Tugas Besar

IF2230 - Sistem Operasi

Milestone 03 of 11

"This too shall pass"

Pembuatan dan Eksekusi Program yang Menggunakan System Call

Dipersiapkan oleh :
Asisten Lab Sistem Terdistribusi

Didukung Oleh :

Waktu Mulai :

Rabu, 6 April 2022, 16.00 WIB

Waktu Akhir :

Sabtu, 30 April 2022, 23.59 WIB

I. Latar Belakang

Sudah beberapa minggu berlalu, banyak kejadian - kejadian yang terjadi. Perang, krisis ekonomi, harga bahan baku yang naik, tugas [redacted] mingguan, tugas [redacted], tugas [redacted], dll. Anda bisa tenang karena suatu saat semua ini akan berakhir. Selain itu Kaiba yang dilarikan ke rumah sakit di Korea sudah mulai membaik dan bisa kembali pulang ke Indonesia, tanah air tercinta kita. Tidak perlu lama untuk menunggu kedatangannya dengan jet pribadinya dan langsung menuju tempat bekerja begitu ia mendarat. Dapat diakui ia memang seseorang yang pekerja keras, namun sulit rasanya bekerja siang malam sepertinya di saat Anda mempunyai tugas-tugas yang rilis di setiap minggunya. Setibanya Kaiba di sana, Anda bisa melihat kaiba tengah duduk di kursi bosnya dengan penampilannya yang baru setelah ia menjalani operasi plastik di Korea. Tampak sesuatu yang berbeda disini, Anda bisa melihat seekor naga bermata biru tengah duduk di pundak Kaiba. Untuk sesaat Anda tidak mempercayai mata Anda dan menganggap semua ini hanyalah ilusi yang disebabkan oleh kurangnya tidur dikala tugas melanda. Tetapi tidak, semua yang Anda lihat itu adalah kenyataan. Tampaknya ledakan yang terjadi beberapa minggu yang lalu menyebabkan *mini black hole* yang membuka gerbang realitas antar dimensi untuk sementara waktu yang memungkinkan makhluk dari dimensi lain untuk melakukan perjalanan ke dimensi ini.

ia terlihat bahagia 👍

Kaiba mulai mengumumkan bahwa ini akan menjadi tugas terakhir darinya karena ia sadar bahwa hidup tidak hanya tentang bekerja ataupun mendapatkan pengakuan orang lain setinggi tingginya, ada hal lain yang lebih berharga daripada itu. Kaiba ingin hidup damai di daerah pedesaan bersama dengan naga barunya, Kanna-chan. Jika Anda berhasil menyelesaikan keseluruhan tugas ini, Kaiba akan mewariskan perusahaannya kepada Anda. Mampukah Anda mewarisi perusahaan dari Kaiba ini? Berikut merupakan tugas terakhir Anda.

II. Deskripsi Tugas

Setelah milestone 2 menambahkan fasilitas *filesystem* pada sistem operasi, milestone 3 akan membuat *binary executable* terpisah dari *kernel space*. Hal-hal yang akan dilakukan pada milestone ini adalah

- Membuat *library* sistem operasi
- Membuat *syscall executeProgram*
- Membuat aplikasi shell yang independen dari kernel
- Membuat sistem *message passing* antar program
- Menambahkan fitur eksekusi program pada aplikasi *shell*
- Membuat *utility program*
- Membuat eksekusi *multiple program* secara sekuensial

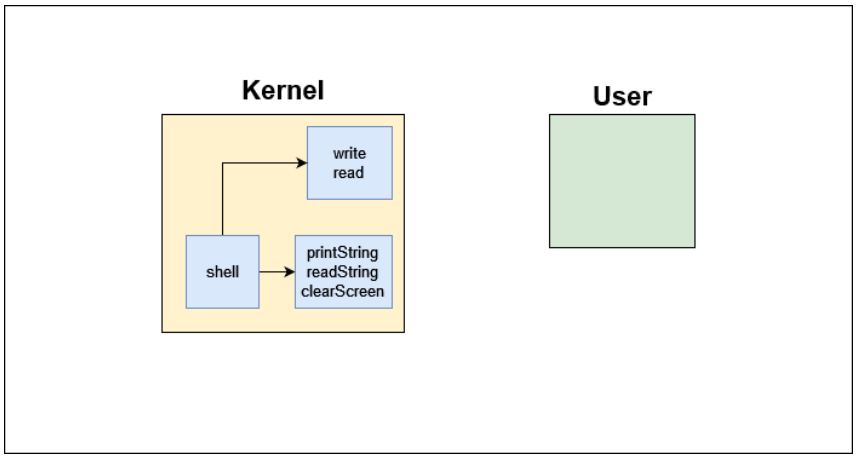
Fitur-fitur diatas akan menggunakan *syscall write* dan *read*. Jika kedua *syscall* belum berjalan dengan baik, direkomendasikan untuk melakukan *bugfix* terlebih dahulu. Kesalahan *read* pada *file binary executable* dapat menyebabkan eksekusi program yang bermasalah.

Pada milestone ini tidak terdapat *header* atau library tambahan pada kit. Repository kit akan digunakan untuk *bugfix* jika ditemukan permasalahan pada kit milestone 1 atau 2. Terdapat beberapa bug kecil untuk *tc_lib* pada milestone 2, *patch* telah dipush pada repository.

