

# **Tugas Besar**

## **IF2230 - Sistem Operasi**

**Milestone 3**

*"Ho/OS"*

**Pembuatan Sistem Operasi x86  
Paging, User Mode, dan Shell**

Dipersiapkan oleh  
Asisten Lab Sistem Terdistribusi





# Guidebook IF2230 - Milestone 3

Dokumen ini akan digunakan sebagai panduan pengerjaan dan referensi tugas besar IF2230 - 2023. Sama seperti guidebook sebelumnya, dokumen akan menyediakan [Table of Contents](#) dan penjelasan untuk semua pengerjaan tugas besar.

Semua bagian pada milestone 3 terdapat pada guidebook ini, bagian yang menggunakan simbol titik (Contohnya • **MMU & TLB**) di header menandai **bagian tersebut hanya penjelasan bonus** dan beberapa **fun fact** pada sistem operasi sehingga dapat dilompati jika dirasa tidak perlu.

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

Spesifikasi IF2230 - Milestone 3	:  IF2230 - Milestone 3
Repository kit milestone 3	: <a href="#">Sister20/kit-OS-milestone-3-2023</a>
QnA & FAQ	:  IF2230 - FAQ + QnA Tubes
Form asistensi	: <a href="#">Form Asistensi Tugas Besar IF2230 - Sistem Operasi</a>

## Table of Contents

Guidebook IF2230 - Milestone 3.....	2
Table of Contents.....	3
3.1. Paging.....	4
• Paging & Memory.....	4
• MMU & TLB.....	5
• Page Frame Size.....	6
3.1.0. Paging - Overview.....	7
3.1.1. Paging Data Structures.....	9
3.1.2. Higher Half Kernel.....	10
3.1.3. Activate Paging.....	11
Tips Paging.....	12
3.2. User Mode.....	13
3.2.1. External Program - Inserter.....	14
3.2.2. User GDT & Task Segment State.....	15
3.2.3. Simple Memory Allocator.....	17
3.2.4. Simple User Program.....	18
3.2.5. Execute Program.....	20
3.2.6. Launching User Mode.....	21
Tips User Mode.....	22
• Security at CPU-level.....	23
3.3. Shell.....	24
3.3.1. System Calls.....	25
3.3.2. Shell Implementation.....	29
Tips Shell.....	32
Pong!.....	33
• Hot plug & Device Recognition.....	34
• Modern OS.....	35
Referensi Tambahan.....	36

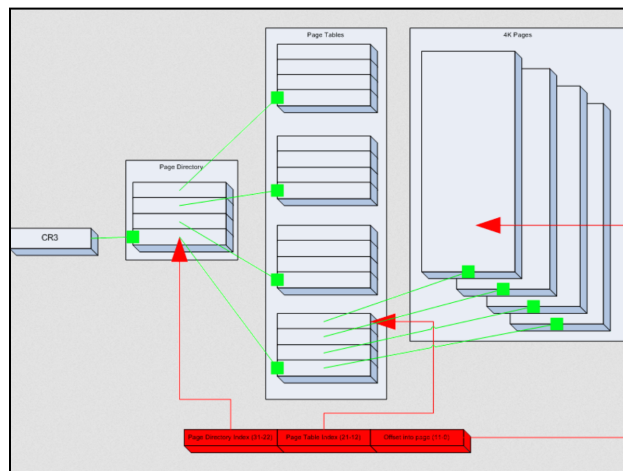
## 3.1. Paging

Sistem operasi yang dibuat akan menggunakan paging sederhana untuk melakukan memory mapping. **Paging yang akan dibuat hanya didesain untuk satu user program dan kernel.**

Page size yang digunakan adalah 4 MiB yang hanya memiliki 1 tingkat tabel: **Page Directory**. Dikelas IF2230 - Sistem Operasi biasanya diajarkan paging dengan 2 tingkat table, **Page Directory & Page Table**

Sebagian besar bagian ini akan dijelaskan dengan detail karena kesalahan setup struktur data paging sering kali menyebabkan **Triple Fault** atau behavior yang aneh.

### • Paging & Memory



Source : [Paging - OSDev](#)

Paging adalah suatu sistem pada arsitektur CPU yang memiliki tujuan untuk mengabstraksikan address. Salah satu contoh kegunaan paging yang paling jelas adalah **Virtual Memory**.

Instruksi pada program A dapat menggunakan (**virtual**) **address yang sama** dengan program B, jika sistem operasi telah menyediakan layanan Virtual Memory dengan baik.

Selain dari virtual memory, paging juga menyediakan cara untuk menandai sebuah block memory dengan privilege level tertentu. Meskipun segmentasi memori telah disediakan GDT, sebagian besar sistem operasi tidak menggunakan segmentasi pada GDT, hanya **Descriptor Privilege Level** GDT yang digunakan. Sehingga umumnya paging telah menjadi *replacement* untuk segmentasi GDT.

Masih berhubungan dengan abstraksi memory, pada tingkat kernel, sebenarnya **physical memory tidak dijamin kontigu**. Physical memory akan bergantung dengan konfigurasi sistem dan **Memory Mapped I/O** yang disebutkan pada milestone 1.

Pada tugas besar ini, digunakan setting default QEMU, memory 128 MiB. Layout dari physical memory dapat dicek dengan GRUB seperti berikut

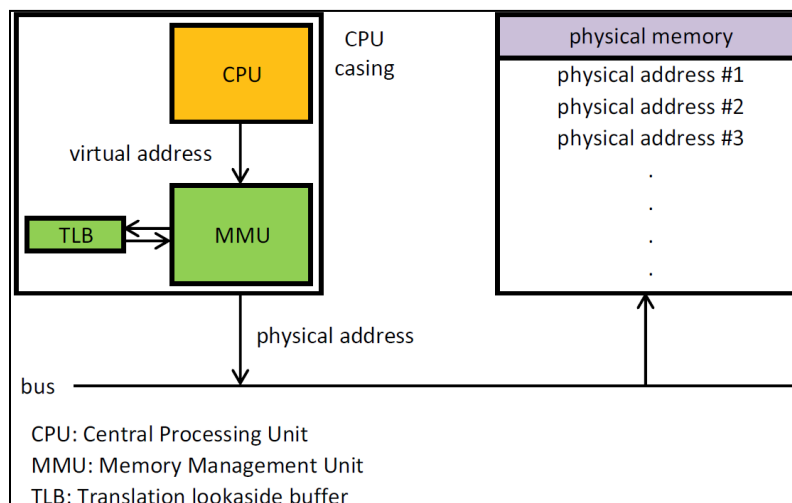
```
grub> displaymem
EISA Memory BIOS Interface is present
Address Map BIOS Interface is present
Lower memory: 639K, Upper memory (to first chipset hole): 129920K
[Address Range Descriptor entries immediately follow (values are 64-
Usable RAM: Base Address: 0x0 X 4GB + 0x0,
Length: 0x0 X 4GB + 0x9fc00 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 + 0xfffc0000,
Length: 0x0 X 4GB + 0x40000 bytes
```

GRUB - **displaymem** command

Perhatikan bahwa ada “lubang” dan “reserved” memory pada tampilan tersebut. Virtual memory mengabstraksikan informasi tersebut dari user program. User program hanya mengetahui virtual memory yang dapat digunakan. Sistem operasi bertugas untuk melakukan mapping virtual address dan physical memory yang valid.

## • MMU & TLB

Seperti yang dijelaskan dikelas, memory dan paging pada x86 akan dikelola oleh **Memory Management Unit (MMU)** dan **Translation Lookaside Buffer (TLB)**.



Source : [MMU - Wikipedia](#)

Proses translasi virtual address dan manajemen **caching tingkat page** (Cache ini berbeda dengan CPU cache yang dipelajari pada **IF2130 - Orkom**) akan dihandle oleh MMU dan TLB.

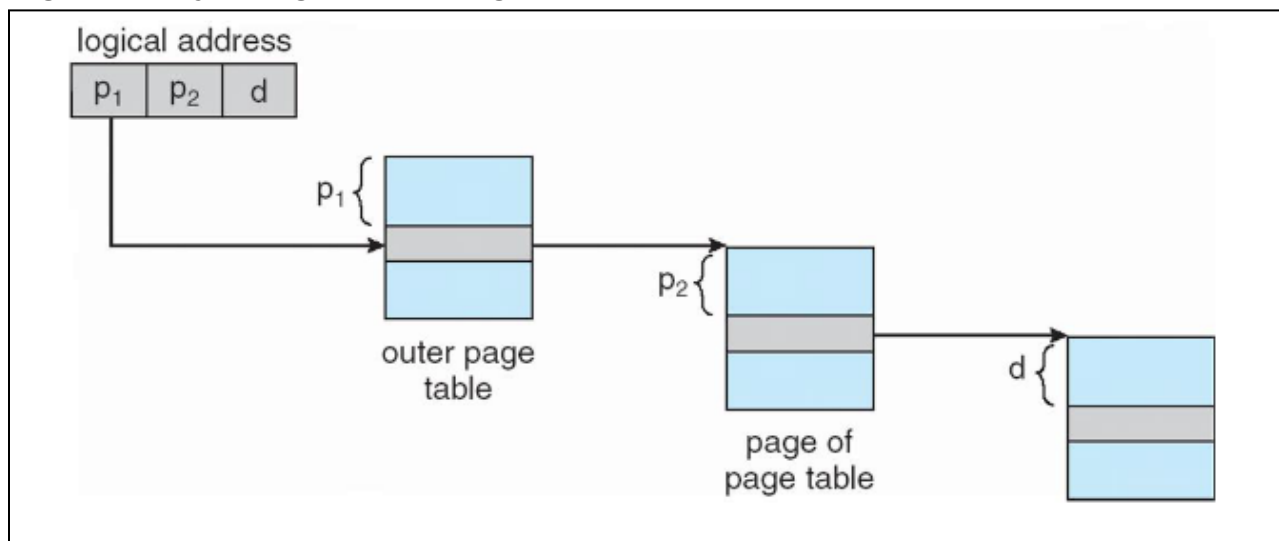
Pada x86 terdapat beberapa instruksi yang dapat digunakan untuk mengirimkan perintah ke TLB dan MMU, salah satunya yang disertakan pada **flush\_single\_tlb()**, instruksi **invlpg** yang menginvalidasi **Page Directory Entry**.

Pada x86, MMU akan melakukan translasi address berdasarkan **Page Directory** yang telah diset pada CPU. CPU menyimpan address ini pada register **CR3**. Tentunya sistem operasi yang dibuat akan menggunakan sistem yang sederhana, hanya satu **Page Directory** yang digunakan untuk sistem.

Pada sistem yang menggunakan **Multitasking** dan **Context Switching**, Page Directory ini dapat diganti oleh kernel dengan Page Directory milik sebuah proses lain. Jadi ketika melakukan pergantian context ke proses lain, kernel akan mengganti Page Directory sesuai dengan process selanjutnya, agar MMU dapat mentranslasikan virtual address ke physical address yang telah dialokasikan ke proses lain tersebut.

