

Kedua interface yang akan diimplementasikan akan bersifat [Blocking](#), CPU akan melakukan [Busy Waiting](#) hingga operasi R/W disk selesai.

Sebagian besar kode untuk bagian ini akan disediakan untuk mempermudah penggerjaan. Kode disk driver akan menggunakan instruksi port I/O 16-bit untuk mengirim dan menerima data dari disk controller. Tambahkan kode berikut pada [portio.c](#)

```
portio.c

void out16(uint16_t port, uint16_t data) {
    __asm__(
        "outw %0, %1"
        : // <Empty output operand>
        : "a"(data), "Nd"(port)
    );
}

uint16_t in16(uint16_t port) {
    uint16_t result;
    __asm__ volatile(
        "inw %1, %0"
        : "=a"(result)
        : "Nd"(port)
    );
    return result;
}
```

Deklarasi interface disk driver telah disediakan pada [disk.h](#) sehingga hanya perlu menambahkan implementasi dari interface. Setelah mengimplementasikan [disk.c](#), lanjutkan hingga bagian [Disk Image](#) selesai untuk menguji dan mengenal dev & debug cycle yang akan ditemui nantinya.

Berikut adalah implementasi untuk **disk.c**

disk.c

```
#include "header/driver/disk.h"
#include "header/cpu/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);
        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();
        for (uint32_t j = 0; j < HALF_BLOCK_SIZE; j++)
            out16(0x1F0, ((uint16_t*) ptr)[HALF_BLOCK_SIZE*i + j]);
    }
}
```

• ATA: PATA & SATA

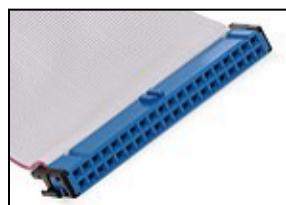
Berbeda dengan 16-bit real mode yang memiliki layanan disk I/O dari [BIOS Interrupt Call](#), tidak tersedia layanan serupa pada protected mode x86 sehingga disk driver akan diimplementasikan menggunakan interface **Advanced Technology Attachment** atau **ATA**.

ATA merupakan interface standard yang sering digunakan pada komputer untuk media penyimpanan lama. Pada komputer modern, salah satu host bus adapter yang mengimplementasikan ATA yang masih lazim digunakan adalah **Serial ATA** atau **SATA**.



2.5-inch SATA drive, [SATA - Wikipedia](#)

SATA sendiri merupakan evolusi dari host bus adapter **Integrated Drive Electronic (IDE)** yang juga disebut **Parallel ATA** atau **PATA**. Jika pernah mengoprek komputer lama, semestinya pernah melihat sebuah kabel pita yang cukup lebar untuk menghubungkan motherboard ke storage device



Ancient cable, [PATA - Wikipedia](#)

Kabel tersebut sering dipakai untuk menghubungkan HDD ke motherboard pada era komputer lama hingga sekitar awal **Intel Pentium 3** (2004). Terlepas perbedaan antara physical electrical interface (serial / parallel), PATA dan SATA sama-sama mengimplementasikan ATA interface untuk pengoperasiannya oleh CPU.

Perlu dicatat bahwa driver pada bagian sebelumnya menggunakan **ATA PIO** yang bersifat **blocking**. Hal ini disebabkan operasi blocking I/O akan lebih sederhana untuk digunakan pembuatan file system.

2.2. Disk Image

Perlu diperhatikan bahwa sistem operasi pada saat ini **belum memiliki non-volatile storage**. Data yang ada akan hilang ketika OS dimatikan. Bagian ini akan menambahkan otomasi untuk membuat sebuah disk image untuk media penyimpanan persistent dari sistem operasi.

Tambahkan recipe berikut pada **makefile**

```
makefile

DISK_NAME      = storage

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

Command diatas akan membuat sebuah disk image dengan format raw binary berukuran **4 MiB**. Disk image ini akan di-attach ke QEMU sebagai media penyimpanan HDD. Semua operasi **write_blocks()** akan bersifat **persistent** meskipun QEMU dimatikan.

Perlu diketahui sebaiknya recipe **disk** tidak selalu dijalankan ketika melakukan **run**. Lebih baik menjalankan recipe tersebut secara **manual**, karena **semua data pada disk image akan terhapus** ketika command **qemu-img** dijalankan.

Tambahkan parameter seperti berikut pada semua script yang menjalankan QEMU untuk menghubungkan disk image ke QEMU

```
makefile & .vscode/.tasks

qemu-system-i386 -s -S -drive file=storage.bin,format=raw,if=ide,index=0,media=disk
-cdrom OS2025.iso
```

Untuk menguji apakah disk image dan driver dapat berjalan, tambahkan kode berikut pada kernel

```
kernel.c

void kernel_setup(void) {
    load_gdt(&_gdt_gdtr);
    pic_remap();
    activate_keyboard_interrupt();
    initialize_idt();
    framebuffer_clear();
    framebuffer_set_cursor(0, 0);

    struct BlockBuffer b;
    for (int i = 0; i < 512; i++) b.buf[i] = i % 16;
    write_blocks(&b, 17, 1);
    while (true);
}
```

Buka **storage.bin** menggunakan hexeditor untuk mengecek apakah operasi penulisan berhasil

Address	Value	Decoded Text	Data Inspector
000021B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	binary 00000000 octal 000 uint8 0 int8 0 uint16 256 int16 256 uint24 131328 int24 131328 uint32 50462976 int32 50462976 int64 506097522914230528 uint64 506097522914230528 float16 0.0000152587890625 bfloat16 2.350988701644575e-38 float32 3.820471434542632e-37 float64 7.949928895127363e-275 UTF-8
000021C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	UTF-16 Ä
000021D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
000021E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
000021F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00002200	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	
00002210	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	
00002220	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	
00002230	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	
00002240	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	
00002250	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	
00002260	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	
00002270	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	
00002280	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	
00002290	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	
000022A0	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	
000022B0	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	
000022C0	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	
000022D0	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	
000022E0	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	

Pengecekan storage.bin menggunakan vscode hexeditor

Gambar diatas menunjukkan contoh operasi penulisan `write_blocks()` yang berhasil. Kode kernel akan menuliskan 1 block ke logical block address 17. Berikut adalah kalkulasi untuk mendapatkan lokasi penulisan pada **storage.bin**

- Definisi ukuran 1 block pada buku ini: **1 block = 512 bytes = 0x200 bytes**
- LBA menggunakan zero-indexed sehingga block 0 terletak pada Byte **0 (0x0)** hingga **511 (0x1FF)**
- Maka logical block address ke 17 memiliki byte offset $0x200 * 17 = 0x200 * 0x11 = 0x2200$
- Offset akan berguna untuk melakukan lompat dengan hexeditor (Keybind umum: **Ctrl + G**)

Pastikan untuk menguji disk driver dan mencoba menggunakan hexeditor seperti diatas.

Nantinya hal ini menjadi bagian dari dev-debug cycle sewaktu membuat implementasi [File System: EXT2 - IF2130 Edition.](#)

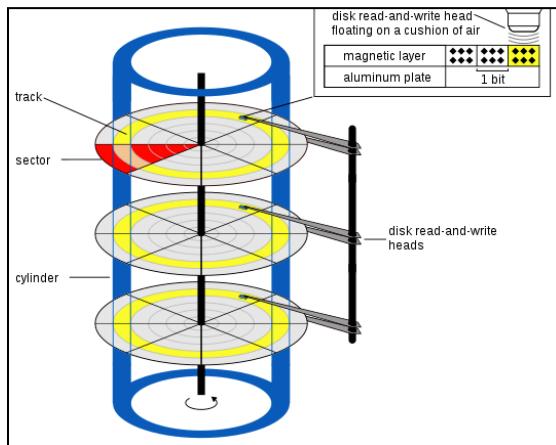
Selain itu disarankan untuk membaca dua bagian tambahan selanjutnya terlebih dahulu:

[Disk Addressing: LBA & CHS](#)

• Disk Addressing: LBA & CHS

Media penyimpanan non-volatile HDD atau SSD akan terlihat seperti **penyimpanan kontigu tanpa label atau metadata apapun**. Satu-satunya mekanisme yang disediakan oleh media penyimpanan tersebut adalah disk addressing. Terdapat dua skema disk addressing yang sering digunakan dalam konteks file system dan sistem operasi: **Logical Block Addressing (LBA)** dan **Cylinder-Head-Sector (CHS)**.

CHS merupakan skema addressing yang didasarkan dari media penyimpanan berbasis piringan



Sumber : [CHS - Wikipedia](#)

Dengan menggunakan addressing CHS pada media penyimpanan piringan, file system dapat melakukan optimisasi letak file secara fisik. Karena RPM pada media penyimpanan piringan bernilai konstan, bagian terluar akan terbaca lebih banyak dalam waktu yang sama ($l = (\omega t)r$ dan panjang segmen piringan l sebanding dengan jumlah data terbaca). Perbedaan kecepatan pembacaan ini menyebabkan kecenderungan untuk menaruh file-file penting seperti file OS dibagian terluar drive.

Namun pada komputer modern, CHS jarang digunakan meskipun menggunakan media penyimpanan piringan seperti HDD. Skema disk addressing **Logical Block Address** mengabstraksikan media penyimpanan dengan menyediakan unit terkecil bernama **Block** untuk pembacaan dan penulisan.

Driver yang diberikan akan menggunakan skema addressing LBA sehingga **semua data akan disimpan dalam kelipatan bilangan bulat dari block**. File yang sebenarnya memiliki ukuran lebih kecil dari 1 block akan menggunakan 1 block pada physical storage.

Ukuran satu block selalu kelipatan dua dan dapat berbeda dari sistem ke sistem tetapi pada buku ini akan didefinisikan bahwa **1 block = 512 bytes**. Namun banyak sistem modern yang menggunakan pengaturan 1 block = 4096 bytes terutama Windows.

- **EXT2: Block & Inode**

EXT2 (Second Extended File System) adalah salah satu file system populer yang digunakan dalam sistem operasi Linux. Di dalam EXT2, dua konsep kunci yang memainkan peran penting adalah **Block** dan **Inode**. Berikut adalah penjelasan tentang keduanya:

1. Block

- Block adalah unit dasar penyimpanan data dalam file system. File besar dipecah menjadi bagian-bagian kecil yang disebut block.
- Ukuran block biasanya berkisar antara 1 KB hingga 4 KB. Ukuran yang lebih besar mengurangi overhead metadata, tetapi bisa menyebabkan lebih banyak *wasted space* (fragmentasi).
- Block digunakan untuk menyimpan konten aktual dari file, seperti teks, directory table, gambar, dan lainnya
- **Bitmap Block:** Bitmap adalah struktur metadata yang digunakan untuk melacak block mana yang sudah terpakai dan yang masih kosong.

2. Inode

- Inode adalah struktur data yang menyimpan informasi metadata tentang file atau direktori. Inode **bukan** data file itu sendiri.
- Informasi dalam Inode pada umumnya mencangkup:
 - Lokasi block data yang terkait dengan file.
 - Informasi tentang pemilik file (user ID dan group ID).
 - Izin file (read, write, execute).
 - Timestamp (waktu modifikasi, waktu akses, dll.).
 - Ukuran file.
- **Indirection:**
 - Jika file kecil, inode dapat menunjuk langsung ke block yang berisi data.
 - Untuk file besar, inode juga menggunakan indirect pointer (single, double, triple indirect) untuk merujuk ke block tambahan secara bertingkat.

Setiap file atau direktori di EXT2 memiliki satu inode. Inode berisi daftar block yang digunakan file tersebut. Block itulah yang menyimpan data aktualnya. Inode memastikan efisiensi dalam penyimpanan dan pengelolaan file karena metadata dan data terpisah.

2.3. Volatile & Non-Volatile Memory

Bagian ini akan menjadi short refresher untuk volatile & non-volatile memory. Pada konteks file system, selalu ingat untuk membedakan **RAM** yang bersifat **volatile** dan **media penyimpanan / storage (HDD, SSD, dll)** yang bersifat **non-volatile / persistent**. Pada panduan ini, storage yang digunakan sistem operasi berwujud sebuah file bernama **storage.bin**.

Semua variabel yang didefinisikan pada kode C akan hidup pada volatile memory (RAM)

```
int main(void) {
    uint8_t abc;
    char c;
    char *s = "hello";
    return 0;
}
```

Semua variabel (abc, c, s) diatas hanya akan ada ketika sistem operasi dijalankan dan hilang ketika QEMU diberhentikan. Hal ini sangat penting untuk diingat ketika mengimplementasikan file system. Developer file system harus mengetahui mana yang terletak pada RAM, mana yang storage.

Semisal ada kode membaca data dari storage menggunakan `read_blocks()` dan melakukan manipulasi pada variabel

```
int main(void) {
    struct BlockBuffer b;
    read_blocks(&b, 42, 1);
    for (int i = 0 ; i < 128; i++) b.buf[i] = 1;
    return 0;
}
```

Setelah kode diatas dijalankan:

- Apakah data yang ada pada storage berubah?
- Bagaimana caranya untuk mengubah data pada storage?

Kedua pertanyaan seharusnya dapat dijawab setelah mencoba dan memahami bagian [Disk Driver](#) dan [Disk Image](#).

Jika masih belum bisa menjawab pertanyaan diatas dengan yakin, **sangat disarankan untuk kembali bagian sebelumnya** untuk mencoba-coba dengan interface disk driver terlebih dahulu.

2.4. FS: Design of EXT2 - IF2130 Edition

Selamat datang ke salah satu subsistem kompleks pada buku ini!

Berbeda dengan bagian-bagian sebelumnya yang memberikan beberapa implementasi secara langsung, bagian ini hanya akan menggambarkan bagaimana file system **EXT2 - IF2130 Edition** bekerja dan behavior-nya.

Untuk memudahkan penggerjaan file system, **semua definisi struktur data akan disediakan pada kit**. Selain itu cek juga bagian [References](#) yang mencantumkan beberapa sumber lain jika dirasa penjelasan yang ada pada buku ini kurang jelas.

Kit yang berisikan definisi struktur data dan konstanta tersedia pada [ch2/4 - Filesystem/](#)

Penting: Jika merasa masih belum paham, tetap lanjutkan membaca hingga akhir bagian ini sebelum mengulangi lagi.

Terdapat beberapa penjelasan untuk “FAQ” dan ilustrasi tambahan pada bagian-bagian selanjutnya yang mungkin membantu sebelum melakukan pembacaan yang kedua.

2.4.1. Overview & Terminology

Sebelum melanjutkan membahas file system EXT2 yang akan diimplementasikan, berikut adalah beberapa istilah-istilah penting yang akan digunakan pada penjelasan desain EXT2

Istilah	Arti pada EXT2 - IF2130 Edition
Directory	Sinonim dengan Folder. Sebuah struktur yang digunakan untuk mengkatalogkan file dan direktori lain. Struktur ini akan menyimpan referensi ke file dan folder lain.
File	Sekelompok data yang memiliki identifier dan metadata. File adalah unit penyimpanan data terkecil pada file system.
Metadata	Data yang mendeskripsikan file seperti filename, filesize, attribute, dan lain-lain.
Block	Unit dasar penyimpanan data dalam partisi. Setiap file dan directory disimpan dalam block yang memiliki ukuran tetap, misalnya 1 KB, 2 KB, atau 4 KB.
Block Group	Unit manajemen yang berisi sejumlah block data dan inode. EXT2 membagi partisi menjadi beberapa block group untuk meningkatkan efisiensi akses data dan mengurangi fragmentasi.
Superblock	Struktur data penting yang menyimpan informasi global tentang file system, seperti ukuran sistem file, jumlah blok, jumlah inode, ukuran block, serta informasi status dan metadata lainnya
Bitmap	Struktur data yang digunakan untuk melacak alokasi block dan inode dalam sistem file. Berisi map binary, 1 jika terisi, 0 jika kosong.
Inode	Struktur data yang menyimpan informasi tentang file atau direktori, seperti ukuran file, hak akses, waktu modifikasi, jumlah link, serta daftar pointer ke block data yang menyimpan isi file tersebut.

EXT2 merupakan file system yang menggunakan bitmap dan tabel inode untuk menyimpan informasi alokasi dan metadata akan disimpan pada tabel lain milik directory ([File & Directory](#)).

Dua struktur yang menjadi building block file system adalah [File & Directory](#). Hanya dua struktur ini yang bersifat transparan ke pengguna. Pengguna hanya dapat melihat dan memanipulasi file & directory melewati interface yang disediakan.

Semua file dan folder akan memiliki [Ancestor Node](#) yang sama bernama [Root Directory](#). Membaca file system EXT2 dimulai dengan membaca root directory yang akan memiliki subdirectory dan hubungan ke semua file & folder lain.

Bagian selanjutnya akan terdapat ilustrasi **Logical View** dan **Physical View**. Physical view akan memperlihatkan bagaimana struktur data akan terlihat pada **memory / hex editor** tanpa label atau penanda apapun pada data. Logical view akan menunjukkan nama / label dari data yang sebenarnya tidak ada pada media penyimpanan dan digunakan hanya sebagai pembantu ilustrasi.

2.4.2. Block Group

EXT2 menggunakan struktur data **Block Group** untuk menyimpan informasi **alokasi inode kosong**. Kit akan menamai struktur data ini dengan `EXT2BlockGroupDescriptor`. Sebelum melanjutkan penjelasan, berikut adalah contoh `EXT2BlockGroupDescriptor` yang telah di-init dan masih kosong

Logical View

Storage						
Block Group	Superblock	Group Descriptor	Block Bitmap	Inode Bitmap	Inode Table	Data Blocks
0	✓	✓	✓	✓	✓	✓
1	✗	✓	✓	✓	✓	✓
2	✗	✓	✓	✓	✓	✓
...

Physical View

Storage					
Block Group	Descriptor Entry	Block Bitmap	Inode Bitmap	Inode Table Start	Data Block Start
0	0x00000001	0xFF00FF00	0xFFFF0000	0x00001000	0x00002000
1	0x00000002	0xFFFF0000	0xFF00FF00	0x00003000	0x00004000
2	0x00000003	0xFF00FF00	0xFFFF0000	0x00005000	0x00006000
...

EXT2 menggunakan struktur data Block Group untuk mengelola alokasi inode dan blok secara efisien. Setiap block group memiliki beberapa komponen penting, termasuk **Superblock**, **Group Descriptor**, **Block Bitmap**, **Inode Bitmap**, **Inode Table**, dan **Data Blocks**. Namun, tidak semua block group memiliki superblock, karena hanya beberapa yang menyimpan salinan superblock sebagai cadangan untuk keperluan pemulihan data.

Pada Logical View, setiap block group memiliki struktur yang sama, kecuali untuk superblock yang hanya terdapat di beberapa block group tertentu. Komponen seperti Group Descriptor, Block

Bitmap, Inode Bitmap, Inode Table, dan Data Blocks selalu ada dalam setiap block group. Simbol menunjukkan bahwa bagian tersebut ada dan digunakan, sementara menunjukkan bahwa bagian tersebut tidak ada di block group tertentu.

Sementara itu, pada Physical View, representasi penyimpanan secara fisik menunjukkan bagaimana data benar-benar disusun di disk. Setiap block group memiliki alamat penyimpanan unik, ditunjukkan dengan nilai 0x00000001 pada block group pertama.

Struktur Block Group pada EXT2 memberikan keuntungan dalam hal efisiensi manajemen file dibandingkan sistem seperti FAT32. Dengan adanya [EXT2BlockGroupDescriptor](#), sistem dapat dengan mudah mengelola lokasi bitmap, tabel inode, dan blok data, memungkinkan akses lebih cepat serta pengurangan fragmentasi pada filesystem.

2.4.3. File & Directory

Tentunya penjelasan [EXT2BlockGroupDescriptor](#) masih memiliki beberapa pertanyaan lanjutan

- Jika BGD tidak membawa informasi data dari file, **dimana data dari file disimpan?**
- BGD juga tidak menyimpan data directory, **bagaimana mengetahui struktur directory?**

Struktur data [EXT2BlockGroupDescriptor](#) memang tidak menyimpan data dari file langsung, **tetapi terdapat struct EXT2DirectoryEntry** yang menyimpan struktur direktori dan pointer ke data sebuah file. Mungkin penjelasan tersebut masih terasa abstrak, berikut adalah kasus contoh: 1 file dan 1 folder pada root

Directory Tree	
root/	 - ini_folder/ - ini_file

Directory Entry

(dalam sebuah directory, root/)

Bytes offset	Size bytes	Description
Directory Entry 0		
0	4	inode = 783362
4	2	rec_len = 12
6	2	name_len = 1
8	1	name = .
9	3	Padding
Directory Entry 1		
12	4	inode = 1109761
16	2	rec_len = 12
18	2	name_len = 2
20	2	name = ..
22	2	Padding
Directory Entry 2		
24	4	inode = 7833624

28	2	rec_len = 14
30	2	name_len = 10
32	10	name = ini_folder
42	2	Padding
Directory Entry 3		
44	4	inode = 783363
48	2	rec_len = 16
50	2	name_len = 8
58	8	name = ini_file
60	2	Padding

Inode File

ini_file	
i_mode	Flag untuk folder/file.
i_size	Size file dalam byte.
i_blocks	Jumlah total block 512-byte yang direservasikan untuk node (tidak sama dengan BLOCK_SIZE).
i_block	15 x 32-bit pointer yang menunjuk ke block-block yang berisi data untuk inode

2.4.3.1 File EXT2

File adalah entitas yang menyimpan data aktual. Dalam EXT2, setiap file direpresentasikan oleh struktur data yang disebut dengan **inode**. Inode menyimpan metadata tentang file, termasuk tipe file, izin akses, userID dan groupID, ukuran file, timestamp, serta pointer ke blok data pada disk. Data aktual disimpan dalam blok-blok data, dan inode menyimpan pointer ke blok-blok ini. Pointer ini bisa berupa **direct pointer** untuk file yang berukuran kecil atau **indirect pointer** untuk file yang berukuran lebih besar yang nantinya akan menunjuk ke blok yang berisi **pointer** lain ke blok data.

Pencarian file pada EXT2, sistem file mencari **inode file** berdasarkan nama file yang diberikan dengan menelusuri direktori yang ada. Setelah **inode** ditemukan, sistem file akan membaca **metadata** yang ada dan menggunakan pointer dalam inode untuk mengakses blok data yang berisi konten file.

2.4.3.2 Directory EXT2

Direktori dalam EXT2 adalah file khusus yang digunakan untuk mengorganisir file dan subdirektori lainnya. Direktori berisi daftar **entry** yang disebut dengan **direntry** (directory entry), yang menghubungkan nama file atau subdirektori dengan **inode**-nya. Setiap direntry berisi nama file atau subdirektori, nomor inode yang terkait, dan panjang entry. Pada direktori yang ada terdapat 2 entry khusus yang selalu ada yaitu, . (dot) dan .. (dot-dot), yang mana . (dot) merujuk ke dirinya sendiri sedangkan ..(dot-dot) merujuk ke direktori parent. Ketika mengakses direktori, sistem file akan membaca **inode directory**, yang menyimpan pointer ke blok data berisi **entry-entry directory** lainnya. Dengan membaca blok data ini, sistem file akan menampilkan daftar file dan subdirektori lain yang ada di dalam.

2.4.4. Root Directory & Interaction

Penjelasan sebelumnya membahas bagaimana file dan folder disimpan dan dibaca.

Namun masih ada pertanyaan terkait pembacaan awal file system yang belum terjawab

- **Bagaimana cara mengetahui semua directory tree?**

Disini **Root Directory** (atau `/`) yang berperan sebagai ancestor node semua file & folder dan dijamin selalu ada berperan pada file system. Pembacaan awal untuk membentuk directory tree file system dimulai dari membaca root directory terlebih dahulu. Setelah itu digunakan algoritma traversal rekursif ke semua subdirectory untuk mendapatkan satu level tree dibawah hingga terminasi.

Poin utama pada EXT2 - IF2130 Edition telah dijelaskan satu per satu secara independen. Untuk menggambarkan bagaimana poin ini berinteraksi satu sama lain dan membentuk satu keutuhan file system, berikut adalah contoh sebuah file system EXT2 - IF2130 Edition yang telah diisi oleh beberapa file dan folder

Directory Tree
<pre>root/ └─ folder1/ ├─ nestedf1/ │ ├─ tpazolite │ └─ xi ├─ folder2/ └─ laur └─ sasakure</pre>

Berikut adalah contoh isi dari directory table root dan folder1. Setiap kolom akan berisikan nama file atau folder.

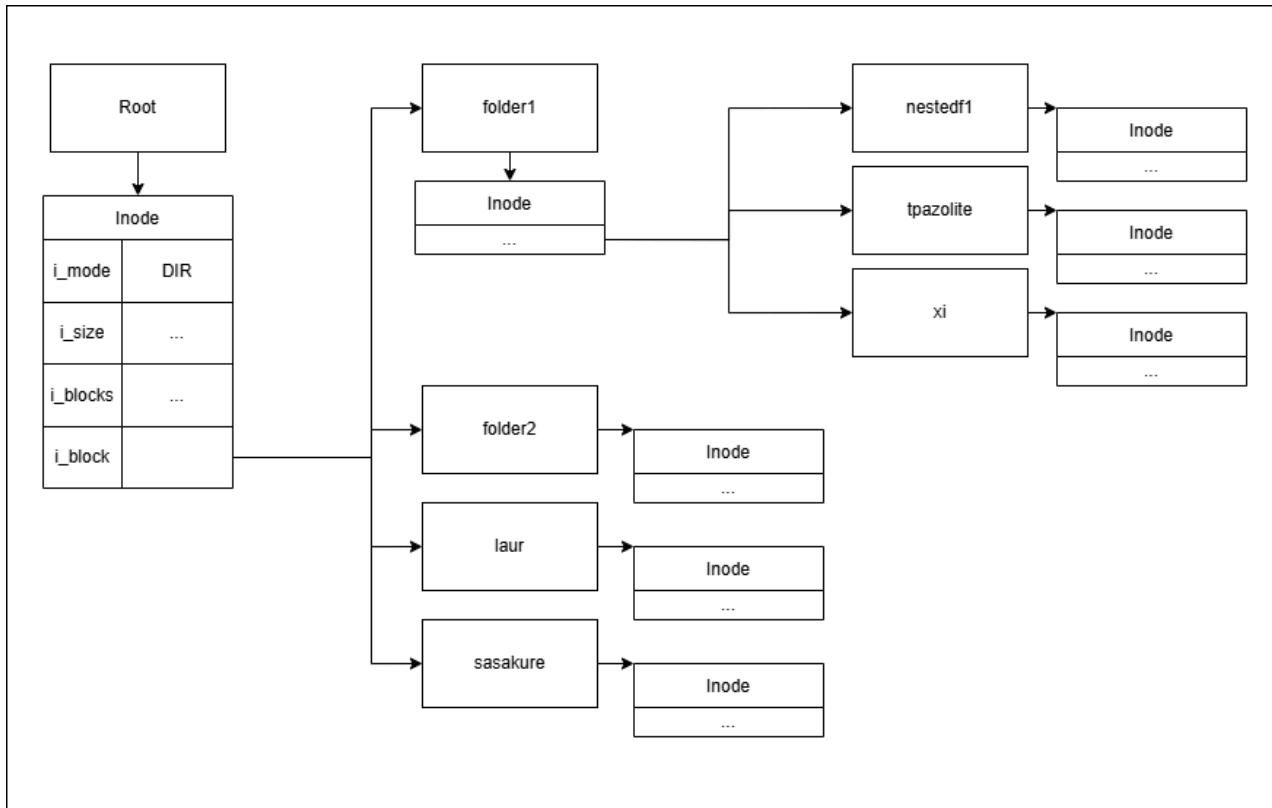


Diagram ini menunjukkan **struktur pohon direktori** dalam sistem file ext2. Dalam **ext2**, setiap file dan direktori direpresentasikan oleh sebuah **Inode**. Bagian paling atas adalah **Root Directory** (direktori utama), yang berisi sebuah Inode. Inode menyimpan metadata seperti:

- **i_mode**: Menunjukkan bahwa ini adalah sebuah **direktori (DIR)**.
- **i_size**, **i_blocks**, dan **i_block**: Menyimpan informasi tentang ukuran direktori, jumlah blok yang digunakan, dan pointer ke blok data.

Setiap file dan direktori memiliki **Inode** sendiri. Bedanya, **i_block** pada file menunjuk ke block yang menyimpan isi data file tersebut, sedangkan pada **i_block** pada direktori menunjuk ke block yang menyimpan list of EXT2DirectoryEntry.

Bagaimana caranya data tersebut “disimpan” dalam pointer-pointer pada **i_block**? **i_block** hanya menyimpan 15x32 bit integer sebagai pointer. Dengan block size 512 bytes, secara teori setiap inode hanya dapat menyimpan $15 \times 512 = 7,68$ KB. Hal ini cukup untuk file dan direktori yang tidak memerlukan banyak data. Bagaimana jika filenya berukuran 1MB, 2MB, 100MB atau bahkan 1GB?

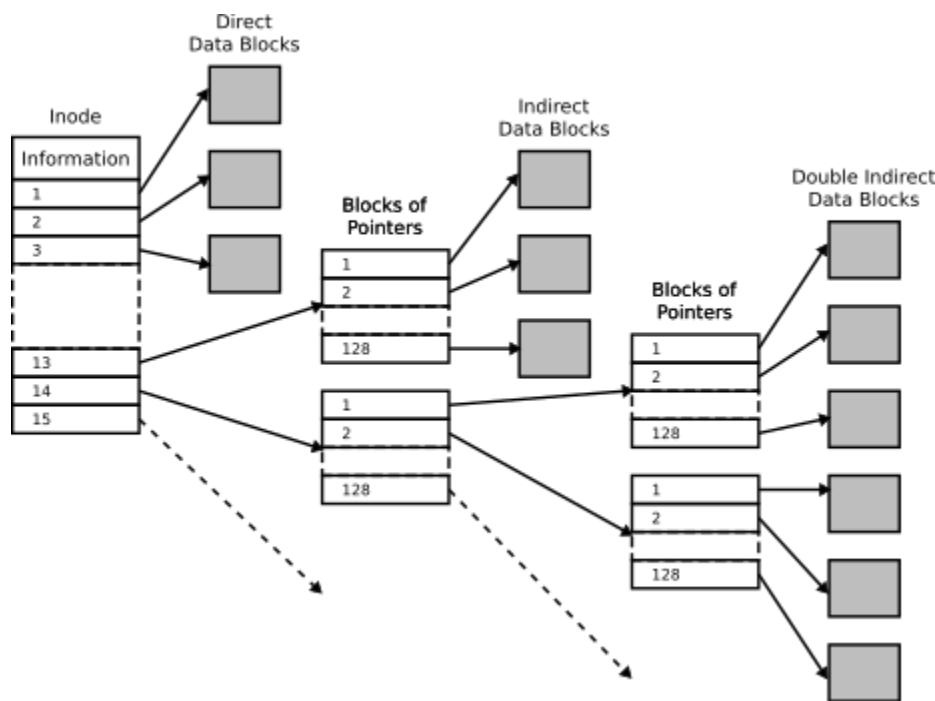
Untuk mencapai hal tersebut, **i_block** menggunakan mekanisme indirect passing:

- 12 pointer pertama adalah **direct block**, menunjuk langsung ke blok data file.

- Entri ke-13 adalah pointer pertama **indirect block**, menunjuk ke sebuah blok yang berisi array dari ID block yang menyimpan data.
- Entri ke-14 adalah pointer pertama **doubly-indirect block**, menunjuk ke sebuah blok yang berisi array dari ID indirect block.
- Entri ke-15 adalah pointer triply-indirect block, menunjuk ke sebuah blok yang berisi array dari doubly-indirect block ID.

Dengan struktur seperti ini, dan setiap block of pointer menyimpan $512/4 = 128$ buah 32-bit pointer, setiap inode dapat menyimpan sebanyak:

$$12 \times 512 + 128 \times 512 + 128^2 \times 512 + 128^3 \times 512 \text{ bytes} = 6 \text{ KB} + 64 \text{ KB} + 8 \text{ MB} + 1 \text{ GB} = 1 \text{ GB data.}$$



Diambil dari <https://en.wikipedia.org/wiki/Ext2>

Sebagai referensi pembantu terkait bagaimana inode menyimpan data, dapat ditonton video singkat berikut: [Inode Structure](#)

Side note:

- *Direct pointer: pointer ke block*
- *Indirect pointer: pointer ke block of pointer ke block*
- *Doubly indirect pointer: pointer ke block of pointer ke block of pointer ke block*
- *Triply indirect pointer: pointer ke block of pointer ke block of pointer ke block of pointer ke block*

2.4.5. Design & Constraint

Sesuai nama, file system **EXT2 - IF2130 Edition** adalah EXT2 yang dimodifikasi. Modifikasi-modifikasi yang ditambahkan bertujuan untuk mempermudah implementasi. Selain modifikasi yang disebutkan, behavior file system akan sama dengan EXT2.

Berikut merupakan beberapa modifikasi desain EXT2 yang akan diimplementasikan:

- **Boot Sector & EXT2 Configuration**
 - Boot sector hanya menyimpan `fs_signature`
 - Isi `fs_signature` diperbolehkan dimodifikasi (misalnya isi dengan nama sendiri)
 - Parameter file system akan dihardcode dengan macro
- **Reserved Inode**
 - Boot sector - IF2130 File System Signature: `fs_signature`
- **Inode Entry**
 - Setiap folder dan file pada EXT2 **memiliki tepat 1 inode**
 - Root folder **tidak dapat dihapus**
 - Directory inode disebut **kosong jika dan hanya jika** memiliki 2 entry pada awal:
 - Entry ke-0: Direktori itu sendiri (self reference, ".")
 - Entry ke-1: Parent folder ("..")
 - Detail isi dari kedua entry adalah sebagai berikut:
 - Entry ke-0 memiliki inode ke direktori itu sendiri
 - Entry ke-1 memiliki inode ke parent inode yang sesuai
 - Selain itu, detail informasi lain dibebaskan
 - Khusus untuk root, parent folder adalah dirinya sendiri
 - **Kedua entry tidak dapat dihapus ataupun diganti** setelah pembuatan folder
 - File dan folder ditulis dengan alokasi block yang diperlukan, dan menyimpan referensi pointer ke block-block tersebut pada inode. Block pointer yang tidak dipakai bernilai 0
- **Inode Table**
 - Entry disebut **kosong jika dan hanya jika** inode bitmap yang menunjuk pada table tersebut bernilai 0
 - Entry bertipe folder akan memiliki attribute `i_mode EXT2_FT_DIR`

2.5. FS: Initializer

Pastikan setidaknya telah membaca bagian sebelumnya untuk melanjutkan implementasi.

Implementasi EXT2 dilarang mengimport kecuali: `disk.h`, `ext2.h`, `stdint.h`, `stdbool.h`, dan `string.h`. Selain belajar untuk membatasi scope dari sebuah kode, Chapter 3 - [External Program: Inserter](#) akan memperlihatkan dampak dari software engineering yang baik.

Storage yang dibuat masih baru dan kosong sehingga file system perlu menuliskan beberapa initial values yang dijelaskan pada bagian sebelumnya. Implementasikan interface berikut

- `EXT2SuperBlock`

Buatlah satu definisi static `EXT2SuperBlock` pada `ext2.c`. Variabel ini akan digunakan driver EXT2 untuk menyimpan state file system pada memory (RAM).

- `create_ext2()`

Prosedur `create_ext2()` akan melakukan inisialisasi *filesystem* dengan cara menulis *filesystem signature* di awal storage. Kemudian prosedur ini juga akan membuat sebuah superblock untuk root.

- `is_empty_storage()`

Prosedur `is_empty_storage()` melakukan komparasi antara boot sector dengan `fs_signature`. Bernilai `true` jika komparasi nilai `fs_signature` dan `boot sector` menghasilkan inequality.

- `initialize_filesystem_ext2()`

Prosedur `initialize_filesystem_ext2()` melakukan `create_ext2()` ketika `is_empty_storage()` bernilai `true`. Di luar kondisi tersebut baca isi dari memori di address 1 ke dalam superblock serta memori di address 2 ke dalam block group descriptor table.

Selain keempat hal diatas, akan ada beberapa fungsi dan prosedur opsional yang disediakan deklarasinya pada `ext2.h`. Fungsi dan prosedur ini tidak wajib diimplementasikan. Template dasar untuk `ext2.c` dan definisi struktur data file system telah disediakan pada kit.

2.6. FS: CRUD

Sebelum implementasi file system CRUD interface, ubah KERNEL_STACK_SIZE dari 4 KiB ke **2 MiB** pada **kernel-entrypoint.s**. Operasi file system CRUD dan fitur-fitur selanjutnya akan menghabiskan ukuran stack size 4 KiB dengan cepat sehingga butuh stack yang lebih besar.

Ubah kode assembly pada **kernel-entrypoint.s** seperti berikut

```
kernel-entrypoint.s

...
KERNEL_STACK_SIZE    equ 2097152          ; size of stack in bytes
MAGIC_NUMBER         equ 0x1BADB002        ; define the magic number constant
FLAGS                equ 0x0              ; multiboot flags
CHECKSUM             equ -MAGIC_NUMBER      ; calculate the checksum
; (magic number + checksum + flags should equal 0)

section .bss
align 4 ; align at 4 bytes
...
```

File system akan menyediakan 4 CRUD interface dengan prosedur: `read()`, `read_directory()`, `write()`, `delete()`. Semua interface akan menggunakan satu parameter dengan tipe data `EXT2DriverRequest`.