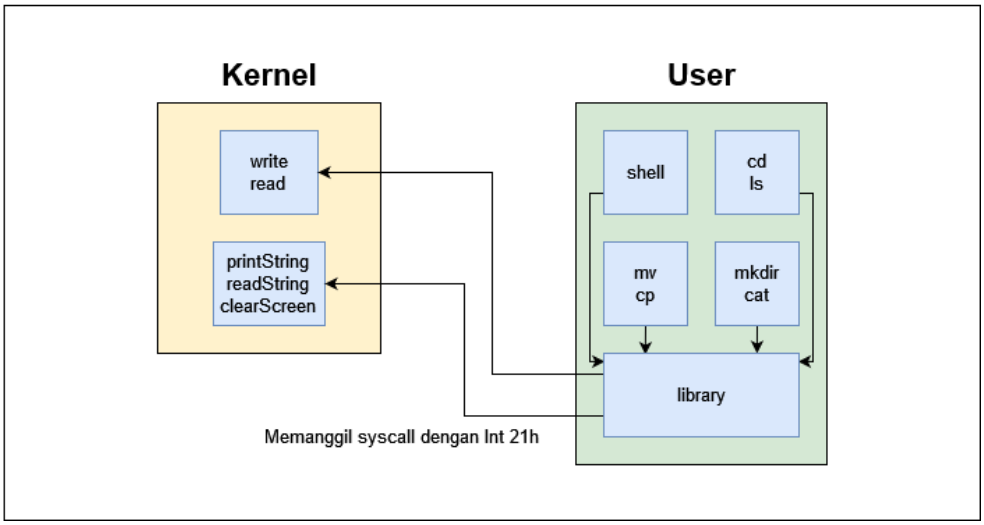
III. Langkah Pengerjaan

3.1. Membuat Library dan executeProgram

Kernel yang telah dibuat sudah memiliki fasilitas *text I/O* ke layar, *filesystem*, dan *shell* sebagai *user interface* untuk melakukan operasi umum. Sistem operasi yang telah dibuat seluruhnya berjalan diatas *kernel space* seperti ilustrasi berikut



Pada bagian ini akan dibuat sebuah *library* yang dapat diimport oleh program *user space* dan pemindahan *shell* dari *kernel space* ke *user space*. Sistem operasi telah menyediakan *syscall* yang dapat dipanggil dengan *interrupt 21h* oleh *user space program*. *User program* akan dapat menggunakan *syscall* yang disediakan sistem operasi seperti illustrasi berikut



3.1.1. Membuat Library

Karena sistem operasi yang dibuat cukup sederhana, sebagian besar *library* akan berisikan *wrapper* kecil untuk *interrupt 21h*. Terdapat beberapa *library* yang wajib diimplementasikan yaitu

- *string* wajib memiliki *strlen*, *strcpy*, dan *strcmp*
- *textio* wajib memiliki *puts* dan *gets*
- *fileio* wajib memiliki *write* dan *read*
- *program* wajib memiliki *exec* dan *exit*

Untuk *behaviour* setiap fungsi *library* dibebaskan kepada pemagang tetapi disarankan untuk mengikuti *behaviour* pada standar *POSIX*. Jika dibutuhkan diperbolehkan untuk membuat *library* lain atau menambahkan *function* lain pada *library*.

Library akan menggunakan fungsi `interrupt()` untuk memanggil *syscall* yang telah dibuat. Pindahkan `interrupt()` dari *kernel.asm* ke file assembly baru *interrupt.asm* dan pada setiap library perlu ditambahkan *function declaration* `interrupt()` seperti berikut

interrupt.asm

```
global _interrupt

_interrupt:
  push bp
  mov  bp, sp
  mov  ax, [bp+4]
  push ds
  mov  bx, cs
  mov  ds, bx
  mov  si, intr
  mov  [si+1], al
  pop  ds
  mov  ax, [bp+6]
  mov  bx, [bp+8]
  mov  cx, [bp+10]
  mov  dx, [bp+12]

intr: int 0x00

  mov ah,0
  pop bp
  ret
```

textio.c

```
extern int interrupt(int int_number, int AX, int BX, int CX, int DX);

int puts(char *string) {
  interrupt(0x21, ..., ..., ..., ...);
  // . . .
}
```

Setelah memindahkan fungsi `interrupt()` dari *kernel.asm*, berikut adalah contoh kompilasi dan linking *interrupt.asm* pada proses pembuatan kernel

```
bcc -ansi -c -o out/kernel.o src/c/kernel.c
nasm -f as86 src/asm/kernel.asm -o out/kernel_asm.o
nasm -f as86 src/asm/interrupt.asm -o out/lib_interrupt.o
ld86 -o out/kernel -d out/kernel.o out/kernel_asm.o out/lib_interrupt.o
```

Catatan : Pastikan argumen *object file* pertama *ld86* adalah hasil kompilasi dari *source code* dengan `main()`. Dalam kasus diatas jika object file *kernel.o* bukan *object file* pertama (ex. `ld86 -d out/int.o out/kernel.o`), sistem operasi akan *stuck* pada layar setelah boot.

3.1.2. Pembuatan *syscall* executeProgram

Sebelum melanjutkan untuk membuat aplikasi terpisah pada *user space*, diperlukan cara untuk menjalankan sebuah program. Tambahkan kode `launchProgram()` ke *kernel.asm*. Tambahkan kode yang ditandai kuning pada `handleInterrupt21()` dan tambahkan fungsi baru `executeProgram()` pada *kernel.c*

kernel.asm

```
global _launchProgram

_launchProgram:
  mov bp,sp
  mov bx,[bp+2]
  mov ax,cs
  mov ds,ax
  mov si,jump
  mov [si+3],bx
  mov ds,bx
  mov ss,bx
  mov es,bx
```

```
  mov sp,0xfff0
  mov bp,0xfff0

jump: jmp 0x0000:0x0000
```

Fun fact : Inti dari kode *assembly bootloader.asm* adalah melakukan `readSector(kernel)` dari drive sector **KSTART**, baca sebanyak **KSIZE** sector, dan tuliskan pembacaan kernel pada memory segmen **KSEG**. Setelah kernel berada pada memory **KSEG**:**0x0000**, `launchProgram(KSEG)` untuk menjalankan kernel.