## • Page Frame Size

Pada kelas IF2230 - Sistem Operasi, umumnya dijelaskan paging dengan 2-level paging, **Page Directory** → **Page Table** → **Page Frame**



Source : PPT Memory Management - IF2230 - Sistem Operasi

Sedikit berbeda tetapi masih menggunakan prinsip yang sama, tugas besar ini menggunakan **4 MiB page frame size**. Kelebihan dari 4 MiB adalah relatif sederhananya mekanisme pengolahan memory dengan **Virtual Address** ketika ditranslasi akan dibagi menjadi **Page Directory index** dan **memory offset**. Pada protected mode x86, paging dengan 4 MiB hanya membutuhkan Page Directory (outer page table pada gambar) pada register CR3 sehingga **Page Table** tidak diperlukan.

Besarnya page size ini akan cepat menghabiskan memory dari sistem, terutama QEMU secara default hanya menggunakan 128 MiB.

### 3.1.0. Paging - Overview

Sebagian besar bagian paging akan disediakan dari kit. Hal ini disebabkan relatif banyaknya kode assembly dan *requirement precise alignment* ketika membuat fitur paging.

Untuk menghindari kebingungan, pada tugas besar ini akan digunakan dua terminologi :

- **Virtual Address**
- **Physical Address**

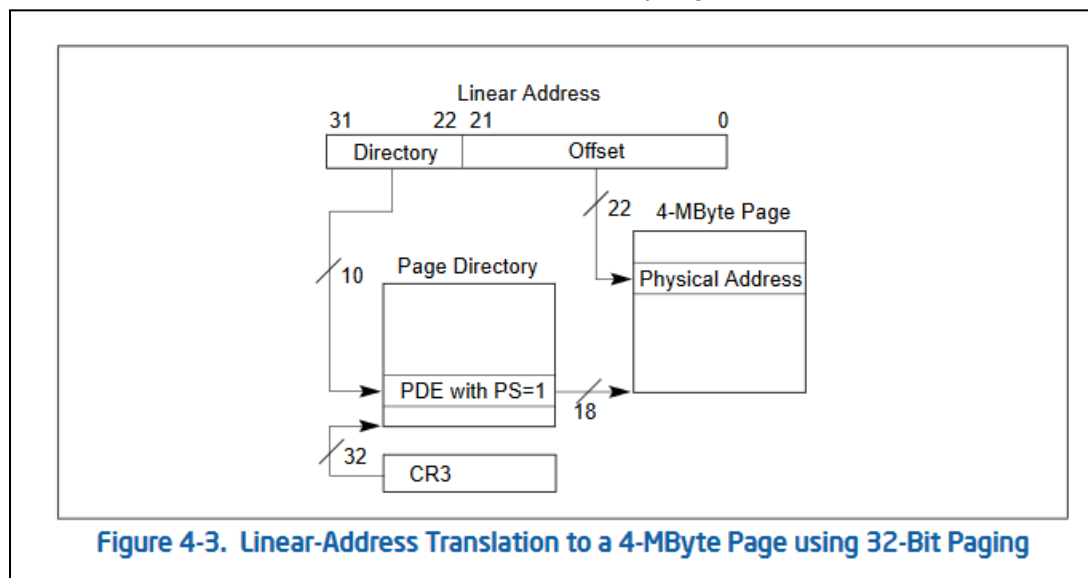
**Paging akan memetakan virtual address ke physical address**, sesuai dengan entry pada Page Directory yang dibaca oleh MMU

**Penting:** Pengetahuan umum bahwa 32-bit hanya dapat mengakses  $2^{32}$  bytes = 4 GB memory, tetapi sebenarnya batasan tersebut umumnya hanya pada **Virtual Memory untuk User Process**. Physical Address dapat lebih besar dari pada 32-bit, bergantung kepada CPU dan Memory Bus

Pada tugas besar ini, akan mengikuti desain memory berikut:

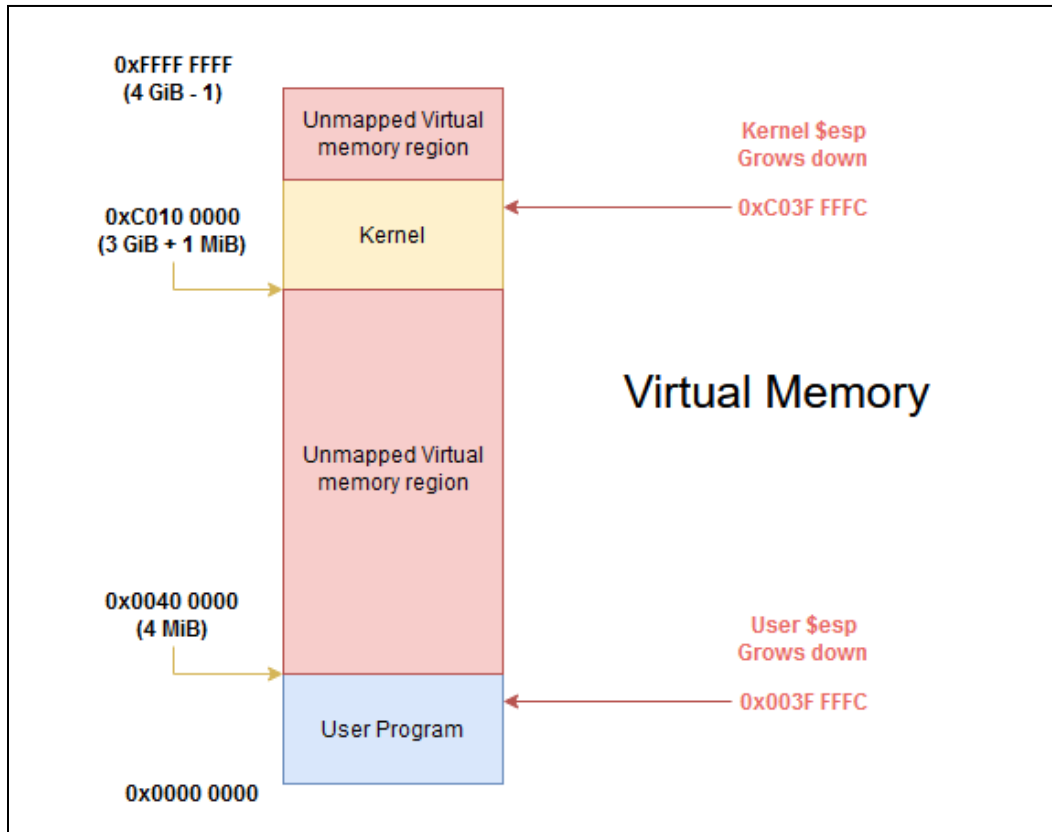
- Fitur segmentasi yang masih ada pada protected mode x86 tidak digunakan
- GDT hanya akan digunakan untuk **descriptor privilege level** dan **TSS**
- **Hanya ada 1 Page Directory** untuk seluruh sistem
- Akan digunakan page frame size **4 MiB**
- Layout memory kernel adalah : **Higher Half Kernel**

Berikut adalah ilustrasi dari Intel Manual untuk translasi yang dilakukan MMU pada 4 MB paging



Intel Manual Vol 3a - Chapter 4 - Paging - 4 MB paging

Meskipun terasa sedikit aneh, bit ke-22 sebenarnya adalah lokasi tepat bit **4 MiB**. Contohnya  $4 * 1024 * 1024 = 0x0040\ 0000 = 0b100\ 0000\ 0000\ 0000\ 0000\ 0000$  (22 nol dibelakang angka 1).  $0x000000 \gg 22$  akan bernilai **0**,  $0x400000 \gg 22$  akan bernilai **1**,  $0x800000 \gg 22$  bernilai **2**, dan seterusnya. Sehingga seperti pada gambar, translasi virtual address ke physical address cukup sederhana. 22 bit diatas dapat langsung di-right shift untuk mendapatkan Page Directory Index.



Sistem operasi yang dibuat akan menggunakan **Higher Half Kernel**, yaitu kernel yang **dipetakan didaerah virtual address paling tinggi**. Lebih tepatnya, kernel akan dipetakan pada virtual address **0xC010 0000** (3 GiB keatas).

Higher half kernel akan membuat virtual memory **0x0000 0000** hingga **0x0040 0000** (atau 4 MiB awal) kosong. Nantinya hal ini akan memudahkan proses menjalankan **user space program**.

Perlu dicatat, meskipun virtual address kernel akan terletak pada **0xC010 0000**, physical address kernel masih terletak pada **0x10 0000** (1 MiB awal). Oleh karena itu, ketika awal sistem operasi berjalan, Page Directory wajib memiliki **Identity Paging** (Paging yang memetakan virtual address ke physical address yang sama) pada kernel.

Identity paging kernel ini (0x0000 0000 hingga 0x0040 0000) hanya akan digunakan sementara ketika sistem operasi dimulai. Nantinya akan dihapus oleh kode assembly yang diberikan pada bagian **Activate Paging**.



### 3.1.1. Paging Data Structures

Bagian ini akan mengimplementasikan deklarasi struktur data yang akan digunakan untuk paging. Template dasar akan disediakan pada **kit/src/paging/**

Implementasikan ketiga struktur data paging berikut

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

Masih sama seperti kedua milestone sebelumnya, definisi struktur data ini dapat dicek pada **Intel Manual Vol 3a - Chapter 4 - Paging**

Pada manual, struktur data ini bernama **PDE: 4MB Page**

## 3.1.2. Higher Half Kernel

Seperti yang dijelaskan pada bagian awal, sistem operasi akan menggunakan **Higher Half Kernel**. Sebenarnya hal yang diperlukan untuk mengganti address kernel adalah mengganti **linker script** yang digunakan **ld**. **Namun bagian ini harus dijalankan dengan paging yang sudah aktif**. Setelah mengganti linker script, lanjutkan bagian selanjutnya untuk menyalakan paging sistem

Update linker script menjadi seperti berikut

### linker.ld

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

/* relocation on address at 0xC001 0000, but load address (physical location) still at 0x100000
*/
SECTIONS {
    . = 0xC0100000;           /* use relocation address (memory references) at 0xC010 0000 */

    /* Optional variable that can be used in kernel, starting address of 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 */
    }

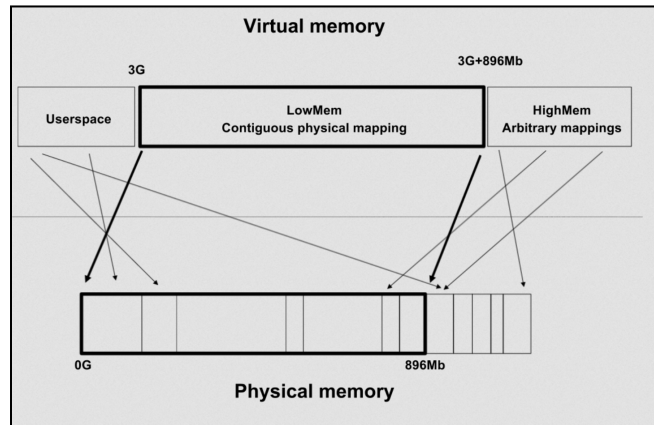
    .data ALIGN (0x1000) : AT (ADDR (.data) - 0xC0000000)
    {
        *(.data)             /* all data sections from all files */
    }