```
struct EXT2DriverRequest {
    void     *buf;
    char     *name;
    uint8_t  name_len;
    uint32_t parent_inode;
    uint32_t buffer_size;

    bool is_directory;
} __attribute__((packed));
```

Setiap interface akan menggunakan informasi pada `EXT2DriverRequest` untuk memenuhi operasinya. Behavior detail setiap interface akan dijelaskan pada bagian selanjutnya.

2.6.1. Read

Interface `read()` akan membaca file yang terletak pada pointer dari inode yang terletak pada `EXT2DirectoryEntry`. Inode ini akan dibaca dan dicari file dengan nama `request.name`.

Entry yang ditemukan **harus bertipe file** dan mengembalikan error jika yang ditemukan adalah folder. Interface mengecek ukuran `request.buffer_size` cukup untuk `EXT2DirectoryEntry.buf` dan mengembalikan error jika tidak cukup.

Entry yang sudah dicek dan tidak mengembalikan error akan membaca dan memasukkan semua partisi file ke dalam `request.buf`.

read()	
Error Code	Pengertian
0	Operasi berhasil
1	Bukan sebuah file
2	Buffer tidak cukup
3	File tidak ditemukan
4	Parent folder tidak dapat dibaca / invalid
-1	Error selain kategori diatas

Interface `read_directory()` memiliki cara kerja yang sama, tetapi hanya menerima target bertipe folder. Atribut filetype dari `EXT2DirectoryEntry` harus bertipe folder pada `read_directory()`. Validasi dan error check akan tidak berbeda jauh `read()`. Setelah validasi, `read_directory()` membaca inode dan memasukkannya pada `request.buf`.

read_directory()	
Error Code	Pengertian
0	Operasi berhasil
1	Bukan sebuah folder
2	Folder tidak ditemukan
3	Parent folder tidak dapat dibaca / invalid
-1	Error selain kategori diatas

2.6.2. Write

Interface ini akan menuliskan sebuah file atau folder berdasarkan request yang diterima. Interface `write()` mengecek apakah terdapat datablock pada `request.parent_inode`. Cek apakah nama `request.name` sudah digunakan oleh file atau folder di directory yang sama. Kembalikan error jika entry dengan nama sama sudah ada pada parent folder.

Penulisan file akan memiliki behavior berikut

- Partisi dari file akan alokasikan pada free space dengan [First Fit Algorithm](#)
- Jumlah block yang ditulis **tepat** $\text{ceil}(\text{request.buffer_size} / \text{BLOCK_SIZE})$
- Mekanisme penulisan partisi file dengan cari lokasi(-lokasi) kosong pada block bitmap, tulis di bitmap, allocate node baru, lalu tulis di datablock

Setelah menuliskan file atau folder pada datablock, `write()` akan membuat sebuah `DirectoryEntry` baru pada `EXT2Inode` milik parent. Atribut dari entry baru akan disesuaikan dengan `request`.

Ketika semua manipulasi variabel pada memory selesai, lakukan **commit** dengan menuliskan semua metadata file system ke storage.

write()	
Error Code	Pengertian
0	Operasi berhasil
1	Entry dengan nama sudah ada
2	Parent folder tidak dapat dibaca / invalid
-1	Error selain kategori diatas

2.6.3. Delete

Interface `delete()` akan menghapus file dan folder sesuai dengan `request.name`. Validasi awal request tidak akan berbeda jauh dengan interface lain. File dan folder akan dihapus dengan mengosongkan block pada `EXT2Inode` dan menghapus entry pada parent `EXT2Inode`.

Penghapusan folder diperlukan validasi tambahan untuk mengecek apakah folder kosong. Tolak request jika folder tidak kosong.

delete()	
Error Code	Pengertian
0	Operasi berhasil
1	Entry tidak ditemukan
2	Folder yang akan dihapus tidak kosong
3	Parent folder tidak dapat dibaca / invalid
-1	Error selain kategori diatas

2.6.4. CRUD Implementation & Testing

Implementasikan semua interface sesuai dengan behavior yang dijelaskan pada bagian sebelumnya. Gunakan **hexeditor** untuk mengecek storage.

Implementasi CRUD interface wajib memperhatikan **Data Integrity**. Data yang ditulis dan dibaca oleh interface wajib menjaga data tidak berubah sesuai data sebelum operasi CRUD. [Chapter 3](#) akan mengasumsikan interface ini memenuhi data integrity. Pastikan implementasi telah teruji pada beberapa edge case berikut

- Urutan partisi block
- Bytes di dekat boundary block (byte ke-0 dan byte ke-2047)
- Isi dari bytes setiap block (pastikan tidak ada perbedaan dengan penulisan)

Telah disediakan sebuah image media penyimpanan yang diisi dengan file system **EXT2 - IF2130 Edition**. Image sample ini hasil dari [External Program: Inserter](#) yang menggunakan reference implementation EXT2 - IF2130 Edition.

Image sample ini dapat digunakan untuk menguji apakah implementasi sudah benar atau belum. Jika ingin menggunakan sample untuk testing, pastikan `fs_signature` sesuai dengan yang diimplementasikan pada file system. Namun, perlu diperhatikan bahwa ini contoh salah satu implementasi filesystem, dan implementasi tersebut dapat berbeda-beda. Jangan terlalu terpaku pada sample ini ketika melakukan testing filesystem yang telah Anda buat.

Image tersedia pada kit **ch2/4 - Filesystem/** bernama **sample-image.bin**

Struktur folder, file, dan alokasi block **sama persis** dengan contoh pada [2.4.4. Root Directory](#).

Directory Tree	
root/	
- folder1/	
- nestedf1/	
- tpazolite	
- xi	
- folder2/	
- laur	
- sasakure	

Hexeditor pada **sample-image.bin** pada lokasi inode root

00002A00	01 00 00 00 0C 00 01 02 2E 00 00 00 00 01 00 00 00	.	.	.
00002A10	0C 00 02 02 2E 2E 00 00 02 00 00 00 10 00 05 01	.	.	.
00002A20	73 68 65 6C 6C 00 00 00 03 00 00 00 10 00 07 02	s h e l l	.	.
00002A30	66 6F 6C 64 65 72 31 00 07 00 00 00 10 00 07 02	f o l d e r 1	.	.
00002A40	66 6F 6C 64 65 72 32 00 08 00 00 00 10 00 04 01	f o l d e r 2	.	.
00002A50	6C 61 75 72 00 00 00 00 09 00 00 00 14 00 08 01	l a u r	.	.
00002A60	73 61 73 61 6B 75 72 65 00 00 00 00 00 00 00 00	s a s a k u r e	.	.
00002A70	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.	.	.
00002A80	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.	.	.
00002A90	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.	.	.

Hexeditor pada **sample-image.bin** pada lokasi inode folder1

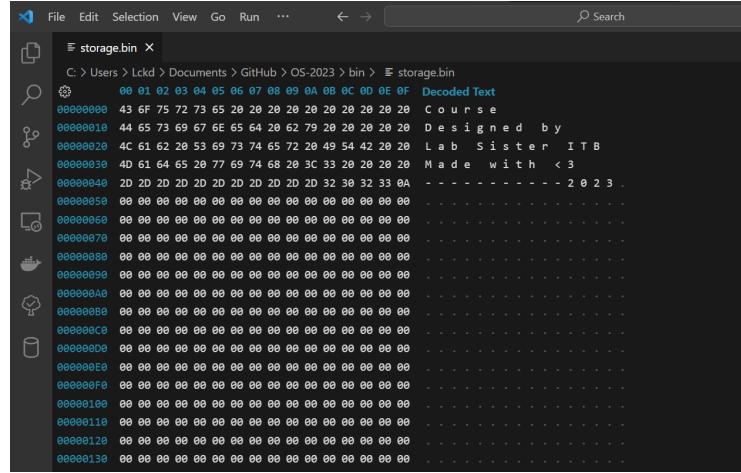
Ganti parameter disk yang dijelaskan pada [2.2. Disk Image](#) untuk menggunakan image ini pada sistem operasi

```
qemu-system-i386 -s -S -drive  
file=sample-image.bin,format=raw,if=ide,index=0,media=disk -cdrom OS2025.iso
```

Catatan: Reference implementation menggunakan parent inode “.” dan “..” menunjuk ke parent. Untuk nama parent directory “..”, reference implementation hanya menuliskan “..”. Beberapa detail file seperti ukuran akurat dari filesize, posisi directory entry.

Tips: File System

- **Hex editor secara efektif wajib digunakan** untuk proses implementasi file system. Hex editor akan digunakan untuk mengecek apakah penulisan ke storage berhasil atau tidak.



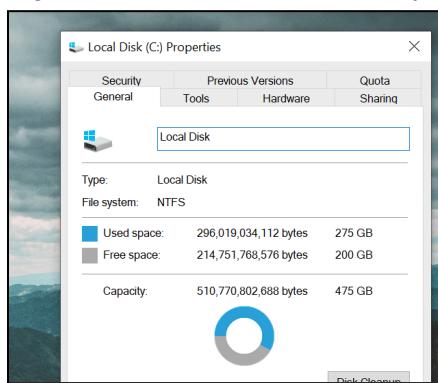
The screenshot shows a hex editor window in Visual Studio Code. The file being edited is named "storage.bin". The status bar at the bottom of the editor window displays the message "Successfully writing into boot sector". The main pane shows the raw binary data in hex format, with some ASCII text decoded for readability. The decoded text includes "Course", "Designed by", "Lab Sister ITB", and "Made with <3>". The file path is visible in the top left corner of the editor window.

Visual Studio Code - Hexeditor - Successfully writing into boot sector

- Gunakan **Ctrl+G** untuk melakukan jump ke offset pada hexeditor
- Gunakan **do-while** construct untuk menuliskan ke storage

• Standardization Mess: SI & Binary Prefix

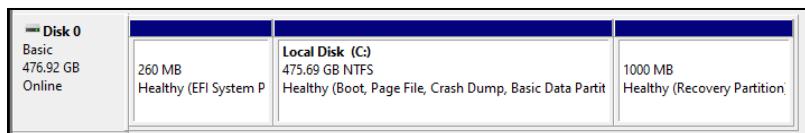
Pernahkah bertanya media penyimpanan pada komputer, smartphone, dan sebagainya selalu terlihat memiliki penyimpanan yang lebih kecil daripada ukuran yang dipromosikan?



Laptop dengan “512GB” NVMe SSD pada Windows 10

Banyak yang memberikan alasan hal ini disebabkan oleh “digunakan oleh sistem”. Meskipun hal ini benar, sebenarnya terdapat alasan lain yang lebih signifikan. Bagian yang digunakan oleh “sistem” seperti file sistem operasi dan metadata file system (misalnya **File Allocation Table** pada FAT32) umumnya akan terlihat sebagai used space. Beberapa penggunaan lain oleh sistem mungkin membutuhkan partisi sendiri seperti [EFI Partition](#) yang dibutuhkan untuk booting.

Alasan sebenarnya yang paling berkontribusi adalah **standar prefix byte**. Dari awal penetapan standar hingga saat ini, penggunaan prefix masih belum konsisten dari sistem operasi ke sistem operasi dan industri. Beberapa sistem operasi memilih menggunakan prefix ala SI (Kilo K, Mega M, Giga G) sebagai **Binary Prefix**. Jika prefix SI adalah kelipatan 10^3 setiap peningkatan prefix, binary prefix menggunakan kelipatan 2^{10} atau 1024.



Sistem operasi Windows hingga panduan ini ditulis (2025) masih menggunakan prefix SI (GB) untuk menunjukkan gibibytes. Gambar diatas diambil pada media penyimpanan dan sistem yang sama. Dengan menggunakan kalkulator, dapat diverifikasi bahwa sebenarnya SSD berukuran **512 GB** (dengan prefix SI) sesuai dengan yang dipromosikan.