kernel.h

```
extern void launchProgram(int segment);
void executeProgram(struct file_metadata *metadata, int segment);
```

kernel.c

```
void handleInterrupt21(int AX, int BX, int CX, int DX) {
    switch (AX) {
        case 0x0:
            printString(BX);
            break;
        case 0x1:
            readString(BX);
            break;
             .
             .
             .
        case 0x6:
            executeProgram(BX, CX);
            break;
        default:
            printString("Invalid Interrupt");
    }
}
```

```
void executeProgram(struct file_metadata *metadata, int segment) {
  enum fs_retcode fs_ret;
  byte buf[8192];

  metadata->buffer = buf;
  read(metadata, &fs_ret);
  if (fs_ret == FS_SUCCESS) {
    int i = 0;
    for (i = 0; i < 8192; i++) {
      if (i < metadata->filesize)
        putInMemory(segment, i, metadata->buffer[i]);
      else
        putInMemory(segment, i, 0x00);
    }
    launchProgram(segment);
  }
  else
    printString("exec: file not found\r\n");
}
```

Syscall `executeProgram()` akan menggunakan atribut nama dan *parent index* pada `metadata`. Atribut nama dan *parent index* digunakan untuk mencari file pada lokasi tertentu.

Catatan : Pada tingkat mesin, seluruh informasi akan disimpan dalam satuan *byte*. Tidak ada penanda bahwa serangkaian byte tersebut adalah sebuah *text file (.txt)*, *zip*, *executable*, *jpg, pdf,* dst. Juga tidak ada larangan apa interpretasi dari *serangkaian byte* tersebut, sebuah *string* dapat dianggap sebagai suatu gambar *jpg* atau rangkaian instruksi (seperti *exploit arbitrary code execution* untuk mengeksekusi *shellcode* pada bonus praktikum 3 IF2130 Orkom).

3.2. Membuat aplikasi *shell*

Setelah sistem operasi menyediakan *standard library* sistem operasi dan *syscall*, saatnya untuk memisahkan aplikasi *shell* dari *kernel space*. Sesuai dengan ilustrasi dan penjelasan pada bagian *library*, *interrupt 21h* akan digunakan sebagai *interface syscall* sistem operasi.

3.2.1. *Shell* sederhana

Aplikasi *shell* ini akan menggunakan *syscall* `executeProgram()` dan *library* yang telah dibuat sebelumnya. Buatlah file baru untuk *source code shell* yang terpisah seperti berikut

shell.c

```
#include "header/std_type.h"
#include “header/string.h”
```

```c
int main() {
  puts("Halo shell!\r\n");
  while (true);
}
```

Build shell menggunakan *bcc* dan *ld86*. Berikut adalah contoh kompilasi shell

```
bcc -ansi -c -o out/shell.o src/c/shell.c
bcc -ansi -c -o out/string.o src/c/string.c
nasm -f as86 src/asm/interrupt.asm -o out/lib_interrupt.o
ld86 -o out/shell -d out/shell.o out/lib_interrupt.o out/string.o
```

Setelah aplikasi shell terbuild, gunakan shell pada kernel untuk melakukan testing dan migrasi. Berikut adalah contoh untuk melakukan eksekusi program dari kernel

kernel.c

```c
void shell() {
  char input_buf[64];
  char path_str[128];
  byte current_dir = FS_NODE_P_IDX_ROOT;
  while (true) {
    printString("OS@IF2230:");
    printCWD(path_str, current_dir);
    printString("$");
    readString(input_buf);

    if (strcmp(input_buf, "cd")) {
      // Utility cd
    }
    else if (strcmp(input_buf, "ls")) {
      // Utility ls
    }
      .
      .
      .
    else if (strcmp(input_buf, "test")) {
      struct file_metadata meta;
      meta.node_name    = "shell";
      meta.parent_index = 0xFF;
      executeProgram(&meta, 0x2000);
    }
    else
      printString("Unknown command\r\n");
  }
```

```
}
```

Gunakan segment `0x2000` khusus untuk aplikasi shell. Penjelasan lebih lanjut terkait pemilihan segment terdapat pada bagian tips. Lakukan *build-run* pada sistem operasi untuk menginisiasi map. Masukkan program *shell* kedalam *system.img*. Proses insersi program *shell* tersebut ke *system.img* dapat menggunakan `insert_file()` dan `create_folder()` yang disediakan pada *tc_lib*. Penggunaan kedua fungsi tersebut dapat dilihat pada *tc_gen.c*. Dengan kedua fungsi tersebut buatlah sebuah folder *bin* pada *root* dan file *shell* yang berada pada folder tersebut. Berikut adalah contoh penggunaan *tc_lib*

```
void shell(byte buf[2880][512]) {
    create_folder(buf, "bin", 0xFF);
    insert_file(buf, "shell", 0);
}
```

Catatan : Terdapat bug pada *tc_lib* milestone 2 (File dengan P = 0 & S = 0 akan *dioverwrite* karena kesalahan pengecekan), *tc_lib* terbaru sudah diupload pada repository kit.

Jika kedua fungsi yang disediakan pada *tc_lib* tidak sesuai kebutuhan, diperbolehkan untuk membuat *external program* sendiri yang berjalan diatas *host*. Tidak ada batasan untuk bahasa dan *compiler / interpreter* (*bcc, gcc, clang, g++, cl, pl, python, scratch, bf, manual pake tangan, …*) yang digunakan tetapi pastikan program mengikuti konvensi *filesystem* yang telah dibuat.

Jalankan *run* setelah melakukan insersi aplikasi shell ke *system.img* untuk menjalankan sistem operasi. Ketik `test` pada shell kernel untuk melakukan tes eksekusi program. Jika pada layar terlihat output dari fungsi `puts()` maka eksekusi program berhasil.

Tips : Proses pembuatan sistem operasi dan memasukkan program diatas dijalankan secara manual. `fillMap()` dari *build-run*, mengeksusi program inserter, dan menjalankan sistem operasi dengan *run* dapat digabung dan otomatisasi menjadi satu. Manipulasi *filesystem* **map** dapat dilakukan `byte imagebuffer[2880][512];` pada *tc_gen.c*. Indeks pertama adalah *sector number* dan indeks kedua adalah byte dalam sektor tersebut.