    .bss ALIGN (0x1000) : AT (ADDR (.bss) - 0xC0000000)
    {
        *(COMMON)            /* all COMMON sections from all files */
        *(.bss)              /* all bss sections from all files */
        kernel_loader.o(.bss)
        _linker_kernel_stack_top = .;
    }

    /* Optional variable that can be used in kernel, show end address of kernel */
    _linker_kernel_virtual_addr_end = .;
    _linker_kernel_physical_addr_end = . - 0xC0000000;
}
```

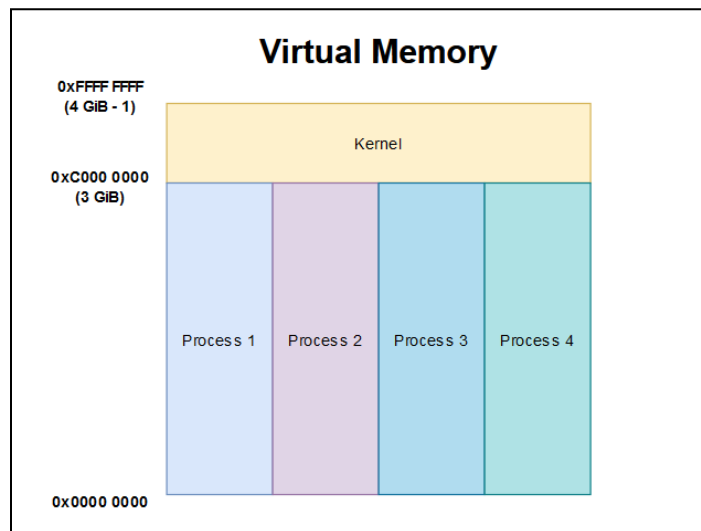
## • Linux & Modern OS Virtual Address Space

Desain Higher Half kernel juga digunakan oleh Linux, Windows, dan sebagian besar OS. Dalam kasus Linux 32-bit, Linux biasanya secara default akan menggunakan **0xC000 0000** hingga **0xFFFF FFFF** untuk kernel, sama seperti OS yang dibuat.



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

Namun pada Linux juga ada fitur untuk melakukan penggantian ukuran offset tersebut yang terdapat pada `/boot/config` dengan pilihan **1 GB User / 3 GB Kernel**, **2 GB User / 2 GB Kernel**, atau default **3 GB User / 1 GB Kernel**. Sedangkan Windows 32-bit menggunakan 2 GB / 2 GB untuk virtual address space. Kedua OS juga menggunakan style yang sama untuk 64-bit, Kernel terletak pada Virtual Memory bagian atas



Ilustrasi : Multiple process dengan Page Directory & Table sendiri-sendiri

Berikut adalah beberapa sumber terkait virtual address space

Windows 32-bit: <https://learn.microsoft.com/en-us/windows/win32/memory/virtual-address-space>

ARM Linux 32-bit: <https://www.arm.linux.org.uk/developer/memory.txt>

### 3.1.3. Activate Paging

Karena addressing akan menjadi sedikit repot jika paging diaktifkan pada kode C, sehingga diperlukan untuk menyalakan paging **sebelum kernel\_setup()**. Sehingga untuk menyalakan paging diperlukan kode assembly.

Sebelumnya sudah ada kode assembly **kernel\_loader.s** yang diimplementasikan pada milestone 1, source file inilah yang nantinya digunakan untuk menyalakan paging.

Ide dasar kode assembly untuk menyalakan paging adalah:

1. Mengatur register CR3 menunjuk ke physical address **\_paging\_kernel\_page\_directory**
2. Menyalakan flag PSE pada register CR4 untuk 4 MB paging
3. Menyalakan paging
4. Melompat ke virtual address kernel
5. Menghapus identity paging kernel
6. Menyiapkan stack dan memanggil kode C **kernel\_setup()**

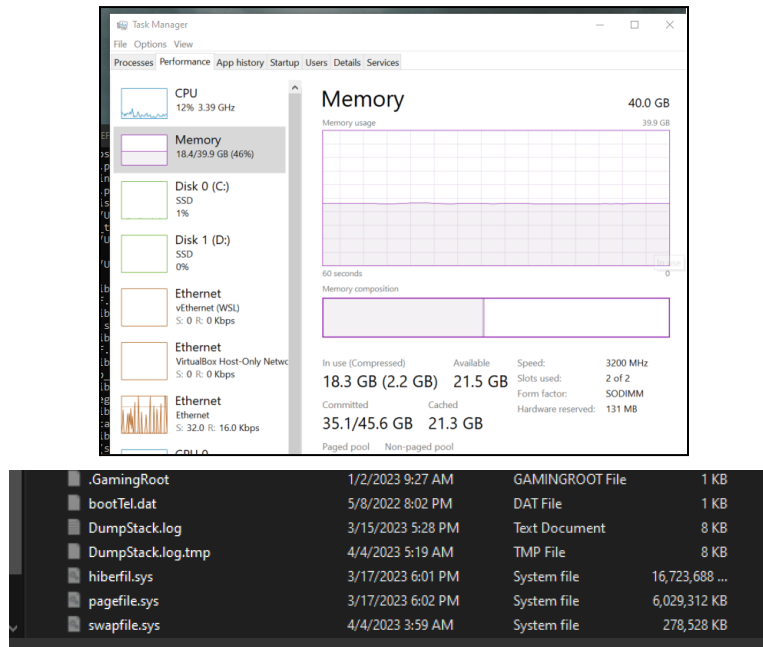
Kode assembly lengkap akan dicantumkan pada **kit/src/paging/kernel\_loader.s**

Jika sudah struktur data paging sudah benar dan paging diaktifkan, semestinya kernel akan berjalan dengan normal seperti sebelumnya. Hanya saja, semua address akan dalam bentuk virtual address dan kode kernel akan mereferensi address dengan awalan **0xC000 0000**, sesuai dengan **Higher Half Kernel**.

Oleh karena itu, **address framebuffer** pada konstanta **MEMORY\_FRAMEBUFFER** yang telah dibuat pada milestone 1 perlu di-update ke **0xC00B 8000**

Jika tidak, operasi framebuffer akan menyebabkan Page Fault karena virtual address **0x0000 0000 - 0x0040 0000** tidak dimap ke physical address manapun. Selain itu, tidak akan ada penulisan apapun ke layar (semestinya hanya terlihat layar booting GRUB saja), karena penulisan ke memory gagal

## • Windows Memory Management & Paging



**Windows 10 64-bit Task Manager - Memory**

Of course, sistem operasi modern seperti Windows juga menggunakan *paging* dan kernel mengimplementasikan modul *memory management*. Seperti yang dijelaskan dikelas, Paging juga dapat digunakan untuk mekanisme **Page File**.

Pernahkan mencoba untuk menjalankan suatu aplikasi yang sangat *memory intensive* dan aplikasi masih berjalan? (meskipun mungkin kinerja sangat buruk) Dengan virtual memory yang disediakan dari paging, kernel dapat handle **CPU Exception Page Fault (0xE)** dan memindahkan memory yang tidak dipakai ke file storage sehingga memory tersebut dapat digunakan untuk proses lain. Dalam kasus Windows, nama file tersebut adalah **pagefile.sys** yang terdapat pada root directory dari setiap partition. **Committed** pada Task manager adalah jumlah dari available physical memory (**RAM**) dan pagefile size (5.5 GB Pagefile + 39.5~ GB RAM + 500~ MB Reserved for OS kernel & modules = 45~ GB)

Operasi penukaran page ke disk-memory tidaklah murah, *in fact*, jika pernah bermain *game* atau *memory intensive app* dan mengalami *stuttering*, salah satu penyebab utama hal tersebut adalah **Page Fault** yang berulang kali yang disebut **Thrashing**. Seperti yang diketahui ketika mendevlop sistem operasi, **Interrupt dan Exception akan menyebabkan eksekusi user program sepenuhnya terhenti** hingga kernel interrupt handler selesai mengerjakan operasinya

Terhentinya eksekusi user program yang terus menerus disebabkan **Page Fault** entah karena **kekurangan available physical memory (RAM)** atau **kernel page swap algorithm** yang kurang baik merupakan penyebab utama **fps stuttering** dalam *game* dan aplikasi

## Tips Paging

- **Higher Half Kernel** dan **Activate Paging** semestinya dikerjakan secara bersamaan
- Jika **Triple Fault** ketika mengaktifkan paging, pastikan alignment 4 KiB terpenuhi
- Debugging untuk tahap ini akan sedikit *tricky*, perubahan relocation address dari **Higher Half Kernel** akan menyebabkan **debugger tidak bekerja dengan baik hingga paging aktif dengan benar**
- Setelah paging aktif, **debugger akan menggunakan virtual address**
- Jika ingin mengetes paging, dapat digunakan **debugger**. Debugger yang juga menggunakan **Virtual Address** juga akan gagal ketika membaca memory yang belum terinisiasi dengan baik
- Cara lain adalah mereferensi langsung address dengan variabel pointer. Akan menyebabkan **Page Fault** jika **Page Frame** masih belum dimap dengan **Physical Address**
- Jika mengalami error seperti berikut

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

Edit `linker.ld` section `.bss`, **kernel\_loader.o(.bss)** menjadi **bin/kernel\_loader.o(.bss)**

- Jika mengalami permasalahan yang aneh, gunakan QnA dan asistensi

## 3.2. User Mode

Sistem operasi yang dibuat hingga bagian ini semestinya sudah siap untuk membuat sebuah program user. Program ini akan berjalan pada **User Mode** atau **Ring 3**. Berbeda dengan kernel yang memiliki akses ke resource, program user akan menggunakan layanan-layanan yang disediakan kernel melewati **System Calls**.

System Calls akan dibahas pada bagian selanjutnya. Bagian ini akan menyiapkan pembuatan executable user program dan mengimplementasikan loader dari user program.

Karena sistem operasi yang dibuat tidak memiliki sistem loading program yang kompleks, user program akan berbentuk **flat binary executable** sehingga dapat dieksekusi langsung dengan “**jump**” sederhana.

### 3.2.1. External Program - Inserter

Sebelumnya pada milestone 2, telah dibuat file system **FAT32 - IF2230 edition**. Jika kode yang diimplementasikan secara **Decoupled** (ala **IF2210 - OOP** dan **IF2250 - RPL**) dengan baik, semestinya bagian ini akan sederhana. Dengan menggunakan hasil implementasi pada milestone 2, kode **fat32.c** dapat di-reuse sebagai inserter file eksternal.

Eksternal disini berarti **program yang berjalan diatas host OS**. Program inserter akan memasukkan file eksternal ke dalam image secondary memory (storage.bin) yang telah dibuat pada milestone 2. Tentunya agar sistem operasi yang dibuat dapat membaca data tersebut, file eksternal juga harus ditulis ke dalam storage dengan cara yang sama pada milestone 2.

Beruntungnya, **fat32.c** semestinya hanya membutuhkan interface disk driver **read\_blocks()** dan **write\_blocks()**, sehingga mudah diganti dengan implementasi **read\_blocks()** dan **write\_blocks()** yang lain.

Kode main driver dari program inserter tersedia pada **kit/inserter/external-inserter.c**

Jika mengerjakan bonus CMOS, buatlah copy **fat32-no-cmos.c** yang distrip fitur CMOS atau tetap menggunakan **fat32.c** dengan implementasi **cmos** direplace dengan library **time.h** yang ada pada **Host OS** (sama seperti r/w blocks). Lagi, jika **cmos** dibuat sedemikian rupa sehingga cukup decoupled, semestinya hanya perlu menghapus/modifikasi beberapa line saja

Tambahkan kode berikut pada makefile

**makefile**

**inserter:**