```
>>> 476.92 * (2**30)
512088950702.08
>>>
```

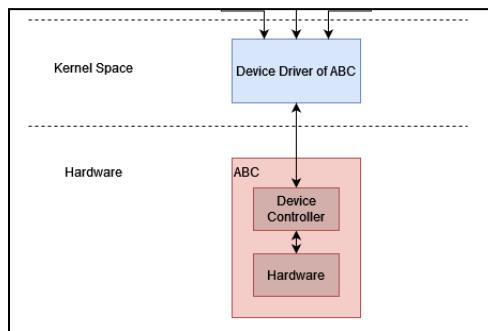
Namun sayangnya tidak konsistennya penggunaan prefix antar OS dan industri sering membuat bingung pembaca yang tidak mengetahui hal ini.

Beberapa detail dapat dicek pada [Binary Prefix - Wikipedia](#).

• Controller, Driver, Scam

Setelah mengimplementasikan [Keyboard Driver](#) dan [File System: EXT2 - IF2130 Edition](#), pembaca seharusnya akan tergambar seberapa dalam hardware dapat dikontrol sistem operasi dan device driver.

[Device Driver](#) sebenarnya menggunakan sebuah interface yang disediakan oleh **Device Controller**. Device controller adalah salah satu komponen hardware pada device yang mengontrol langsung behavior hardware device sesuai dengan program [Firmware](#) yang dipasang.



Bagaimana perilaku keyboard menyimpan keypress pada internal buffer dan bagaimana storage device menyimpan data ke physical disk atau NAND flash akan sepenuhnya diatur oleh device controller. **Kernel dan device driver hanya dapat mempercayai informasi dari controller.**

Meskipun abstraksi seperti ini sangat umum pada *computer engineering*, terdapat **scam** sangat umum di-Indonesia yang melakukan abuse kepercayaan kernel dan device driver.

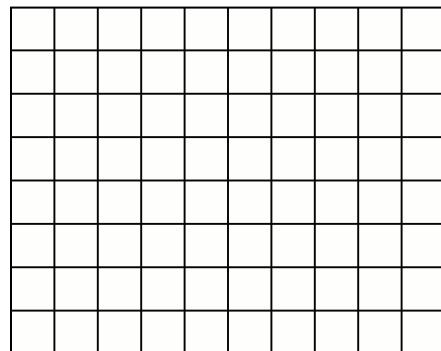


2TB for 145k, too good to be true, no?

Sayangnya scam ini sangat marak di Indonesia dan banyak yang mengasumsikan “data aman” asal tidak melewati ukuran tertentu. Meskipun storage controller akan melaporkan ukuran penyimpanan “2TB”, tidak ada jaminan firmware akan menuliskan dan menyimpan data secara predictable. Secara efektif pembeli mendapatkan sebuah **Electronic Waste** karena device controller tidak dapat diprogram ulang dengan mudah oleh pengguna awam dan diluar jangkauan skill sebagian besar *software engineer* (Contoh: [Reverse Engineering Samsung SSD 840](#)).

Buyer tips: Jangan, pernah, membeli, storage dengan controller yang dimodifikasi. Titik.

- **Fragmentation**



Fragmentation-Defragmentation process, [Defragmentation - Wikipedia](#)

Fragmentation adalah fenomena yang terjadi secara natural pada media penyimpanan volatile dan non-volatile pada komputer. Fragmentation terjadi ketika partisi dari file ditulis secara tidak sekuensial atau berurutan pada physical block. Hal ini secara natural terjadi karena operasi CRUD file system tidak menjamin data akan ditulis secara sekuensial.

Lama kelamaan, fragmentasi dari keseluruhan file system akan terakumulasi dan berdampak kepada **kecepatan sequential read yang buruk**. Dampak dari fragmentasi ini akan sangat terasa pada media penyimpanan berbasis piringan seperti Hard Disk Drive (HDD), DVD, dll. Namun hal ini tidak terlalu berpengaruh* pada media penyimpanan berbasis [NAND Flash](#) seperti SSD.

Defragmentation adalah proses menyusun partisi file pada media penyimpanan agar menjadi terurut rapi dan kontigu. Proses ini sangat penting untuk menjaga kinerja throughput dari disk-based storage sehingga sistem operasi modern seperti Windows akan menjadwalkan secara otomatis pada background untuk melakukan defragmentasi.

Namun terdapat sifat NAND flash-based storage yang menjadi argumen utama untuk tidak melakukan defragmentasi. Setiap chip NAND flash akan memiliki batasan siklus penulisan-penghapusan sebelum chip tersebut tidak dapat dihapus lagi dan unreliable. Hal ini menyebabkan defragmentasi yang menggunakan operasi read-write-delete dalam jumlah besar akan cenderung menghabiskan umur penulisan-penghapusan dari chip NAND flash.

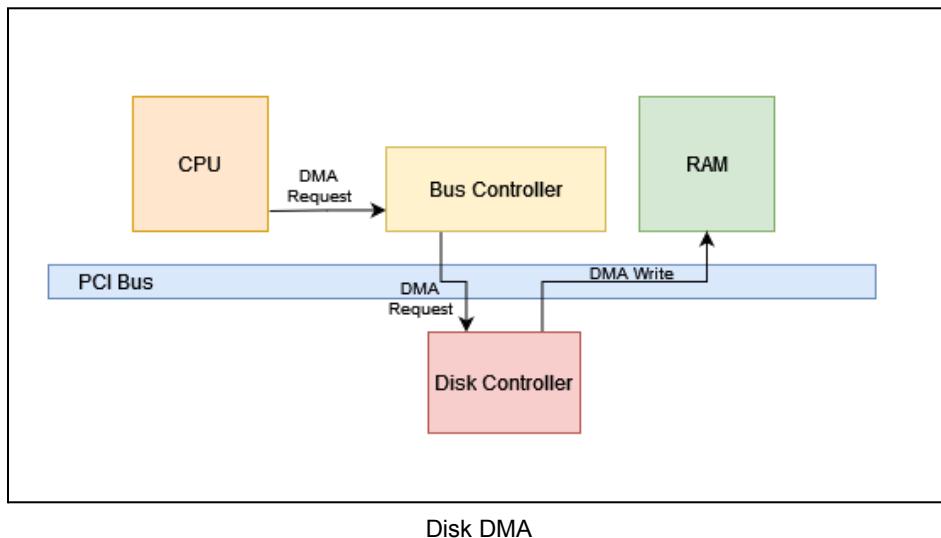
NAND flash-based storage modern bagus umumnya menggunakan [Device Controller](#) yang melakukan [Wear Leveling](#) secara transparan. Operasi penulisan pada [LBA](#) tertentu mungkin dipetakan ke physical block address yang berbeda agar setiap chip memiliki siklus penulisan-penghapusan yang sama.

*Tentunya sistem kompleks akan memiliki banyak asterisk, anecdote: [Flash Drive Fragmentation Performance](#)

2.7. - Extras: Hardware

- **Direct Memory Access**

Direct Memory Access (DMA) adalah fitur komputer yang memperbolehkan komponen komputer mengakses secara langsung RAM. DMA akan mengurangi overhead I/O pada CPU karena komponen hardware lain tidak perlu mengirim hardware interrupt setiap kali perlu mengambil dan menulis data ke RAM, hanya request dan notifikasi selesai saja.



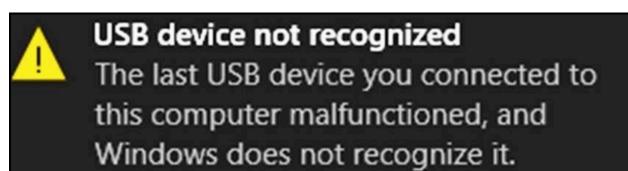
Sebenarnya ATA PIO yang diimplementasikan pada [**Disk Driver**](#) dapat menggunakan direct memory access untuk mempercepat I/O dan tidak menghabiskan CPU cycle untuk melakukan operasi blocking. Namun karena kompleksitas yang bertambah disebabkan adanya komponen **Asynchronous I/O**, panduan ini tetap menggunakan ATA PIO sederhana yang blocking.

Meskipun fitur direct memory access terlihat sangat bagus dalam segi kinerja throughput data, terdapat masalah CPU [**Cache Coherence**](#) dan segi keamanan untuk mencegah komponen eksternal dapat melakukan [**DMA Attack**](#). Kompleksitas implementasi DMA akan menjadi lebih kompleks dengan konsiderasi tambahan ini.

● Device Recognition & Hot Plug

Pengguna komputer umumnya *taken for granted* peripheral yang dapat dicolokkan ke port komputer akan bekerja jika memiliki bentuk yang sama. USB, Ethernet, HDMI, DisplayPort, dan lain-lain akan bekerja jika ditancapkan ke komputer modern.

Padahal sebenarnya sistem operasi modern memiliki berbagai macam built-in driver yang mengimplementasikan banyak standar dan protokol sehingga pengguna dapat langsung melakukan **Plug and Play** dan mengenal device tersebut secara otomatis. Jika komunikasi sistem operasi dengan device tidak berhasil karena protokol berbeda atau tidak ada driver, sistem operasi akan menandai device ini sebagai tidak dikenal.



Windows 10 - USB Device not recognized

Solusi untuk hal ini adalah memastikan koneksi physical device terhubung dengan baik (kabel tidak putus, konektor bersih, dll) dan device driver telah terpasang pada sistem operasi.

Selain kemudahan plug and play pada sistem operasi modern, banyak port eksternal pada komputer modern juga didesain dapat mendukung **Hot plug** dari sisi hardware. Sistem operasi modern akan melengkapi fitur hot plug ini dari sisi software. Sebenarnya fitur plug and play dan hot plug berhubungan karena sistem operasi perlu mengimplementasikan device manager untuk mencatat device yang terhubung dan alokasi resource pada komputer.

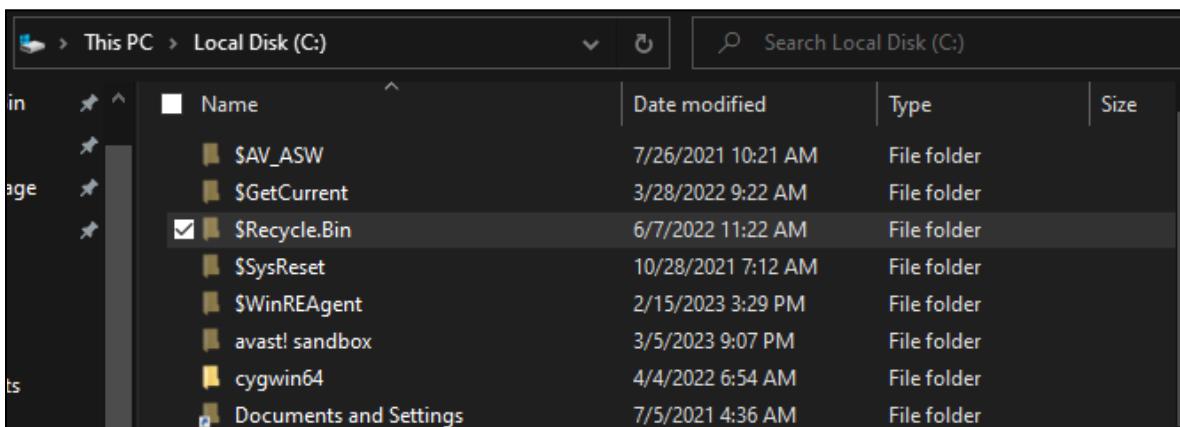
Selain kompleksitas manajemen device, setiap device mungkin membutuhkan cara *clean-up* nya sendiri-sendiri. Sistem operasi modern mengimplementasikan [Page Cache](#) yang memetakan media penyimpanan ke physical memory yang tidak dipakai oleh user. Cache ini berguna untuk meningkatkan kinerja I/O ke media penyimpanan, tetapi sistem operasi wajib untuk menuliskan data yang telah dimodifikasi ke media penyimpanan ketika selesai.

Shutdown pada sistem operasi modern dan operasi seperti “**Eject Removable Disk**” akan memastikan hal-hal seperti page cache telah dituliskan ke media penyimpanan dan tidak ada user program yang sedang aktif mengakses media penyimpanan ini.

• Digital Forensic

Digital Forensic adalah pemulihan, ekstraksi, analisis data dari peralatan digital seperti komputer dan smartphone. Biasanya sumber informasi didapatkan dari non-volatile storage atau pada kasus tertentu volatile storage. Metode ekstraksi informasi dari sumber informasi ini membutuhkan pengetahuan mendalam bagaimana [File System](#) dan memory management [Paging](#) bekerja.

File system umumnya tidak melakukan **overwrite data pada storage ketika file deletion**. Hanya metadata file yang disimpan oleh struktur data file system yang dihapus. Salah satu alasan mengapa penghapusan file tidak menghapus sepenuhnya data pada disk adalah operasi overwrite bersifat linear dengan ukuran file dan akan membutuhkan waktu yang sangat lama.



Operasi file seperti move and delete hanya akan memanipulasi metadata saja untuk menghemat operasi yang mahal secara waktu. Penghapusan sebuah file pada file system Windows modern hanya memindahkan ke **Recycle Bin** dan permanent delete menghapus metadata saja. Dengan hati-hati, data yang tidak memiliki metadata ini dapat dilakukan recovery untuk keperluan forensik.

Untuk menghindari hal tersebut, terdapat metode pada sisi software bernama [Data Destruction](#). Beberapa implementasi data destruction menggunakan operasi overwriting berulang kali hingga semua partisi file yang ingin dihapus telah hilang. Selain sisi software ini, salah satu metode yang sangat reliable tetapi mahal adalah **menghancurkan langsung physical device**.

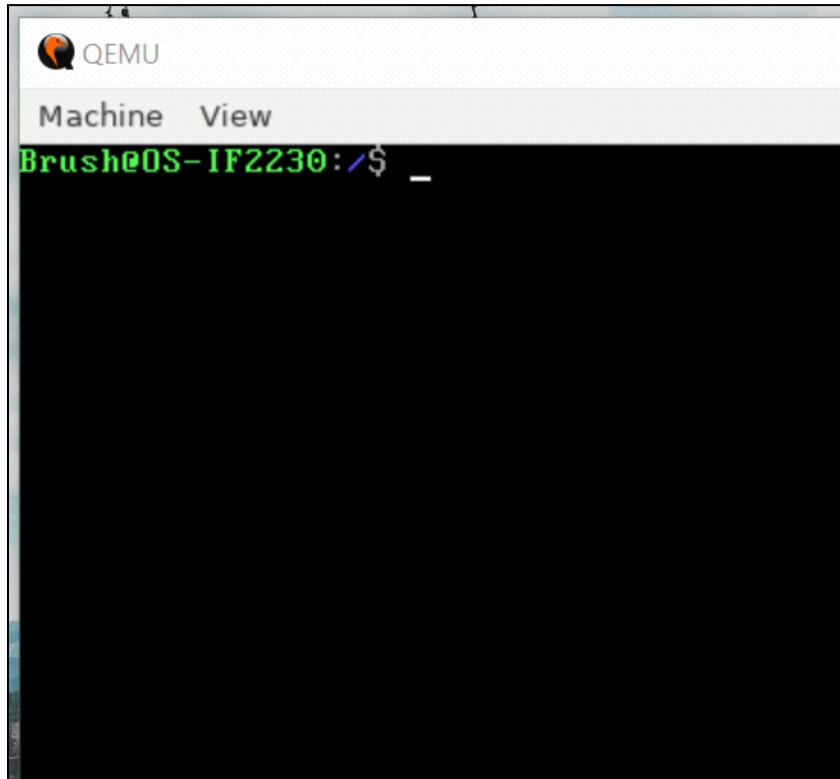
Melakukan recovery data dari volatile memory juga hal yang mungkin. Kompetisi [Capture the Flag](#) yang berformat Jeopardy umumnya memiliki kategori khusus untuk forensik memory volatile yang menggunakan berbagai macam trik dan technical skill terkait bagaimana memory digunakan pada sistem operasi modern. Hal-hal seperti analisis crash dump dari memory membutuhkan pengetahuan yang mendalam bagaimana sistem operasi bekerja dan cara penyimpanan data.

Ch. 3 - Paging, User Mode, Shell

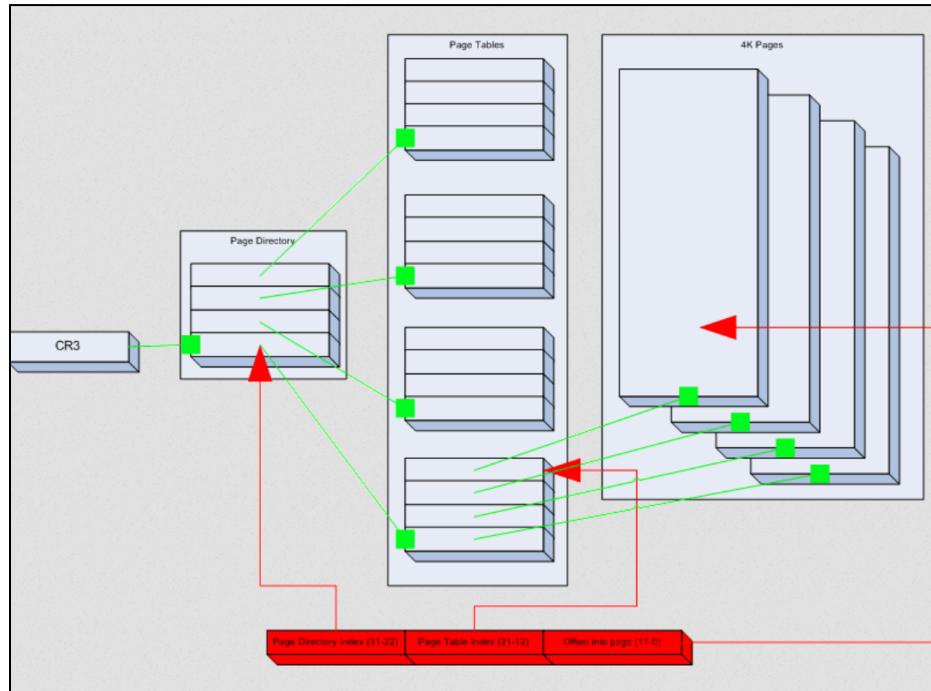
Chapter 3 memiliki target utama membuat sebuah **User Program** yang menyediakan [Command Line Interface](#). Sebelum membuat user program, kernel wajib mengimplementasikan [Paging](#) dan [User Mode](#) terlebih dahulu.

Setelah kedua bagian selesai, bagian **Shell** akan mengimplementasikan [System Calls](#) dan [Command Line Interface](#) untuk menyediakan interface kepada pengguna.

Sistem operasi pada akhir Chapter 3 akan memiliki shell yang dapat digunakan pengguna untuk melakukan command-command dan mengoperasikan komputer.



3.1. Paging



Source : [Paging - OSDev](#)

Paging adalah sistem manajemen memory yang **mengabstraksikan Physical Memory**. Dengan menambahkan layer abstraksi **Virtual Memory**, user program dapat menggunakan virtual address yang sama dengan user program lain. Selain itu kernel juga dapat menggunakan storage non-volatile untuk mensimulasikan ukuran RAM yang lebih besar dengan **Memory Paging**.

Selain dari virtual memory, paging menyediakan fitur untuk memberikan flag-flag proteksi pada memory seperti menandai sebagian address pada memory dimiliki kernel atau user.

Sebelum mengimplementasikan, penjelasan **Paging: Overview** akan mereview beberapa istilah dan desain yang akan diimplementasikan pada kernel. Implementasi awal dimulai dari **Data Structure: Page Table** dan diakhiri dengan membuat **Higher Half Kernel**. Bagian **Activate Paging** akan menguji implementasi dengan menyalakan fitur paging pada CPU. Paging diakhiri dengan membuat sebuah **Memory Manager** yang akan digunakan pada pembuatan **Shell** dan **Chapter 4** nantinya.

3.1.0. Paging: Overview

Paging mengabstraksikan physical memory dengan cara memetakan [Linear Address ke Physical Address](#). Sama seperti GDT dan IDT, struktur data paging akan didasarkan dengan tabel yang memiliki entry. Akan terdapat beberapa istilah tambahan yang digunakan pada bagian paging

Istilah	Arti pada Paging
Page Table	Struktur data berbentuk tabel dengan entry yang membawa informasi mapping antara linear address ke physical address.
Page Directory	Salah satu skema implementasi page table yang digunakan pada buku ini
Page Frame / Page Size	Region physical memory yang dapat dipetakan oleh entry page table.
Virtual, Linear, Physical Memory	Merefere kepada tipe memory yang didasarkan dengan skema addresingnya. Digambarkan pada bagian tambahan x86 Memory Addressing . Dengan tidak digunakannya memory segmentation GDT pada panduan ini, Paging akan disebut memetakan Virtual Address ke Physical Address .

Paging menggunakan struktur data **Page Table** untuk menyimpan informasi pemetaan memory. Pemetaan 1:1 antara virtual dan physical address sangat tidak efisien sehingga setiap entry page table mewakili region physical memory dengan ukuran tertentu yang disebut **Page Frame**.

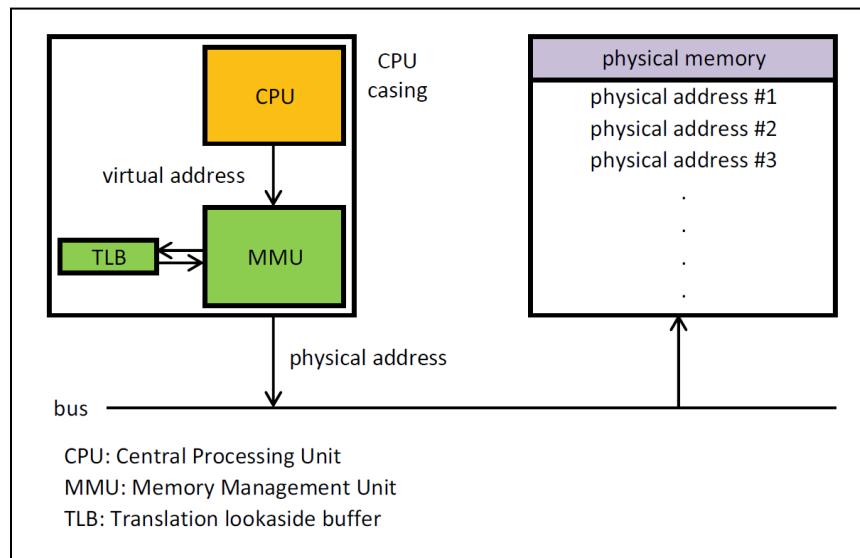
Paging merupakan fitur yang sangat krusial pada komputer modern sehingga x86 menggunakan komponen hardware tambahan khusus pada CPU untuk meningkatkan kinerja memory & paging: [MMU & TLB](#). Berbeda dengan GDT dan IDT yang hanya diakses CPU dan diubah oleh kernel, struktur data **page table juga akan diakses MMU dan TLB**. Hal ini penting diketahui karena manipulasi page table oleh kernel (yang dieksekusi CPU) mungkin tidak terbaca oleh **TLB**.

Fitur paging yang diimplementasikan akan berfokus untuk memenuhi kebutuhan [User Mode](#) dan [Process](#). Beberapa konfigurasi paging dipilih untuk memudahkan implementasi.

Panduan ini akan menggunakan skema paging dan memory berikut

- **Memory Segmentation**
 - Fitur memory segmentation GDT **tidak digunakan**
 - GDT hanya akan digunakan untuk **DPL** (Privilege) dan [TSS](#)
 - Oleh karena itu secara efektif **Virtual Address = Linear Address**
- **Kernel**
 - Kernel akan memiliki **1 Page Directory** khusus untuk dirinya sendiri
 - Layout memory kernel adalah [Higher Half Kernel](#)
- Page frame size berukuran **4 MiB**
- [Page Table Structure](#) adalah **Flat / 1-Level Page Table**

- **MMU & TLB**



Source : [MMU - Wikipedia](#)

Memory Management Unit (MMU) merupakan komponen hardware yang terintegrasi pada CPU yang bertugas untuk melakukan translasi virtual address. Ketika CPU membutuhkan akses ke memory RAM, MMU bertugas untuk mentranslasikan virtual address dengan membaca page table yang sedang aktif.

Translation Lookaside Buffer (TLB) adalah bagian dari MMU dan menjadi **cache translasi virtual-physical address**. Virtual address yang sering digunakan & manipulasi akan disimpan pada TLB untuk mengurangi lookup manual pada page table. Perlu dicatat bahwa TLB berbeda dengan [CPU Cache](#) yang menyimpan **value dari physical address**.

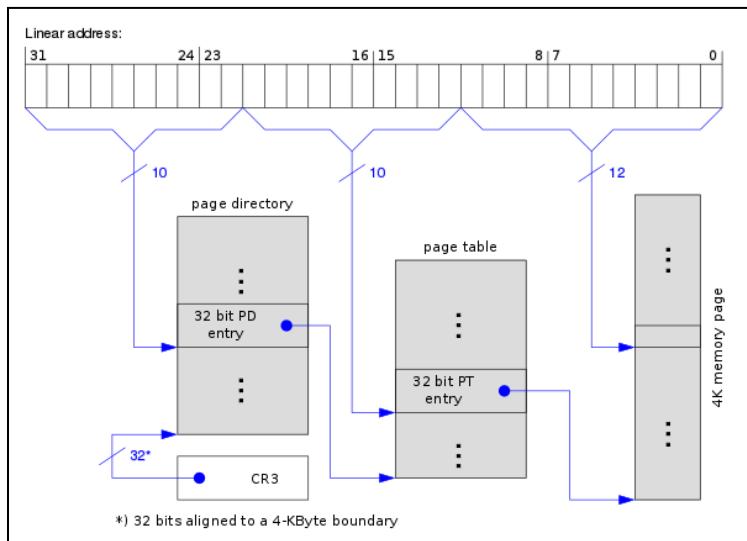
Pada arsitektur x86, CPU memiliki instruksi `invlpg` yang digunakan untuk flush cache TLB ketika terdapat perubahan pada page table. Page table yang dibaca MMU akan menggunakan address yang terdapat CPU control register **CR3** yang dapat diubah kernel.

[Process](#) dan [Context Switching](#) akan memanipulasi page table untuk memberikan fitur [Process Isolation](#) sehingga semua process tidak dapat membaca memory process lain. Setiap process akan memiliki struktur data [Process Control Block](#) sendiri yang menyimpan informasi page table process.

• Page Table Structure

Paging memetakan virtual address ke physical address. Tentunya paging bukanlah sebuah sulap yang tiba-tiba mengubah virtual ke physical address. Paging menggunakan sebuah struktur data berbentuk tabel yang digunakan untuk mencatat semua pemetaan virtual-physical.

Implementasi page table memiliki bermacam-macam skema yang memiliki kekurangan dan kelebihan masing-masing. Pada arsitektur x86, implementasi page table menggunakan skema [Hierarchical Page Table](#).



Hierarchical Page Table: x86 2-Level PT, [Page Table - Wikipedia](#)

Ilustrasi diatas menggunakan x86 2-level page table. Setiap page table akan memiliki dua struktur data: **Page Directory** sebagai level pertama dan **Page Table** sebagai level kedua. Page frame pada skema ini berukuran **4 KiB**. Pada skema ini, setiap virtual address akan ditranslasikan menjadi index entry Page Directory, Page Table, dan offset.

Panduan ini akan menggunakan 1-level page table yang hanya memiliki struktur data **Page Directory**. Karena hanya memiliki 10-bit (Cek [Page Table: Address Translation](#)), page frame berukuran **4 MiB**. Dengan menggunakan 1-level page table, implementasi paging akan lebih sederhana.

Besarnya page frame yang digunakan akan mempengaruhi granularitas manajemen memory kernel. Dengan page size 4 MiB, kernel hanya dapat melakukan manajemen memory dalam kelipatan bilangan bulat 4 MiB. Ukuran ini tergolong sangat besar dan akan cepat menghabiskan memory dari sistem jika setiap proses tidak menggunakan semua memory 4 MiB. Sama seperti slack space yang dipengaruhi dengan ukuran inode size pada [EXT2: Block & Inode](#).

- **Page Table: Address Translation**

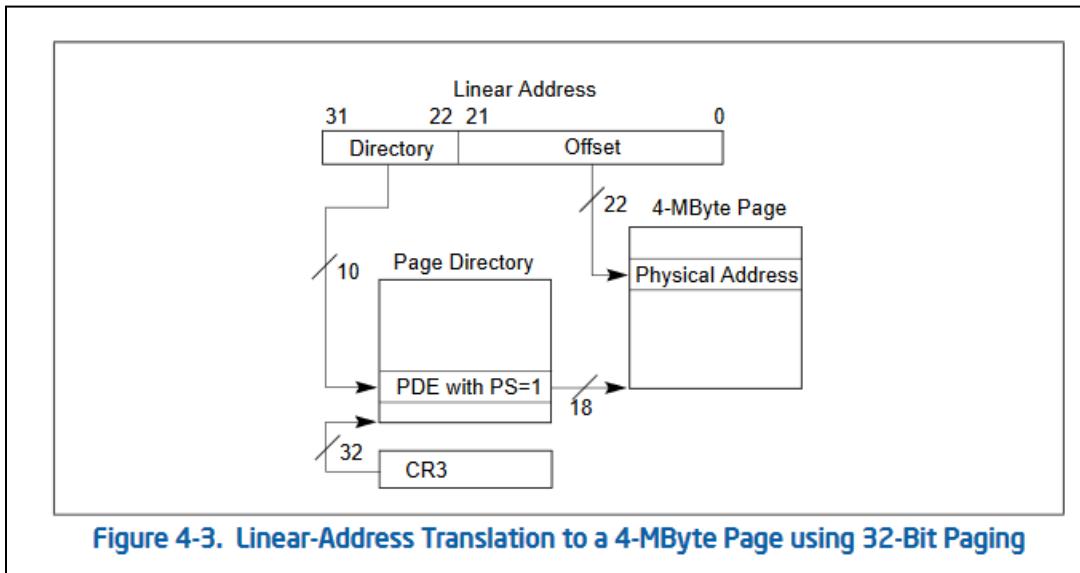


Figure 4-3. Linear-Address Translation to a 4-MByte Page using 32-Bit Paging

Intel Manual Vol 3a - Chapter 4 - Paging: 4 MB paging, (Note: Interval bit inklusif)

Ilustrasi diatas adalah skema 1-level page table. Sekilas pemilihan bit ke-22 hingga 31 untuk index page directory mungkin terasa aneh. Namun jika diteliti lebih lanjut, semua skema paging x86 menggunakan **bit yang bernilai sama dengan ke ukuran page size**.

Perlu diingat kembali bahwa 2^{10} bernilai 1024 atau dalam prefix binary disebut 1 kibi. Selain itu operasi bit shift ekuivalen dengan perkalian dan pembagian angka 2. Dalam konteks ini, perhitungan digit akan dimulai dari 0 (zero-indexed). Semisal terdapat angka binary 00010000, angka ini ekuivalen dengan operasi $1 \ll 4$ atau $2^4 = 16$.

Bit ke-22 sama dengan $1 \ll 22$ atau $2^{22} = 2^2 \cdot 2^{10} \cdot 2^{10} = 2^2$ mebi atau 4 mebi. Setiap **kelipatan bit ke-22 adalah kelipatan 4 mebi**. Ukuran ini sama seperti ukuran page frame. Bit ke-21 hingga ke-0 digunakan sebagai offset.

Page directory akan mentranslasikan bit ke-31 hingga ke-22. Bit tersebut digunakan sebagai index page directory yang setiap framenya berukuran 4 MiB / 0x400000 bytes. Offset pada bit-21 hingga ke-0 memiliki panjang bit 22 sehingga dapat melakukan addressing dari nilai 0x0 **hingga 0x3FFFFF** yang cukup untuk melakukan addressing 1 page frame.

3.1.1. Data Structure: Page Table

Skema page table yang akan diimplementasikan adalah 1-level page table yang memiliki page size 4 MiB. Struktur data yang perlu diimplementasikan untuk skema ini adalah **Page Directory** saja.

Sama seperti struktur data GDT dan IDT, struktur data page table **sensitif dengan alignment** dan **harus memiliki definisi yang sama persis dengan Intel Manual x86**. Kesalahan pada struktur data paging sering kali menyebabkan **Triple Fault**.

Kerangka dasar struktur data page table dan definisi page directory untuk kernel akan disediakan pada **ch2/1 - Paging/**

Cek detail 1-level page table pada **Intel Manual Vol 3a - Chapter 4 - Paging: 32-Bit Paging**. Struktur data **PageDirectoryEntry** akan disebut dengan **PDE: 4MB Page** pada manual.

Lengkapi definisi ketiga struktur data page table berikut

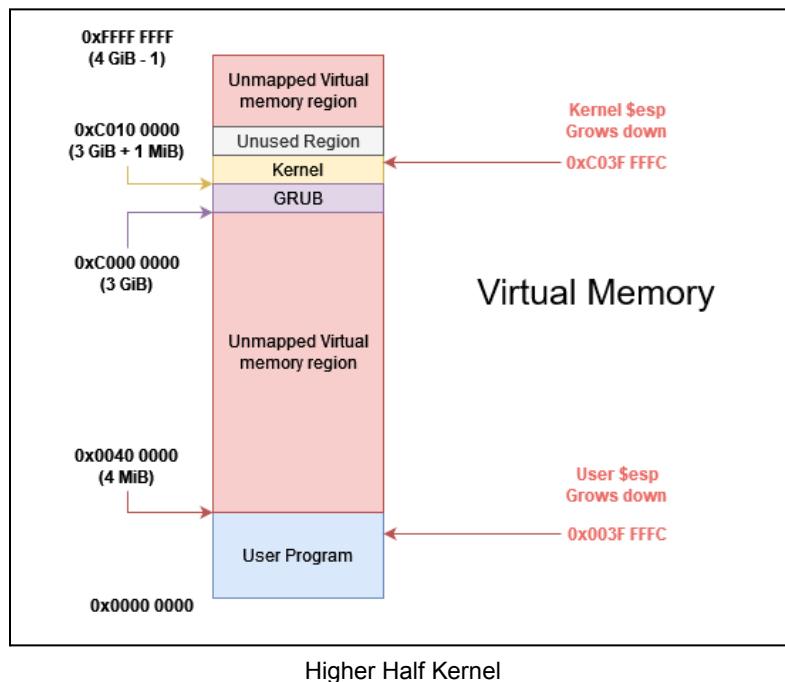
- **PageDirectoryEntryFlag**
- **PageDirectoryEntry**
- **PageDirectory**

3.1.2. Higher Half Kernel

Nantinya sistem operasi akan memiliki program user [Shell](#) yang juga akan hidup pada memory yang sama. Shell pada [Chapter 2](#) akan meminjam page directory milik kernel untuk memetakan memory milik shell. Kernel yang diimplementasikan dari [Chapter 0](#) hingga [Chapter 1](#) dimasukkan kedalam memory `0x100000` sesuai yang dituliskan pada bagian [Chapter 0: C Kernel & Linker](#).

Dengan menggunakan pemetaan virtual-physical address yang dimiliki paging, address `0x0` hingga `0x400000` awal dapat dipetakan ke tempat lain sehingga user program dapat menggunakan address tersebut. Letak pemetaan virtual address kernel ini akan diserahkan kepada pembuat kernel.

Panduan ini akan menggunakan desain bernama [Higher Half Kernel](#). GRUB dan kernel yang menggunakan physical address `0x0` hingga `0x400000` dipetakan ke virtual address yang dimulai dari `0xC0000000` hingga `0xC0400000`.



Dengan memetakan kernel ke address tertinggi, user program dapat menggunakan virtual address `0x0` hingga `0xC0000000`.

Untuk mengimplementasikan higher half kernel, diperlukan beberapa langkah tambahan. Sebelum pemetaan higher half kernel dilakukan, page directory wajib memiliki satu entry **Identity Paging** (Pemetaan virtual-physical address yang sama) untuk page frame 0. Hal ini dibutuhkan karena kernel masih di load oleh GRUB pada address `0x100000`.

Identity paging page frame 0 hanya akan digunakan tahap booting dan akan dihapus oleh kode assembly yang diberikan pada bagian [Activate Paging](#). Ubah linker script yang telah dibuat untuk mengganti [Relocation Address](#) kernel ke higher half. Bagian ini tidak dapat dites sendirian, lanjutkan ke bagian selanjutnya untuk menguji implementasi.

linker.ld

```
ENTRY(loader)           /* the name of the entry label */
/* Relocation address 0xC0010000, but load address (physical location) 0x100000 */
SECTIONS {
    . = 0xC0100000;          /* use relocation address at 0xC0100000 */
    /* Linker variable that can be used in kernel */
    _linker_kernel_virtual_addr_start = .;
    _linker_kernel_physical_addr_start = . - 0xC0000000;
    .multiboot ALIGN (0x1000) : AT (ADDR (.multiboot) - 0xC0000000) {
        *(.multiboot)      /* put GRUB multiboot header at front */
    }

    .setup.text ALIGN (0x1000) : AT (ADDR (.setup.text) - 0xC0000000) {
        *(.setup.text)     /* initial setup code */
    }

    .text ALIGN (0x1000) : AT (ADDR (.text) - 0xC0000000) {
        *(.text)           /* all text sections from all files */
    }

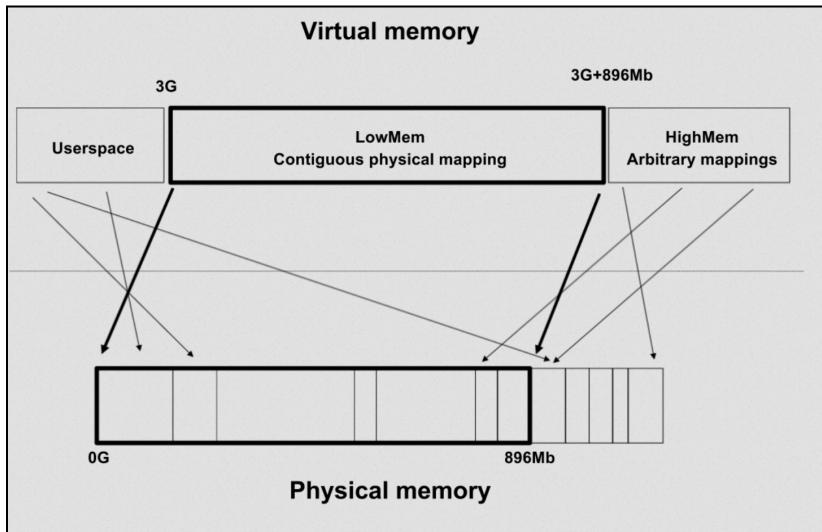
    .rodata ALIGN (0x1000) : AT (ADDR (.rodata) - 0xC0000000) {
        *(.rodata*)        /* all read-only data sections from all files */
    }

    .data ALIGN (0x1000) : AT (ADDR (.data) - 0xC0000000) {
        *(.data)           /* all data sections from all files */
    }

    .bss ALIGN (0x1000) : AT (ADDR (.bss) - 0xC0000000) {
        *(COMMON)          /* all COMMON sections from all files */
        *(.bss)             /* all bss sections from all files */
        [a-z\/*]*kernel-entrypoint.o(.bss)
        _linker_kernel_stack_top = .;
    }
    /* Linker variable that can be used in kernel */
    _linker_kernel_virtual_addr_end = .;
    _linker_kernel_physical_addr_end = . - 0xC0000000;
}
```

• Modern OS: Virtual Address Space

Higher half kernel juga digunakan oleh Linux & Windows. Linux 32-bit secara default akan menggunakan `0xC0000000` hingga `0xFFFFFFFF` untuk kernel.



Linux 32-bit Memory Mapping, Source : [Linux Kernel Labs - Memory Mapping](#)

Mapping pada Linux dapat diganti menggunakan konfigurasi `/boot/config` dengan pilihan

- **1 GB User - 3 GB Kernel**
- **2 GB User - 2 GB Kernel**
- **3 GB User - 1 GB Kernel**

Sedangkan Windows 32-bit menggunakan **2 GB - 2 GB** untuk [Virtual Address Space \(Windows\)](#).

3.1.3. Activate Paging

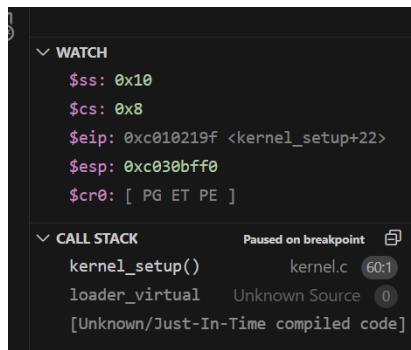
Proses menyalakan paging membutuhkan instruksi yang presisi untuk mengatur memory ketika transisi virtual-physical address. Instruksi harus akurat ketika menggunakan memory reference, apakah address yang dimaksud adalah physical atau virtual. Oleh karena itu, semua instruksi untuk menyalakan paging akan diimplementasikan pada assembly.

Berikut adalah step-step yang dilakukan kode assembly untuk menyalakan paging

1. Set register **CR3** untuk menyimpan physical address `_paging_kernel_page_directory`
2. Menyalakan flag **PSE** pada register **CR4** untuk 4 MiB paging
3. Menyalakan paging dengan flag **PG** pada **CR0**
4. Melompat ke virtual address kernel pada higher half (`0xc0100000`)
5. Menghapus identity paging kernel
6. Menyiapkan stack dan memanggil kode C `kernel_setup()`

Beruntungnya semua step ini dapat dilakukan sebelum kode C `kernel_setup()` dipanggil sehingga proses menyalakan **fitur paging akan sepenuhnya disediakan pada kit**. Kit akan menyediakan `kernel-entrypoint.s` yang telah mengimplementasikan step-step diatas untuk menyalakan paging.

Ketika `kernel_setup()` dipanggil oleh kode assembly `loader_entrypoint()`, sistem sudah dalam keadaan paging menyal dan kernel akan diletakkan pada higher half.



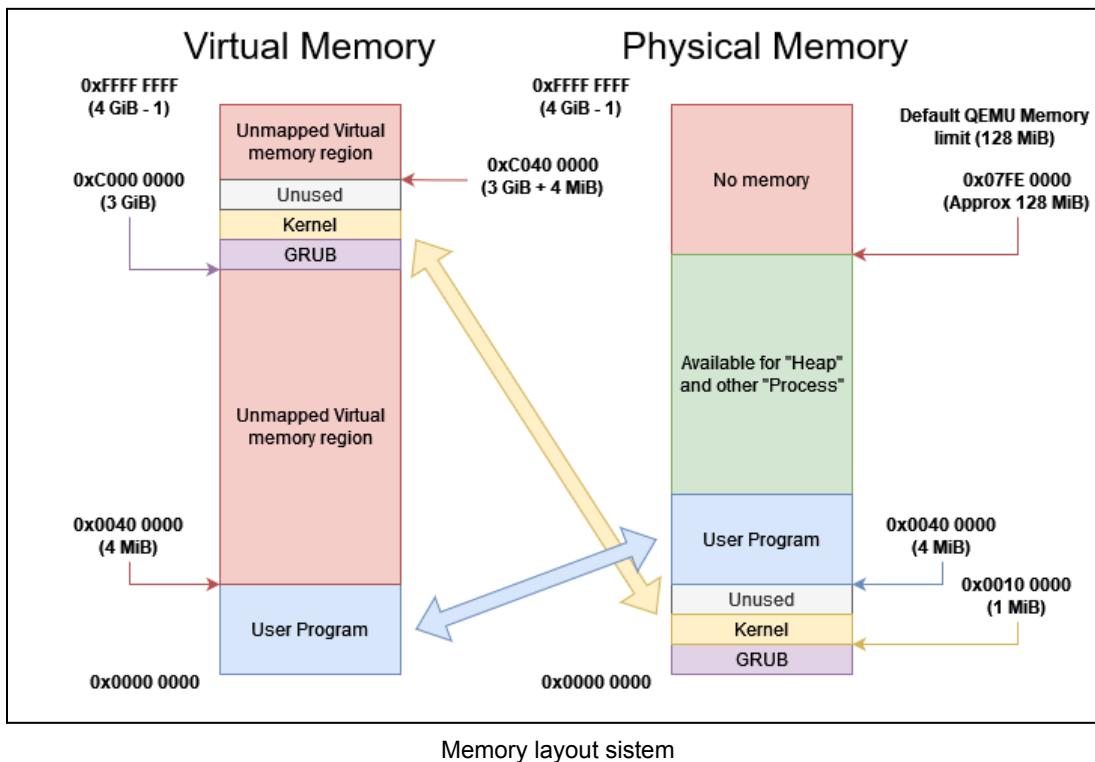
Debugger watch expression ke register CPU

Jika sistem operasi berjalan seperti biasa setelah menambahkan bagian ini (tidak **Triple Fault** sehingga bootloop), **fitur paging sudah aktif**. Dengan menggunakan debugger akan terlihat seperti gambar diatas dimana register `$eip` dan `$esp` berada pada region `0xc0000000`.

Address framebuffer `FRAMEBUFFER_MEMORY_OFFSET` **perlu diupdate ke `0xC00B8000`**. Jika tidak operasi framebuffer akan menyebabkan CPU exception **Page Fault** karena virtual address `0x0 - 0x00400000` tidak dipetakan ke physical address.

3.1.4. Memory Manager

Setelah paging aktif, diperlukan sebuah sistem untuk mengalokasikan dan memetakan physical-virtual memory. Bagian ini akan mengimplementasikan **Memory Manager** yang akan melakukan manajemen physical memory.



Berikut adalah rencana layout memory yang akan digunakan untuk user program

- Virtual memory pada address **0x0** hingga **0xC0000000** kosong tetapi dapat dipetakan
- Physical memory yang tersedia terletak pada **0x400000** hingga **0x7FE0000**
- Skema higher half memetakan physical frame 0 (kernel) ke virtual address **0xC0000000**

Memory manager akan menggunakan dan melakukan manajemen ruang kosong pada physical memory **0x400000** hingga **0x7FE0000**. Kernel akan menggunakan interface yang disediakan memory manager untuk melakukan alokasi page frame tanpa menyentuh physical memory.

3.1.4.1. Frame Allocator

Interface `paging_allocate_user_page_frame()` menerima virtual address dan mengalokasikan physical page frame ke virtual address tersebut pada `PageDirectory` yang diberikan. Page frame akan dicatat sebagai milik user space. Sama seperti **File System: Write**, interface ini harus mencatat physical page frame yang sudah digunakan dan mana yang masih kosong.

Strategi alokasi dibebaskan, First-fit, Best-fit, dan lain-lain. Permasalahan alokasi ini memiliki nama sendiri yaitu Bin Packing Problem. Bin dalam konteks ini berarti frame yang berukuran sama.

Pemetaan virtual-physical yang disediakan interface ini akan digunakan kembali untuk mengalokasikan page frame ketika membuat **Process**.

3.1.4.2. Frame Deallocator

Interface `paging_free_user_page_frame()` menerima virtual address dan `PageDirectory` dan melepaskan virtual page frame. Dengan pemetaan virtual-physical address yang didapatkan dari `PageDirectory` yang menggunakan page frame tersebut, physical page frame dapat dicatat sebagai siap digunakan kembali untuk alokasi selanjutnya.

3.1.4.3. Free Memory Check

Interface `paging_allocate_check()` mengecek apakah masih tersedia physical memory pada sistem untuk memenuhi alokasi dengan menerima argumen berupa berapa frame yang dibutuhkan. Pengecekan dapat dilakukan menggunakan informasi yang ada pada `page_manager_state` dan requested memory size.

• Physical Memory Hole

Pada tingkat kernel yang langsung berhubungan dengan hardware, akan terlihat behavior-behavior yang unik pada tingkat kernel yang diabstraksikan sehingga layer atas tidak merasakannya. Salah satu behavior unik hardware yang berhubungan dengan paging adalah **physical memory akan memiliki hole / lubang yang tidak bisa digunakan**.

Konfigurasi default dari QEMU akan menggunakan RAM berukuran **128 MiB**. Program bootloader seperti GRUB biasanya akan menyediakan mekanisme untuk mendapatkan layout physical memory.