3.2.2. Message passing

Shell akan mengeksekusi program lain yang berada pada *segment* berbeda. Karena hal tersebut diperlukan sebuah cara untuk menyalurkan informasi / *message passing* antar program.

Cara untuk *message passing* dibebaskan, salah satu cara yang mudah adalah membuat file untuk menyimpan argumen atau informasi lainnya. Berikut adalah contoh struktur data *message*

```
struct message {
    byte current_directory;
    char arg1[64];
    char arg2[64];
    char arg3[64];
    int next_program_segment; // Dapat digunakan untuk bagian 3.5
    byte other[317];
};
```

Berikut contoh dari penulisan message menggunakan syscall *write*

```
struct message test;
struct file_metadata metadata;
struct fs_retcode retcode;
int i = 0;

test.current_directory = 0xFF;
for (i = 0; i < 64; i++) {
  test.arg1[i] = 0x41;
  test.arg2[i] = 0x42;
  test.arg3[i] = 0x43;
}
for (i = 0; i < 319; i++)
  test.other[i] = 0x61;

metadata.buffer       = &test;
metadata.node_name    = "message";
metadata.filesize     = 512;
metadata.parent_index = FS_NODE_P_IDX_ROOT;

write(&metadata, &retcode);
```

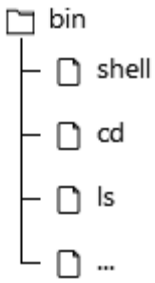
Selain menggunakan syscall *write*, dapat juga mendefinisikan sektor tertentu untuk menyimpan data seperti *filesystem* **map**, **node**, dan **sector** yang terletak pada `0x100`, `0x101-0x102`, dan `0x103`. Pilih sektor yang tidak digunakan oleh sistem operasi (ex. sektor `0x104` keatas) untuk menyimpan *message*. Gunakan *readSector* dan *writeSector* untuk mengakses file tersebut.

Cara lain seperti membuat fungsi *assembly* untuk membaca informasi pada *segment* berbeda juga dapat digunakan untuk implementasi *message passing*. Terdapat beberapa tips untuk assembly pada bagian tips.

3.2.3. Executing binary file

Pada bagian ini shell harus dapat mengeksekusi program secara *local* dan *global*. Buatlah

sebuah folder *bin* pada *root* untuk menampung seluruh executable yang dijalankan secara *global*.



Eksekusi program secara *global* dilakukan dengan cara mengetikkan nama *executable* pada folder *bin* (ex. Jika terdapat executable bernama *cp* pada bin, eksekusi program dilaksanakan dengan mengetik *cp* pada sembarang *current working directory*).

Eksekusi program secara *local* dilakukan dengan mengetikkan *./executable* pada shell. Shell akan mencari program pada *current working directory* dengan nama *executable* untuk dieksekusi.

Spesifikasi untuk bagian ini adalah terdapat folder *bin* pada *root* untuk menyimpan *executable,* shell dapat mengeksekusi program secara *global* dan *local*. Kedua mode eksekusi wajib tetap dapat menerima *message* dari shell.

3.3. Utility program

Setelah aplikasi shell telah berjalan secara independen dari kernel, hapus shell yang berada pada kernel dan eksekusi program shell seperti berikut

kernel.c

```
int main() {
  struct file_metadata meta;
  fillKernelMap();
  makeInterrupt21();
  clearScreen();

  meta.node_name    = "shell";
  meta.parent_index = 0x00;      // Pastikan parent_index adalah folder bin
  executeProgram(&meta, 0x2000);
}
```

Utility yang dibuat pada bagian ini sama dengan *utility* yang dibuat pada milestone 2. Seluruh

behaviour wajib sama seperti spesifikasi pada milestone 2. Berikut adalah *utility* yang wajib diimplementasikan

- **cd**
- **ls**
- **mkdir**
- **cat**
- **cp**
- **mv**

Letakkan *utility* pada folder *bin* agar dapat dieksekusi secara *global* oleh shell. Spesifikasi untuk *utility* adalah wajib dapat berkomunikasi dengan shell menggunakan sistem *message passing* yang telah dibuat sehingga tidak memerlukan penulisan argumen ulang ketika didalam program, pada akhir program panggil kembali shell pada segmen `0x2000`.

Contoh dibawah mengasumsikan terdapat deklarasi fungsi `get_message()` pada *utils.h* yang akan dilink dengan ld86 nantinya. Berikut adalah contoh implementasi *utillity cp*

cp.c

```
#include "header/filesystem.h"
#include "header/std_type.h"
#include "header/textio.h"
#include “header/fileio.h”
#include “header/program.h”
#include “header/utils.h”

int main() {
  struct message msg;
  get_message(&msg);

  /*
      TODO : Implementasi cp
  */

  exit();
}
```

Pada akhir setiap program panggil `exit()` yang digunakan untuk mengeksekusi program selanjutnya. Berikut adalah contoh `exit()`

```
void exit() {
  struct file_metadata meta;
  meta.node_name    = "shell";
  meta.parent_index = 0x00;

  exec(meta, 0x2000);
}
```

Pada akhir program tunggal, lakukan eksekusi shell lagi pada **`0x2000`**.

3.5. *Multiple Program*

Pada bagian ini akan dibuat eksekusi berantai seperti *`cd folder1 ; ls ; cp file2 f4`* yang menggunakan **`exit`**`()` dan *message passing* yang telah dibuat. Gunakan “ *`;`* “ (termasuk spasi untuk mempermudah proses *lexing*) antar command sebagai pemisah program. Token “ *`;`* “ akan di *reserve* sebagai eksekusi *multiple program* sehingga argumen “*`;`*” tidak valid (ex. *`cp folder2 ; ; cd ; ; ls`* tidak valid). Tips : *Split argumen vector* dengan token “ *`;`* “.