```
@$(CC) -Wno-builtin-declaration-mismatch -g \
    $(SOURCE_FOLDER)/stdmem.c $(SOURCE_FOLDER)/fat32.c \
    $(SOURCE_FOLDER)/external-inserter.c \
    -o $(OUTPUT_FOLDER)/inserter
```

Program ini dapat dijalankan dengan syntax berikut

```
./inserter <file to insert> <parent cluster index> <storage>
```

Ketika file **<file to insert>** tidak ada, inserter akan membuat sebuah folder dengan nama sama ke file system.

Jika **FAT32 - IF2230 edition** sebelumnya sudah dibuat dengan baik, mestinya bagian ini cukup sederhana. Perbaikilah jika masih ada error pada implementasi **fat32.c**

Nantinya program ini akan digunakan untuk memasukan user program dan file-file lainnya kedalam sistem operasi yang dibuat



Jika mengalami error seperti Segmentation Fault atau yang lain ketika menjalankan inserter, gunakan **Debugger** untuk melakukan debug

Tambahkan konfigurasi bernama **Inserter** untuk debugger VSCode 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
  }
}
```

### 3.2.2. User GDT & Task Segment State

Meskipun segmentasi dari GDT tidak akan digunakan pada sistem operasi ini, GDT masih diperlukan untuk privilege level. Bagian ini akan menambahkan dua entry baru pada GDT untuk **User Mode** dan satu entry baru untuk **Task Segment State**.

**Task Segment State** sendiri merupakan struktur data yang digunakan untuk task switching. Namun tugas besar ini tidak akan menggunakan sebagian besar fitur tersebut, nantinya TSS hanya akan digunakan untuk kepentingan **System Calls**.

Ketika user program melakukan system call melewati Inter Privilege Interrupt, address ISR dapat dicek pada IDT, tetapi IDT tidak menyimpan informasi stack pointer kernel. TSS akan menyimpan informasi ini sehingga ketika ada Inter Privilege Interrupt, address ISR untuk lompat tersedia pada IDT dan TSS menyimpan register **esp** & **ss** milik kernel.

Berikut adalah deklarasi 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** kosong pada **interrupt.c**. Satu entry TSS ini akan digunakan untuk keperluan sistem operasi. **\_interrupt\_tss\_entry** akan di edit lebih lanjut pada bagian **System Calls**. Tambahkan juga implementasi **set\_tss\_kernel\_current\_stack()** berikut

**interrupt.c**

```
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;
}
```

Setelah `_interrupt_tss_entry` sudah ada, sekarang GDT dapat ditambahkan untuk **User Mode descriptor** dan **TSS descriptor**.

Tambahkan 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);
```

Setelah menambahkan deklarasi diatas, tambahkan kode berikut pada `gdt.c`

- 2 entry User GDT, sama persis seperti kernel, hanya saja privilege bernilai 0x3
- Entry ke 5 TSS dan `gdt_install_tss()`

Berikut adalah gambaran untuk `gdt.c` yang baru, sesuaikan dengan implementasi masing-masing

```
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;
}
```

### 3.2.3. Simple Memory Allocator

Sebelum melanjutkan ke pembuatan user program, nantinya program tersebut **harus di-load ke memory** terlebih dahulu sebelum dijalankan. Namun pada titik ini, meskipun kernel sudah memiliki paging, kernel masih belum memiliki fungsi untuk mengalokasikan memory.

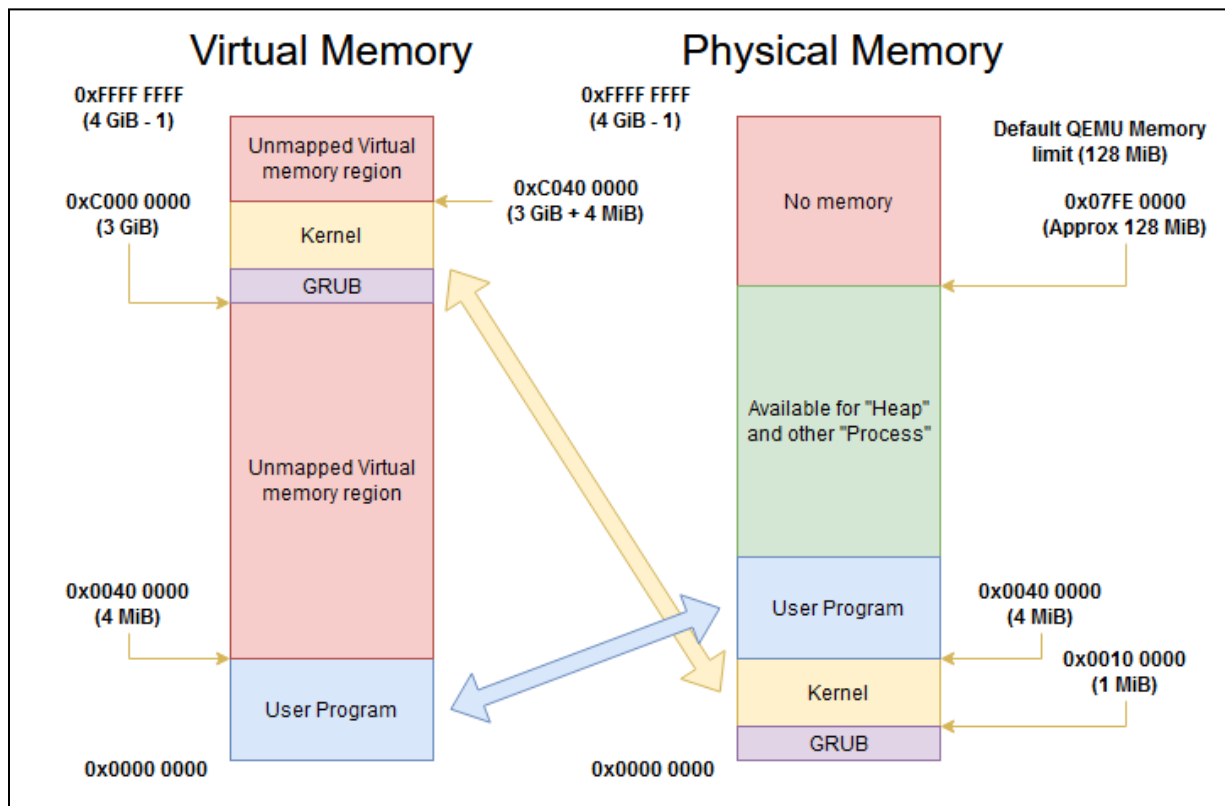
Untuk tugas besar ini, tidak diperlukan fungsi memory allocator yang kompleks (ala **malloc()**), hanya sebuah fungsi sederhana yang mengalokasikan beberapa page frame untuk user

```
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 + 0xfffc0000,
Length: 0x0 X 4GB + 0x40000 bytes
```

GRUB - displaymem

Berdasarkan command **displaymem** pada GRUB diatas, terlihat bahwa **usable RAM** atau **physical memory** yang dapat digunakan dimulai dari **0x10 0000** (1 MiB) hingga **0x10 0000 + 0x07EE 0000 = 0x07FE 0000**.

Berikut adalah rencana layout memory yang akan digunakan untuk user program



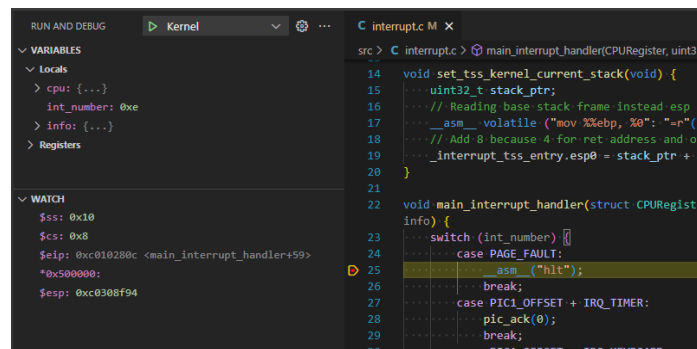
Dengan menggunakan informasi tersebut, selesaikan implementasi `allocate_single_user_page_frame()` pada `paging.c` dengan menggunakan `update_page_directory_entry()` yang sudah didefinisikan

Metode alokasi dan pencatatan physical memory yang terisi dibebaskan. Flag yang bernilai 1 adalah `present_bit`, `write_bit`, `user_bit`, dan `pagesize_4_mb` (sesuaikan dengan nama yang digunakan)

Gunakan Debugger / Variabel Pointer untuk mengetes apakah Page Frame sudah teralokasi atau tidak. Jika tidak, **Debugger akan gagal** dan variabel pointer menyebabkan CPU Exception **Page Fault**.

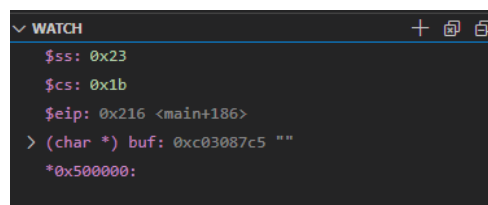
Berikut contoh mencoba mengubah memory `0x500000` dengan pointer `*((uint8_t*) 0x500000) = 1;`

Kondisi pada gambar adalah hanya `0 - 0x40 0000` dan `0xC000 0000 - 0xC040 0000` terpetakan ke physical memory

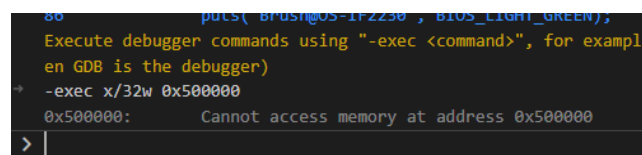


**CPU Exception - Page Fault (Interrupt Vector 0xE)**

Berikut adalah contoh error dengan debugger



**Watch expression tidak mengeluarkan apapun**



**Examine dengan gdb mengeluarkan error**

### 3.2.4. Simple User Program

Sekarang, *basic ingredient* untuk user program sudah siap sehingga bagian ini akan membuat user program yang sangat sederhana, hanya menuliskan **0xDEADBEEF** ke register **eax**.

User program ini akan berformat **flat binary executable**. Sehingga untuk menghindari *annoyance* yang disebabkan urutan linking dapat berpengaruh, user program akan di-link dengan kode assembly yang hanya berisikan label **\_start** yang memanggil fungsi C **main()**. Linker script akan menjamin label **\_start** terletak paling atas dari binary executable.

Berikut adalah linker script, assembly, dan kode C untuk user program

#### user-linker.ld