```
grub> displaymem
EISA Memory BIOS Interface is present
Address Map BIOS Interface is present
Lower memory: 639K, Upper memory (to first chipset hole): 129920K
[Address Range Descriptor entries immediately follow (values are 64-
Usable RAM: Base Address: 0x0 X 4GB + 0x0,
Length: 0x0 X 4GB + 0x9fc00 bytes
Reserved: Base Address: 0x0 X 4GB + 0x9fc00,
Length: 0x0 X 4GB + 0x400 bytes
Reserved: Base Address: 0x0 X 4GB + 0xf0000,
Length: 0x0 X 4GB + 0x10000 bytes
Usable RAM: Base Address: 0x0 X 4GB + 0x100000,
Length: 0x0 X 4GB + 0x7ee0000 bytes
Reserved: Base Address: 0x0 X 4GB + 0x7fe0000,
Length: 0x0 X 4GB + 0x20000 bytes
Reserved: Base Address: 0x0 X 4GB + 0xffffc0000,
Length: 0x0 X 4GB + 0x40000 bytes
```

GRUB - displaymem command

Interface CLI GRUB menyediakan command `displaymem` untuk menampilkan layout physical memory. GRUB dapat dikonfigurasi untuk menyediakan informasi ini kepada kernel jika dibutuhkan dengan [Multiboot Header](#) (diimplementasikan pada `kernel-entrypoint.s .multiboot`).

Pada gambar diatas terlihat lubang-lubang memory yang ditandai dengan “**Reserved**”. Semua lubang memory yang ada tidak dapat digunakan oleh kernel maupun program lain. Selain itu, masih ingat dengan [Memory Mapped I/O](#)? I/O yang menggunakan memory ini juga tidak dapat digunakan oleh program lain untuk menyimpan data.

Virtual memory mengabstraksikan hal-hal ini dari user program. Kernel yang mengimplementasikan paging akan menyusun dan mengalokasikan physical memory yang tersedia sehingga user program dapat menggunakan memory layaknya tidak ada lubang sama sekali.

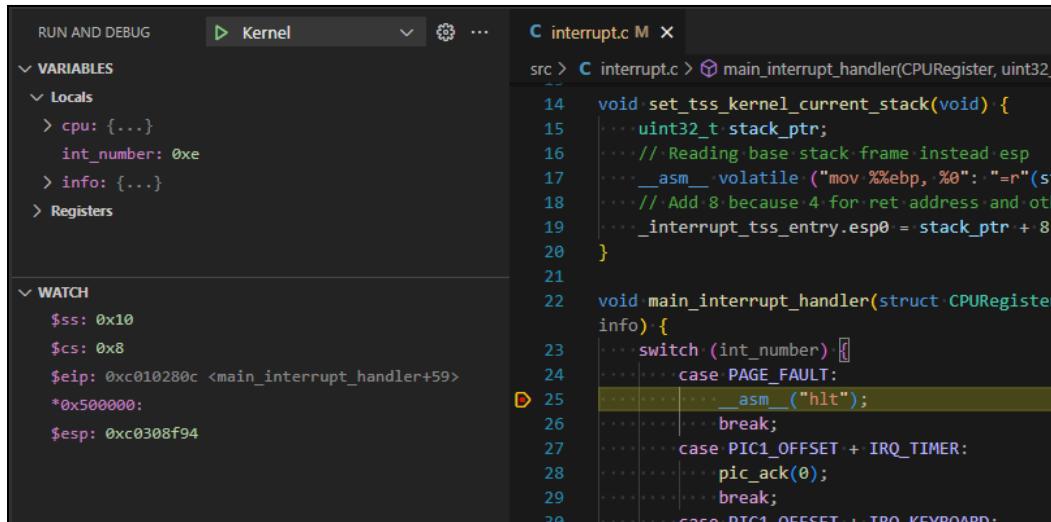
Tips: Paging

Gunakan debugger dan pointer untuk mengetes apakah page frame sudah terpetakan dengan baik atau tidak. Jika alokasi masih belum benar, **debugger akan gagal** dan variabel pointer menyebabkan CPU exception **Page Fault**.

Berikut contoh mencoba mengubah memory **0x500000** (virtual page frame 1) dengan pointer

```
*((uint8_t*) 0x500000) = 1;
```

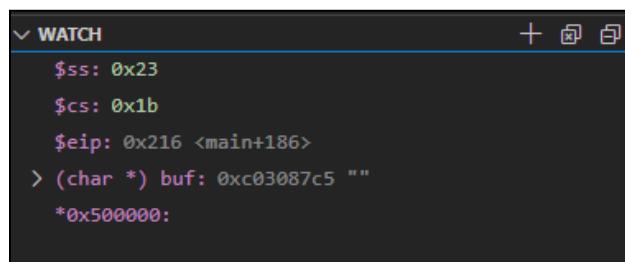
Kondisi ketika gambar diambil adalah virtual frame 1 belum dipetakan ke physical frame manapun. Dari kondisi tersebut, seharusnya debugger akan memperlihatkan page fault



The screenshot shows a debugger interface with the following details:

- RUN AND DEBUG**: Shows "Kernel" selected.
- VARIABLES**:
 - Locals**:
 - > **cpu**: {...}
 - int_number: 0xe
 - > **info**: {...}
 - > **Registers**
 - WATCH**:
 - \$ss: 0x10
 - \$cs: 0x8
 - \$eip: 0xc010280c <main_interrupt_handler+59>
 - *0x500000:
 - \$esp: 0xc0308f94
- interrupt.c M X**: The assembly code for the interrupt handler is displayed. Line 25 contains the instruction `asm ("hlt");`. A yellow arrow points to this line, indicating it is the current instruction being executed.

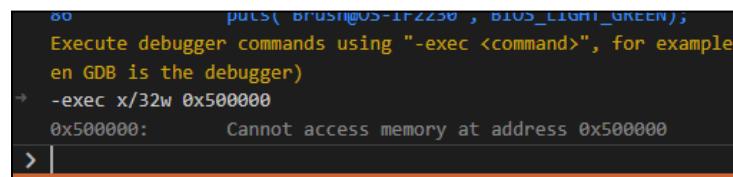
CPU Exception - Page Fault (Interrupt Vector 0xE)



The screenshot shows a "WATCH" window with the following entries:

- \$ss: 0x23
- \$cs: 0x1b
- \$eip: 0x216 <main+186>
- > (char *) buf: 0xc03087c5 ""
- *0x500000:

Watch expression tidak mengeluarkan apapun



```
60          puls( BRUSH@US-1F2Z3D , BIOS_LIGHT_GREEN );
Execute debugger commands using "-exec <command>", for example
en GDB is the debugger)
→ -exec x/32w 0x500000
0x500000:      Cannot access memory at address 0x5000000
```

Examine dengan gdb mengeluarkan error

Extra: Paging

Jika membutuhkan physical memory tambahan, alokasi physical memory QEMU dapat dimodifikasi dengan menambahkan flag `-m <Physical Memory>` ketika eksekusi QEMU, misalnya perintah berikut akan menjalankan QEMU dengan physical memory **1024 MiB**

```
qemu-system-i386 -m 1024M -drive file=drive.img,format=raw,media=disk,id=disk -s -S  
-cdrom os2025.iso
```

Jangan lupa untuk mengubah semua command yang menjalankan QEMU. Task vscode pada `.vscode/.tasks` juga perlu diganti agar debugger vscode akan menggunakan parameter memory yang sesuai.

Panduan ini dapat diselesaikan dengan ukuran physical memory dari pengaturan default (128 MB). Namun jika membutuhkan physical memory untuk kebutuhan tambahan dan alasan tertentu, edit dan sesuaikan bagian paging sesuai dengan kebutuhan.

Frequent Issue: Paging

- Jika **Triple Fault** ketika mengaktifkan paging, pastikan alignment 4 KiB terpenuhi
- [Higher Half Kernel](#) dan [Activate Paging](#) perlu dikerjakan secara bersamaan
- Setelah paging aktif, **debugger akan menggunakan virtual address**
- Debugger akan sama menggunakan state yang dimiliki CPU pada sistem operasi. Jika CPU pada OS tidak bisa mengakses virtual address tertentu, maka debugger juga akan gagal.
- Jika mengalami error seperti berikut

```
Starting build...
make build
ld: cannot find kernel_loader.o
make: *** [makefile:48: kernel] Error 1
```

Edit **linker.ld**. Pada .bss ubah **kernel-entrypoint.o(.bss)** menjadi **bin/kernel-entrypoint.o(.bss)**

• Memory Paging

Dengan abstraksi virtual memory, lokasi penyimpanan data secara fisik dari sebuah virtual page frame dapat diletakkan pada RAM atau media penyimpanan. Penyimpanan virtual memory pada secondary memory yang non-volatile ini bernama **Memory Paging**.

Penggunaan media penyimpanan non-volatile ini akan mensimulasikan memori yang tersedia pada komputer lebih besar daripada main memory (RAM) yang terpasang. User program hanya perlu untuk menggunakan virtual address space yang dimilikinya tanpa memikirkan physical memory yang terpasang pada komputer. Arsitektur yang menggunakan 64-bit virtual address dapat menyediakan 2^{64} bytes atau sekitar 18 exabytes untuk satu virtual address space.

sda	8:0	0	389.8M	1	disk
sdb	8:16	0	2G	0	disk [SWAP]
sdc	8:32	0	1T	0	disk /snap

Partisi swap pada Linux

	DumpStack.log.tmp	4/4/2023 5:19 AM	TMP File	8 KB
	hiberfil.sys	3/17/2023 6:01 PM	System file	16,723,688 ...
	pagefile.sys	3/17/2023 6:02 PM	System file	6,029,312 KB
	swapfile.sys	4/4/2023 3:59 AM	System file	278,528 KB

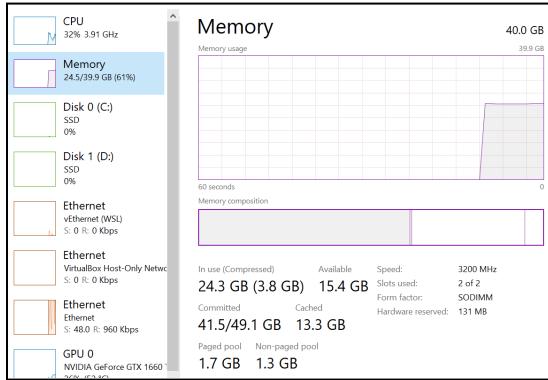
Page file pada windows

Setiap sistem operasi mungkin memberikan nama yang berbeda untuk fitur [Memory Paging](#) seperti **Pagefile** pada Windows atau **Swapfile** pada Linux. Implementasi memory paging setiap OS juga berbeda-beda. Linux dapat menggunakan sebuah **swapfile** atau **swap partition**, sedangkan Windows menggunakan sebuah file bernama **pagefile.sys**.

Kernel yang mengimplementasikan [Paging](#) dan memory paging dapat memindahkan sebagian physical page frame yang jarang digunakan ke non-volatile storage agar melepaskan sebagian RAM untuk keperluan lain. Algoritma manajemen memory paging wajib memperhatikan kecepatan secondary memory jauh dibawah primary memory RAM untuk menghindari kinerja sistem yang buruk.

Jika user program mengakses virtual memory yang sedang berada pada secondary storage, MMU akan mengirimkan CPU Exception **Page Fault** yang akan di handle oleh kernel untuk memasukkan data dari secondary ke primary memory. Operasi swap ketika page fault sangat berdampak pada kinerja user program, termasuk pada [Windows: Memory & Performance](#).

• Windows: Memory & Performance



Windows 10 64-bit Task Manager - Memory

Note: Pada konteks RAM digunakan prefix binary (GiB) meskipun ditulis ala prefix SI (GB)

Sistem operasi modern seperti Windows tentunya menggunakan **Paging** dan **Pagefile**. Gambar diatas memperlihatkan Windows 10 Task Manager yang berjalan pada laptop dengan RAM 40 GB dengan konfigurasi pagefile 9.4GB.

Committed adalah jumlah total memory (RAM + Pagefile) yang digunakan. **In use (Compressed)** adalah jumlah physical page frame yang sedang ada pada RAM. Aplikasi yang jarang digunakan akan dimasukkan kedalam pagefile dan akan diambil hanya ketika aplikasi tersebut aktif kembali.

Nilai dari committed adalah batas atas dari ukuran sebuah aplikasi dapat menggunakan memory. Jika ada aplikasi yang menghabiskan memory hingga diatas committed, kernel Windows akan menolak request dan sebagian besar akan membuat aplikasi berhenti berjalan.

Operasi swap antara RAM dan storage tergolong mahal. Setiap kali terjadi **Page Fault**, kernel wajib **menggunakan operasi swap ini secara langsung**. Ingat sequence diagram yang ada pada **Interrupt**. Interrupt handler bersifat time-sensitive sehingga implementasi ISR umumnya bersifat ringan dan meminimalkan operasi mahal secara waktu. Namun operasi swap mau tidak mau harus melakukan **blocking total thread pada process** hingga I/O dari storage selesai.

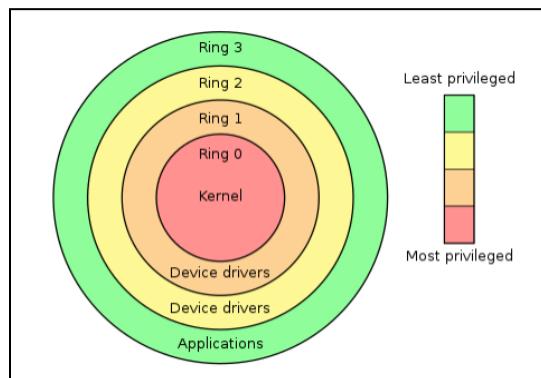
Pernahkah bermain sebuah game atau aplikasi yang membutuhkan requirement RAM yang jauh di atas komputer yang digunakan? Jika game atau aplikasi masih dapat dijalankan, mungkin akan sering mengalami **frame-per-second stuttering** atau **not responding**. Alasan utama terjadinya hal tersebut adalah operasi swap. CPU dan GPU tidak dapat melanjutkan eksekusi program karena thread blocking ketika page fault sehingga akan mengalami **Bottleneck** pada kecepatan storage. Page fault yang berulang kali ini disebut **Thrashing** dan merupakan penyebab utama komputer terasa lambat.

3.2. User Mode

User Mode dan **Kernel Mode** adalah konsep utama dari sistem operasi modern. Dengan adanya kernel dan user mode, program dalam user mode tidak dapat membuat komputer reboot ketika error atau menggunakan hardware seenaknya (Terjadi pada era [MS-DOS](#) & [Real Mode x86 16-bit!](#)).

Protected mode x86, seperti judul dari buku ini, protected mode menambahkan fitur [Paging](#), Segmentation ([GDT](#)), 32-bit, dan lain-lain yang menjadi peningkatan dari Real Mode yang hanya 16-bit dan tidak memiliki proteksi sama sekali. Fitur-fitur ini diperlukan untuk membuat kernel-user space. Pada titik ini, sistem operasi seharusnya sudah mengimplementasikan semua bagian yang dibutuhkan untuk membuat separasi user & kernel mode.

Flashback ke bagian [Global Descriptor Table](#). Kernel pada panduan ini hanya akan menggunakan 2 ring proteksi: **Ring 0 untuk Kernel Mode** dan **Ring 3 untuk User Mode**.



Source: [Protection Ring - Wikipedia](#)

Hasil akhir pada bagian ini akan membuat sebuah program yang hidup pada **Ring 3**. Selanjutnya program akan di-dekorasi hingga menjadi [Shell](#).

Implementasi kernel-user mode dimulai dengan membuat [External Program: Inserter](#) pada Host OS untuk memasukkan sembarang file dan folder ke dalam **storage.bin**. Dilanjutkan dengan membuat [GDT: User & Task State Segment Descriptor](#) yang diperlukan CPU untuk melakukan perpindahan privilege.

[Simple User Program](#) yang akan dibuat hanya infinite loop dalam format [flat binary executable](#). Kernel perlu memiliki mekanisme untuk [Execute Program](#) terlebih dahulu sebelum [Launching User Mode](#) menggunakan user program yang dibuat.

3.2.1. External Program: Inserter

Ketika membaca bagian ini, diharapkan implementasi [FS: Initializer](#) dan [FS: CRUD](#) untuk EXT2 - IF2130 Edition tidak melanggar dengan menggunakan batasan header selain yang disebutkan. Jika implementasi telah menerapkan [Modular & Reusability](#), bagian ini akan menjadi singkat.

Istilah eksternal pada konteks ini adalah program yang berjalan pada **Host OS**. Bagian ini membuat sebuah program bernama **inserter** yang memiliki fungsi untuk memasukkan sembarang file dan folder ke dalam storage dengan file system EXT2 - IF2130 Edition. Media penyimpanan dalam konteks ini adalah virtual storage **storage.bin**.

Kode main program inserter merupakan program CLI sederhana yang menerima beberapa argumen untuk membuat file atau folder.

```
./inserter <file or folder to insert> <parent inode index> <storage>
```

Kode main dari program inserter tersedia pada **ch3/3 - User Mode/external-inserter.c**. Modifikasi main function **inserter** sesuai dengan kebutuhan.

Gunakanlah command berikut untuk membuat executable **inserter**

```
makefile
```

```
inserter:  
    @$(CC) -Wno-builtin-declaration-mismatch -g -I$(SOURCE_FOLDER) \  
          $(SOURCE_FOLDER)/stdlib/string.c \  
          $(SOURCE_FOLDER)/filesystem/ext2.c \  
          $(SOURCE_FOLDER)/external/external-inserter.c \  
    -o $(OUTPUT_FOLDER)/inserter
```

Jika menemukan **Segmentation Fault** ketika menjalankan inserter, gunakan debugger. Tambahkan konfigurasi bernama **Inserter** untuk debugger seperti berikut

launch.json

```
{  
    "name": "Inserter",  
    "type": "cppdbg",  
    "request": "launch",  
    "program": "${workspaceFolder}/bin/inserter",  
    "args": ["shell", "2", "storage.bin"],  
    "stopAtEntry": false,  
    "cwd": "${workspaceFolder}/bin",  
    "environment": [],  
    "preLaunchTask": "Build Inserter",  
    "externalConsole": false,  
    "MIMode": "gdb",  
    "setupCommands": [  
        {  
            "description": "Enable pretty-printing for gdb",  
            "text": "-enable-pretty-printing",  
            "ignoreFailures": true  
        },  
        {  
            "description": "Set Disassembly Flavor to Intel",  
            "text": "-gdb-set disassembly-flavor intel",  
            "ignoreFailures": true  
        },  
        {  
            "text": "set output-radix 16",  
            "description": "Use hexadecimal output"  
        },  
    ]  
}
```

tasks.json

```
{  
    "type": "cppbuild",  
    "label": "Build Inserter",  
    "command": "make",  
    "args": ["inserter"],  
    "options": {"cwd": "${workspaceFolder}"},  
    "group": {  
        "kind": "build",  
        "isDefault": true  
    }  
}
```

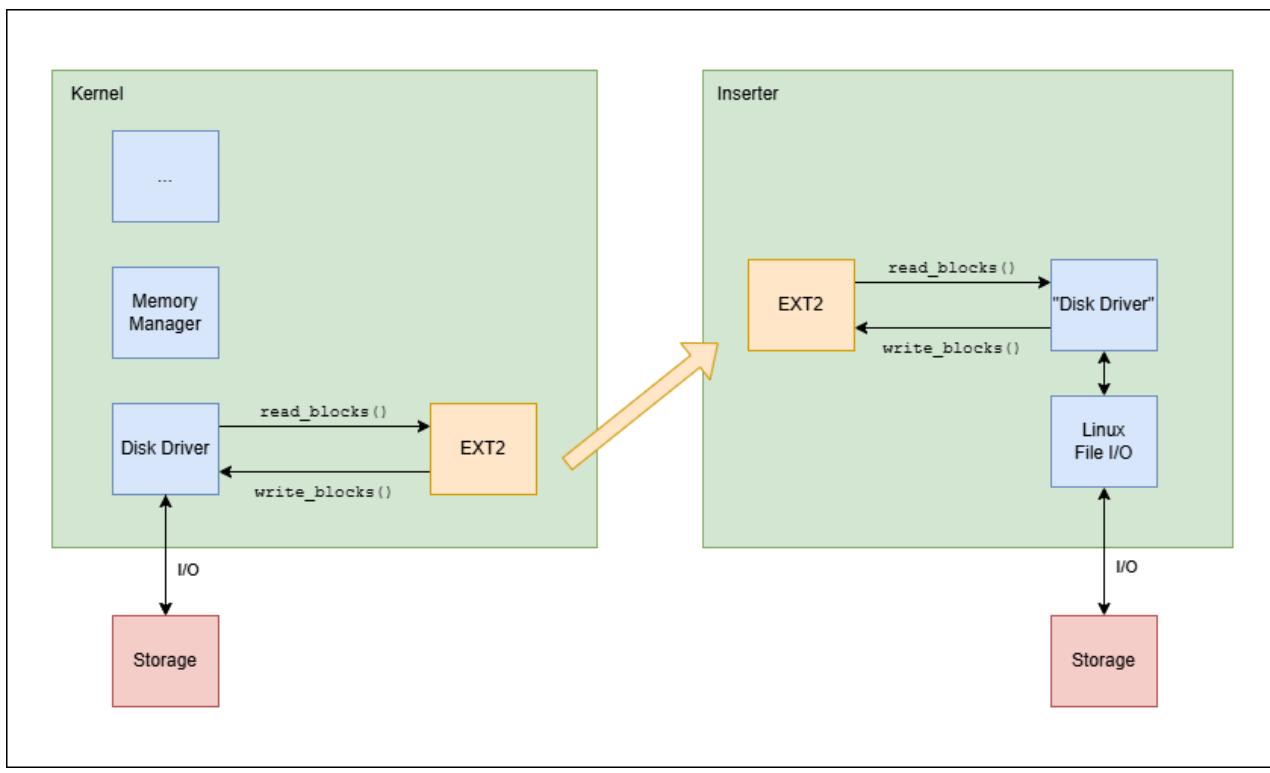
- **Modular & Reusability**

Mendapatkan solusi adalah tahap pertama. Langkah selanjutnya adalah memastikan solusi bersifat **Modular & Reusable**. Dengan membuat kode dapat di-reusable, kode dapat digunakan pada sistem lain dan tidak perlu copy-paste dan mengubah sebagian kecil kode.

Apa alasan implementasi file system tidak boleh menggunakan header selain yang disebutkan?

Meskipun C tidak menyediakan keyword dan language feature secara langsung, deklarasi fungsi pada header dapat digunakan seperti [Interface pada Java](#) atau [Concepts pada C++](#).

Interface akan dijadikan **sebuah perjanjian** bahwa pemanggilan fungsi akan membutuhkan informasi tertentu dan mengembalikan nilai apa. Kode implementasi EXT2 - IF2130 Edition **hanya membutuh**



Reusing EXT2

[**Disk Driver**](#) yang diberikan telah membuat implementasi spesifik digunakan didalam sistem operasi. Tentunya implementasi interface ini tidak dapat digunakan pada host OS Linux atau Windows sehingga membutuhkan implementasi spesifik OS. Kode [**external-inserter.c**](#) menyediakan main program dan juga mengimplementasikan interface disk driver diatas Linux file I/O sehingga implementasi EXT2 akan tetap dapat digunakan dengan normal.

3.2.2. GDT: User & Task State Segment Descriptor

CPU menggunakan GDT [Segment Selector](#) untuk menandai **Current Privilege Level (CPL)** dan [Segment Descriptor](#) yang sedang aktif. Beberapa detail yang disembunyikan pada assembly [Load GDT](#) akan dibutuhkan untuk mengimplementasikan user mode.

Pada protected mode, segment register sebenarnya memiliki peran berbeda untuk segmentation. Semua CPU segment register (cs, ds, ss, es, dan lain-lain) digunakan sebagai penyimpan struktur data GDT segment selector. Bit ke-3 hingga ke-15 digunakan sebagai index GDT table dan bit ke-0 hingga ke-1 digunakan untuk **CPL**.

Informasi diatas akan digunakan secara langsung pada [Execute Program](#) untuk memasuki user mode dan melakukan eksekusi user program. Bagian ini hanya akan mengimplementasikan segment descriptor yang diperlukan untuk membuat user space.

Kernel perlu menyimpan register yang berhubungan dengan kernel stack pointer (**esp** dan **ss**) sebelum berpindah ke user mode. Informasi ini akan digunakan ketika ada perpindahan dari user ke kernel dan disimpan pada struktur data [Task State Segment \(TSS\)](#). Struktur data Task State Segment akan berperan penting pada [Inter Privilege Interrupt](#).

Namun CPU perlu mengetahui dimana lokasi struktur data TSS terletak pada memory untuk membaca informasinya, sama seperti **GDTR & IDTR**. Entry pada tabel GDT selain menyimpan segment descriptor biasa, juga dapat menyimpan **TSS Descriptor**. GDT nantinya akan ditambahkan 1 TSS descriptor yang akan digunakan seluruh sistem. Selain itu, User Mode akan membutuhkan 2 [User Segment Descriptor](#) tambahan yang membawa flag user, satu untuk data dan satu untuk kode.

3.2.2.1. Task State Segment

Struktur data task state segment menyimpan banyak informasi yang digunakan untuk perpindahan privilege dan task yang penting untuk **User Mode**, **System Call**, dan **Chapter 4: Context Switch**.

Namun karena alasan **Hardware & Software Context Switch**, panduan ini hanya akan menggunakan sebagian kecil TSS untuk keperluan software context switch nantinya.

Definisi lengkap dan informasi detail Task-State Segment terdapat pada **Intel Manual Vol 3a - Chapter 8: 8.2.1 Task-State Segment (TSS)**

Berikut adalah definisi struktur data TSS, tambahkan pada **interrupt.h**

```
interrupt.h

extern struct TSSEntry _interrupt_tss_entry;

/***
 * TSSEntry, Task State Segment. Used when jumping back to ring 0 / kernel
 */
struct TSSEntry {
    uint32_t prev_tss; // Previous TSS
    uint32_t esp0;    // Stack pointer to load when changing to kernel mode
    uint32_t ss0;    // Stack segment to load when changing to kernel mode
    // Unused variables
    uint32_t unused_register[23];
} __attribute__((packed));

// Set kernel stack in TSS
void set_tss_kernel_current_stack(void);
```

Definisikan `_interrupt_tss_entry` pada **interrupt.c**. Hanya satu definisi entry TSS yang diperlukan untuk seluruh sistem. Definisi TSS ini diperlukan untuk **System Calls**.

Tambahkan juga implementasi `set_tss_kernel_current_stack()` berikut

```
interrupt.c

struct TSSEntry _interrupt_tss_entry = {
    .ss0 = GDT_KERNEL_DATA_SEGMENT_SELECTOR,
};

void set_tss_kernel_current_stack(void) {
    uint32_t stack_ptr;
    // Reading base stack frame instead esp
    __asm__ volatile ("mov %%ebp, %0": "=r"(stack_ptr) : /* <Empty> */);
    // Add 8 because 4 for ret address and other 4 is for stack_ptr variable
    _interrupt_tss_entry.esp0 = stack_ptr + 8;
}
```

3.2.2.2. User Segment Descriptor

Dua user segment descriptor yang diperlukan tidak jauh berbeda dengan kernel segment descriptor yang ada pada [GDT & GDTR Definition](#). Satu descriptor untuk user code segment dan satu lain untuk user data segment.

Tambahkan segment descriptor pada `gdt.c`. Ingat bahwa urutan definisi berpengaruh.

- 2 entry user descriptor, sama seperti kernel descriptor, dengan privilege bernilai 0x3
- 1 entry TSS dan `gdt_install_tss()`

```
gdt.c

static struct GlobalDescriptorTable global_descriptor_table = {
    .table = {
        /* TODO: Null Descriptor */,
        /* TODO: Kernel Code Descriptor */,
        /* TODO: Kernel Data Descriptor */,
        /* TODO: User Code Descriptor */,
        /* TODO: User Data Descriptor */,
        {
            .segment_high      = (sizeof(struct TSSEntry) & (0xF << 16)) >> 16,
            .segment_low       = sizeof(struct TSSEntry),
            .base_high         = 0,
            .base_mid          = 0,
            .base_low          = 0,
            .non_system        = 0,           // S bit
            .type_bit          = 0x9,
            .privilege         = 0,           // DPL
            .valid_bit         = 1,           // P bit
            .opr_32_bit        = 1,           // D/B bit
            .long_mode         = 0,           // L bit
            .granularity       = 0,           // G bit
        },
        {0}
    }
};

void gdt_install_tss(void) {
    uint32_t base = (uint32_t) &_interrupt_tss_entry;
    global_descriptor_table.table[5].base_high = (base & (0xFF << 24)) >> 24;
    global_descriptor_table.table[5].base_mid  = (base & (0xFF << 16)) >> 16;
    global_descriptor_table.table[5].base_low  = base & 0xFFFF;
}
```

Tambahkan juga deklarasi berikut pada **gdt.h**.

gdt.h

```
#define GDT_USER_CODE_SEGMENT_SELECTOR 0x18
#define GDT_USER_DATA_SEGMENT_SELECTOR 0x20
#define GDT_TSS_SELECTOR              0x28

// Set GDT_TSS_SELECTOR with proper TSS values, accessing _interrupt_tss_entry
void gdt_install_tss(void);
```

Macro yang ditambahkan di atas akan digunakan untuk memilih segment descriptor pada GDT. Macro didefinisikan dengan asumsi urutan definisi sama seperti yang digambarkan pada komentar pada halaman sebelumnya.

3.2.3. Simple User Program

Agar *make sense*, masuk ke user mode perlu diiringi dengan sebuah user program. Tidak ada operasi yang penting pada user program. Kernel hanya membutuhkan sesuatu yang dapat dipanggil untuk user mode.

Sama dengan sistem operasi modern lain, user program yang akan diimplementasikan **disimpan pada filesystem** dan dieksekusi oleh kernel. Kernel akan menyiapkan virtual memory dan semua state yang dibutuhkan untuk mengeksekusi user program.

Format user program yang digunakan adalah **Flat Binary Executable** yang sederhana. Format ini sepenuhnya hanya berisikan instruksi machine code tanpa header tambahan. Konsekuensi dari tidak adanya header adalah urutan linking oleh [Linker](#) akan berpengaruh.

Isi dari simple user program hanya sebuah instruksi untuk memasukkan nilai ke dalam register `eax` dan mengembalikan nilai return. Berikut adalah implementasi untuk simple user program

```
user-shell.c
```

```
int main(void) {
    __asm__ volatile("mov %0, %%eax" : /* <Empty> */ : "r"(0xDEADBEEF));
    return 0;
}
```

Akan disediakan assembly dan linker script sederhana untuk menghindari linking yang tidak dapat diprediksi dan menyamakan [C: Program Entry point](#) seperti biasanya: fungsi `main()`.