3.5.1. Modifikasi *message passing*

Sesuaikan sistem *message passing* yang telah dibuat untuk keperluan *message passing multiple program*. Implementasi *message passing* untuk *multiple program* dibebaskan kepada pemagang.

Berikut adalah beberapa tips dan gambaran untuk *message passing multiple program*

- *Message passing* dilakukan menggunakan banyak *instance message.* Simpan *message* dengan *write* dengan nama *_msg0*, *_msg1*, … . Gunakan kode assembly berikut untuk mendapatkan lokasi segmen sekarang, setiap segmen akan terpetakan dengan indeks *_msg* (ex. _msg0 untuk segment **`0x2000`**, *_msg1* untuk **`0x3000`**, dst)

  utils.h

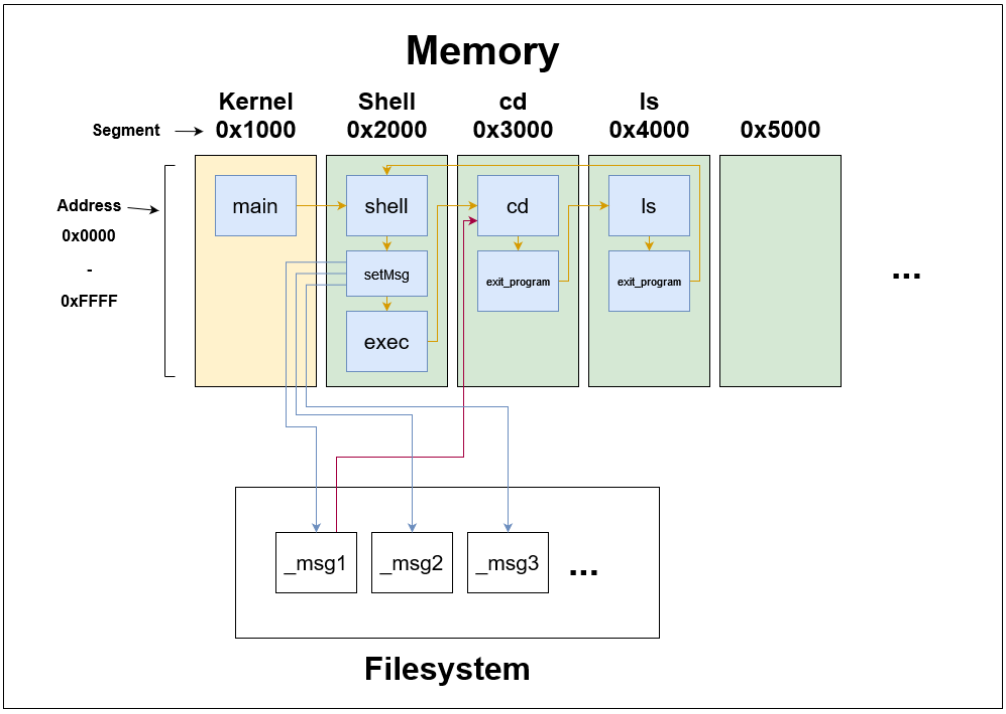
```
extern int getCurrentSegment();
```

  utils.asm

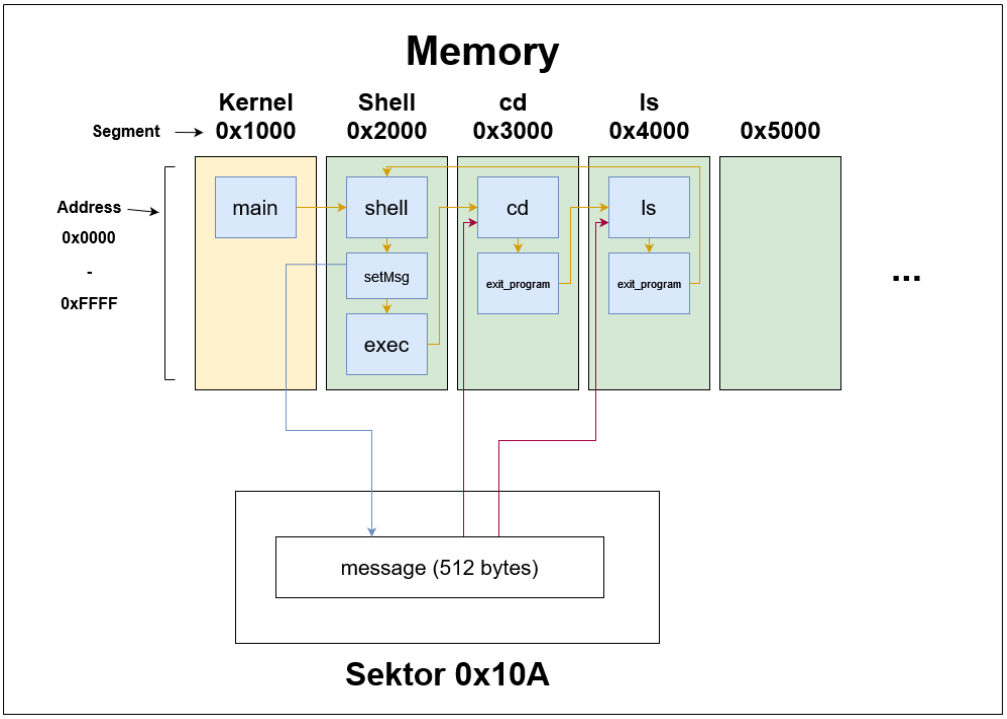
```
global _getCurrentSegment

_getCurrentSegment:
  mov ax, ds
  ret
```

  Berikut adalah ilustrasi untuk *message passing* dengan banyak *message* dengan *read/write*, panah kuning menunjukkan alur instruksi, panah biru menunjukkan operasi *write*, panah merah menunjukkan operasi *read*



- Jika tidak menginginkan banyak *message* untuk setiap program, gunakan semaksimal mungkin 1 *instance message* untuk menyimpan informasi semua program. *Message* tersebut dapat digunakan untuk menyimpan *state* dari eksekusi sistem operasi. Berikut adalah illustrasi menggunakan *message passing* dengan *read/writeSector*



- Jika menggunakan metode selain diatas, sesuaikan sistem *message passing* agar

support multiple program.