```
ENTRY(_start)
OUTPUT_FORMAT("binary")

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

    .text ALIGN(4):
    {
        user-entry.o(.text) /* Put user-entry at front of executable */
        *(.text)
    }

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

    .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")
}
```

#### user-entry.s

```
global _start
extern main

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

#### user-shell.c

```
#include "lib-header/stdtype.h"

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

Ketiga file diatas akan di-link menjadi **flat binary** format oleh ld. User program dapat di-compile & link dengan cara seperti berikut

#### makefile

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

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

Lengkapilah juga resep **insert-shell** pada makefile diatas jika dibutuhkan. Jika tahap inserter dan kode user program sudah berjalan dengan baik, user program dapat dimasukkan ke dalam sistem operasi dengan **make disk** dan **make insert-shell**

### 3.2.5. Execute Program

Setelah ada cara untuk mengalokasikan memory untuk user program dan user program sederhana sudah dibuat, tentunya perlu cara untuk melakukan eksekusi instruksi user program.

Sayangnya cara untuk mengeksekusi user program dari kernel terasa seperti *band aid solution*. Tidak ada instruksi langsung (seperti **jmp**) yang dapat digunakan untuk melakukan hal ini. Tugas besar ini akan menggunakan metode **iret** untuk eksekusi user program dengan privilege Ring 3.

Tambahkan kode assembly berikut pada **kernel\_loader.s** pada **section .text**

#### kernel\_loader.s

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

    mov     ecx, [esp+4] ; Save this 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\_loader.h** agar dapat dipanggil oleh **kernel.c**

#### kernel\_loader.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
 */
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); // Note : Already implemented in kernel_loader.asm
```



### 3.2.6. Launching User Mode

Dengan semuanya sudah ter-setup: File system, Memory, Execute program, dan User program yang sudah masuk pada storage, sekarang waktunya mencoba mengeksekusi

Tambahkan kode pada kernel seperti berikut

kernel.c

```
void kernel_setup(void) {
    enter_protected_mode(&_gdt_gdtr);
    pic_remap();
    initialize_idt();
    activate_keyboard_interrupt();
    framebuffer_clear();
    framebuffer_set_cursor(0, 0);
    initialize_filesystem_fat32();
    gdt_install_tss();
    set_tss_register();

    // Allocate first 4 MiB virtual memory
    allocate_single_user_page_frame((uint8_t*) 0);

    // Write shell into memory
    struct FAT32DriverRequest request = {
        .buf          = (uint8_t*) 0,
        .name          = "shell",
        .ext           = "\0\0\0",
        .parent_cluster_number = ROOT_CLUSTER_NUMBER,
        .buffer_size   = 0x100000,
    };
    read(request);

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

    while (TRUE);
}
```

Berikut adalah kasus umum ketika kode kernel ini dijalankan, pause QEMU terlebih dahulu jika masih berjalan (Tombol kiri atas QEMU, Machine → Pause)

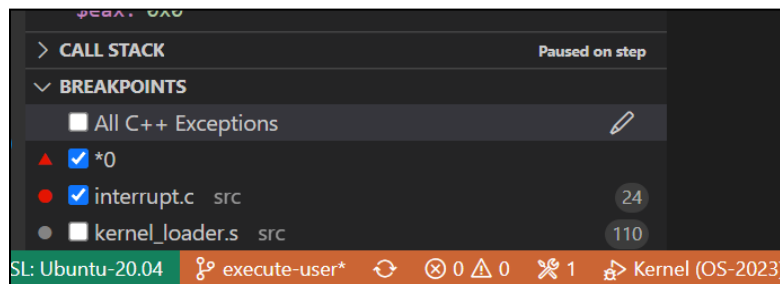
- Tidak terjadi apapun, **eip** menunjuk pada instruksi kernel → pastikan **read()** berhasil
- QEMU crash → TSS belum terinisiasi dengan baik
- QEMU Triple Fault & Bootloop → Ada kemungkinan kesalahan pada setup user mode
- Terkena **General Protection Fault** → Kemungkinan GDT User Mode masih kurang tepat
- Jika QEMU diam saja, **eip** menunjuk ke address sekitar 0x0 hingga 0x100  
Jika iya, berarti sudah berhasil masuk ke user program

Dalam kondisi sama diatas, jika pada user mode OS Triple Fault ketika menekan Keyboard (Keyboard Interrupt) atau setelah menunggu beberapa waktu (Timer Interrupt), lanjutkan **3.3.1 System Calls**. Hal tersebut disebabkan karena TSS **ss0** masih belum disetup

## Tips User Mode

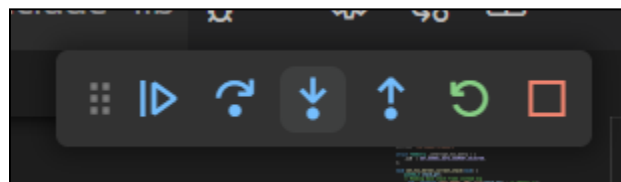
- **Penting:** Karena kernel dan user program nantinya akan di compile secara terpisah, jika telah mengupdate shell, jangan lupa untuk memanggil **make disk** dan **make insert-shell**
- Gunakan QnA atau asistensi jika mengalami bug / behavior yang aneh
- **Bagian user mode akan sangat dependen dengan file system.** Jika ada error pada read / write sehingga data instruksi yang di load ke memory menjadi **reordered** atau **corrupted**, user program akan tidak berjalan sesuai dengan ekspektasi
- Binary shell dapat didump dengan command **objdump -D -b binary -m i386 shell**
- Jika menggunakan **kernel\_execute\_user\_program()** yang diberikan, ukuran stack user program akan sama dengan **4 MiB - size(shell)**, dengan **size(shell)** adalah ukuran binary executable dari shell
- Karena program user di-compile secara eksternal dan semua symbol distrip (karena flat binary format), debugger vscode tidak bekerja secara langsung pada source file **user-shell.c**

Detail fix untuk ini akan dijelaskan pada bagian shell, tetapi untuk sekarang dapat digunakan **breakpoint secara langsung pada virtual address**



Contoh breakpoint pada virtual address **0x0000 0000**

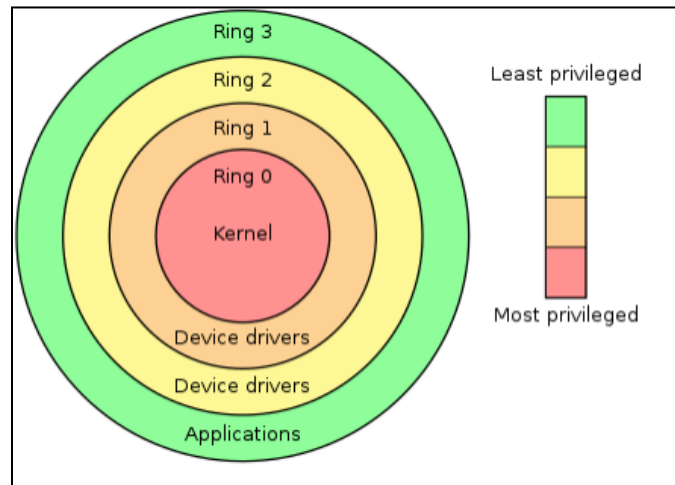
- Gunakan juga **step into** ketika menggunakan debugger pada assembly view, step over terkadang akan melompati instruksi karena menganggap serangkaian instruksi tersebut adalah function call



Step into pada VSCode debugger

- **Security at CPU-level**

Protection ring yang diperlihatkan pada milestone 1 biasanya diimplementasikan pada tingkat CPU. Untuk sistem operasi x86, umumnya hanya menggunakan ring terdalam dan terluar, **Ring 0 / Kernel mode** dan **Ring 3 / User mode**.



Source: [Protection Ring - Wikipedia](#)

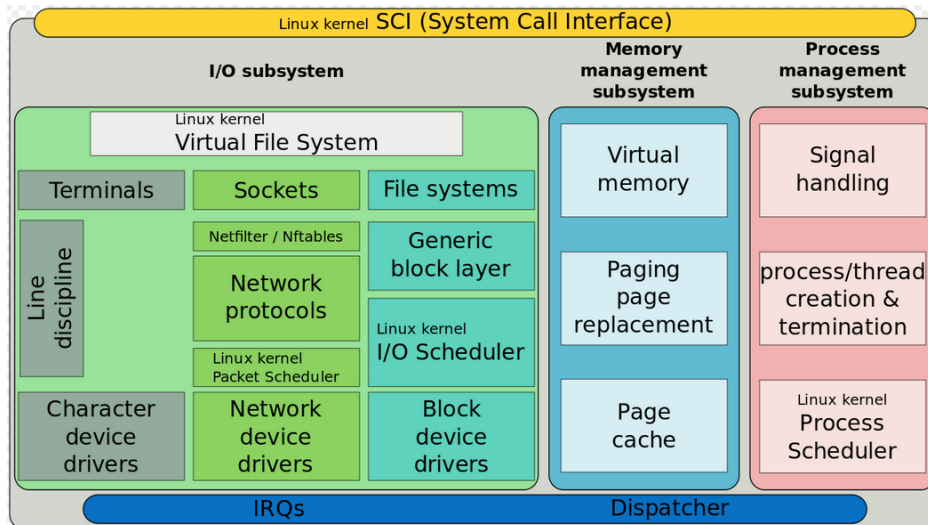
Dengan *kernel mode* dan *user mode*, CPU akan membatasi akses program user ke resources yang terhubung pada komputer. Kernel sistem operasi bertugas untuk mengatur apakah user diperbolehkan atau tidak mengakses resource tertentu. Jika tidak diperbolehkan oleh kernel, maka user hanya dapat menerima.

Karena pada tingkat CPU (atau dalam kata lain, tingkat *hardware*), keamanan ini biasanya sangat sulit untuk ditembus. Daripada mencoba untuk menembus keamanan pada tingkat ini, sebagian besar exploit terletak pada **keamanan tingkat software**. Contohnya pada tingkat software seperti stack overflow yang disebabkan **gets()** atau implementasi user privilege checking yang kurang baik dari sistem operasi.