```
crt0.s
```

```
global _start
extern main

section .text
_start:
    call main
    jmp $
```

```

user-linker.1d

ENTRY(_start)

SECTIONS {
    . = 0x00000000; /* Assuming OS will load this program at virtual address */

    .text ALIGN(4): {
        [a-zA-Z]*crt0.o(.text) /* Put the entrypoint in the front of executable */
        *(.text)
    }

    .data ALIGN(4): {
        *(.data)
    }

    .bss ALIGN(4): { /* 9 May Edit: Fixes for static storage variables */
        *(COMMON)
        *(.bss)
    }

    .rodata ALIGN(4): {
        *(.rodata*)
    }
    _linker_user_program_end = .;
    ASSERT (_linker_user_program_end <= 1 * 1024 * 1024), "Error: User program
linking result is more than 1 MiB"
}

```

Ketiga file akan link menjadi **flat binary executable** format menggunakan **1d** seperti berikut

```

makefile

user-shell:
    @$(ASM) $(AFLAGS) $(SOURCE_FOLDER)/crt0.s -o crt0.o
    @$(CC) $(CFLAGS) -fno-pie $(SOURCE_FOLDER)/user-shell.c -o user-shell.o
    @$(LIN) -T $(SOURCE_FOLDER)/user-linker.1d -melf_i386 --oformat=binary \
        crt0.o user-shell.o -o $(OUTPUT_FOLDER)/shell
    @echo Linking object shell object files and generate flat binary...
    @size --target=binary $(OUTPUT_FOLDER)/shell
    @rm -f *.o

insert-shell: inserter user-shell
    @echo Inserting shell into root directory...
    @cd $(OUTPUT_FOLDER); ./inserter shell 2 $(DISK_NAME).bin

```

Dengan menggunakan resep **insert-shell**, shell dapat dimasukkan ke dalam storage dengan file system **EXT2 - IF2130 Edition**. Storage yang telah diisi dengan inserter akan dibaca oleh kernel dan [Execute Program](#) user program yang telah dimasukkan.

● C: Program Entrypoint

Tutorial awal dari pemrograman menggunakan C pasti akan dimulai dengan mengenalkan “Hello, World!” legendaris dan mengenalkan bahwa fungsi `main()` merupakan fungsi pertama yang dipanggil pada bahasa C

```
#include <stdio.h>

int main() {
    printf("Hello, World!");
    return 0;
}
```

Untuk sebagian besar user program, informasi `main()` merupakan fungsi yang dipanggil pertama sudah cukup. Namun untuk kernel dan compiler developer, terdapat hal yang penting diketahui bahwa fungsi ini bukanlah entrypoint program yang sebenarnya. Bahasa C dan C++ menggunakan fungsi `_start` sebagai entrypoint dari semua user program.

Fungsi `_start` diimplementasikan pada sebuah object file bernama `crt0.o` yang biasanya disimpan pada library C Linux dan otomatis di link untuk semua user program. Fungsi ini digunakan untuk menyiapkan runtime, hal spesifik sistem operasi dan arsitektur sebelum memanggil fungsi `main()`.

Kernel dan user program yang dibuat pada panduan ini sejak awal di compile dan link menggunakan gcc flag `-nostartfiles`. Flag ini akan meminta gcc untuk tidak melakukan linking hasil kompilasi dengan `crt0.o` sistem. Oleh karena itu, pada bagian [Simple User Program](#), diperlukan implementasi sendiri `crt0.o` sederhana.

Tidak ada kebutuhan yang kompleks pada user program pada panduan ini. Implementasi `crt0.o` hanya memastikan `_start` selalu menjadi entrypoint dengan diletakkan pada bagian paling depan dari executable. Flat binary executable yang tidak memiliki informasi tabel address symbol akan menjadikan bagian terawal binary sebagai entrypoint.

Penjelasan tambahan untuk C & C++ entrypoint: embeddedartistry.com/overview-before-main

3.2.4. Execute Program

Dari semua yang dibutuhkan untuk menjalankan sebuah program pada user mode hanya terdapat satu bagian yang tersisa: **Execute Program**.

Eksekusi program ini tidak dapat diimplementasikan dengan C karena membutuhkan manipulasi register dan stack sebelum menggunakan instruksi khusus `iret`. Arsitektur x86 tidak menyediakan instruksi khusus untuk melakukan lompat ke user program sehingga trik instruksi `iret` akan digunakan untuk hal ini.

Tambahkan kode assembly berikut pada `kernel-entrypoint.s` pada **section .text**

```
kernel-entrypoint.s
```

```
global kernel_execute_user_program ; execute initial user program from kernel
kernel_execute_user_program:
    mov eax, 0x20 | 0x3
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax

    ; Using iret (return instruction for interrupt) technique for privilege change
    ; Stack values will be loaded into these register:
    ; [esp] -> eip, [esp+4] -> cs, [esp+8] -> eflags, [] -> user esp, [] -> user ss
    mov ecx, [esp+4] ; Save first (before pushing anything to stack) for last push
    push eax ; Stack segment selector (GDT_USER_DATA_SELECTOR), user privilege
    mov eax, ecx
    add eax, 0x400000 - 4
    push eax ; User space stack pointer (esp), move it into last 4 MiB
    pushf ; eflags register state, when jump inside user program
    mov eax, 0x18 | 0x3
    push eax ; Code segment selector (GDT_USER_CODE_SELECTOR), user privilege
    mov eax, ecx
    push eax ; eip register to jump back

    iret
```

Tambahkan deklarasi berikut pada **kernel-entrypoint.h** agar dapat dipanggil oleh **kernel.c**

```
kernel-entrypoint.h

// Optional linker variable : Pointing to kernel start & end address
// Note : Use & operator, example : a = (uint32_t) &_linker_kernel_stack_top;
extern uint32_t _linker_kernel_virtual_addr_start;
extern uint32_t _linker_kernel_virtual_addr_end;
extern uint32_t _linker_kernel_physical_addr_start;
extern uint32_t _linker_kernel_physical_addr_end;
extern uint32_t _linker_kernel_stack_top;

/**
 * Execute user program from kernel, one way jump. This function is defined in asm
source code.
 *
 * @param virtual_addr Pointer into user program that already in memory
 * @warning           Assuming pointed memory is properly loaded with instruction
 */
extern void kernel_execute_user_program(void *virtual_addr);

/**
 * Set the tss register pointing to GDT_TSS_SELECTOR with ring 0
 */
extern void set_tss_register(void); // Implemented in kernel-entrypoint.s
```

3.2.5. Launching User Mode

Semua yang dibutuhkan untuk menjalankan user mode sudah terpenuhi sehingga yang dibutuhkan selanjutnya adalah merangkai bagian-bagian tersebut.

Task state segment perlu dipasang terlebih dahulu dan dilanjutkan dengan mengalokasikan virtual memory untuk user program. File system `read()` akan membaca storage dan memasukkan user program ke memory. Terakhir, nilai TSS diperbarui dengan kondisi kernel stack dan diselesaikan dengan melompat ke user program.

Karena untuk memasuki user mode membutuhkan banyak struktur data dan instruksi yang presisi, sering kali ketika menjalankan bagian ini ditemui isu-isu yang sama. Beberapa isu dan clue untuk resolusinya terdapat pada [Frequent Issue: User Mode](#).

Tambahkan kode pada kernel seperti berikut

```
kernel.c

void kernel_setup(void) {
    load_gdt(&_gdt_gdtr);
    pic_remap();
    initialize_idt();
    activate_keyboard_interrupt();
    framebuffer_clear();
    framebuffer_set_cursor(0, 0);
    initialize_filesystem_ext2();
    gdt_install_tss();
    set_tss_register();

    // Allocate first 4 MiB virtual memory
    paging_allocate_user_page_frame(&_paging_kernel_page_directory, (uint8_t*) 0);

    // Write shell into memory
    struct EXT2DriverRequest request = {
        .buf              = (uint8_t*) 0,
        .name             = "shell",
        .inode            = 1,
        .buffer_size      = 0x100000,
        .name_len         = 5,
    };
    read(request);

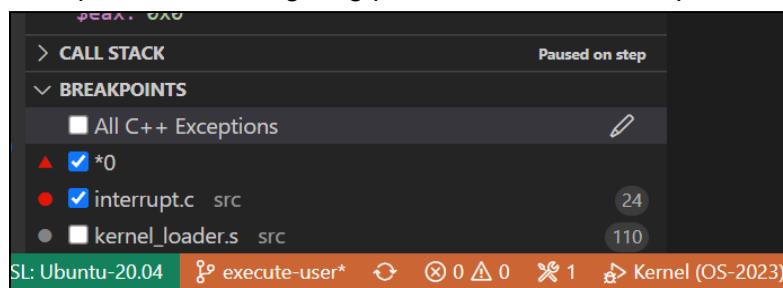
    // Set TSS $esp pointer and jump into shell
    set_tss_kernel_current_stack();
    kernel_execute_user_program((uint8_t*) 0);

    while (true);
}
```

Tips: User Mode

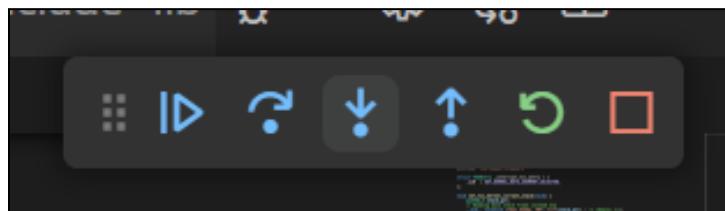
- **Penting:** Kernel dan user program akan di compile secara terpisah, jangan lupa untuk memanggil `make disk` dan `make insert-shell` ketika mengubah source code user program
- CRUD file system **harus menjaga data integrity**. Pembacaan user program yang **reordered** atau **corrupted** akan membuat masalah ketika dijalankan.
- Binary shell dapat didump dengan command `objdump -D -S -b binary -m i386 shell`
- Karena user program di compile secara terpisah dan semua Symbol distrip (karena flat binary format), debugger akan kebingungan membaca executable ini.

Detail fix untuk ini akan dijelaskan pada [Shell: Debugger](#), tetapi untuk sekarang dapat digunakan breakpoint secara langsung pada virtual address seperti berikut



Contoh breakpoint pada virtual address **0x00000000**

- Gunakan **step into** ketika menggunakan debugger pada assembly view. Step over akan didasarkan oleh kode C, assembly yang berkorespondensi dengan function call akan dianggap sebagai kesatuan dan dilompati oleh step over.



Step into pada VSCode debugger

Frequent Issue: User Mode

Berikut adalah isu yang sering kali ditemukan ketika menjalankan user mode. Pastikan sekali lagi [CRUD File System](#) telah bekerja dengan baik. Implementasi file system yang tidak diuji dengan baik seringkali menjadi masalah ketika mengerjakan bagian ini.

Pause QEMU terlebih dahulu jika masih berjalan (Tombol kiri atas QEMU, Machine → Pause)

- Tidak terjadi apapun, `eip` menunjuk pada instruksi kernel
Pastikan `read()` berhasil membaca file ke memory dan `kernel_execute_user_program()` telah dipanggil pada `kernel.c`.
- QEMU crash
Struktur data Task State Segment belum terinisiasi atau terdefinisi dengan baik.
- **Triple Fault & Bootloop**
Ada kemungkinan kesalahan pada segment descriptor atau struktur data CPU. Pastikan modifikasi pada [GDT: User & Task State Segment](#) tidak mengubah kernel descriptor.
- Terkena **General Protection Fault / Page Fault**
Ada kemungkinan kesalahan pada paging dan virtual memory. Uji dengan mengecek memory pada address 0 seperti [Tips: Paging](#).
- Jika QEMU diam saja, `eip` menunjuk ke address sekitar 0x0 hingga 0x100
Berhasil masuk ke user program dengan user mode

Untuk mencari akar permasalahan, gunakan debugger dan step satu per satu kode kernel. Uji satu per satu bagian seperti yang disebutkan pada list diatas. Bagian selanjutnya akan mendekorasi simple user program ini menjadi user interface sistem operasi [Shell](#).

3.3. Shell

```
* status: started
mars@marsmain /usr/portage/app-shells/bash $ ping -q -c1 en.wikipedia.org
PING rr.esams.wikimedia.org (91.198.174.2) 56(84) bytes of data.

--- rr.esams.wikimedia.org ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 2ms
rtt min/avg/max/mdev = 49.820/49.820/49.820/0.000 ms
mars@marsmain /usr/portage/app-shells/bash $ grep -i /dev/sda /etc/fstab | cut --fields=-3
/dev/sda1          /boot
/dev/sda2          none
/dev/sda3          /
mars@marsmain /usr/portage/app-shells/bash $ date
Sat Aug  8 02:42:24 MSD 2009
mars@marsmain /usr/portage/app-shells/bash $ lsmod
Module           Size  Used by
rndis_wlan       23424  0
rndis_host        8696   1 rndis_wlan
cdc_ether         5672   1 rndis_host
usbnet          18688   3 rndis_wlan,rndis_host,cdc_ether
parport_pc       38424  0
fglrx          2388128  20
parport         39648   1 parport_pc
iTCO_wdt        12272  0
i2c_i801         9380   0
mars@marsmain /usr/portage/app-shells/bash $
```

CLI shell, source : [Bash \(Unix shell\) - Wikipedia](#)

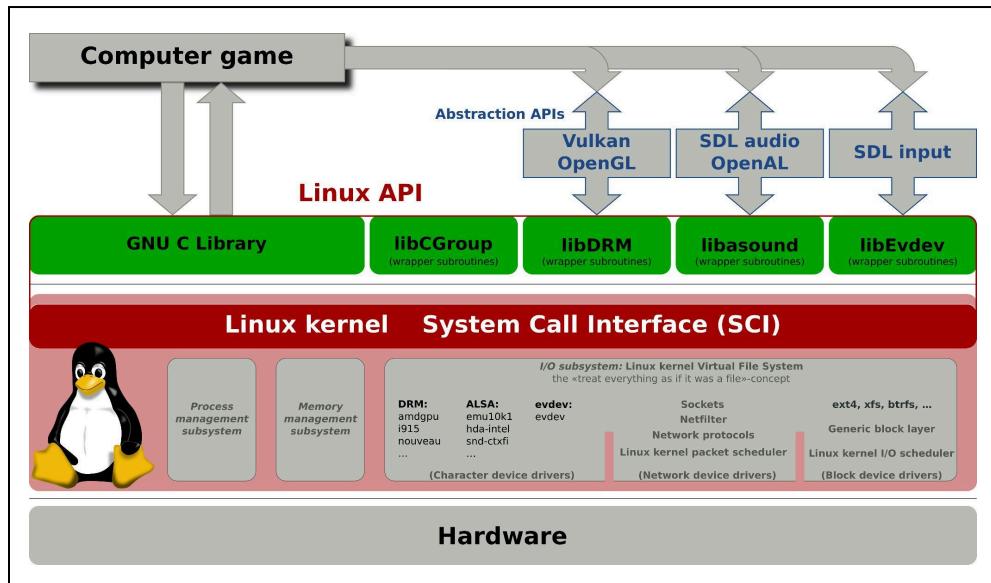
Shell adalah bagian terluar dari sistem operasi yang menggunakan layanan **Kernel** untuk menyediakan user interface. Bagian ini akan mengimplementasikan shell yang menyediakan interface dalam bentuk **Command Line Interface (CLI)**.

Pada titik ini, kernel sudah menyediakan cukup layanan yang dapat digunakan untuk membuat sebuah shell yang berjalan pada user mode. Bagian ini merupakan satu-satunya bagian implementasi user interface yang dapat langsung digunakan oleh pengguna. Modifikasi sendiri dan gunakan kreativitas ketika mengimplementasikan bagian ini.

Shell yang diimplementasikan akan dijalankan pada **User Mode** sehingga shell tidak memiliki akses penuh resource sistem. Oleh karena itu, kernel perlu mengimplementasikan **System Calls** yang menyediakan operasi I/O dan layanan lain ke user program. Setelah system calls disediakan oleh kernel, dilanjutkan mendekorasi simple user program sebelumnya menjadi shell dengan **Command Line Interface Shell**.

3.3.1. System Calls

System Calls (syscall) adalah cara utama kernel menyediakan layanan I/O dan hardware kepada user program. Shell yang diimplementasikan akan menggunakan system calls untuk menuliskan karakter ke layar, menerima input keyboard, mengoperasikan file system, dan lain-lain.



Game → System Call → Kernel → Hardware, Source: [Linux Kernel Interfaces - Wikipedia](#)

Pada protected mode x86, salah satu cara umum untuk mengimplementasikan system calls adalah menggunakan [Inter-Privilege Interrupt](#). Implementasi system calls menggunakan interrupt tidak jauh berbeda dengan menambahkan ISR baru sehingga cukup sederhana.

Perlu diingat, **program user mode tidak dapat menulis data ke memory kernel atau I/O**. Percobaan user untuk melakukan akses memory kernel akan menyebabkan CPU exception [General Protection Fault](#) (atau Segmentation Fault). Hal ini berbeda dengan **kernel yang dapat menuliskan apapun pada semua memory**. Nantinya syscall menggunakan fakta ini untuk menuliskan data yang didapatkan dari I/O ke memory milik user.

Sebelum mengimplementasikan syscall, perlu untuk [Designing System Calls](#) terlebih dahulu. Perancangan system calls akan dipandu pada bagian selanjutnya, tetapi detail implementasi akan diserahkan kepada pembaca. Setelah itu, [Inter-Privilege Interrupt Syscall](#) memberikan contoh implementasi. Hasil implementasi diuji dengan [Calling Syscall](#) pada user program.

• Security: User Mode & System Calls

Program user tidak memiliki akses apapun kecuali melakukan komputasi, *the end.*

Apakah implementasi seperti ini aman secara security?

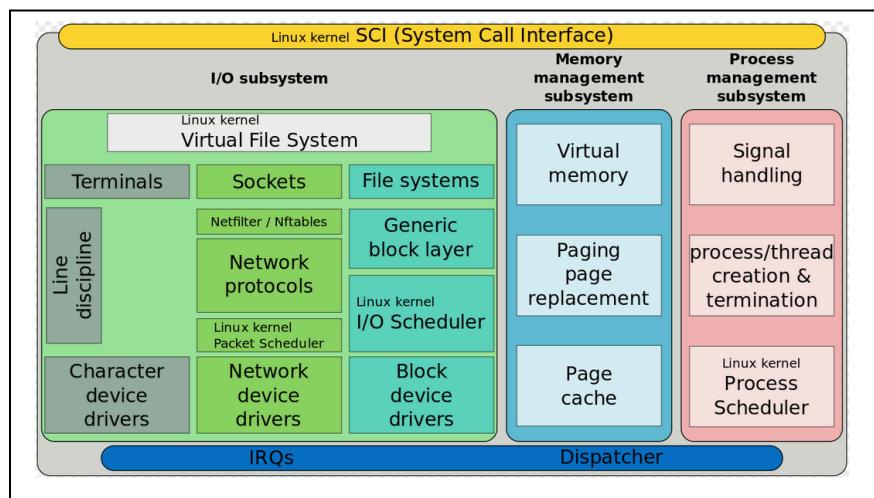
Oh tentu saja, tidak ada data yang masuk dan keluar akan menjamin keamanan informasi.

Apakah user program akan berguna?

Sangat berguna sebagai **pemanas ruangan** yang tidak efisien.

Bagaimana caranya user program memiliki kegunaan selain menjadi pemanas ruangan?

Dengan memberikan akses terbatas ke dunia luar menggunakan **System Calls**



Source: [tc \(linux\) - Wikipedia](#)

System calls adalah **Interface** yang disediakan kernel untuk user program mengendalikan dan menerima data ke I/O hardware. Dengan system calls, user program dapat mengeluarkan hasil komputasi melalui I/O interface yang disediakan sehingga menjadi lebih berguna. Pemanggilan system calls memerlukan input data dari user program untuk melakukan operasinya.

Wajib untuk menganggap **input dari user tidak aman** dan membutuhkan validasi. Implementasi system calls pada sisi kernel wajib melakukan sanitasi input, pemrosesan data, dan berbagai hal lain untuk mencegah user melakukan hal yang tidak diinginkan. Hal ini tidak berbeda jauh ketika mengimplementasikan **Web Frontend** yang menerima data dari user. Perbedaan yang signifikan pada system calls adalah bug dan kesalahan validasi dapat dijadikan malicious actor untuk melakukan **Code Injection** pada kernel space.

Tentunya syscall yang diimplementasikan tidak wajib untuk memperhatikan bagian security dengan detail. Namun tidak ada salahnya berlatih menuliskan kode yang juga memperhatikan keamanan.

3.3.1.1. Designing System Calls

Bagaimana detail perilaku dan interface dari system calls berbeda-beda untuk setiap sistem operasi. Panduan ini akan memberikan contoh desain syscall **UNIX-like** yang dapat digunakan sebagai contoh.

- Syscall dipanggil menggunakan **Inter-Privilege Interrupt** dengan angka **0x30**
- Syscall akan menggunakan **CPU general purpose register** sebagai parameter
 - **eax, ebx, ecx, edx**
- Register **eax** akan digunakan untuk layanan syscall yang diinginkan
- Register **ebx, ecx, edx** digunakan sesuai dengan kebutuhan (dicast ke pointer, etc)

Berikut adalah tabel untuk layanan syscall yang disediakan kernel

Service	eax	Parameter		
		ebx	ecx	edx
FS <code>read()</code>	0	Pointer <code>EXT2DriverRequest</code>	Pointer return code	-
FS <code>read_directory()</code>	1	Pointer request	Pointer return code	-
FS <code>write()</code>	2	Pointer request	Pointer return code	-
FS <code>delete()</code>	3	Pointer request	Pointer return code	-
Keyboard I/O <code>getchar()</code>	4	Pointer char	-	-
Text output <code>putchar()</code>	5	Char value	Text color	-
Text output <code>puts()</code>	6	Pointer char buffer	Char count	Text color
Activate keyboard input	7	-	-	-
...

Catatan: `cpu.stack.esp` & `cpu.stackebp` adalah register kernel ketika dipanggil, bukan user

Sistem operasi yang diimplementasikan tidak perlu untuk mengikuti standar POSIX / UNIX secara penuh. Rancang dan tambahkan sendiri layanan seperti `move_text_cursor()` atau `clear_screen()` pada system calls ini.

3.3.1.2. Inter-Privilege Interrupt Syscall

Mengimplementasikan system calls menggunakan inter-privilege interrupt tidak berbeda jauh dengan menambahkan interrupt handler pada [Interrupt Descriptor Table](#).

Tambahkan entry baru `IDTGate` ke IDT pada `initialize_idt()`. Interrupt syscall yang akan dihandle `int_number` bernilai `0x30`. **Pastikan privilege bernilai `0x3`** agar interrupt number dapat diakses pada user mode. Selain yang disebutkan, nilai `IDTGate` akan sama dengan yang sebelumnya dibuat pada [1.2.4. Load IDT & Testing Interrupt](#).

Tambahkan kode seperti berikut pada `interrupt.c`

```
interrupt.c

void syscall(struct InterruptFrame frame) {
    switch (frame.cpu.general.eax) {
        case 0:
            *((int8_t*) frame.cpu.general.ecx) = read(
                *(struct EXT2DriverRequest*) frame.cpu.general.ebx
            );
            break;
        case 4:
            get_keyboard_buffer((char*) frame.cpu.general.ebx);
            break;
        case 6:
            puts(
                (char*) frame.cpu.general.ebx,
                frame.cpu.general.ecx,
                frame.cpu.general.edx
            ); // Assuming puts() exist in kernel
            break;
        case 7:
            keyboard_state_activate();
            break;
    }
}
```

Implementasikan sendiri untuk `puts()` menggunakan framebuffer yang telah dibuat sebelumnya. Sesuaikan kode diatas dengan kebutuhan atau implementasi yang telah dibuat sebelumnya

3.3.1.3. Calling Syscall

System calls akan dipanggil dengan instruksi assembly `int` pada user mode. Interrupt `0x30` yang dipasang pada [Inter-Privilege Interrupt Syscall](#) akan dipanggil CPU untuk menangani [Software Interrupt](#) yang disebabkan instruksi `int`.

Tambahkan prosedur C `syscall()` dan contoh pemanggilan `syscall read()` berikut

```
user-shell.c

#include <stdint.h>
#include "header/filesystem/ext2.h"

#define BLOCK_COUNT 16

void syscall(uint32_t eax, uint32_t ebx, uint32_t ecx, uint32_t edx) {
    __asm__ volatile("mov %0, %%ebx" : /* <Empty> */ : "r"(ebx));
    __asm__ volatile("mov %0, %%ecx" : /* <Empty> */ : "r"(ecx));
    __asm__ volatile("mov %0, %%edx" : /* <Empty> */ : "r"(edx));
    __asm__ volatile("mov %0, %%eax" : /* <Empty> */ : "r"(eax));
    // Note : gcc usually use %eax as intermediate register,
    //         so it need to be the last one to mov
    __asm__ volatile("int $0x30");
}

int main(void) {
    struct BlockBuffer     bl[2]    = {0};
    struct EXT2DriverRequest request = {
        .buf              = &bl,
        .name             = "shell",
        .inode            = 1,
        .buffer_size      = BLOCK_SIZE * BLOCK_COUNT,
        .name_len          = 5,
    };
    int32_t retcode;
    syscall(0, (uint32_t) &request, (uint32_t) &retcode, 0);
    if (retcode == 0)
        syscall(6, (uint32_t) "owo\n", 4, 0xF);

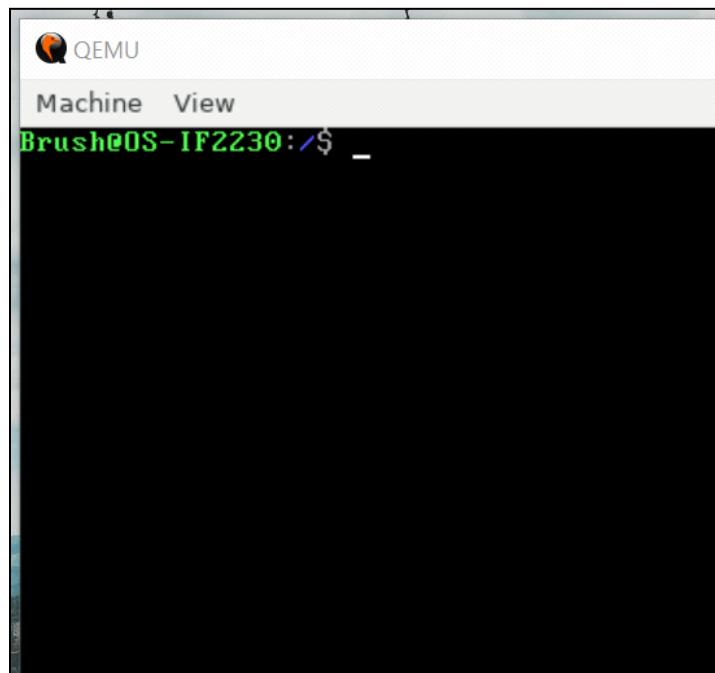
    char buf;
    syscall(7, 0, 0, 0);
    while (true) {
        syscall(4, (uint32_t) &buf, 0, 0);
        syscall(5, (uint32_t) &buf, 0xF, 0);
    }

    return 0;
}
```

Contoh diatas akan memanggil system calls file system `read()` dan memasukkan data kedalam variabel `c1`. Jika implementasi interrupt handler `syscall` sudah benar, shell akan membaca dari storage dan melakukan keyboard input-output secara terus menerus.

3.3.2. Command Line Interface

Chapter 2 akhiri dengan mengimplementasikan **Command Line Interface** yang dapat dioperasikan pengguna sistem operasi. Pengguna sistem operasi akan dapat berinteraksi dengan sistem operasi untuk melakukan beberapa fungsionalitas dasar melalui CLI ini.



Writing command in shell

Berbeda dengan bagian-bagian lainnya yang membutuhkan *presisi tinggi*, shell akan hidup pada user mode sehingga lebih fleksibel. Selain itu bagian ini dapat menggunakan semua subsistem yang telah dibuat pada bagian-bagian sebelumnya untuk membuat fitur-fitur yang diinginkan.

Bagian ini hanya akan memperbaiki [Shell: Debugger](#) agar dapat digunakan untuk melakukan debugging shell menggunakan gdb pada vscode. Setelah itu akan dipaparkan [Shell: Specification](#) dasar yang harus dipenuhi ketika mengimplementasikan shell.

3.3.2.1. Shell: Debugger

Berhubungan dengan [Tips: User Mode](#), shell tidak memiliki debugger symbol sehingga gdb tidak mengetahui apapun tentang executable. Namun permasalahan ini dapat diperbaiki dengan membuat dua executable berbeda: **flat binary** dan [ELF32-i386 executable](#).

Executable ELF32-i386 ini hanya digunakan untuk debugging symbol dari `gcc` dan `nasm`. Nantinya informasi ini akan di-load gdb dan digunakan untuk debugging shell pada dalam sistem operasi. Untuk pengguna lldb, cek instruksi [Apple Silicon/Build Configuration](#).

Edit command pada **makefile** dan tambahkan instruksi launch tambahan pada **launch.json**

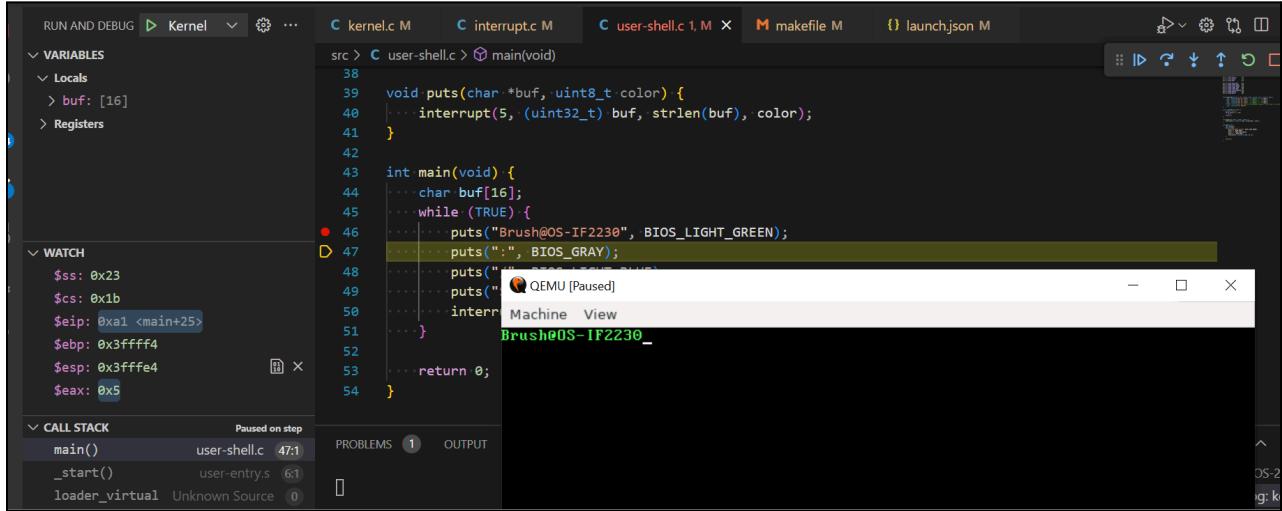
```
makefile

user-shell:
    @$(ASM) $(AFLAGS) $(SOURCE_FOLDER)/crt0.s -o crt0.o
    @$(CC) $(CFLAGS) -fno-pie $(SOURCE_FOLDER)/user-shell.c -o user-shell.o
    @$(CC) $(CFLAGS) -fno-pie $(SOURCE_FOLDER)/stdlib/string.c -o string.o
    @$(LIN) -T $(SOURCE_FOLDER)/user-linker.ld -melf_i386 --oformat=binary \
        crt0.o user-shell.o string.o -o $(OUTPUT_FOLDER)/shell
    @echo Linking object shell object files and generate flat binary...
    @$(LIN) -T $(SOURCE_FOLDER)/user-linker.ld -melf_i386 --oformat=elf32-i386 \
        crt0.o user-shell.o string.o -o $(OUTPUT_FOLDER)/shell_elf
    @echo Linking object shell object files and generate ELF32 for debugging...
    @size --target=binary $(OUTPUT_FOLDER)/shell
    @rm -f *.o
```

```
launch.json

"customLaunchSetupCommands": [
    {
        "text": "target remote localhost:1234",
        "description": "Connect to QEMU remote debugger"
    },
    {
        "text": "symbol-file kernel",
        "description": "Get kernel symbols"
    },
    {
        "text": "add-symbol-file shell_elf",
        "description": "Get shell symbols"
    },
    {
        "text": "set output-radix 16",
        "description": "Use hexadecimal output"
    }
]
```

Tambahkan breakpoint pada **user-shell.c** dan coba jalankan sistem operasi. Jika sudah ditambahkan dengan baik, debugger akan menghentikan QEMU sesuai dengan breakpoint yang diletakkan pada vscode



The screenshot shows the VS Code interface during a debug session. The current file is `user-shell.c`. A red dot indicates a breakpoint is set on line 46. The code on line 46 is:

```
46 puts("Brush@OS-IF2130", BIOS_LIGHT_GREEN);
```

The `VARIABLES` panel shows registers and memory locations. The `WATCH` panel lists variables being monitored. The `CALL STACK` panel shows the current stack trace. The status bar at the bottom indicates "Paused on step".

Breakpoint pada line 46 **user-shell.c**, step over dan tuliskan “**Brush@OS-IF2130**” ke layar

3.3.2.2. Shell: Specification

Spesifikasi dari shell akan disengaja longgar sehingga memberikan ruang untuk membuat fitur tambahan jika menginginkan. Implementasikan shell dengan spesifikasi berikut

Spesifikasi dasar shell

- Shell adalah **sebuah user program yang berjalan pada user mode**
- Shell akan menggunakan **REPL (Read-Eval-Print Loop)**
- Shell menerapkan buffer untuk output (akan sangat membantu untuk *pipelining*)
- Shell mengimplementasikan **Pipeline (Pipeline (Unix) - Wikipedia)**
- Shell menuliskan **Current Working Directory** setiap line
- Awal current working directory terletak pada **Root**
- Shell tidak wajib parse relative pathing (Contohnya: `cd ../folder1/nestedf1/`)

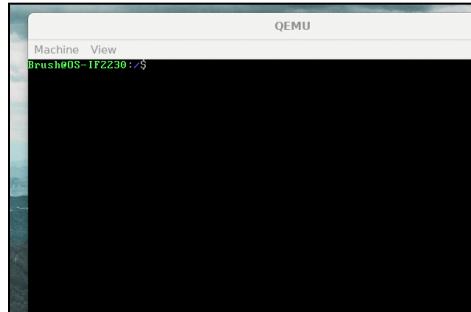
Implementasikan built-in utility command dengan UNIX-like behavior

- **cd** - Mengganti current working directory (termasuk .. untuk naik)
- **ls** - Menuliskan isi current working directory
- **mkdir** - Membuat sebuah folder kosong baru pada current working directory
- **cat** - Menuliskan sebuah file sebagai text file ke layar (Gunakan format LF newline)
- **cp** - Mengcopy suatu file (Folder menjadi bonus)
- **rm** - Menghapus suatu file (Folder menjadi bonus)
- **mv** - Memindah dan merename lokasi file/folder
- **find** - Mencari file/folder dengan nama yang sama **diseluruh file system**
- **grep** - Mencari sebuah pattern dari input berupa file atau keluaran pipeline, lalu menuliskan hasil penemuannya masing-masing pada satu line (regex menjadi bonus)

Fitur lain yang tidak disebutkan (splash screen, UI, utility lain, dll) dibebaskan.

Catatan: Panduan ini mengasumsikan **ukuran executable shell adalah kurang dari 1 MiB**. Mestinya batasan ini lebih dari cukup untuk penggerjaan bagian ini. Diperbolehkan mengganti batas ukuran shell jika dibutuhkan.

Tips: Shell



Never gonna let you down

- Selalu ingat melakukan `make disk` dan `make insert-shell` ketika mengubah kode shell
- Jika membutuhkan `sleep()` untuk fitur tertentu (splash screen, fake loading bar, etc), salah satu trik yang paling mudah adalah gunakan saja **nested loop doing nothing**
- Jika keyboard berhenti berfungsi, **pastikan semua IRQ** (Timer, Keyboard, dll) telah di-ACK
- IF2130 biasanya diambil bersamaan dengan **IF2211 Strategi Algoritma** yang mempelajari graf. Ingat, [EXT2, Graph, Tree](#) memiliki hubungan antara satu sama lain.
- Argumen syscall dapat dicek dengan breakpoint pada fungsi syscall dan menggunakan command berikut pada debug console