3.5.2. Modifikasi *exit*

Gunakan fungsi `exit()` untuk memanipulasi message jika dibutuhkan dan mengeksekusi program selanjutnya. Tambahkan pemrosesan message pada `exit()` sebelum melakukan `exec()` selanjutnya. Pada program terakhir, eksekusi *shell* pada segment `0x2000`. Contohnya `cd folder1 ; ls ; cp file2 f4` alur eksekusi program menjadi *shell (segment* `0x2000`*)* → *cd (segment* `0x3000`*)* → *ls (segment* `0x4000`*)* → *cp (segment* `0x5000`*)* → *shell (segment* `0x2000`*)*.

Untuk *policy* pengalokasian segment yang akan digunakan program dibebaskan. Asumsikan sistem operasi hanya dapat menjalankan maksimal *5 program* dalam satu eksekusi *multiple program*.

Catatan : Gunakan segment `0x3000`, `0x4000`, …, `0x7000` untuk mengeksekusi program selain kernel dan shell. *Reserved memory space* pada *real mode x86* dapat dibaca pada Memory Map OSDev.

3.6. Konstrain dan Batasan

Untuk mempermudah pengerjaan milestone ini, berikut adalah konstrain dan batasan yang digunakan

- Setiap *user program* memiliki ukuran maksimum 8192 bytes (Ukuran maksimum *1 entry file* yang di *support* oleh *filesystem*).
- Setelah *user program* selesai, sistem operasi akan meng-*exec shell* pada segmen `0x2000`.
- *User program* dilarang untuk memanggil langsung *syscall* (ex. Implementasi kode *write* di*copy-paste* dan dipanggil langsung pada *shell.c*). Gunakan *interrupt 21h* yang telah dibuat.
- Antar program dapat berkomunikasi dengan sistem *message passing* yang telah dibuat.
- Ukuran default kernel adalah 15 sektor (7680 bytes). *Upper bound* ukuran kernel pada tugas besar ini adalah 31 sektor (15872 bytes). Jika merubah ukuran kernel pastikan batas `fillMap()`, **KSIZE** bootloader.asm, batas *syscall read* / *write*, dan batas-batas lain telah disesuaikan.
- Bergantung kepada banyaknya *entry filesystem* **node** yang digunakan folder dan executable (*bin, shell, cp*, …), kondisi benar *test case* yang diberikan pada milestone 2 dapat berubah dikarenakan pergeseran *parent index* `insert_file()` dan `create_folder()`.

Epilog

Pada suatu malam, udara Bandung terasa lebih dingin daripada sebelumnya, Anda dapat merasakannya dengan jelas di permukaan pipi Anda, dengan satu gerakan tangan Anda mulai menggunakan jubah merah Anda untuk mengatasi udara dingin tersebut, selain udara dingin Anda juga bisa merasakan betapa kosongnya hati Anda saat ini. Banyak hal yang Anda lakukan untuk menghilangkan kekosongan ini, tetapi tidak satupun yang dapat mengatasi kekosongan tersebut, sebagian besar adalah tindakan tindakan yang dapat menyebabkan degradasi intelegensi. Semakin lama Anda menjadi semakin hampa, hingga suatu ketika Anda teringat kembali tentang ledakan dan gerbang dimensi itu. Anda merasakan keinginan yang sangat kuat untuk membuka kembali gerbang tersebut dan mencari belahan hati Anda. Setiap langkah langkah pada saat itu Anda ulang secara detail dan … *duarrr* ledakan terjadi lagi dan gerbang dimensi berhasil terbuka kembali. Kini Anda bisa mengunjungi realitas realitas lainnya yang tak terbatas jumlahnya. Siapkah Anda berpetualang di realitas tanpa batas ini?

Saatnya ~~berdelusi~~ menjelajahi realitas tanpa batas

Dengan berakhirnya milestone ini, rangkaian pengerjaan tugas besar IF2230 Sistem Operasi telah selesai. Sampai jumpa di realitas berikutnya, *good luck have fun* :)

ps. Judul mengggunakan basis desimal

dan binary

IV. Penilaian

1. ***Library & executeProgram* (30)**
    a. Library *string* (5)
    b. Library *textio* (5)
    c. Library *fileio* (5)
    d. Library *program* (5)
    e. *executeProgram* (10)

2. **Aplikasi shell dan *utility* program (40)**
    a. Shell (25)
        i. Sederhana dan executable (10)
        ii. Execute program *global* & *local* (15)
            1. *Global* (5)
            2. *Local* (10)

    b. *Utility* sesuai spesifikasi (15)
        i. cd (2.5)
        ii. ls (2.5)
        iii. mkdir (2.5)
        iv. cat (2.5)
        v. cp (2.5)
        vi. mv (2.5)

3. **Multiple Program (30)**
    a. *Multiple program* tanpa argumen (ex. *ls ; ls ; ls*) (15)
    b. *Multiple program* dengan message passing (15)

V. Pengumpulan dan Deliverables

1. Untuk tugas ini Anda diwajibkan menggunakan *version control system* **git** dengan menggunakan sebuah *repository* ***private*** di Github Classroom "**Lab Sister 20**" (gunakan surel *student* agar gratis). Invitation ke dalam Github Classroom "**Lab Sister 20"** akan diberikan saat tugas dirilis.

2. Kit untuk semua milestone tugas besar IF2230 tersedia pada repository GitHub berikut link repository.

3. Kreativitas dalam pengerjaan sangat dianjurkan untuk memperdalam pemahaman. Penilaian sepenuhnya didasarkan dari kriteria penilaian, bukan detail implementasi.