Namun tidak menutup juga adanya kesalahan desain mikroarsitektur CPU yang menyebabkan vulnerability tingkat hardware seperti **Meltdown** dan **Spectre**. Fatalnya, untuk vulnerability tingkat hardware, umumnya tidak ada “obat” patch yang ada pada tingkat software. Solusinya adalah mengganti hardware tersebut dengan versi yang lebih baru.

## • User Mode & System Calls

Secara singkat dan simplifikasi, **program user tidak memiliki akses apapun terhadap dunia luar**. Tapi bukankah program seperti **Steam, Whatsapp, Discord** merupakan user program yang bisa berkomunikasi dengan user menggunakan interface UI dan komputer lain menggunakan Internet? Ya, meski user program by default arsitektur x86 tidak memiliki akses ke dunia luar, sistem operasi yang digunakan akan memberikan API ke dunia luar yang dapat digunakan aplikasi. API ini umumnya diakses menggunakan **System Calls**



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

System calls menjadi interface antara kernel mode dan user mode program. Kernel bertugas untuk mengevaluasi apakah permintaan program user ditolak atau dipenuhi. Selain itu umumnya user juga dapat menikmati langsung layanan-layanan tersebut, tanpa mengetahui implementasinya.

Tentunya hal ini tidaklah gratis, setiap layer abstraksi yang ditambahkan akan menambah overhead performa sistem. Tapi, *generally small price to pay* untuk **Security & Convenience**

Seperti yang dirasakan pada tugas besar ini, bayangkan saja ketika kita tidak menggunakan sistem operasi dan ingin membuat website, kita harus membuat **Networking Stack, Scheduler, Memory Management** terlebih dahulu.

Pada tugas besar ini, **General & Basic Memory Protection** telah terbentuk dari **Protected Mode, GDT, dan Paging**. Character device driver sederhana ekuivalen dengan **Text Framebuffer** dan **Keyboard Driver**. File system & Block device driver telah diimplementasikan dengan **FAT32 - IF2230 edition**.

Jadi pada titik ini, sistem operasi yang dibuat sebenarnya sudah memiliki basic functionality dari sistem operasi pada umumnya.

### 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 $ █
```

Source : [Bash \(Unix shell\) - Wikipedia](#)

Bagian ini bisa dikatakan bagian mulainya *sandbox*. Shell yang diimplementasikan akan hidup dalam **Ring 3** atau **User mode**. Tentunya shell tidak memiliki akses penuh resource sistem; File system, Keyboard driver, dan manipulasi framebuffer sepenuhnya dimiliki dan dikontrol oleh kernel. User tidak memiliki akses kepada I/O tersebut secara langsung. Maka dari itu, agar shell berguna (tidak sekedar *infinite loop doing nothing*), diperlukan **System Calls**

Kemungkinan besar, **bagian syscall dan bagian implementasi shell bersifat *back and forth***. Ketika membuat shell mungkin ada fitur syscall yang belum ada dan ketika membuat syscall mungkin tidak mengetahui kebutuhan yang dibutuhkan shell

Milestone 1 dan 2 sebenarnya mempersiapkan requirement-requirement yang dibutuhkan untuk bagian ini. Sistem operasi hasil milestone 1 dan 2 tidak memiliki **User Interface**. Meskipun ada beberapa layanan yang sudah disediakan kernel, user tidak memiliki cara untuk mengakses layanan tersebut

**Shell adalah sebuah user program yang menyediakan interface ke manusia atau program.** Bagian ini akan mengimplementasikan shell berdasarkan **Command Line Interface (CLI)**.

By default, kernel tugas besar ini akan membaca executable bernama “**shell**” pada root directory.

Bagian ini merupakan *build-up* dari semua yang dikerjakan. **Semua yang dibuat pada milestone 1, 2, dan 3 akan digunakan pada bagian ini**

Berikut adalah beberapa fitur milestone sebelumnya yang secara langsung digunakan pada shell

- **File System**
- **Interrupt Descriptor Table**
- **Text Framebuffer Driver**
- **Keyboard Driver**

Mungkin ada beberapa implementasi bagian yang kurang cocok ketika digunakan di-Syscall dan Shell, edit lagi dan sesuaikan implementasi-implementasi sebelumnya sehingga memenuhi kebutuhan shell

Meskipun terdapat 8 utility command, jika sudah mengerjakan bagian-bagian sebelumnya dengan baik, **bagian ini semestinya tidak terlalu susah untuk dikerjakan secara paralel.**

Gunakan fakta bahwa **pengerjaan tugas besar berkelompok, bukan individu**, untuk mempermudah hidup

***Anyway, good luck, have fun!***  
;)

### 3.3.1. System Calls

Kernel akan menyediakan **System Calls** melewati **Inter Privilege Interrupt**. Sama seperti **CPU Exception** dan **IRQ** (Hardware Interrupt) yang sebelumnya dikerjakan pada milestone 2, **Inter Privilege Interrupt** merupakan sebuah interrupt yang terjadi ketika ada pergantian **Privilege Level**.

Sebelum melanjutkan, perlu diingat lagi bahwa user mode terbatas pada memory dengan privilege user, sedangkan **kernel memiliki akses penuh ke sistem**. Apa arti konkrit dari ini? **Pointer ke user memory akan dapat dibaca/tulis oleh kernel, tetapi tidak sebaliknya**. Percobaan user untuk melakukan akses memory diluar privilege-nya akan menyebabkan CPU exception **General Protection Fault** a.k.a. in more familiar name: **Segmentation Fault**.

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

#### Segmentation Fault

Handler Inter Privilege Interrupt sama persis seperti sebelumnya, sebuah **InterruptGate** pada IDT. Hanya saja **InterruptGate** pada IDT yang dibuat sebelumnya pada milestone 2 masih diset pada privilege 0, user mode program yang mencoba melakukan Interrupt dengan handler yang ditandai privilege level 0 akan menyebabkan **GP Fault**.

Lakukan hal berikut untuk menyiapkan **InterruptGate** agar dapat dipanggil oleh user program

- Edit **initialize\_idt()** yang dibuat pada milestone 2 agar **int\_vector** 0x30 hingga 0x3F memiliki privilege level 0x3. **Selain yang disebutkan, sama persis dengan sebelumnya**

Setelah beberapa **InterruptGate** pada IDT diperbolehkan untuk dipanggil user program, sekarang waktunya menambahkan interrupt handler pada **main\_interrupt\_handler()**

Tambahkan kode seperti berikut pada **interrupt.c**

```
interrupt.c

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

void main_interrupt_handler(struct CPURegister cpu, uint32_t int_number, struct
InterruptStack info) {
    switch (int_number) {
        case PIC1_OFFSET + IRQ_KEYBOARD:
            keyboard_isr();
            break;
        case 0x30:
            syscall(cpu, info);
            break;
    }
}

void syscall(struct CPURegister cpu, __attribute__((unused)) struct InterruptStack
info) {
    if (cpu.eax == 0) {
        struct FAT32DriverRequest request = *(struct FAT32DriverRequest*) cpu.ebx;
        *((int8_t*) cpu.ecx) = read(request);
    } else if (cpu.eax == 4) {
        keyboard_state_activate();
        __asm__("sti"); // Due IRQ is disabled when main_interrupt_handler() called
        while (is_keyboard_blocking());
        char buf[KEYBOARD_BUFFER_SIZE];
        get_keyboard_buffer(buf);
        memcpy((char *) cpu.ebx, buf, cpu.ecx);
    } else if (cpu.eax == 5) {
        puts((char *) cpu.ebx, cpu.ecx, cpu.edx); // Modified puts() on kernel side
    }
}
```

Kode tersebut merupakan contoh implementasi syscall untuk operasi **file system read**, **keyboard input**, dan **text output** dari template.

Implementasikan sendiri untuk kode **puts()** menggunakan fungsi framebuffer yang telah dibuat sebelumnya. Selain itu sesuaikan kode diatas dengan kebutuhan atau implementasi yang telah dibuat sebelumnya



## • Coding Convention & Compiler

Seperti yang disebutkan pada milestone 1, pada kernel development, operasi casting integer value ke address bisa dikatakan “**halal**” (kode diatas menggunakan banyak integer to pointer dan sebaliknya). Umumnya operasi ini *sangat tidak dianjurkan* untuk digunakan pada user program dan *straight-up* “**haram**”, berdasarkan C programming convention umum

Kernel memiliki kewajiban untuk mengelola memory, jika tidak melakukan hal tersebut, bagaimana caranya untuk mengelola memory?

Meskipun hal-hal yang biasanya “*haram*” menjadi “*halal*”, **tidak berarti kalian bisa menjadi careless**. Berhati-hatilah setiap kali melakukan operasi-operasi “*haram*” jika masih belum memahami konsep pointer, memory, dan sebagainya

Mengulangi yang dituliskan pada disclaimer milestone 1, sekali lagi, **tidak ada handholding disini**. Tidak ada *safeguard* di tingkat kernel, mau menulis di-memory kernel & boot loader? Boleh, Mau mendelete metadata file system? Boleh, Membuang CPU cycle untuk menghabiskan energi? Boleh juga

**Maka dari itu sangat tidak direkomendasikan untuk mematikan flag -Werror pada gcc**. Flag ini akan **mengubah semua warning menjadi error**. Meskipun memang terkadang terasa *frustrating*, compiler warning adalah satu-satunya protection ketika kernel development

***Compiler warning is not for protecting the computer, but you***

Berikut adalah desain syscall yang dapat digunakan sebagai panduan

- Desain syscall adalah **UNIX-like**
- Syscall akan menggunakan **General Purpose CPU register** sebagai parameter
  - **eax, ebx, ecx, edx**
- Register **eax** akan digunakan untuk layanan syscall yang diinginkan
- Register **ebx, ecx, dan edx** akan digunakan sesuai dengan kebutuhan layanan (akan dicast ke pointer, etc)
- Berikut adalah tabel untuk layanan syscall yang disediakan kernel

eax	Service	Parameter		
		ebx	ecx	edx
0	File system read()	<b>FAT32DriverRequest</b> pointer	Return code pointer	- (Unused)
1	FS read_directory()	Ptr request	Ptr retcode	-
2	FS write()	Ptr request	Ptr retcode	-
3	FS delete()	Ptr request	Ptr retcode	-
4	Keyboard input - fgets()	Ptr char buffer	Buffer size	-
5	Text output - puts()	Ptr char buffer	Char count	Text color
...	...	...	...	...

**Sekali lagi, template yang diberikan tidak wajib diimplementasikan**, tetapi bisa menjadi panduan dan ide dasar pengerjaan. Diperbolehkan menambah, menghapus, membuat sendiri syscall sesuai dengan kebutuhan

Catatan : **cpu.esp & cpu.ebp** adalah register **esp** dan **ebp** kernel ketika dipanggil, bukan user

Argumen syscall dari user 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}
>
```

String dapat dicek dengan perintah seperti berikut (atau menggunakan **gdb, -exec x/s buf**)

Berikut adalah contoh **shell** yang memanggil beberapa syscall (berdasarkan template)