```
→ -exec print (struct FAT32DriverRequest) (*cpu.ebx)
$2 = {buf = 0x3ff7e8, name = "ikanaide", ext = "\000\000", parent_cluster_number = 0x2, buffer_size = 0x800}
>
```

- Gunakan cast operator pada **Debug Console** untuk membaca memory sebagai struct

```
code + (struct FAT32DirectoryTable) cl
  ↘ {...}
    ↘ table
      ↘ [0]
        > name
        > ext
          attribute: 0x10
          user_attribute: 0xaa
46 >
```

c1 (ClusterBuffer) di cast ke **DirectoryTable** agar dapat dibaca

- Gunakan juga **gdb print** atau casting untuk menuliskan [Null-Terminated String](#)

```
→ -exec print ((struct FAT32DirectoryTable) cl).table[0].name
$1 = "root\000\000\000"
46 > |
```

c1 (ClusterBuffer) dicast dan dibaca nama dari entry ke 0

```
debugger
→ (char *) buf
> 0x3ff7c1 "nandemonai"
>
```

buf casted into char pointer

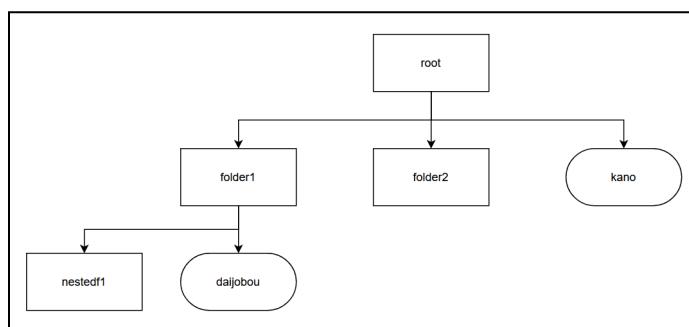
• EXT2, Graph, Tree

Graf G didefinisikan dengan sebuah objek dengan himpunan V yang disebut simpul atau node dan himpunan E yang disebut sisi atau edge. Setiap sisi adalah sebuah himpunan pasangan simpul.

Pohon T didefinisikan sebagai graf yang semua pasangan titik hanya memiliki **tepat 1 jalur** yang menghubungkannya.

Tentunya kedua hal diatas tidak asing setelah mempelajari matematika diskrit dan struktur data. Sering kali graf dan pohon digambarkan dengan sebuah struktur matematis atau data yang secara sengaja dibuat untuk memenuhi definisi. Padahal definisi abstrak ini mencakup banyak hal yang sekilas tidak terlihat seperti graf maupun tree.

Inti dari EXT2 adalah sebuah **Linked List**. Setiap partisi file akan menunjuk ke partisi selanjutnya. Setiap folder akan menunjuk ke file dan folder selain dirinya dan parent. Namun ternyata hanya dengan definisi ini, EXT2 memenuhi kriteria dari sebuah **Tree**



Your friend, Tree, formed by collections and tables of linked list

Karena EXT2 membentuk sebuah tree, semua sifat, algoritma, dan teorema yang ada pada tree dapat diterapkan pada sistem ini. Misalnya untuk mengimplementasikan **find** pada shell dapat digunakan **Depth First Search** atau **Breadth First Search** sederhana untuk mencari file dan folder yang sesuai.

Alasan **EXT2 - IF2130 Edition** meminta entry 0 adalah ke diri sendiri dan entry 1 pointer ke parent adalah untuk mempermudah traversal tree ketika melakukan operasi file system. Entry 0 digunakan sebagai metadata dari node itu sendiri, sedangkan entry 1 akan digunakan ketika ingin melakukan traversal ke parent node.

Hal ini memperlihatkan definisi abstrak graf dan pohon tidak harus dipenuhi dengan struktur data buatan yang sengaja seperti “**BinTree**” dan “**Graph**” pada mata kuliah struktur data. Banyak sekali objek yang dapat memenuhi definisi ini dan kita dapat menerapkan teknik yang telah dipelajari digraf untuk menyelesaikan masalah.

Extra - Ch. 3: Security

• Code Injection

Pernahkah bertanya mengapa exploit buffer overflow pada praktikum IF2130 Organisasi & Arsitektur Komputer dapat bekerja? Alasan utamanya adalah instruksi CPU juga sebuah data pada [von Neumann Architecture](#).

Code Injection adalah manipulasi eksekusi program dengan input data dari penyerang. Buffer overflow yang digunakan pada IF2130 menggunakan fakta bahwa fungsi library C `gets()` tidak melakukan validasi ukuran dari buffer sehingga dapat penyerang dapat memanipulasi return address dan melakukan shellcode code injection.

```
extern void flag();  
  
int main() {  
    char buf[10];  
    gets(buf);  
    return 0;  
}
```

Seperti yang disebutkan juga pada IF2130, terdapat berbagai mekanisme pengamanan dari sisi hardware dan software. Pada sisi hardware, CPU modern menyediakan fitur [NX-bit](#) untuk milarang CPU mengeksekusi sebagian virtual memory. Fitur ini berhubungan dan diimplementasikan pada [Paging](#).

Pada sisi software, sistem operasi dapat menyediakan fitur seperti [Address Space Layout Randomization](#) (ASLR) dan compiler dapat menyediakan [Stack Canary](#) untuk mempersulit code injection melalui buffer overflow.

Tentunya untuk web development, code injection menggunakan buffer overflow C hampir jarang akan terjadi. Namun tidak berarti code injection tidak terjadi pada web development. Pada konteks development web backend terdapat **SQL Injection** dan frontend terdapat **XSS** yang umum untuk diperhatikan.

Sama persis, baik web development atau kernel development, **user input harus di sanitasi**. Setiap request dari user atau external source wajib dilakukan validasi dan memastikan input tidak digunakan secara langsung sebagai kode pada aplikasi.

Istilah code injection biasanya lebih berfokus pada menghentikan injeksi kode melewati user input. Namun program **Malware** juga dapat melakukan *code injection* melewati konsep pada sistem operasi bernama [Hooking](#).

• Hooking

Hooking adalah manipulasi perilaku dari suatu proses atau aplikasi dengan cara intercept pesan, event, pemanggilan fungsi, dan system calls. Hook akan dijalankan terlebih dahulu ketika ada pesan, event, atau fungsi yang seharusnya dipanggil.

Banyak sekali kegunaan dari hook, dari membuat in-game overlay, debugger seperti gdb, menambahkan fitur tambahan pada aplikasi tanpa mengetahui source code, hingga hal malicious seperti membaca keyboard input yang dilakukan Keylogger atau memanipulasi hasil operasi system calls aplikasi lain.



MSI Afterburner In-Game Overlay in Factorio

Tentunya tidak asing dengan **In-Game Overlay** jika pernah bermain game. Overlay memanipulasi graphics pipeline yang dimiliki game dengan hooking. Game atau aplikasi yang membutuhkan operasi grafik kompleks akan menggunakan library seperti **DirectX**, **OpenGL**, atau **Vulkan**. Overlay akan memanipulasi pemanggilan graphics library untuk menggambar overlay terlebih dahulu sebelum diserahkan kepada pemrosesan GPU.

Hooking dilakukan dengan banyak cara seperti

- Modifikasi source code
- Manipulasi shared library (DLL & shared object manipulation)
- Layanan hooking dari syscall ([Win32 Hooks API](#))
- Mengganti dispatch table dari executable ([Import Address Table pada PE format/.exe](#))

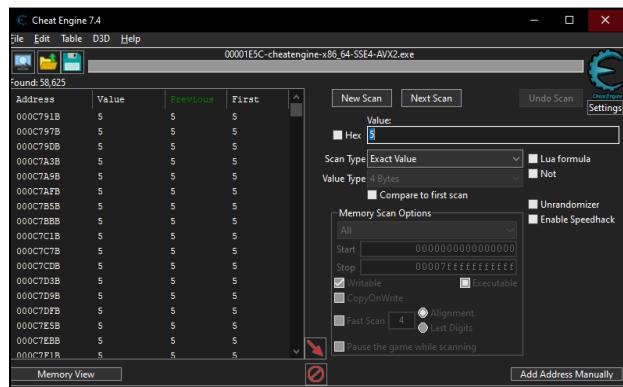
Seperti yang terlihat, metode untuk memanipulasi eksekusi dari sebuah program merupakan hal yang sangat berguna. Namun manipulasi ini menjadi berbahaya jika digunakan oleh aktor yang tidak bertanggung jawab. Oleh karena itu program yang melakukan hooking secara berlebihan mungkin dapat dideteksi oleh program anti-virus sebagai **malware**.

Artikel tambahan terkait In-Game Overlay: fredermott.com/in-game-overlay

● Memory Manipulation

Pernahkah menggunakan [Cheat Engine](#) (CE) untuk melakukan cheat pada game?

Percaya atau tidak, membuat aplikasi game cheat dari scratch membutuhkan pengetahuan banyak terkait komputer terutama memory dan sistem operasi. Aplikasi cheat perlu menggunakan dan memanipulasi OS-specific syscall untuk memory management dan [Hooking](#).



Cheat Engine 7.4

Dasar dari game cheat adalah **memory manipulation**. Aplikasi seperti Cheat Engine didesain untuk mempermudah manipulasi memory dan manipulasi eksekusi. CE menggunakan trik yang dijelaskan pada [Hooking](#) untuk membuat overlay dan manipulasi game tick.

Cheat engine menggunakan banyak trik dan mekanisme untuk melakukan manipulasi process lain. Namun fungsionalitas penuh CE hanya dapat diakses ketika menjalankan sebagai administrator. User Windows yang memiliki status administrator hampir memiliki akses penuh ke sistem layaknya berjalan pada **kernel-mode** meskipun berbeda konsep.

Dengan menggunakan privilege layaknya kernel-mode, CE akan dapat mengakses sembarang proses yang ada pada komputer. Manipulasi atribut pada game dilakukan dengan cara mencari value pada memory misalnya atribut "Health" bernilai 105. Setelah menemukan (virtual) address yang menyimpan atribut ini, CE dapat memanipulasi nilai sesuai yang diinginkan.

Debugger menggunakan mekanisme tidak jauh berbeda dengan cheat engine diatas. Hanya saja debugger memiliki interface yang ditujukan untuk mempermudah pencarian bug dengan menghentikan pada line tertentu source code atau sebagainya.

Layanan memory manipulation dan hooking yang disediakan sistem operasi dapat digunakan untuk operasi yang relatif *benign* seperti diatas. Namun bagaimana jika **malware** menggunakan layanan ini untuk **mencuri akses token dari browser**?

Ch. 4 - Process, Scheduler, Multitasking

Chapter 4 sebagai **Grand Finale** akan mengimplementasikan fitur **Multitasking**. Untuk mencapai fitur ini, Interrupt dan Memory Manager pada chapter sebelumnya akan digunakan lagi dan diperlukan langkah-langkah tambahan seperti menyiapkan **Process** dan **Scheduler**.

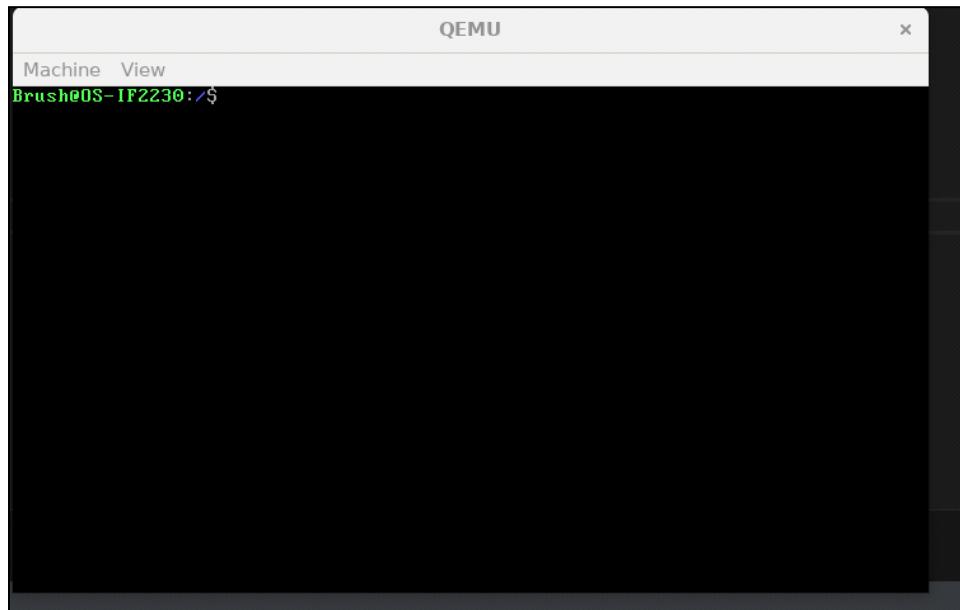
Proses akan diawali dengan pembuatan **Process** yang akan memberikan isolasi resource antar program. **Scheduler** yang akan dibuat melakukan context switch secara periodik dengan algoritma tertentu. Ketika kedua fitur selesai, sistem operasi secara otomatis sudah dapat melakukan **Multitasking**.

Sebelum memasuki Chapter 4, baca dan pahami ulang bagian Paging. Pastikan untuk memahami bagaimana cara melakukan Address Translation dan peran page directory.

Pastikan hasil bagian sebelumnya tidak memiliki banyak bug dan lakukan debug terlebih dahulu sebelum melanjutkan penggeraan **Chapter 4**.

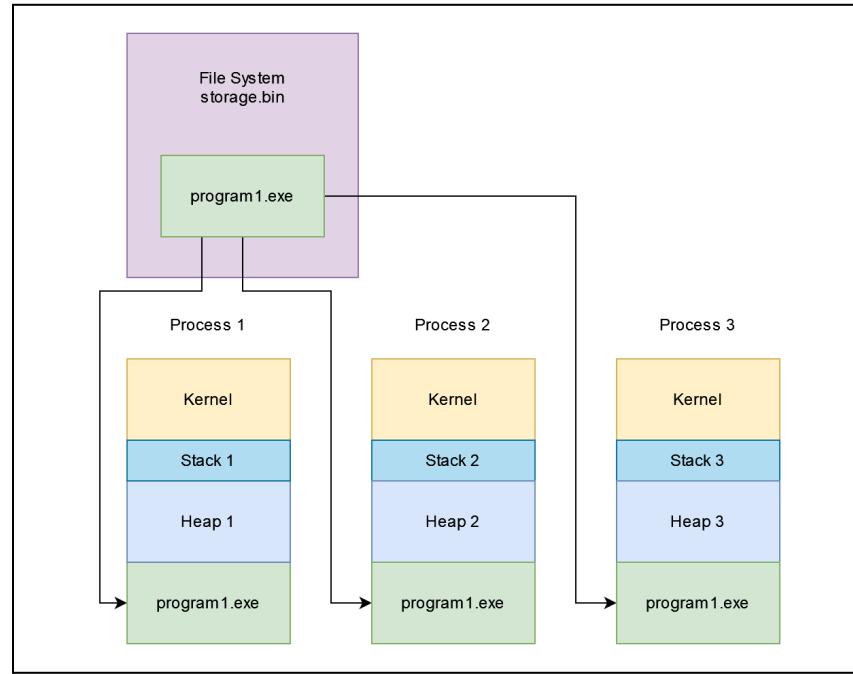
Tingkat kesulitan debugging Chapter 4 akan meningkat secara signifikan karena membutuhkan pemahaman mendalam konsep dan implementasi kode yang ditulis.

Sistem operasi pada akhir **Chapter 4** akan dapat menjalankan banyak program secara konkuren



4.1. Process

Process merupakan konsep yang sangat penting pada sistem operasi dan komputasi modern. **Process** adalah instance program yang dapat dijalankan oleh CPU. Pada sistem operasi modern, process akan dibuat oleh kernel ketika user menjalankan file **Executable Format**.



Selain menyimpan letak instruksi / machine code dari executable format, process juga mencatat resource-resource sistem operasi yang lain seperti networking, file descriptor, lock, status process, handle, dan lain-lain.

Definisi detail dari process akan berbeda dari satu OS ke OS lain. Namun sebagian besar OS modern memiliki kesamaan dengan mengimplementasikan [Process Isolation](#) yang melarang interaksi antar process tanpa operasi [Inter-Process Communication](#) dari kedua process.

Tentunya buku ini hanya memandu untuk membuat process yang sederhana tetapi fungsional untuk mengilustrasikan process yang sebenarnya pada sistem operasi modern.

Sama seperti mayoritas OS modern, bagian ini akan membuat process yang mendukung isolasi memory, membawa informasi [Context](#), dan metadata yang diperlukan agar [Scheduler](#) kernel dapat melakukan [Context Switch](#). Implementasi process akan diuji dengan menjalankan beberapa program pada [Multitasking](#).

4.1.0. Correction: Kit Chapter 1

Terdapat issue konsistensi dan kesalahan penamaan variabel pada kit **Chapter 1** yang dapat berdampak misleading ketika penggeraan **Chapter 4**. Permasalahan terletak pada bagian kit Interrupt sehingga terdapat 2 file yang perlu diganti:

- Assembly `intsetup.s` yang tersedia pada **ch4/correction/intsetup.s**
- Header `interrupt.h`, bagian struktur data `CPURegister`

interrupt.h

```
/**  
 * CPURegister, store CPU registers values.  
 *  
 * @param index    CPU index register (di, si)  
 * @param stack    CPU stack register (bp, sp)  
 * @param general CPU general purpose register (a, b, c, d)  
 * @param segment CPU extra segment register (gs, fs, es, ds)  
 */  
struct CPURegister {  
    struct {  
        uint32_t edi;  
        uint32_t esi;  
    } __attribute__((packed)) index;  
    struct {  
        uint32_t ebp;  
        uint32_t esp;  
    } __attribute__((packed)) stack;  
    struct {  
        uint32_t ebx;  
        uint32_t edx;  
        uint32_t ecx;  
        uint32_t eax;  
    } __attribute__((packed)) general;  
    struct {  
        uint32_t gs;  
        uint32_t fs;  
        uint32_t es;  
        uint32_t ds;  
    } __attribute__((packed)) segment;  
} __attribute__((packed));
```

Ganti kedua bagian yang telah disebutkan dengan file dan definisi struktur data `CPURegister` di atas. Seharusnya sistem operasi tetap dapat berjalan dengan normal setelah mengganti kit. Issue ini hanya mempengaruhi penggunaan `cpu.stack.ebp` dan `cpu.stack.esp`.

4.1.1. Multi Virtual Address Space

Sebelum melanjutkan ke pembuatan process, fitur [Paging](#) yang telah dibuat perlu ditambahkan fitur untuk mendukung banyak virtual address space dengan melakukan operasi pembuatan, penghapusan, dan penggantian page directory aktif.

Tambahkan interface manipulasi page directory berikut pada header paging

```
paging.h

/* --- Process-related Memory Management --- */
#define PAGING_DIRECTORY_TABLE_MAX_COUNT 32

/***
 * Create new page directory prefilled with 1 page directory entry for kernel higher
half mapping
 *
 * @return Pointer to page directory virtual address. Return NULL if allocation failed
 */
struct PageDirectory* paging_create_new_page_directory(void);

/***
 * Free page directory and delete all page directory entry
 *
 * @param page_dir Pointer to page directory virtual address
 * @return True if free operation success
 */
bool paging_free_page_directory(struct PageDirectory *page_dir);

/***
 * Get currently active page directory virtual address from CR3 register
 *
 * @note Assuming page directories lives in kernel memory
 * @return Page directory virtual address currently active (CR3)
 */
struct PageDirectory* paging_get_current_page_directory(void);

/***
 * Change active page directory (indirectly trigger TLB flush for all non-global
entry)
 *
 * @note Assuming page directories lives in kernel memory
 * @param page_dir_virtual_addr Page directory virtual address to switch into
 */
void paging_use_page_directory(struct PageDirectory *page_dir_virtual_addr);
```

Dua definisi dan kerangka dasar dari interface diatas disediakan pada **ch4/1 - Process/paging.c**

Tambahkan kit tersebut pada bagian paging sebelumnya dan lengkapi interface diatas sebelum melanjutkan.

4.1.2. Process Control Block

Process Control Block (PCB) merupakan struktur data yang merepresentasikan sebuah process. Setiap process akan memiliki tepat 1 PCB yang menyimpan berbagai macam informasi terkait process seperti metadata, alokasi memory, dan **Context**.

Berbeda dengan konsep-konsep pada bagian sebelumnya seperti [Global Descriptor Table](#), [Interrupt](#), dan [Paging](#) yang merupakan konsep pada CPU, Process merupakan konsep yang ada pada Sistem Operasi. Oleh karena itu, tidak ada ketetapan wajib bagaimana susunan dan desain struktur data PCB untuk process.

Linux's Process Control Block (Source: elixir.bootlin.com/sched.h)

```
struct task_struct {
    unsigned int          __state;

    /* saved state for "spinlock sleepers" */
    unsigned int          saved_state;

    /*
     * This begins the randomizable portion of task_struct. Only
     * scheduling-critical items should be added above here.
     */
    randomized_struct_fields_start

    void                  *stack;
    refcount_t             usage;
    /* Per task flags (PF_*), defined further below: */
    unsigned int            flags;
    unsigned int            ptrace;

    int                   on_rq;

    int                   prio;
    int                   static_prio;
    int                   normal_prio;
    unsigned int           rt_priority;

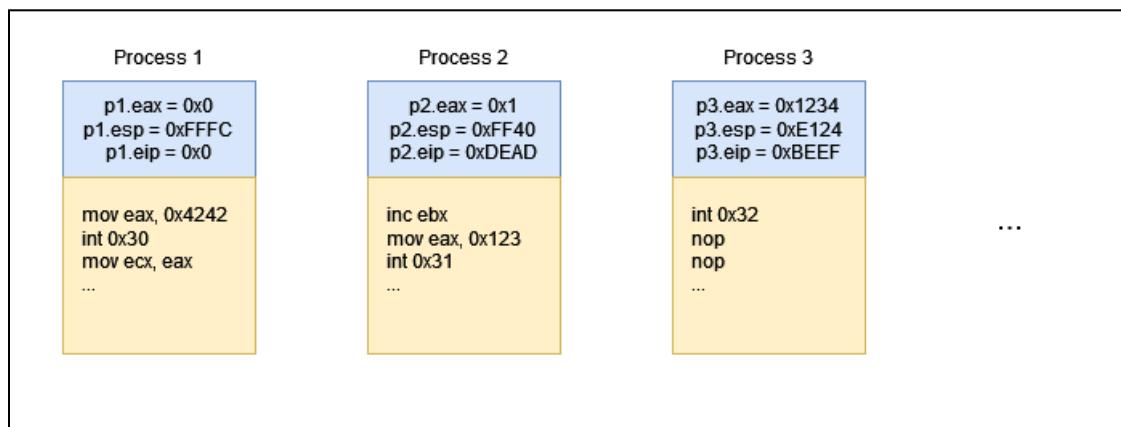
    struct sched_entity      se;
    struct sched_rt_entity    rt;

    ...
}
```

Pada sistem operasi modern, struktur data ini menyimpan berbagai macam informasi dan menjadi cukup kompleks. Kompleksitas ini diperlukan untuk banyak kepentingan seperti scheduler process priority, process exit callback, memory allocation, dan banyak lainnya.

Tentu saja panduan pada buku ini akan mengarahkan struktur data PCB yang minimalis dan sederhana. Namun PCB pada panduan akan tetap fungsional dan menggambarkan bagaimana PCB yang sebenarnya.

Tujuan utama dari bagian **Process** adalah membuat sebuah fitur pada sistem operasi yang memperbolehkan kernel untuk menjalankan banyak program user yang terisolasi dari satu sama lain dan secara konkuren.



Ilustrasi fitur Process yang diinginkan: Register & Memory Isolation

Untuk memenuhi fitur yang diinginkan ini, akan muncul beberapa pertanyaan lain seperti:

- Bagaimana register dan memory antar process berbeda-beda?
- Adakah fitur yang mendukung isolasi register dan memory antar process?
- Apakah perlu menyimpan informasi tambahan untuk fitur ini?

Process perlu menyimpan rangkaian informasi bernama **Process Context** untuk menjawab semua pertanyaan diatas. Fitur process isolation dan konkurensi akan dipenuhi dengan context. Namun context saja tidak cukup dan masih terdapat beberapa pertanyaan lanjutan yang berasal dari bagian **Scheduler**.

- Bagaimana scheduler mengetahui status eksekusi process?
- Apakah terdapat metode untuk identifikasi process?
- Bagaimana cara kernel melepaskan resource yang dimiliki process ketika terminasi?

Dari pertanyaan diatas, status eksekusi dan alokasi resource process juga perlu dicatat oleh kernel. **Process State** menyimpan status eksekusi process yang dibutuhkan scheduler dan **Memory & Metadata** menyimpan informasi alokasi memory dan process metadata lain.

Deklarasi dasar beberapa interface akan disediakan pada **ch4/1 - Process**

4.1.2.1. Process Context

Process Context menyimpan informasi diperlukan untuk melakukan eksekusi sebuah program. Dalam konteks buku ini, yang diperlukan untuk disimpan context adalah **Register** dan **Memory**. Untuk memory, cukup menyimpan address dari page directory yang telah dibuat pada [Multi Virtual Address Space](#).

Buatlah sebuah definisi struktur data **Context** yang menyimpan semua x86 CPU register dan virtual address dari page directory.

Struktur data **CPURegister** yang disediakan pada kit interrupt tidak menyimpan semua CPU register yang ada pada x86 CPU. Beberapa register yang tidak ada tersebut sangat penting untuk operasi context switch nantinya.

Berikut adalah gambaran kasar dari struktur data context

```
/**  
 * Contain information needed for task to be able to get interrupted and resumed later  
 *  
 * @param cpu          All CPU register state  
 * @param eip          CPU instruction counter to resume execution  
 * @param eflags        Flag register to load before resuming the  
 *                      execution  
 * @param page_directory_virtual_addr CPU register CR3, containing pointer to active  
 *                                   page directory  
 */  
struct Context {  
    // TODO: Add important field here  
};
```

4.1.2.2. Process State

Process State utamanya akan digunakan untuk kepentingan [Scheduler](#). Setiap process akan tepat berada pada satu process state yang ada pada sistem operasi. Scheduler akan menggunakan informasi ini didalam algoritma scheduling untuk menentukan process mana yang akan dijalankan selanjutnya.

Jumlah, definisi, dan cara penggunaan process state pada PCB sepenuhnya diatur oleh sistem operasi.

Definisikan process state yang tersedia pada sistem operasi.

Reference implementation menggunakan 3 process state yang cukup untuk kepentingan scheduler sederhana. Gunakan macro atau C enum untuk mendefinisikan process state. Dianjurkan untuk menggunakan code style seperti berikut jika menggunakan C enum.