4. Pada tugas besar ini akan digunakan bahasa *ANSI C* dan *nasm x86*. Disarankan untuk membuat mayoritas sistem operasi menggunakan *ANSI C* tetapi juga diperbolehkan untuk membuat kode *assembly* sendiri.

5. Setiap kelompok diwajibkan untuk membuat tim dalam Github Classroom dengan **nama yang sama pada spreadsheet kelompok**. Mohon diperhatikan lagi cara penamaan kelompok agar bisa sama dengan nama repository Anda nanti.

6. Meskipun commit tidak dinilai, lakukanlah *commit* yang wajar dan sesuai *best practice* (tidak semua kode satu *commit*).

7. File yang harus terdapat pada *repository* adalah file-file *source code* dan *script* (jika ada) sedemikian rupa sehingga jika diunduh dari github dapat dijalankan. Dihimbau untuk tidak memasukkan *binary* dan *temporary intermediate files* hasil kompilasi ke *repository* (manfaatkan **.gitignore**).

8. Kelompok tetap sama dan dapat dilihat pada link berikut.

9. **Mulai** Rabu, 6 April 2022, 16.00 WIB waktu server.
   Deadline Sabtu, 30 April 2022, 23.59 WIB waktu server.
   Setelah lewat waktu *deadline*, perubahan kode akan dikenakan pengurangan nilai.

10. Pengumpulan dilakukan dengan membuat ***release*** dengan tag **v.3.0.0** pada repository yang telah kelompok Anda buat sebelum deadline. Pastikan tag sesuai format.
    Repository team yang tidak memiliki tag ini akan dianggap tidak mengumpulkan Milestone 3.

11. Teknis pengumpulan adalah via kode yang terdapat di *repository* saat *deadline.* Kami akan menindaklanjuti **segala bentuk kecurangan.**

12. Diharapkan untuk mencoba mengerjakan tugas besar ini terlebih dahulu sebelum

mencari sumber inspirasi dari *google*, *repository*, atau teman yang sudah selesai. Namun **dilarang melakukan *copy-paste* langsung** kode orang lain (selain spesifikasi). *Copy-paste* kode secara langsung akan dianggap melakukan kecurangan.

13. Dilarang melakukan kecurangan lain yang merugikan peserta mata kuliah IF2230.
14. Jika ada pertanyaan atau masalah pengerjaan harap segera menggunakan sheets QnA mata kuliah IF2230 Sistem Operasi pada link berikut.

VI. Tips dan Catatan

1. bcc tidak menyediakan *check* sebanyak gcc sehingga ada kemungkinan kode yang Anda buat berhasil *compile* tapi *error*. Untuk mengecek bisa mengcompile dahulu dengan gcc dan melihat apakah *error*.

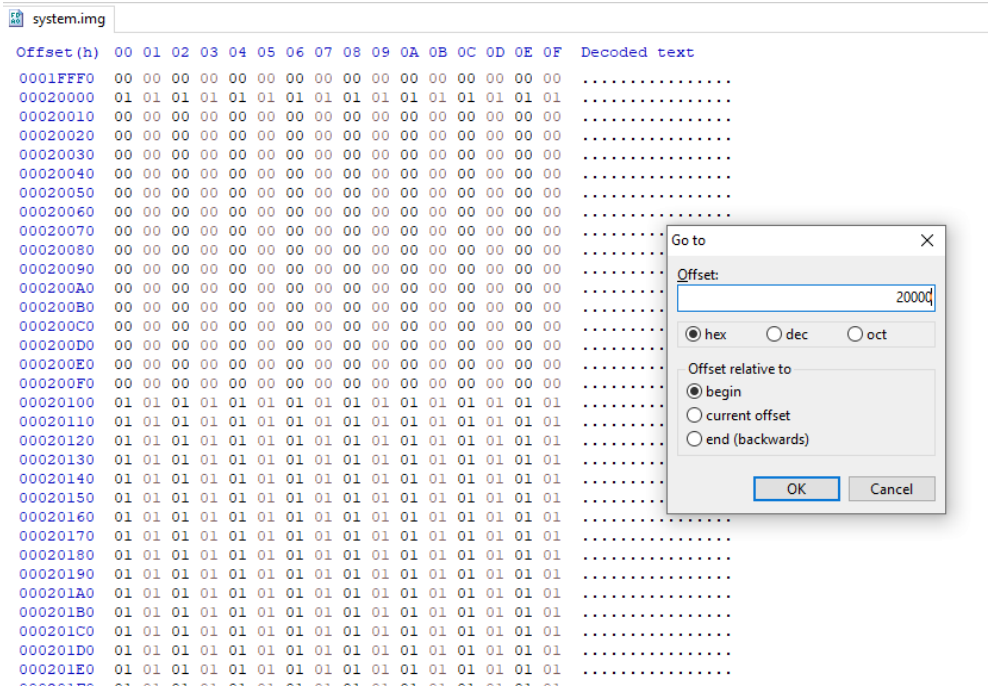
2. Pastikan flag -d selalu ada ketika menjalankan perintah ld86. Flag tersebut membuat ld86 menghapus header pada output file.

3. Konsekuensi dari flag -d adalah **urutan definisi fungsi berpengaruh terhadap program**. Pastikan definisi fungsi pertama kali merupakan `int main()`. Deklarasikan fungsi selain `main()` pada header atau bagian atas program jika ingin memanggil fungsi lain pada `main()`.

4. Compiler bcc dengan opsi ANSI C hanya memperbolehkan deklarasi variabel pada awal scope. Deklarasi variabel pada tengah kode akan mengakibatkan error.

5. Spesifikasi **tidak** mewajibkan untuk *menghandle* seluruh *edge case* dan *abuse case*. Namun *jika memiliki waktu tambahan*, direkomendasikan untuk menambah *handler*.