**user-shell.c**

```
#include "lib-header/stdtype.h"
#include "lib-header/fat32.h"

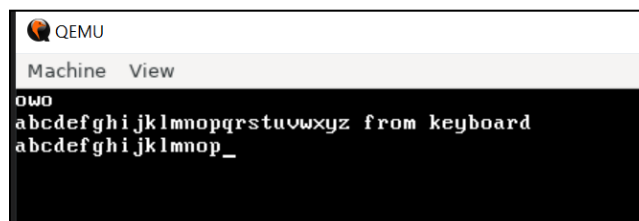
void syscall(uint32_t eax, uint32_t ebx, uint32_t ecx, uint32_t edx) {
    __asm__ volatile("mov %0, %%ebx" : /* <Empty> */ : "r"(ebx));
    __asm__ volatile("mov %0, %%ecx" : /* <Empty> */ : "r"(ecx));
    __asm__ volatile("mov %0, %%edx" : /* <Empty> */ : "r"(edx));
    __asm__ volatile("mov %0, %%eax" : /* <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 ClusterBuffer cl = {0};
    struct FAT32DriverRequest request = {
        .buf = &cl,
        .name = "ikanaide",
        .ext = "\0\0\0",
        .parent_cluster_number = ROOT_CLUSTER_NUMBER,
        .buffer_size = CLUSTER_SIZE,
    };
    int32_t retcode;
    syscall(0, (uint32_t) &request, (uint32_t) &retcode, 0);
    if (retcode == 0)
        syscall(5, (uint32_t) "owo\n", 4, 0xF);

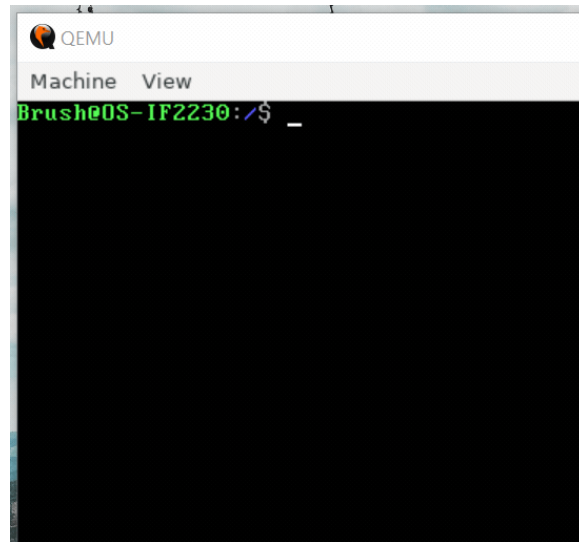
    char buf[16];
    while (TRUE) {
        syscall(4, (uint32_t) buf, 16, 0);
        syscall(5, (uint32_t) buf, 16, 0xF);
    }

    return 0;
}
```

Jika syscall sudah berhasil, semestinya user program tersebut akan mendapatkan hasil **read()** dari kernel dan dapat membaca keyboard seperti screenshot berikut (Asumsi ada file "ikanaide" di file system)



### 3.3.2. Shell Implementation



Bagian ini berhubungan dengan bagian sebelumnya. Jika merasa ingin mengimplementasikan fitur shell tetapi syscall yang dibuat tidak menyediakan, tambahkan syscall baru. Sama seperti spesifikasi, berikut adalah list yang diwajibkan ada pada shell

Behavior dasar shell yang harus ada:

- Shell akan menggunakan desain **REPL** (read-eval-print-loop, seperti shell biasanya)
- Shell dapat menuliskan **Current Working Directory**
- Awal current working directory terletak pada **Root**
- **Shell tidak wajib mem-parse relative pathing** (Contohnya: **cd ../folder1/nestedf1/**)

Spesifikasi meminta built-in utility berikut pada shell, dengan behavior UNIX-like (selain **whereis**):

- **cd** - Mengganti current working directory (termasuk **..** untuk naik)
- **ls** - Menuliskan isi current working directory
- **mkdir** - Membuat sebuah folder kosong baru
- **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
- **whereis** - Mencari **file/folder dengan nama yang sama diseluruh file system**

Dasar dan kerangka dasar program shell sudah diberikan pada bagian **System Calls** sebelumnya. Utility akan diimplementasikan langsung pada shell (bukan program terpisah).

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

Catatan : Jika menggunakan semua rekomendasi milestone 2 dan 3 (Kernel & User stack size, user linker script, 1 cluster FAT), by default tugas besar ini mengasumsikan **ukuran executable shell adalah kurang dari 1 MiB**. Yang dimana mestinya untuk pengerjaan tugas besar ini **jauh lebih dari cukup**. Diperbolehkan mengganti batas ukuran shell jika dibutuhkan

**Penting :** Berhubungan dengan yang disebutkan pada tips user mode, **shell tidak memiliki debug symbol** (sehingga tidak dapat di-debug secara langsung dengan debugger vscode). Namun hal ini dapat diperbaiki dengan membuat dua executable berbeda, satu **flat binary executable** dan satunya **ELF32-i386 executable**

Executable ELF32-i386 ini hanya akan digunakan untuk debugging symbol yang dibuat oleh **gcc** dan **nasm**. Nantinya **gdb** dapat membaca debugging symbol tersebut ketika melakukan debugging.

Edit kode berikut pada **makefile**

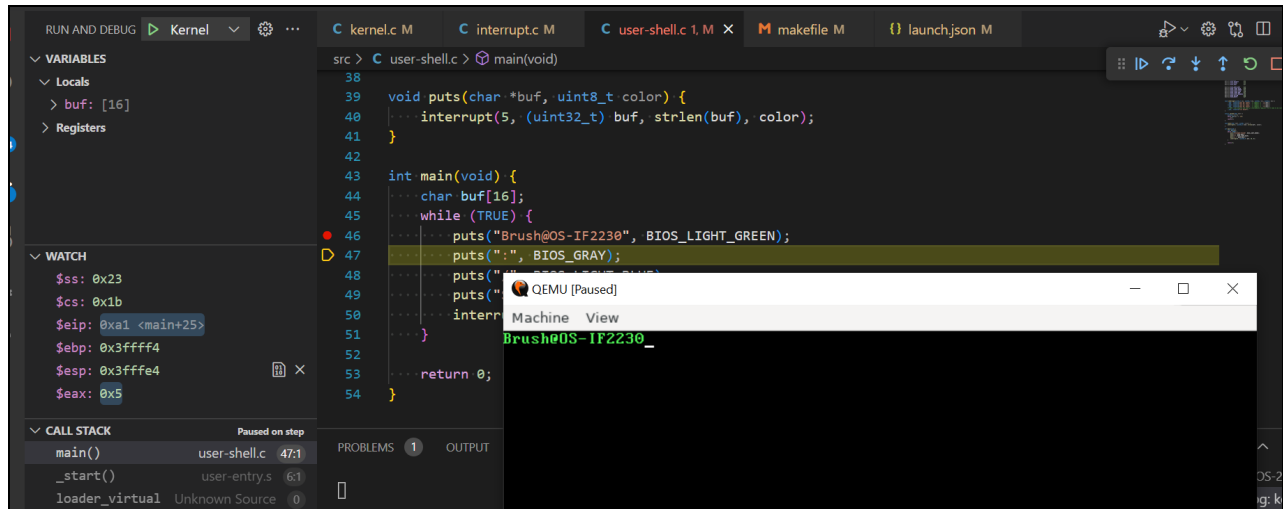
```
makefile

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

Edit **customLaunchSetupCommands** pada **launch.json** seperti berikut

```
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"
    }
],
```



Breakpoint pada line 46 **user-shell.c**, step over dan menuliskan “**Brush@OS-IF2230**” ke layar

Jika sudah berhasil, breakpoint VSCode pada **user-shell.c** akan bekerja dengan normal seperti gambar diatas.

Implementasi shell akan tetap dapat menggunakan debugger untuk mempermudah proses debugging user program.

## Tips Shell

- *Reiterating* dan menekankan lagi, **Tugas besar ini tidak didesain untuk solo**. Jika ada masalah (stuck, bingung, dll), cobalah untuk dibicarakan dengan anggota lain
- Selalu ingat untuk melakukan **make disk & make insert-shell** ketika mengubah kode shell. Jika perlu, untuk sementara pasang resep **disk & insert-shell** pada resep **iso** agar selalu di compile dan insert setiap kali menjalankan debugger
- Jika untuk suatu alasan (*splash screen*, *\*useless\* loading bar*, etc) dibutuhkan untuk **sleep()**, salah satu trik yang paling mudah adalah gunakan saja **nested loop yang doing nothing**. Jumlah index dan nested loop yang dibutuhkan akan bergantung dengan kecepatan sistem jika menggunakan trik ini
- Jika keyboard berhenti berfungsi, **pastikan semua IRQ** (Timer, Keyboard, dll) telah di-**ACK**
- Terkait dengan *open ended-ness* dari shell, handle kasus yang sederhana saja (contohnya **cat** hanya menuliskan file, tolak argumen folder), **tidak perlu edge cases yang ekstrim** (contohnya apakah commands dijamin valid dan tidak menyebabkan *buffer overflow* sehingga dapat di-exploit, etc)
- Jika masih belum yakin apakah edge case terlalu ekstrim atau sudah cukup, gunakan QnA
- Gunakan cast pada **Debug Console** untuk membaca memory sebagai struct

```
code → (struct FAT32DirectoryTable) c1
└─ { ... }
  └─ table
    └─ [0]
      > name
      > ext
      attribute: 0x10
      user_attribute: 0xaa
```

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

- Gunakan juga **gdb print** atau casting untuk menuliskan **null-terminated string**