```
typedef enum PROCESS_STATE {
    // TODO: Add process states
} PROCESS_STATE;
```

4.1.2.3. Memory & Metadata

Untuk mempermudah proses dealokasi memory, PCB dapat ditambahkan struktur data tambahan untuk mencatat virtual memory mana saja dan berapa banyak yang digunakan process. Tambahkan juga informasi process metadata yang penting pada PCB.

Definisikan struktur data **ProcessControlBlock** yang mencakup semua informasi penting tentang process. Satu struktur data **ProcessControlBlock** akan mewakili satu process.

Selain itu deklarasi & definisikan sebuah variabel array statik PCB _process_list berukuran PROCESS_COUNT_MAX.

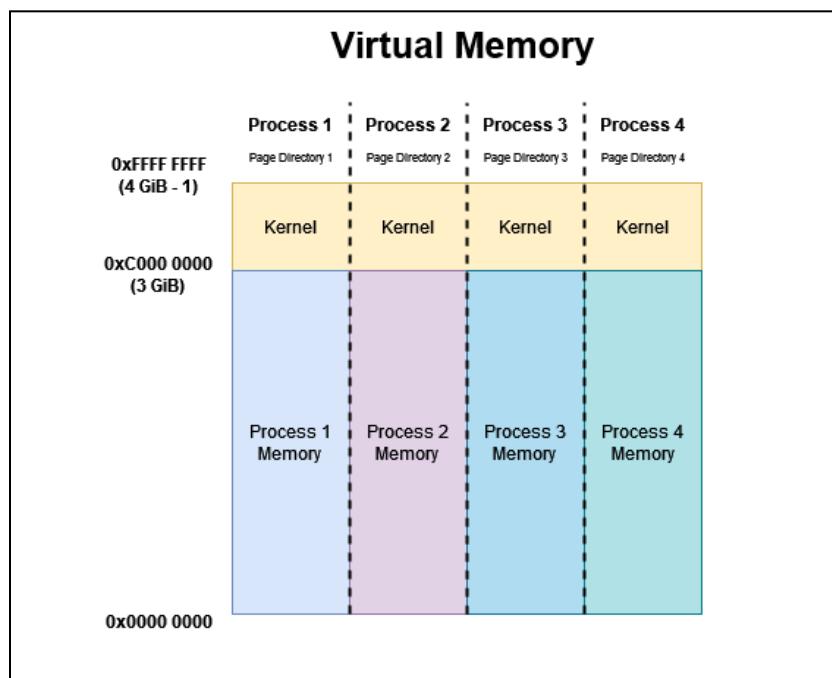
Berikut adalah gambaran kasar untuk struktur data **ProcessControlBlock**

```
/**  
 * Structure data containing information about a process  
 *  
 * @param metadata Process metadata, contain various information about process  
 * @param context Process context used for context saving & switching  
 * @param memory Memory used for the process  
 */  
struct ProcessControlBlock {  
    struct {  
        // ...  
    } metadata;  
  
    struct {  
        void     *virtual_addr_used[PROCESS_PAGE_FRAME_COUNT_MAX];  
        uint32_t page_frame_used_count;  
    } memory;  
};
```

4.1.3. Process Creation

Setelah definisi struktur data [Process Control Block](#) telah dibuat, waktunya menggunakan PCB yang telah dibuat untuk membuat sebuah process. Secara garis besar, pembuatan process user baru akan mengikuti alur berikut:

0. Validasi & pengecekan beberapa kondisi kegagalan
1. Pembuatan virtual address space baru dengan page directory
2. Membaca dan melakukan load executable dari file system ke memory baru
3. Menyiapkan state & context awal untuk program
4. Mencatat semua informasi penting process ke metadata PCB
5. Mengembalikan semua state register dan memory sama sebelum process creation



Multiple process dengan Page Directory sendiri-sendiri

Ilustrasi diatas menggambarkan layout memory untuk masing-masing process yang akan diimplementasikan pada bagian [Virtual Address Space](#). Setiap **user process** akan memiliki **call stack terpisah dan terisolasi antar process**.

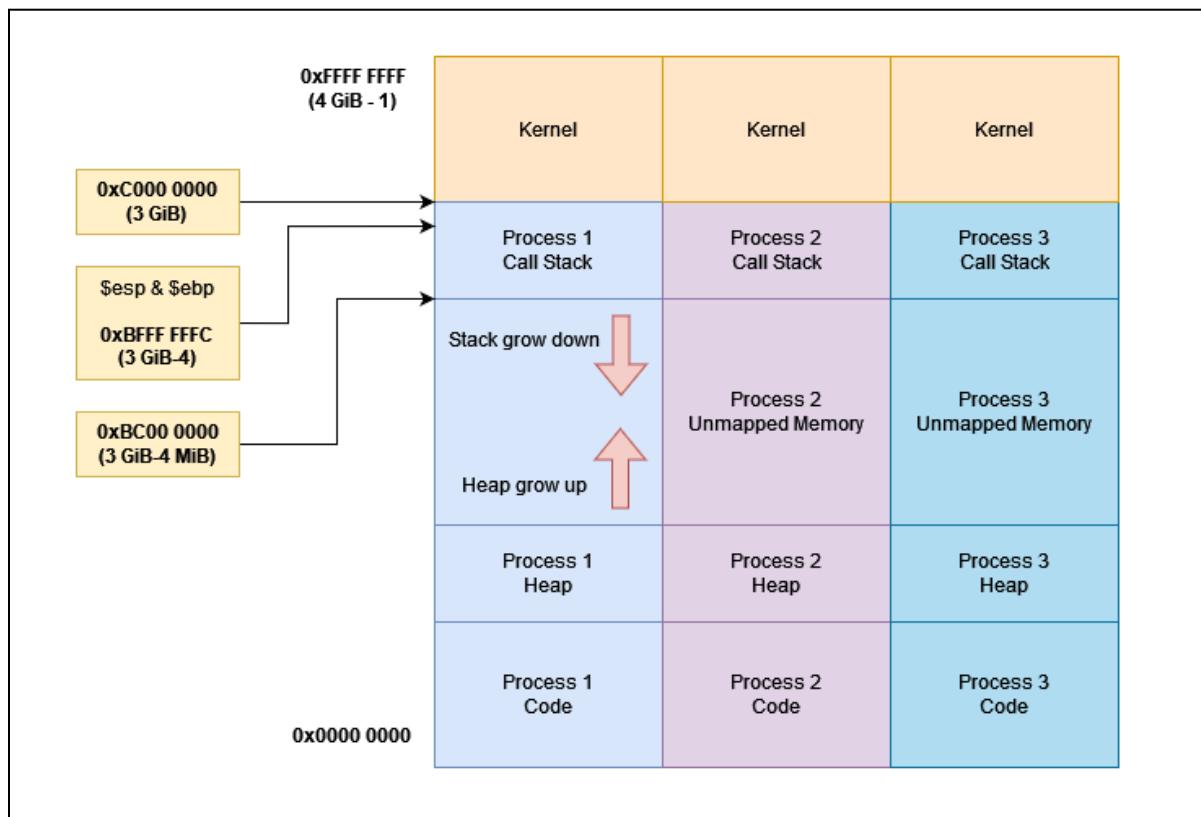
Namun untuk call stack kernel yang digunakan untuk operasi syscall dan interrupt handling, **hanya ada satu kernel call stack pada sistem**. Karena tidak adanya **Reentrancy** pada interrupt handling yang dibuat, kernel call stack yang hanya ada satu tidak menjadi masalah.

Gambaran dasar process creation disediakan pada **ch3/1 - Process/process.c**

4.1.3.1. Virtual Address Space

Dengan menggunakan interface manipulasi page directory yang telah dibuat pada [Multi Virtual Address Space](#), alokasikan virtual memory baru yang dapat digunakan untuk user process. Catat juga setiap alokasi virtual memory ke PCB.

Jika menggunakan linker script yang diberikan panduan [User Mode](#) untuk melakukan linking, user program akan mengasumsikan instruksi program terletak pada lokasi memory **0x0** dan seterusnya. Sesuaikan alokasi virtual memory jika menggunakan layout memory lain.



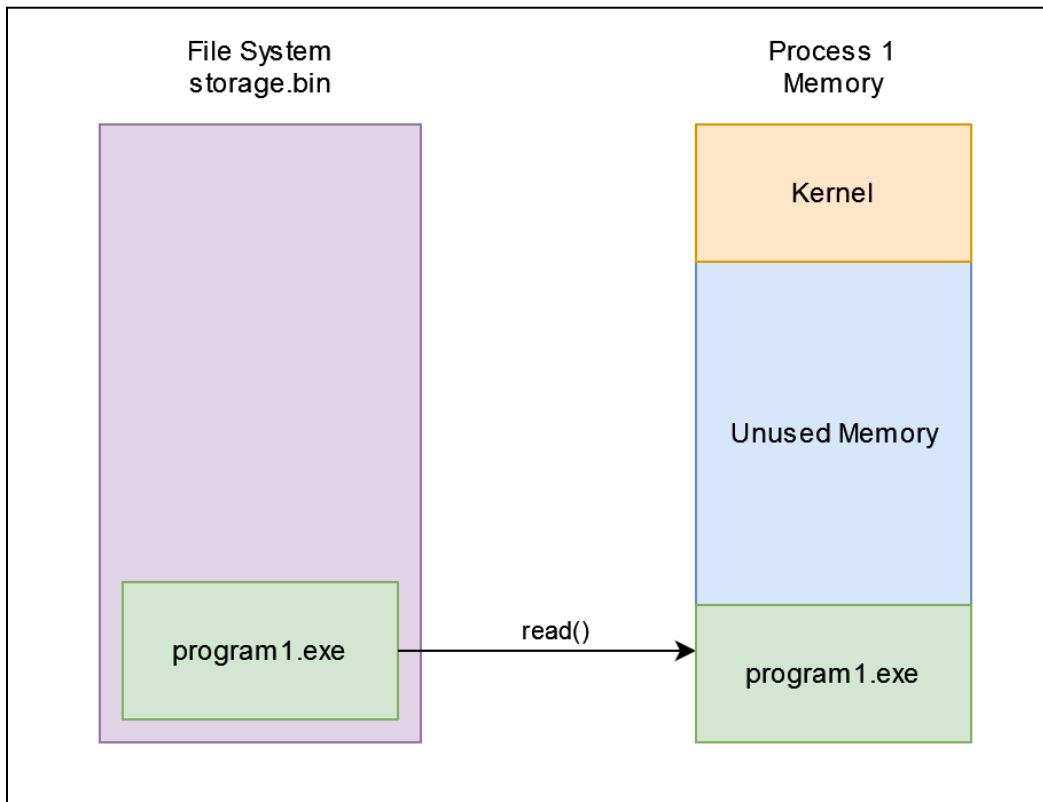
Layout virtual memory beserta user call stack yang akan dibuat

Memory layout diatas tidak jauh berbeda dengan memory layout yang ada pada sistem operasi modern. Call stack bertumbuh ke bawah / address yang lebih kecil dan “Heap” yang bertumbuh ke atas.

4.1.3.2. Load Executable

Setelah virtual memory untuk process telah dialokasikan, untuk sementara ganti page directory ke virtual address space baru dan lakukan pembacaan executable dari file system ke memory. Kembalikan virtual address space ke sebelum proses load executable setelah operasi file system selesai.

Cancel pembuatan process baru jika operasi pembacaan file system gagal. Pengecekan ini penting untuk mencegah process yang rusak atau gagal diinisiasi tetap dijalankan oleh sistem.



Loading executable dari file system ke virtual memory yang baru dialokasikan

4.1.3.3. Context Initialization

Struktur data [Context](#) yang dimiliki oleh setiap process akan digunakan untuk state eksekusi process. Lakukan setup context awal yang akan digunakan process sebagai initial context ketika process pertama kali dijalankan.

Atur semua register sesuai yang dibutuhkan process pertama kali. Segment register seharusnya menggunakan nilai [Segment Selector](#) yang menunjuk ke GDT user data segment descriptor dan memiliki Privilege Level 3. Untuk register eflags, pastikan setidaknya menyalaakan **flag wajib** dan **interrupt flag** yang telah disediakan macronya pada header process.

```
eflags |= CPU_EFLAGS_BASE_FLAG | CPU_EFLAGS_FLAG_INTERRUPT_ENABLE
```

4.1.3.4. Process Metadata & Cleanup

Sisanya adalah melakukan setup semua metadata process sebelum melakukan cleanup dan mengembalikan return code ke function caller. Sesuaikan nilai metadata sesuai struktur yang didefinisikan pada struktur data `ProcessControlBlock`.

Tidak disarankan untuk menggunakan `goto` jika masih belum familiar dengan penggunaan call stack pada C. Keyword ini digunakan sebagian besar hanya untuk error handling & proses cleanup pada fungsi process creation.

4.1.4. Process: Init

Implementasi dari bagian **Process** akan diuji pada bagian ini. **Shell** yang dibuat pada **Chapter 3** akan digunakan diperlukan seperti [init process](#) ala UNIX. Susun kernel setup seperti berikut

```
void kernel_setup(void) {
    load_gdt(&_gdt_gdtr);
    pic_remap();
    initialize_idt();
    activate_keyboard_interrupt();
    framebuffer_clear();
    framebuffer_set_cursor(0, 0);
    initialize_filesystem_ext2();
    gdt_install_tss();
    set_tss_register();

    // Shell request
    struct EXT2DriverRequest request = {
        .buf            = (uint8_t*) 0,
        .name           = "shell",
        .inode          = 1,
        .buffer_size    = 0x100000,
        .name_len       = 5,
    };

    // Set TSS.esp0 for interprivilege interrupt
    set_tss_kernel_current_stack();

    // Create & execute process 0
    process_create_user_process(request);
    paging_use_page_directory(_process_list[0].context.page_directory_virtual_addr);
    kernel_execute_user_program((void*) 0x0);
}
```

Perubahan yang penting pada snippet di highlight dengan warna kuning. Ketiga baris menggunakan fungsi yang dibuat pada [Process Creation](#), [Multi Virtual Address Space](#), dan [User Mode](#). Pastikan kode diatas sesuai dengan alokasi & hasil eksekusi kode yang dapat dicek dengan debugger.

Assembly `kernel_execute_user_program()` yang ada pada `kernel-entrypoint.s` sebenarnya hanya menyiapkan call stack untuk instruksi `iret` yang dibahas pada [x86: Handling Intra & Inter Privilege Interrupt](#) tanpa melakukan penyiapan context. Edit kode assembly dan sesuaikan nilai register stack pointer dengan konvensi yang digunakan pada [Process Creation](#).

Setelah menyesuaikan assembly, harusnya program shell berjalan seperti pada **Chapter 3**.

Metode eksekusi program menggunakan `iret` ini akan digunakan kembali dan menjadi bahasan utama pada bagian [Scheduler](#) dan [Context Switch](#).

Frequent Issue: Process

Pengujian yang ada pada [Process: Init](#) sebenarnya hanya menguji poin-poin berikut:

- Alokasi virtual memory dengan paging
- Penggantian virtual address space dengan page directory berbeda
- Pembacaan executable dari file system ke memory

Sebenarnya ketiga poin diatas tidak terlalu berbeda dengan yang dilakukan pada **Chapter 3**. Informasi tambahan process yang ada pada PCB seperti context dan metadata-nya belum digunakan sama sekali pada titik ini. Informasi-informasi tersebut akan digunakan pada bagian selanjutnya [Scheduler](#), tepatnya [Context Switch](#).

Berikut beberapa issue yang dapat ditemui ketika mencoba bagian [Process: Init](#):

- Mengalami error yang signifikan (page fault, invalid opcode, triple fault, dan sebagainya)

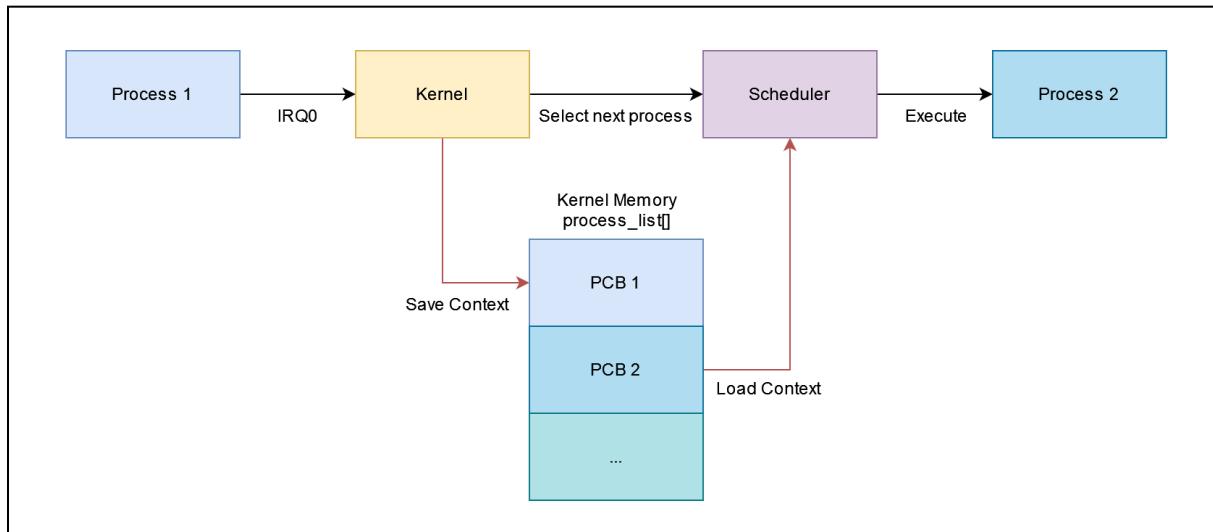
Jika tidak mengalami bug yang sama pada **Chapter 3**, ada kemungkinan operasi alokasi virtual address space atau pembacaan executable memiliki masalah.

- CPU masuk ke address 0x0 & load executable sukses tetapi gagal ketika push / pop

Pastikan instruksi assembly `kernel_execute_user_program()` meletakkan stack pointer di lokasi yang sesuai dengan alokasi [Virtual Address Space](#).

4.2. Scheduler

Scheduler adalah bagian dari kernel yang bertugas untuk mengelola task dan mengalokasikan resource yang ada pada sistem. Dalam konteks penggeraan ini, task adalah **Process** yang dibuat pada bagian sebelumnya dan resource adalah CPU register, satu unit eksekusi CPU, dan virtual memory.



Ilustrasi sederhana context switch oleh scheduler

Karena pada panduan ini hanya terdapat satu **Logical Processor** dan single-threaded process, **Task Scheduler** akan melakukan **Context Switch** secara periodik dengan sangat cepat yang akan memberikan ilusi paralelitas dan **Multitasking**.

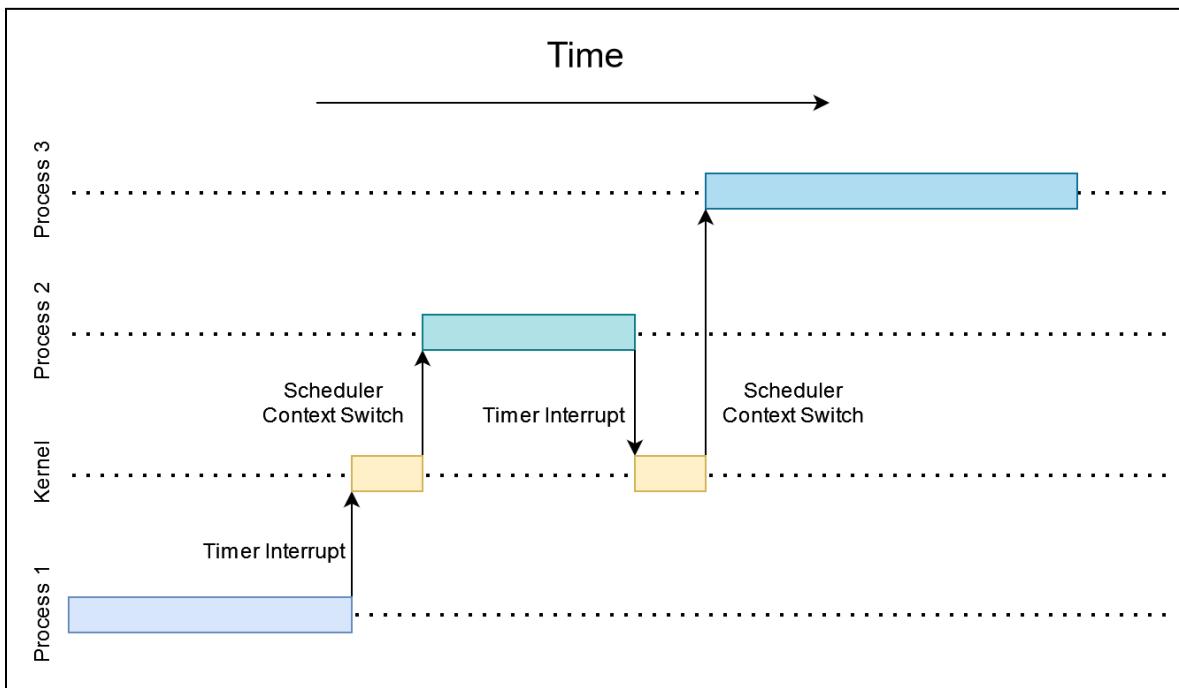
Task Scheduler yang akan dibuat merupakan **Preemptive Scheduler**. Secara periodik menggunakan **IRQ0 - Timer Interrupt**, kernel akan memanggil scheduler untuk memilih process baru yang harus dijalankan. Pemilihan process baru ini bergantung dengan **Scheduling Algorithm** yang diimplementasikan.

Context Switch perlu untuk mengganti CPU register dan harus diimplementasikan dengan instruksi assembly. Operasi ini juga akan menyimpan semua informasi context lama sebelum menggantinya dengan yang baru.

Test: Single Process akan menguji implementasi **Process** dan **Scheduler**. Bagian ini akan menjadi debugging implementasi dan menjadi tantangan terakhir sebelum **Multitasking**. Perlu dicatat bahwa kompleksitas debugging menjadi kompleks dengan context switching dan interrupt yang sangat sensitif dengan delay (termasuk conditional breakpoint & logpoint debugger).

4.2.1. Task Scheduler

Task Scheduler bertugas untuk memilih process yang akan dijalankan selanjutnya. Dengan mengganti eksekusi process terus menerus, kernel akan memenuhi sifat **Concurrency** dan memberikan ilusi **Multitasking**.



Ilustrasi sequence diagram task scheduler

Ilustrasi diatas menggambarkan eksekusi yang diharapkan pada akhir bagian **Scheduler**. Meskipun scheduler dan context switch merupakan operasi yang sangat mahal dan sistem operasi modern banyak melakukan optimisasi untuk mengurangi overhead, panduan ini akan mengimplementasikan **Process** dan **Scheduler** fungsional sesederhana mungkin dan mengorbankan optimisasi demi *readability*.

Karena tipe task scheduler yang diimplementasikan adalah **Preemptive Scheduler**, diperlukan untuk melakukan konfigurasi Programmable Interrupt Controller (PIC) terlebih dahulu agar mengirimkan timer interrupt secara periodik. [IRQ0 - Timer Interrupt & Scheduler Initialization](#) akan mengimplementasikan hal tersebut dan menginisiasi scheduler.

Setelah timer interrupt secara periodik mengirimkan interrupt kepada kernel, scheduler akan menjalankan [Scheduling Algorithm](#) untuk menentukan process baru yang harus dijalankan.

4.2.1.1. IRQ0 - Timer Interrupt & Scheduler Initialization

Timer interrupt akan dinyalakan menggunakan port I/O ke IRQ0. Tambahkan kode berikut untuk melakukan konfigurasi 8259 PIC agar mengirimkan timer interrupt secara periodik pada IRQ0.

```
#define PIT_MAX_FREQUENCY    1193182
#define PIT_TIMER_FREQUENCY  1000
#define PIT_TIMER_COUNTER    (PIT_MAX_FREQUENCY / PIT_TIMER_FREQUENCY)

#define PIT_COMMAND_REGISTER_PIO      0x43
#define PIT_COMMAND_VALUE_BINARY_MODE 0b0
#define PIT_COMMAND_VALUE_OPR_SQUARE_WAVE (0b011 << 1)
#define PIT_COMMAND_VALUE_ACC_LOHIBYTE (0b11 << 4)
#define PIT_COMMAND_VALUE_CHANNEL    (0b00 << 6)
#define PIT_COMMAND_VALUE (PIT_COMMAND_VALUE_BINARY_MODE |
PIT_COMMAND_VALUE_OPR_SQUARE_WAVE | PIT_COMMAND_VALUE_ACC_LOHIBYTE |
PIT_COMMAND_VALUE_CHANNEL)

#define PIT_CHANNEL_0_DATA_PIO 0x40

void activate_timer_interrupt(void) {
    __asm__ volatile("cli");
    // Setup how often PIT fire
    uint32_t pit_timer_counter_to_fire = PIT_TIMER_COUNTER;
    out(PIT_COMMAND_REGISTER_PIO, PIT_COMMAND_VALUE);
    out(PIT_CHANNEL_0_DATA_PIO, (uint8_t) (pit_timer_counter_to_fire & 0xFF));
    out(PIT_CHANNEL_0_DATA_PIO, (uint8_t) ((pit_timer_counter_to_fire >> 8) & 0xFF));

    // Activate the interrupt
    out(PIC1_DATA, in(PIC1_DATA) & ~(1 << IRQ_TIMER));
}
```

Kode diatas akan menggunakan frekuensi default `PIT_TIMER_FREQUENCY` yang bernilai 1000 Hz. Ganti macro tersebut jika ingin menaikkan atau menurunkan frekuensi timer interrupt. Perlu diingat jika frekuensi terlalu tinggi, eksekusi program mungkin akan menjadi lambat.

Sama seperti IRQ lain seperti [IRQ1 - Keyboard Controller](#) dari PIC, pastikan untuk melakukan PIC ACK setiap IRQ0 yang datang dari timer interrupt jika menyalakannya. Jika tidak interrupt lain akan terblokir hingga IRQ0 dikirim ACK.

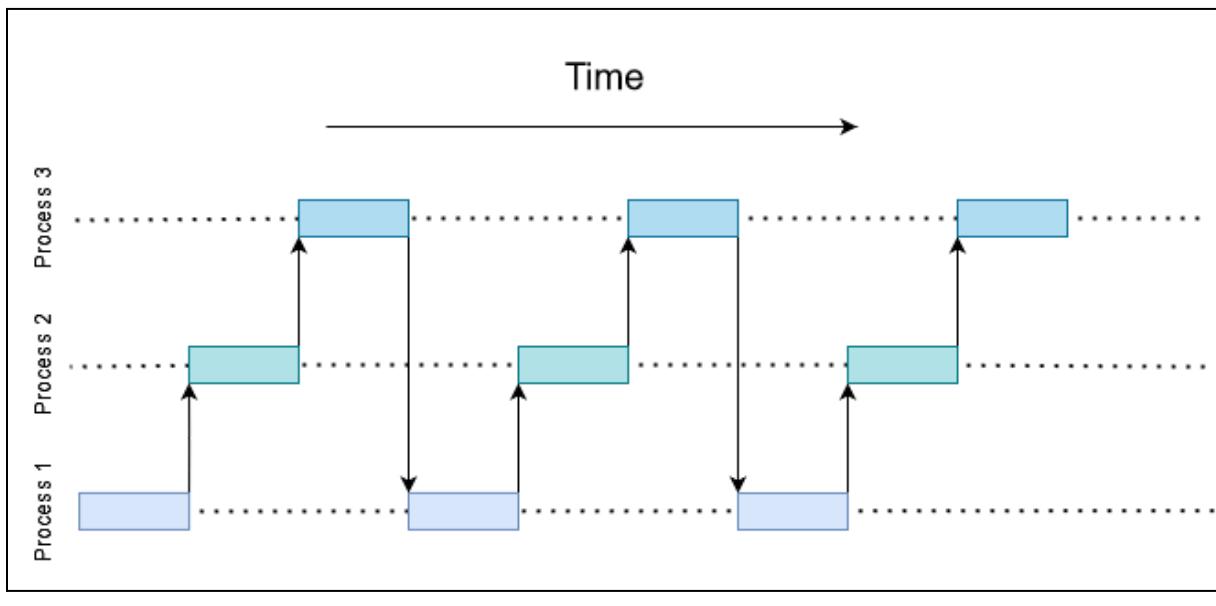
Tambahkan semua inisiasi yang diperlukan scheduler untuk berjalan pada `scheduler_init()` yang nantinya akan dipanggil oleh kernel. Inisiasi tersebut akan dipanggil sekali sebelum “loop” scheduler dijalankan.

4.2.1.2. Scheduling Algorithm

Terdapat berbagai macam **Scheduling Algorithm** yang memiliki kelebihan dan kekurangan masing-masing. Task scheduler yang akan diimplementasikan tentunya membutuhkan setidaknya satu scheduling algorithm yang memilih process selanjutnya yang dijalankan.

Beberapa contoh preemptive scheduling algorithm yang dapat diimplementasikan adalah:

1. Round Robin
2. Priority Scheduling

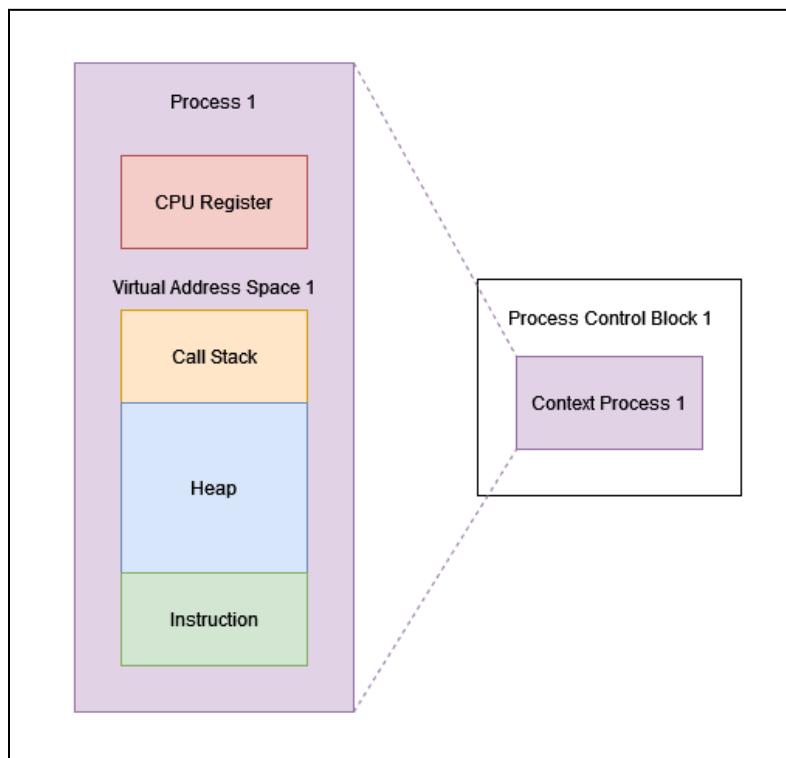


Implementasikan setidaknya satu scheduling algorithm pada task scheduler sistem operasi. Fungsi `scheduler_switch_to_next_process()` akan menggunakan scheduling algorithm untuk memilih dan mengganti process lama dengan yang baru untuk dieksekusi.

Nantinya `scheduler_switch_to_next_process()` akan menggunakan scheduling algorithm dan Context Switch untuk berpindah ke process selanjutnya. Kernel akan memanggil fungsi ini secara periodik dengan timer interrupt sehingga sistem operasi akan berganti process secara periodik, sesuai dengan konsep preemptive scheduler.

4.2.2. Context Switch

Context Switch akan menyimpan dan mengganti semua informasi yang diperlukan untuk menjalankan task yang baru. Penggantian data pada context switch harus tepat dan tidak ada yang terlewat agar eksekusi setiap process tidak hancur.



Ilustrasi context yang harus disimpan ketika context switch

Dari semua bagian **Process** dan **Scheduler**, bagian ini adalah satu-satunya yang memiliki **Portability** yang sangat rendah. Hal ini disebabkan oleh manipulasi CPU register yang sangat dependen dengan arsitektur CPU dan diabstraksikan oleh high-level programming language.

Oleh karena itu, implementasi context switch penggantian [**CPU Register**](#) akan ditulis dalam bahasa assembly x86. Hanya pada bagian ini yang perlu menggunakan assembly, operasi penggantian [**Virtual Address Space & Process**](#) dapat dilakukan pada kode C.

Metode yang akan digunakan pada **Context Switch** ini sama persis dengan yang digunakan **User Mode** `kernel_execute_user_program()` yaitu manipulasi call stack dan instruksi `iret`.

4.2.2.1. CPU Register

Seperti yang ditekankan pada bagian sebelumnya, context switching untuk CPU register membutuhkan assembly x86. **Sangat disarankan untuk** membaca ulang assembly x86, [x86 Calling Convention - cdecl](#), dan [Compiler Explorer - C Calling Convention Example](#).

Fungsi yang akan dibuat pada assembly bernama `process_context_switch()`, function signature tersedia pada header scheduler yang ada pada **ch4/2 - Scheduler/scheduler.h**

Menekankan sekali lagi, operasi ini akan sangat mirip dengan `kernel_execute_user_program()` yang ada pada [User Mode](#) yang menggunakan `iret` dan manipulasi call stack yang digambarkan pada [x86: Handling Intra & Inter Privilege Interrupt](#).

Garis besar yang akan dilakukan `process_context_switch()` adalah:

1. Sebelum melakukan semuanya, simpan base address function argument ctx
2. Lanjutkan dengan setup iret stack dengan push
3. Load semua register dari ctx
4. Cleanup operasi register yang tersisa jika ada
5. Lakukan jump ke process dengan `iret`

Pada bagian ini anda ditugaskan untuk membuat implementasi assembly **context-switch.s**

```
context-switch.s

global process_context_switch

process_context_switch:
    ; TODO : Implement

    iret
```

Tidak langsung berhubungan dengan bagian ini, tetapi ketika menulis assembly perlu menyelesaikan permasalahan yang disembunyikan compiler yaitu [Register Allocation](#). Tidak seperti pada bahasa tingkat tinggi, instruksi assembly hanya dapat memanipulasi register dan memory dengan aturan tertentu.

Operasi seperti menulis ke memory tertentu mungkin membutuhkan intermediate register yang terbatas. Variabel pada bahasa C mungkin disimpan pada CPU register atau memory. Disinilah tantangan untuk menentukan register dan memory mana yang merepresentasikan sebuah variabel.

4.2.2.2. Virtual Address Space & Process

Berbeda dengan [CPU Register](#), untuk mengganti virtual address space cukup untuk mengganti register CR3 yang membawa physical address dari page directory yang akan digunakan. Pada [Process Creation - Load Executable](#) telah melakukan penggantian virtual address space untuk sementara. Operasi yang sama akan digunakan lagi pada bagian ini.

Karena process merupakan konsep pada sistem operasi, definisi pergantian process akan berbeda-beda antar sistem operasi. Sesuaikan protokol pergantian process dengan kebutuhan.

Implementasikan rangkaian pergantian process pada `scheduler_switch_to_next_process()` yang memilih process selanjutnya dan melakukan rangkaian context switch.

Bagian [Test: Single Process](#) akan menguji hasil [Process](#), [Task Scheduler](#), dan [Context Switch](#). Ketiga bagian tersebut akan dirangkai sebagai preemptive task scheduler yang menjalankan single process. [Multitasking](#) akan menambahkan layanan kernel dan program tambahan.

4.2.3. Test: Single Process

Gunakan kernel setup berikut untuk menguji apakah implementasi scheduler dapat melakukan lompatan pertama dan melakukan eksekusi user program secara normal seperti [Chapter 2](#)

kernel.c

```
void kernel_setup(void) {
    load_gdt(&_gdt_gdtr);
    pic_remap();
    initialize_idt();
    activate_keyboard_interrupt();
    framebuffer_clear();
    framebuffer_set_cursor(0, 0);
    initialize_filesystem_ext2();
    gdt_install_tss();
    set_tss_register();

    // Write shell into memory
    struct EXT2DriverRequest request = {
        .buf             = (uint8_t*) 0,
        .name            = "shell",
        .inode           = 1,
        .buffer_size     = 0x100000,
        .name_len        = 5,
    };

    // Set TSS.esp0 for interprivilege interrupt
    set_tss_kernel_current_stack();

    // Create init process and execute it
    process_create_user_process(request);
    scheduler_init();
    scheduler_switch_to_next_process();
}
```

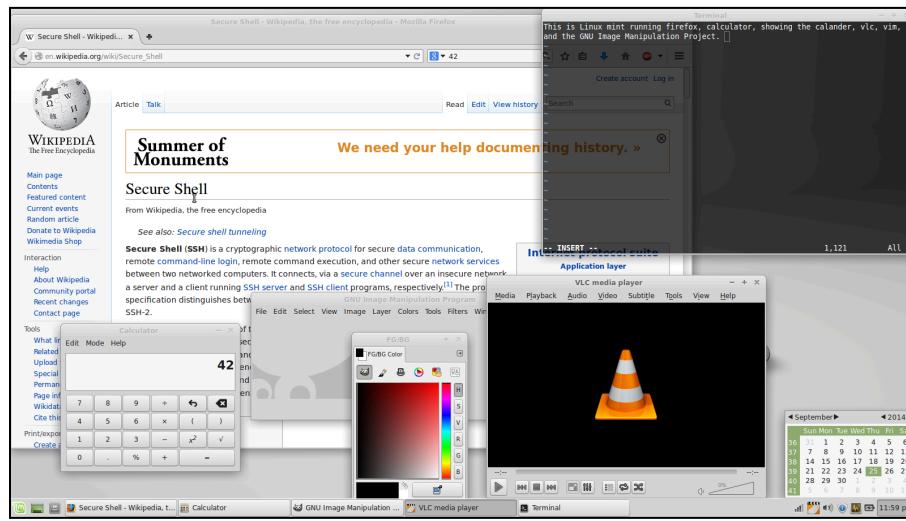
Seharusnya ketika dijalankan shell berjalan normal layaknya [Chapter 3](#). Pada titik ini meskipun timer interrupt berjalan, kernel tidak melakukan apapun setiap kali ada [IRQ0 - Timer Interrupt](#) masuk selain PIC ACK.

Setelah itu, tambahkan operasi context saving & context switching pada interrupt handler IRQ0 agar operasi pergantian process & context switching berjalan setiap kali ada timer interrupt. Jalankan dan biarkan sistem operasi berjalan lebih dari 10 detik, pastikan tidak ada error aneh.

Pastikan bagian ini berjalan normal terlebih dahulu sebelum melanjutkan ke [Multitasking](#). Testing diatas akan menguji apakah implementasi sesuai ekspektasi dan stabilitasnya.

4.3. Multitasking

Multitasking adalah eksekusi banyak task atau process secara konkuren. Semua sistem operasi desktop modern yang banyak digunakan menyediakan fitur multitasking. Pengguna awam mungkin tidak mengetahui bahwa multitasking bukanlah fitur yang *taken for granted*.



Multitasking: Menjalankan banyak program secara bersamaan (Sumber: [Multitasking - Wikipedia](#))

Pada titik ini jika telah mengimplementasikan bagian sebelumnya dengan baik, sebenarnya sistem operasi sudah mendukung untuk multitasking. Fitur Process dan Scheduler cukup untuk memenuhi fitur **Multitasking**.

Hanya saja untuk memperlihatkan fitur multitasking diperlukan beberapa layanan tambahan kernel dan program lain selain shell. Bagian ini menjadi penutup dari rangkaian pengajaran **Chapter 4**.

Melanjutkan dan memodifikasi Simple User Program, bagian Process Entrypoint & Exit akan melengkapi **crt0.s** layaknya program C yang ada pada environment OS modern biasanya.

Process Management & Command akan menambahkan beberapa kernel syscall tambahan untuk melakukan manajemen process. Shell juga akan ditambahkan beberapa command yang menggunakan syscall tersebut untuk memanipulasi process.

Clock akan membuat sebuah program background sederhana menggunakan CMOS dan berjalan di background. Program clock ini akan memperlihatkan sistem operasi dapat berjalan secara konkuren dan melakukan multitasking.

4.3.1. Process Entrypoint & Exit

Seperti yang telah dijelaskan pada **C: Program Entrypoint**, entrypoint dari program C yang sebenarnya adalah symbol `_start` yang didefinisikan pada **crt0.s**. Symbol ini akan menyiapkan environment dan memanggil fungsi C `main()`.

Setelah fungsi `main()` selesai dan mengembalikan return code, eksekusi program dikembalikan ke `_start` yang melakukan cleanup sebelum meminta kernel melakukan terminasi menggunakan system call.

Buatlah syscall exit yang melakukan terminasi proses yang memanggil syscall ini. Handling error code `main()` diserahkan kepada desain system call.

Modifikasi juga **crt0.s** agar memanggil system call ini ketika program C exit.