6. Untuk melihat isi dari *disk* bisa digunakan utilitas hexedit untuk Linux dan HxD untuk Windows. Sektor **map** adalah 0x100, **node** 0x101 & 0x102, dan **sector** 0x103 sehingga nilai offset byte untuk masing-masing filesystem dapat dikalkulasikan dengan perkalian 0x200 (ex. **map** terletak pada byte offset 0x100*0x200 = 0x20000). Pada HxD dan hexedit dapat digunakan CTRL + G untuk melakukan lompat ke offset seperti berikut



7. Walaupun kerapihan tidak dinilai langsung, kode yang rapi akan sangat membantu saat *debugging*.

8. Fungsi-fungsi dari *stdc* yang biasa Anda gunakan seperti **strlen** dan lainnya tidak tersedia di sini. Jika anda mau menggunakannya, anda harus membuatnya sendiri. Direkomendasikan untuk melengkapi definisi operasi-operasi umum pada *std_lib.c*

9. Tahap *debugging* secara *dynamic* dapat menggunakan debugger yang disediakan *bochs* (memiliki fitur *gdb* dasar), menggunakan kondisional `if (a == b) printString("ok\n");` untuk *sanity check*, dan mengimplementasikan `printf()` jika diperlukan. Gunakan *hex editor* untuk melakukan *debugging* pada *storage*.

10. Untuk penjelasan tentang *x86 memory addresing* menggunakan *segment register* telah dijelaskan pada referensi tambahan milestone 1. Informasi mengenai *layout memory* dapat dibaca pada OSDev. Segment `0x0000` digunakan untuk menyimpan *interrupt vector table*. Bootloader memasukkan dan menggunakan segment `0x1000` untuk kernel (konstanta KSEG pada asm). Untuk mencegah permasalahan disebabkan oleh `putInMemory()` yang meng-overwrite *current code segment* / instruksi yang sedang dieksekusi sekarang, gunakan segment yang berbeda dengan segment yang digunakan sekarang. Contohnya kernel berada dan berjalan pada segment `0x1000` akan mengeksekusi shell pada segment `0x2000`.

11. Pada *real mode* terdapat 4 segment register yang dapat digunakan. Segment register diasosiasikan dengan beberapa register sebagai offset. Singkatnya **SS** menjadi segment offset untuk register **BP** dan **SP**. **DS** dan **ES** untuk offset **SI** dan **DI**. Dan segment register **CS** untuk **IP**. Khusus untuk segment register **CS** hanya dapat dimanipulasi oleh instruksi seperti *far jump*, *far call*, *far ret*, *int*, dan beberapa instruksi lain. Detail dari setiap register dapat dibaca pada wikibooks. Fungsi `launchProgram()` "*mengosongkan*" register dan mempersiapkan segmen register ke target sebelum melakukan *jump*. *Message passing* dapat menggunakan proses "pengosongan" untuk memanipulasi register & memory sebelum melakukan *jump*.

12. Sistem yang ditargetkan tugas besar sistem operasi adalah *x86 16-bit*. Pastikan hanya menggunakan *register* dan instruksi 16 bit pada kode *assembly*.

13. Ingat, *assembly* hanya memiliki instruksi tertentu yang menggunakan operan register dan memory yang spesifik. Tidak seperti *high level language* yang membebaskan *assignment operator* ke sembarang variabel, instruksi x86 **mov** juga memiliki keterbatasan seperti instruksi **mov** tidak dapat memindahkan konstanta ke *segment register* secara langsung (ex. `mov ds, 0x2000`) sehingga terlihat pada `launchProgram()`, `bootloader`, dan kode *assembly* lain menggunakan *general purpose register* (**AX**, **BX**, **CX**, **DX**) sebagai penengah sebelum memindahkan ke *segment register*.

14. Secara teknis mode operasi *Real mode x86* tidak menyediakan fitur seperti *virtual memory, multitasking,* dan *privilege user/kernel mode*. Seluruh program pada *Real mode* memiliki akses *r/w* seluruh memori secara langsung dan tidak memiliki perbedaan antara *user* dan *kernel space program*. Milestone ini hanya mensimulasikan *user-kernel space*

pada sistem operasi yang dibuat. Jika menginginkan untuk implementasi yang lebih nyata, anda dapat membuat tugas besar ini dalam *protected mode* (32-bit) atau bahkan *long mode* (64-bit).

15. Perhatikan bahwa *real mode* tidak memiliki layanan proteksi memori, sebuah program dapat memanipulasi seluruh memori sistem. Direkomendasikan untuk tidak melakukan *overwrite* pada segment yang sama dengan *code segment (register cs)* yang sedang dijalankan. Manipulasi memori instruksi yang tidak tepat dapat menyebabkan sistem operasi *hang*.

16. Fun fact : Pada *tc_lib* terdapat tiga fungsi, `writer()`, `create_folder()`, dan `insert_file()`. `insert_file()` dan `create_folder()` *wrapper kecil* `writer()`. Fungsi `writer()` sendiri adalah implementasi asli dari *pseudocode system call write* pada milestone 2.

VII. Referensi

1. Spesifikasi Tugas Besar IF2230 - Sistem Operasi - Milestone 1
2. Spesifikasi Tugas Besar IF2230 - Sistem Operasi - Milestone 2
3. https://en.wikipedia.org/wiki/X86
4. https://wiki.osdev.org/Memory_Map_(x86)
5. https://en.wikipedia.org/wiki/Interrupt_vector_table
6. https://cs.lmu.edu/~ray/notes/nasmtutorial/ (Catatan : Sistem operasi yang dibuat adalah 16 bit)
7. https://stackoverflow.com/questions/40669332/what-is-the-exact-behaviour-of-int-instruction
8. http://www.c-jump.com/CIS77/ASM/Memory/lecture.html x86 Real mode memory addressing
9. https://en.wikibooks.org/wiki/X86_Assembly/16,_32,_and_64_Bits
10. http://hilite.me/ (Pretty print untuk source code)
11. https://ascii-tree-generator.com/ (Pembuatan ilustrasi direktori)