```
→ -exec print ((struct FAT32DirectoryTable) c1).table[0].name
$1 = "root\000\000\000"
```

cl (**ClusterBuffer**) dicast dan dibaca nama dari entry ke 0

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

**buf** casted into char pointer

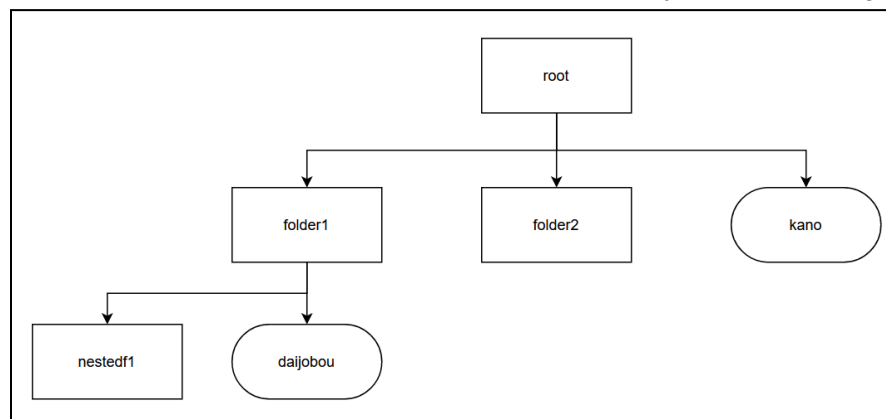
- Karena masih *fresh* dan mata kuliah ini biasanya diambil secara bersamaan, gunakan pengetahuan algoritma graf yang dipelajari dari **IF2211 - Strategi Algoritma**



***Mowanga, Babwana Money!***

Source: stolen from '21's readme, xd

- Mengapa **Graf**? Meskipun FAT32 awalnya adalah linked list, karena setiap cluster memiliki lebih dari 1 pointer (FAT dan DirectoryTable) ke file/folder lain, secara tidak langsung memenuhi kondisi dan membentuk **Graph**, lebih spesifiknya **Directed Acyclic Graph**



***Your friend, Tree, formed by collections and tables of linked list***

- Alasan mengapa **FAT32 - IF2230 edition** meminta entry 0 dari **DirectoryTable** merupakan pointer ke parent adalah **agar file system membentuk Undirected Acyclic Graph atau Tree**, yang mempermudah traversal nantinya
- Strategi traversal file system untuk **whereis** dapat menggunakan **Depth First Search** atau **Breadth First Search** (*true to its name, search*) yang diajarkan pada Stima
- File system **FAT32 - IF2230 edition** tidak secara eksplisit menggunakan struktur data graf seperti yang ada pada **IF2110 - Algoritma & Struktur Data** dan **IF2211 - Strategi Algoritma**, *instead* secara implisit terbentuk dari pointer cluster (edges) dan file/folder (nodes)



Yang memperlihatkan aplikasi graf pada *software engineering* tidak sekedar terbatas pada struktur data yang eksplisit graf

## • Kernel & Operating System

Setelah mengerjakan milestone-milestone ini, dapat mencoba untuk kembali ke pertanyaan awal

### ***Apa itu sistem operasi dan kernel?***

Meskipun “*gambaran high-level*” bagaimana kernel & sistem operasi bekerja bisa diketahui melewati kelas Sistem Operasi atau bahkan seperti pada situs-situs Internet, gambaran-gambaran tersebut umumnya terasa tidak konkrit dan *hand waving*

Setelah menyentuh secara langsung bagaimana membuat sistem operasi, semestinya akan tergambar langsung apa saja yang dikerjakan oleh sistem operasi, seberapa banyak privilege yang dimiliki kernel, dan subsistem-subsistem yang ada pada sistem operasi

Seperti yang diketahui, **kernel memiliki kewajiban untuk mengontrol resource** yang ada pada komputer seperti RAM, Secondary Memory, CPU, I/O, dan lain-lain. Selain itu, kernel menyediakan layanan-layanan melewati **System Calls**. Umumnya hanya kernel tidak dianggap sebagai sistem operasi, **sistem operasi umumnya didefinisikan sebagai kernel dengan utility, UI, subsistem tambahan, dan aplikasi lainnya**



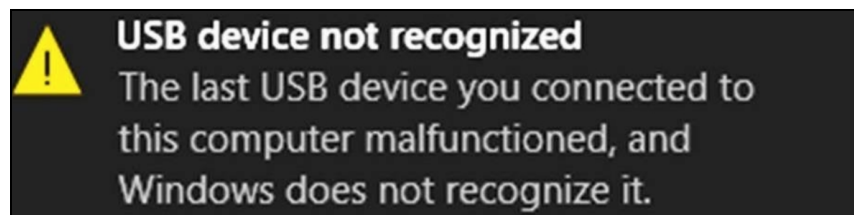
***Watch your computer burn by compiling a penguin***

**Linux Kernel** merupakan salah satu kernel yang sangat populer. Seperti namanya, Linux kernel hanya meliputi bagian utama dari sistem operasi dan dapat ditambahkan subsistem lainnya. Sistem operasi **Android** menggunakan Linux Kernel yang dimodifikasi untuk keperluan mobile device. Selain itu, open source-nya Linux kernel memperbolehkan untuk **membuat sistem operasi sendiri yang menggunakan Linux kernel**. Beberapa contoh sistem operasi yang sangat *barebone* dan menggunakan Linux kernel adalah **Arch, Gentoo, dan LFS / Linux From Scratch**

Meskipun sistem operasi yang dibuat tugas besar ini sederhana, konsep-konsep yang ada masih applicable dengan modern OS. Pengetahuan-pengetahuan yang didapat pada tugas besar ini sangat berguna jika ingin mendalami bidang **Kernel Development** dan **Information Security**. Selain itu OS juga berguna pada **Computer Infrastructure** dan **Distributed System**

## • Hot Plug & Device Recognition

Sebagai pengguna device modern, sebagian besar orang tidak menyadari dan *taken for granted* jika port memiliki bentuk yang sama dengan komputer, kita dapat langsung “*mencolokkan*” dan device dapat bekerja. Entah USB, Ethernet, HDMI, DisplayPort, VGA, dan lain-lain. Padahal sebenarnya sistem operasi modern memiliki banyak built-in driver yang mengimplementasikan banyak sekali macam protokol komunikasi antar device dan komputer. Ketika device terhubung dengan komputer, umumnya pembukaan pada protokol komunikasi akan juga mencakup **Device Recognition**. Jika ada hal yang menyebabkan proses recognition ini gagal (Contohnya, kabel tidak tersambung dengan baik), biasanya menyebabkan device not recognized. Driver yang tidak terinstall dengan baik juga dapat menyebabkan hal ini, OS tidak mengetahui cara berkomunikasi dengan device jika tidak ada driver yang sesuai



Windows 10 - USB Device not recognized

Karena relatif mudahnya hal ini pada sisi user, banyak pengguna yang juga mengasumsikan semua port **hot pluggable**. By default, **tidak ada yang gratis dari sisi bare metal**. Bahkan PC pada abad-20, biasanya port tidak bersifat hot pluggable (termasuk **VGA**, tapi device dengan VGA modern biasanya memiliki pengaman tambahan), sama seperti internal interface seperti SATA atau NVMe. Sistem operasi juga perlu melakukan pencatatan semua device yang telah terhubung dengan baik untuk menyediakan hot plug.

Selain itu, masih berhubungan dengan hot plug, “**Safely remove removable disk**” atau “**Eject removable disk**” merupakan salah satu hal yang muncul karena hot plug. Pada sistem operasi modern, biasanya file system dijalankan secara asynchronous dan banyak proses dapat mengakses suatu direktori. Karena removable storage juga menggunakan file system, sistem operasi wajib mengelola metadata file system tersebut (seperti **FileAllocationTable** pada FAT32). Jika ada penulisan yang belum ditulis sistem operasi dan removable storage dicabut tanpa sepengetahuan sistem operasi, data tersebut dapat hilang, atau yang lebih buruk lagi, metadata pada removable storage dapat kacau.

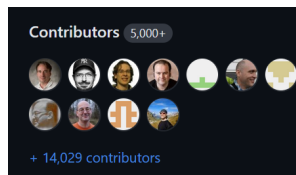
Operasi “**Eject removable disk**” sederhananya memang menyelesaikan semua penulisan dan pembacaan ke storage, dan juga mengecek apakah ada proses yang masih menggunakan storage ini.

## • Modern OS

MS-DOS dirilis pada 1981, 42 tahun ketika dokumen ini ditulis dan Linux dibuat oleh Linus pada 1991, 32 tahun.

Kedua contoh sistem operasi modern tersebut berawal dari tim kecil hingga mungkin ratusan dan puluhan ribu developer yang berkontribusi sekarang. Dari fungsionalitas sistem yang kompleks, berbagai macam memory management, berbagai macam target architecture, hingga *massive amount* standar-standar legal maupun non-legal yang harus dipatuhi.

**Sistem operasi modern tidak dapat dibuat sendirian**, dan sebenarnya tugas besar ini juga berlaku hal yang sama.



### **Linux Kernel GitHub Repository - 14k~ contributors & 1,17 Million Commits**

Meskipun metrik “Man hour” biasanya tidak terlalu akurat, masih dapat digunakan untuk menggambarkan seberapa kompleks suatu sistem. Kedua sistem operasi modern yang diambil sebagai contoh, kurang lebih dalam proses development lebih dari 30 tahun hingga sekarang. Dalam 30 tahun tersebut, lebih dari ribuan orang yang berkontribusi ke dalam proses development sehingga dengan kalkulasi sederhana mungkin sekitar **30.000 tahun man hour**

Meskipun tidak mungkin sekelompok kecil orang atau bahkan organisasi kecil untuk membuat sistem operasi modern, hal tersebut mestinya tidak jadi penghalang untuk seseorang mencoba **memahami cara kerja sistem operasi**.

Tugas besar ini mengimplementasikan sistem operasi “mainan” x86 protected mode. Memang sebagian besar dari tugas besar ini cenderung menyentuh detail secara langsung dan menyentuh **implementation-level** dari konsep-konsep yang ada pada sistem operasi.

Namun, sebelum mencoba untuk memahami konsep-konsep lain sistem operasi secara **high-level**, ada baiknya juga telah mengetahui low-level / implementation detail. Sama seperti matematika, mencoba mempelajari “high-level” seperti kalkulus tapi belum menguasai “implementation details” aljabar dasar. **Possible**, tapi jauh lebih sulit mendapatkan intuisinya dan ketika melakukan kalkulasi / implementasi akan mengalami kendala

Selain itu, beberapa konsep pada tugas besar ini telah diajarkan di kelas, sehingga tugas besar dan kelas juga dapat dianggap sebagai komplementer: *konsep dan implementation*. Jika benar-benar mengerjakan dengan baik, semestinya akan memahami dengan baik konsep seperti **Paging** dan **File system**, baik ide dasar dan implementation-detail

Anggap tugas besar ini sebagai *exercise* dan bahan belajar untuk memahami sistem-sistem yang ada pada komputer, bukan sebuah beban yang harus dikerjakan hanya untuk nilai.

## Referensi Tambahan

### Paging

1. Intel Manual Vol 3A - Chapter 4 - Paging
2. <https://wiki.osdev.org/Paging>
3. [https://www.gnu.org/software/grub/manual/multiboot/html\\_node/multiboot\\_002eh.html](https://www.gnu.org/software/grub/manual/multiboot/html_node/multiboot_002eh.html)
4. <https://github.com/szhou42/osdev>
5. [https://wiki.osdev.org/Memory\\_management](https://wiki.osdev.org/Memory_management)
6. <https://stackoverflow.com/questions/62997536/what-is-the-difference-between-linear-physical-logical-and-virtual-memory-address>
7. [https://wiki.osdev.org/Setting\\_Up\\_Paging](https://wiki.osdev.org/Setting_Up_Paging)

### GDT, User Mode, dan 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](https://wiki.osdev.org/Getting_to_Ring_3)
4. <https://wiki.osdev.org/TSS>

### Shell

1. [https://en.wikipedia.org/wiki/Unix\\_shell](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/whereis.1.html>

### Extras

1. Manimanimanimaniman - <https://github.com/ManimCommunity/manim/>