```
crt0.s

global main

_start:
    call main
    mov ebx, eax
    mov eax, 10    ; Assuming syscall exit is 10
    int 0x30
```

4.3.2. Process Management & Command

Kernel pada titik ini sudah dapat melakukan pembuatan dan manipulasi process. Bagian ini akan menggunakan fitur-fitur kernel tersebut dan meng-expose-nya ke user program dengan system call. Fitur ini akan digunakan untuk mengatur process pada sistem.

Implementasikan syscall-syscall berikut:

- Pembuatan user process baru
- Melakukan terminasi process berdasarkan PID
- Mendapatkan informasi process pada sistem

Setelah itu implementasikan shell command yang menggunakan syscall tersebut:

- **exec** - Menjalankan program yang ada pada filesystem
- **ps** - Menampilkan informasi proses pada sistem
- **kill** - Melakukan terminasi proses berdasarkan pid

Informasi proses setidaknya menampilkan nama process dan identifier. Tidak ada ketetapan untuk cara penulisan dan tampilan informasi ini.

Perlu dicatat bahwa beberapa command di atas meskipun memiliki nama yang sama dengan beberapa command UNIX, behavior yang dideskripsikan mungkin sedikit berbeda. Khusus untuk **exec**, fungsionalitas ini diperbolehkan untuk menggunakan syntax berbeda (relative execution seperti `./a/b/c` atau folder global khusus binary executable ala Linux `/bin/`).

4.3.3. Clock

Clock akan menjadi program pengujian multitasking. Pada dasarnya program ini hanya akan menampilkan jam, menit, dan detik pada interface.



The screenshot shows a terminal window with the title "Machine View". The command "clock" is run, resulting in the output "Clock running..". The terminal window has a dark background and light-colored text. The bottom right corner of the window displays the system time as "23:28:06".

Program clock pada pojok kanan bawah yang dijalankan dengan “clock”

Buatlah sebuah program user seperti shell yang dapat menampilkan informasi waktu pada lokasi layar tertentu. Program ini setidaknya dapat menampilkan jam, menit, dan detik ke layar. Dengan interval update selambat-lambatnya setiap detik.

Implementasikan [driver CMOS](#) terlebih dahulu dan sediakan syscall untuk mengaksesnya dari user program. Kode dari clock sendiri seharusnya tidak terlalu sulit untuk diimplementasikan.

Karena clock adalah user program, seharusnya clock dapat terminasi dan dilakukan exec ulang tanpa merusak berjalannya sistem.

4.3.4. External Application

Aplikasi eksternal adalah program yang berjalan sebagai proses terpisah dari kernel, dieksekusi dalam [User Mode](#), dan berinteraksi dengan kernel melalui [System Call](#). Setelah implementasi [Process Management](#) dan [System Call](#), kita dapat mengembangkan berbagai aplikasi eksternal yang memperluas fungsionalitas sistem operasi tanpa memodifikasi kernel.



Contoh Aplikasi Eksternal berupa gim “snake”

Aplikasi eksternal memanfaatkan berbagai fitur sistem operasi yang telah diimplementasikan sebelumnya seperti process management, paging, filesystem, dan scheduling. Bagian ini akan membahas konsep dan implementasi aplikasi eksternal pada sistem operasi kita.

Implementasikan satu (atau lebih) aplikasi eksternal sederhana yang dapat dijalankan melalui command **exec** dengan kriteria sebagai berikut:

- Aplikasi harus memanfaatkan linker dan entrypoint
- Aplikasi harus menggunakan syscall yang sudah didefinisikan / yang akan dibuat untuk memenuhi kebutuhan aplikasi
- Aplikasi tidak boleh di hardcode ke dalam shell, yang berarti command **exec** harus dapat menerima argumen berupa nama dari aplikasi tersebut
- Setidaknya salah satu aplikasi yang diimplementasikan adalah aplikasi “Hello World!”.

4.3.5. Environment Variables

Agar aplikasi dapat berjalan secara fleksibel di berbagai *device* dan *environment*, sistem operasi menggunakan *environment variables*. *Environment variables* adalah variabel yang disimpan oleh sistem operasi, yang dapat mengubah perilaku program sesuai lingkungan berjalannya tanpa harus memodifikasi kode program.

Salah satu *environment variable* yang sering digunakan adalah **PATH**. Variabel ini berguna untuk menunjukkan lokasi dimana suatu *executable* berada, sehingga untuk menjalankan *executable* tersebut tidak perlu menggunakan *full path*-nya.

Implementasikan mekanisme variabel **PATH** sehingga aplikasi dapat dijalankan dengan nama *file*-nya saja, tanpa perlu menggunakan **exec** atau mem-parsing *command* khusus untuk aplikasi tersebut.

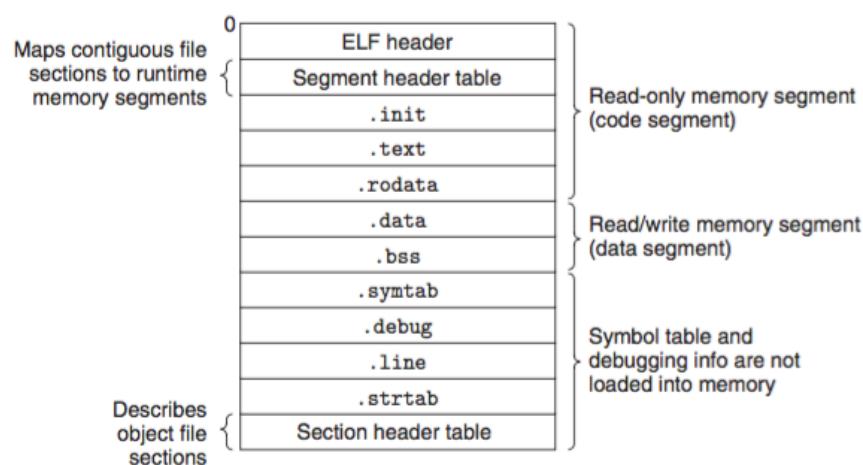
Untuk detail implementasi dibebaskan, namun harus berfungsi sebagai berikut:

- Menyimpan *absolute path* ke satu atau banyak direktori
- Direktori pada path dapat ditambah dan dikurangi melalui sebuah *command* (*read*: [export](#))
- File yang berada di dalam salah satu direktori pada **PATH** dapat dijalankan menggunakan *command* berupa nama dari file tersebut **dari direktori mana pun** pada sistem operasi Misal terdapat file **/bin/clock**, file tersebut dapat dijalankan dengan *command* **clock** dari direktori **/home/user**
- Buat mekanisme sehingga **PATH** bersifat *persistent*, artinya perubahan yang dibuat pada variabel **PATH** akan tersimpan walaupun sistem operasi dimatikan

4.3.6. Separate Shell Commands

Bagian ini akan menjelaskan implementasi *shell* dengan *command* yang terpisah dari *shell* (tidak *built-in*). Tujuan dari adanya *separate shell commands* pada sistem operasi adalah untuk membuat program lebih “modular” dan sistem tidak perlu mengubah *shell* jika ada perubahan, penambahan, atau penghapusan pada suatu *shell commands*.

Pada *real-world cases*, dibutuhkan implementasi [context switch](#) yang baik dan [system call](#) yang sesuai. Agar lebih mudah dalam memisahkan *shell commands*, pastikan bahwa seluruh *commands* yang telah dibuat pada bagian [3.3.2.2. Shell: Specification](#) dapat berjalan dengan baik. Sebagai dasar, berikut adalah struktur sederhana dari [ELF](#) (*Executable and Linkable Format*) structure yang dijelaskan pada ilustrasi berikut.



TLDR (*read: Too Long; Didn't Read*), setiap *executable file* memiliki *metadata*. *Metadata* tersebut dapat berupa *header*, *segment header table*, *read-only memory segment*, *read/write memory segment*, *symbol table*, dan *section header*. Pada bagian ini, buatlah kerangka *executable file* dengan format ELF sebagai struktur dasar dari *separated shell commands*. Gunakan (atau modifikasi) *linker* berikut sesuai dengan kebutuhan *command* yang akan dibuat.

```
linker.ld

ENTRY(_start)

SECTIONS
{
    . = 0x0;

    .text : ALIGN(4K)
    {
        *(.text)
        *(.rodata)
    }
}
```

```

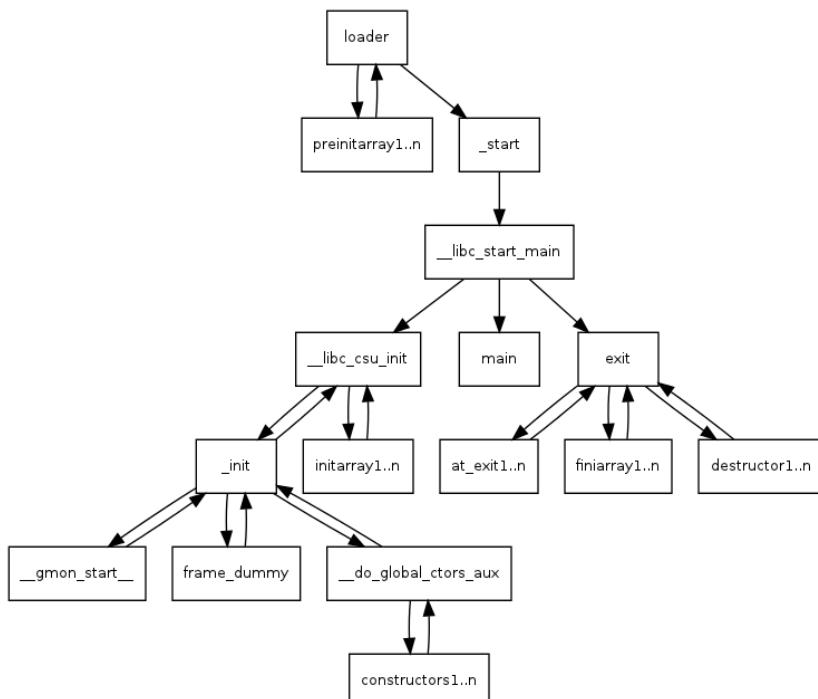
.data : ALIGN(4K)
{
    *(.data)
}

.bss : ALIGN(4K)
{
    *(.bss)
}

/DISCARD/ :
{
    *(.comment)
    *(.note*)
    *(.eh_frame*)
}
}

```

Selain *linker*, buatlah juga [*minimal-startup routine \(crt0\)*](#) agar *command* sudah siap dipakai sebelum *main function* dieksekusi. Pastikan bahwa crt0 yang dibuat sesuai dengan fungsionalitas yang akan dijalankan oleh *commands*. Sebagai ilustrasi, berikut adalah contoh *instructions tree* yang dijalankan sejak *loader* dieksekusi dan juga contoh crt0.s yang dapat digunakan sebagai *baseline* dari *echo command* (boleh dimodifikasi sesuai kebutuhan).



```
crt0-echo.s
```

```
global _start
extern main

section .text
_start:
    ; Kernel stack setup:
    ; ESP -> argc
    ; ESP+4 -> argv[0]
    ; ESP+8 -> argv[1]
    ; ...
    ; ESP+4*(argc+1) -> NULL

    ; Base pointer
    mov ebp, esp

    ; Clear registers
    xor eax, eax
    xor ebx, ebx
    xor ecx, ecx
    xor edx, edx
    xor esi, esi
    xor edi, edi

    ; Call main function
    call main

    ; Exit program (terminate process)
    mov eax, 32          ; Syscall number for process termination
    mov ebx, 0            ; Exit code
    int 0x30

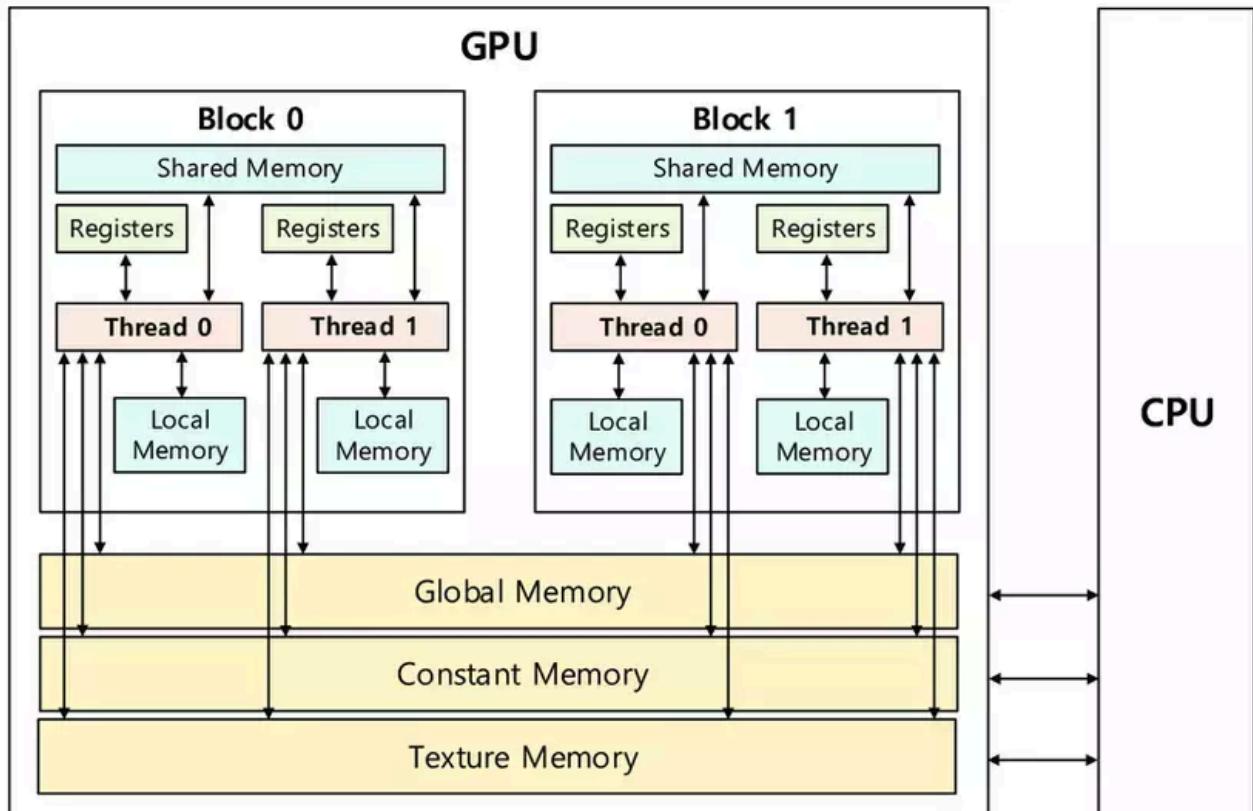
    ; Is it possible to reach here? idk.
    cli
    hlt
```

Setelah semuanya telah ter-setup, pastikan bahwa *commands* tidak menjadi [zombie process](#) atau [orphan process](#) setelah berhasil dieksekusi (sangat disarankan). Hal tersebut bertujuan agar *workload* sistem operasi tidak bertambah dan mencegah terjadinya *unwanted behaviour* dari sistem operasi yang sedang berjalan.

NOTES:

Konsep implementasi dari *separate shell commands* menggunakan hampir semua konsep penting sistem operasi yang telah dijelaskan pada buku ini (*just name it; interrupt, filesystem, shell, system call, process, context switch, etc.*). Pastikan bahwa implementasi [base commands](#) telah berjalan dengan baik. Jika ada *error* yang terjadi, coba untuk melakukan *debug* pada *filesystem* atau *shell*.

Jika *base commands* berjalan dengan baik, tetapi tidak bekerja ketika terpisah dari *shell*, pastikan bahwa implementasi *linker* dan *crt0* sudah tepat. Selain itu, coba untuk melakukan *debug* pada *process* dan juga *consider* untuk membuat *shared-memory* (dibebaskan) dan juga *system call* sebagai *helper* dari *commands* yang ingin diimplementasikan. (pada umumnya terjadi karena *args* tidak terbaca, *request* yang gagal teridentifikasi oleh sistem operasi, atau tidak sinkronnya *memory*)



4.3.7. Grand Finale

Sebagai **Grand Finale** dan menjadi tantangan tersulit, ...

Tambahkan markdown readme pada root repository yang mengandung:

- Nama sistem operasi sebagai H1 (header tingkat 1)
- List kontributor sistem operasi

Selain itu, kreativitas pada readme sangat dianjurkan dan dinantikan oleh asisten. Untuk repository OS-2025, akses admin akan diberikan pada akhir deadline Chapter 4. Diperbolehkan untuk mengganti akses visibilitas dari repository dari private ke public.

It's the fruit of your hard work after all.

Ch. 4 - Extras: Operating System

- **Kernel**

The crux of the book: Apa itu Kernel?

Setelah mengikuti panduan ini, banyak fitur yang telah diimplementasikan

- Global Descriptor Table
- Interrupt Descriptor Table, Handler, PIC
- Framebuffer, Keyboard, Disk, Clock IRQ handler
- File System
- Paging & Memory Manager
- Kernel-User Mode
- Process Control Block & Manager
- Task Scheduler

Sadar atau tidak, fitur-fitur diatas diimplementasikan pada **Kernel** sistem operasi. Definisi dari kernel bermacam-macam antar buku dan sumber, tetapi semuanya memiliki satu ide utama: **Kernel merupakan layer abstraksi software paling rendah.**



Tux, Linux Kernel Mascot

Kernel seperti **Linux** dan yang dibuat pada panduan ini memiliki kewajiban untuk mengontrol & manajemen hardware device. Agar user program dapat melakukan operasi hardware, kernel menyediakan layanan dan operasi melalui **System Calls**.

Dengan menggunakan syscall, program yang berjalan pada user mode dapat menambahkan fitur-fitur sendiri seperti Shell CLI & GUI, Window Manager, Internet Browser, File Explorer, dan lain-lain. Aplikasi-aplikasi user mode ini yang sering digunakan biasanya akan dijadikan satu image dengan instalasi kernel menjadi **Sistem Operasi**.

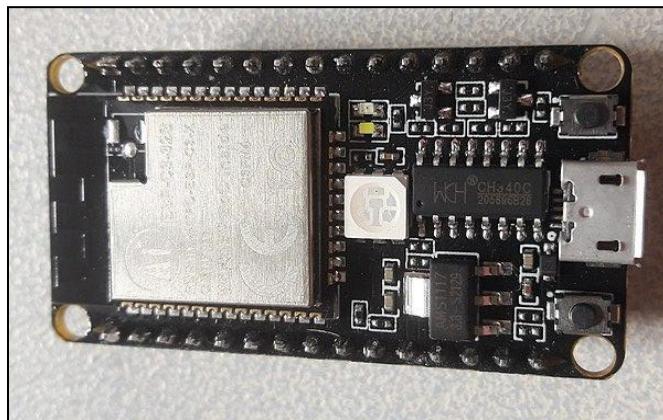
Pada komunitas Linux, terdapat berbagai macam sistem operasi yang menggunakan Linux kernel seperti Ubuntu, Alpine, Arch Linux, Gentoo, dan lain-lain. Sistem operasi yang dibuat menggunakan Linux kernel dan aplikasi-aplikasi tambahan yang disertakan pada instalasi dinamai **Linux Distro**.

Panduan ini membuat sistem operasi yang sangat sederhana dan minimum. Namun konsep-konsep dasar ini tetap digunakan pada OS modern. Jika ingin mempelajari lebih lanjut dan mencari tantangan, cobalah membuat distro Linux sendiri dengan [Linux from Scratch](#).

- **Bare Metal & Embedded System**

Mungkin sudah terlambat, tetapi kernel development pada panduan ini merupakan [Bare Metal](#) programming. Berbeda dengan user program yang dibuat pada abstraksi tingkat atas, program bare metal tidak memiliki layanan sistem operasi atau programming language runtime. Program yang dibuat untuk dijalankan secara langsung pada bare metal disebut **Freestanding Program**.

Pada makefile yang disediakan pada template, terdapat beberapa flag seperti `-nostdlib` dan `-ffreestanding` yang meminta compiler tidak mengasumsikan adanya dan memanggil implementasi C standard library. Flag-flag ini digunakan untuk membentuk sebuah executable yang freestanding dan dapat dijalankan pada bare metal, sesuai dengan kebutuhan **Kernel**.



ESP32 Microcontroller, [Wikipedia - ESP32](#)

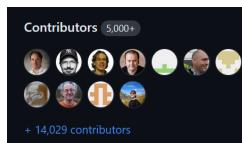
Programmer pada abstraksi tingkat atas hampir tidak akan pernah menyentuh semua hal yang ada pada panduan ini. Sebuah hal yang sangat disayangkan karena konsep-konsep ini digunakan sebagai dasar pada subjek lain seperti **Embedded System** dan **System Programming**.

Sekarang cukup banyak yang tertarik dengan *trending jargon Internet of Things* yang sebenarnya merupakan embedded system. Memang banyak IDE dan development environment yang sudah siap digunakan seperti Arduino IDE, ESP-IDF, dan berbagai macam proprietary software yang disediakan perusahaan pembuat. Namun konser dasar pada bare metal programming tetap penting untuk mengoptimalkan program yang dibuat.

• Operating System

MS-DOS dirilis pada 1981, Mac OS dirilis bersamaan dengan original Macintosh pada 1984, dan Linux dibuat oleh Linus pada 1991. Ketiga sistem operasi dan kernel tersebut berkembang dan masih di-maintain hingga sekarang. Meskipun lebih dari 20 tahun development, ketiga sistem operasi masih dalam status active development untuk improvement dan bug fixes.

Mengulang kalimat pada **Preface**, sistem operasi merupakan sebuah sistem yang kompleks. Buku ini yang memiliki sekitar 200 halaman hanya mencakup sebagian kecil technical detail dari sistem operasi modern. Kompleksitas sistem operasi modern sudah diluar jangkauan dan tidak mungkin dibuat oleh *one-man developer*.



Linux Kernel Repository GitHub - 14k contributors & 1,17M Commits

Sistem operasi modern merupakan hasil kolaborasi dari berbagai macam kontributor individu. Kontribusi tidak hanya dalam bentuk kode, bisa dari review, dokumentasi, legal, standar, dan hal-hal lain yang tidak berhubungan dengan kode. Alangkah baiknya jika penggeraan panduan ini juga dikerjakan dengan **kerjasama tim**.

Memang, membuat sistem operasi modern jauh diluar kapabilitas sekelompok kecil tim maupun organisasi kecil. Namun hal ini bukan menjadi penghalang untuk seseorang **mempelajari dan memahami cara kerja sistem operasi**.

Buku ini ditulis sepenuhnya dengan **tangan** pada 2025 dan didasari alasan **konyol** untuk membuat pembaca tertarik dan termotivasi untuk mempelajari OS.

Secara efektif tidak ada resource dalam bahasa Indonesia yang berani membahas sistem operasi dengan dalam. Hampir semua Perguruan Tinggi di Indonesia hanya malu-malu dan membahas lapisan atas sistem operasi seperti cara menggunakan bash shell tanpa menyentuh inti dari sistem operasi.

Diharapkan dengan adanya buku ini, setidaknya ada resource bahasa Indonesia yang berani menjelaskan sistem operasi secara mendalam dan dapat membantu *subset dari subset* dari penduduk Indonesia yang tertarik dengan sistem operasi.

Epilogue

That's all folks!

Selamat telah menyelesaikan pembuatan sistem operasi sederhana!



200 pages, 32k words, 240k characters

Manually written with care in the era of copy-pasting and LLM noises.

A love letter to OS, low-level programming, and retrocomputing

またいつから会えますように!
また会った時に恥ずかしいからね

So long,
Brush

References

Chapter 0: Toolchain, Kernel, GDT

1. Kitab Intel x86 dan x64 Volume 3A - Intel® 64 and IA-32 Architectures Developer's Manual Volume 3A System Programming Guide
<https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html>

Catatan: Buku ini akan menggunakan **Volume 3A - System Programming Guide Part 1**

Volume lain (1, 2ABCD, 3ABCD, 4) & edisi lengkap (5060 pages) dapat diakses pada link berikut:

<https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>

2. Panduan ini dikembangkan dari dasar buku - <https://littleosbook.github.io>
Catatan penting: Beberapa detail implementasi tidak akurat untuk x86
3. c9x - x86 Instruction Set References
<https://c9x.me/x86/>
4. OSDev - x86 kernel from scratch
https://wiki.osdev.org/Meaty_Skeleton

Chapter 1: Interrupt, Driver

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#scancodesets>
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.asciiitable.com/>
6. https://users.utcluj.ro/~baruch/sie/labor/PS2/Scan_Codes_Set_1.htm

Extras

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

Chapter 2: File System

Filesystem

1. <https://wiki.osdev.org/Ext2>
2. <https://www.nongnu.org/ext2-doc/ext2.html>
3. <https://www.youtube.com/watch?v=ncwn8O70YVg>

Chapter 3: Paging, User Mode, Shell

Paging

1. **Intel Manual Vol 3A - Chapter 4 - Paging**
2. <https://connormcgarr.github.io/paging/>
3. <https://wiki.osdev.org/Paging>
4. https://www.gnu.org/software/grub/manual/multiboot/html_node/multiboot_002eh.html
5. <https://github.com/szhou42/osdev>
6. https://wiki.osdev.org/Memory_management
7. <https://stackoverflow.com/questions/62997536/what-is-the-difference-between-linear-physical-logical-and-virtual-memory-addr>
8. https://wiki.osdev.org/Setting_Up_Paging

User Mode & System Calls

1. **Intel Manual Vol 3A - Chapter 6 - Interrupt and Exception Handling**
2. <https://stackoverflow.com/questions/57926177/what-register-in-i386-stores-the-cpl>
3. https://wiki.osdev.org/Getting_to_Ring_3
4. <https://wiki.osdev.org/TSS>

Shell

1. https://en.wikipedia.org/wiki/Unix_shell
2. <https://man7.org/linux/man-pages/man1/cd.1p.html>
3. <https://man7.org/linux/man-pages/man1/ls.1.html>
4. <https://man7.org/linux/man-pages/man1/mkdir.1.html>
5. <https://man7.org/linux/man-pages/man1/cat.1.html>
6. <https://man7.org/linux/man-pages/man1/cp.1.html>
7. <https://man7.org/linux/man-pages/man1/mv.1.html>
8. <https://man7.org/linux/man-pages/man1/find.1.html>

Extras

1. Manimanimanimananim - <https://github.com/ManimCommunity/manim/>

Chapter 4: Process, Scheduler, Multitasking

Process

1. Intel Manual Vol 3A - Chapter 8 - Task Management
2. <https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-intro.pdf>
3. <https://course.ccs.neu.edu/cs3650/unix-xv6/HTML/S/98.html>
4. https://wiki.osdev.org/Processes_and_Threads
5. [https://en.wikipedia.org/wiki/Process_\(computing\)](https://en.wikipedia.org/wiki/Process_(computing))
6. [https://en.wikipedia.org/wiki/Thread_\(computing\)](https://en.wikipedia.org/wiki/Thread_(computing))

Scheduler

1. Intel Manual Vol 3A - Chapter 8 - Task Management
2. <https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-mechanisms.pdf>
3. <https://course.ccs.neu.edu/cs3650/unix-xv6/HTML/S/66.html>
4. https://wiki.osdev.org/Context_Switching
5. https://en.wikipedia.org/wiki/Context_switch
6. <https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched.pdf>
7. https://wiki.osdev.org/Programmable_Interval_Timer
8. https://wiki.osdev.org/Multitasking_Systems
9. https://wiki.osdev.org/Scheduling_Algorithms
10. <https://www.geeksforgeeks.org/difference-between-preemptive-and-cooperative-multitasking/>
11. https://en.wikipedia.org/wiki/Completely_Fair_Scheduler

Multitasking

1. [https://en.wikipedia.org/wiki/Daemon_\(computing\)](https://en.wikipedia.org/wiki/Daemon_(computing))
2. <https://man7.org/linux/man-pages/man1/ps.1.html>
3. <https://man7.org/linux/man-pages/man1/kill.1.html>
4. <https://man7.org/linux/man-pages/man3/exec.3.html>
5. <https://wiki.osdev.org/CMOS>

Extras

1. **Operating Systems: Three Easy Pieces** - <https://pages.cs.wisc.edu/~remzi/OSTEP/>
2. **Unix-xv6 Source code** - <https://course.ccs.neu.edu/cs3650/unix-xv6/HTML/>

...

Samarsky was just some guy
But he found a neat rock and now he's immortalized
In element number sixty-two
On the periodic table known to me and you
Next to Einstein, Curie, Rutherford
All of the scientific pedigree with excellence assured,
But Samarsky hardly qualifies as a scientist
Yet samarium's his element so add him to the list

...

Samarsky was just some guy
But he found a neat rock and now he's immortalized
In element number sixty-two
On the periodic table known to me and you
Next to Seaborg, Bohr, Copernicus
He did little, but his name will outlive every one of us
For if IUPAC survives until the end of time
Then Samarsky will as well, but is that really such a crime?

...

Samarsky was just some guy
But he found a neat rock and now he's immortalized
In element number sixty-two
On the periodic table known to me and you
Next to Lawrence, Mendeleev, Oganessian
All these others whose achievements they go on and on and on
But the first, the OG, to whom they all must pay respects...
Is Vassily Yevgrafovich Samarsky-Bykovets

[-Lugg's dank music](#)