Tugas Besar
IF2230 - Sistem Operasi
Milestone 02 of ??
"Bingung ya?"
Pembuatan FIlesystem dan Shell Sederhana

Dipersiapkan oleh :
Asisten Lab Sistem Terdistribusi

Didukung Oleh :

Waktu Mulai :
Rabu, 16 Maret 2022, 16.00 WIB

Waktu Akhir :
Selasa, 5 April 2022, 23.59 WIB

I. Latar Belakang

Waktu demi waktu telah berlalu, tanpa terasa 16 jam berlalu dengan sangat cepat. Kaiba menyatakan proses interview ini telah berakhir dan memberikan ucapan terima kasih kepada seluruh peserta. Tampak ruangan meet ini dipenuhi oleh teman - teman Anda yang sudah sangat kelelahan dalam menjalani proses interview ini. Disisi lain, nampaknya Kaiba sedang melihat lihat seluruh hasil yang telah dikerjakan oleh setiap peserta. Tiba - tiba ia berkata

"Baiklah, mari kita coba sistem operasi buatanmu!"

Jika kalian berhasil menyelesaikan seluruh spesifikasi, silahkan baca bagian ini

Anda merasa sangat lelah dengan semua yang telah hal ini, namun Anda berhasil menyelesaikan sistem operasi dengan sangat baik. Anda mulai mencoba dan mengamati sistem operasi sederhana yang telah Anda buat dengan bangga. Tiba tiba terdengar sebuah kalimat

"Baiklah, mari kita coba sistem operasi buatanmu!" Yang ditunjukkan kepada teman Anda.

Ada sedikit perasaan kecewa karena bukan sistem operasi Anda yang akan dicoba. Anda dapat melihat raut wajah teman Anda yang seketika berubah menjadi pucat pasi dan berkeringat dingin. Dalam hati Anda bertanya tanya apa yang salah, "Apakah dia terkena tipes?" tanya Anda pelan dalam hati.

Silahkan lanjut ke BAGIAN 2

Jika kalian belum berhasil menyelesaikan seluruh spesifikasi, silahkan baca bagian ini

Terdengar banyak keluhan, tangisan, dan air mata yang bercucuran dari berbagai peserta. Tentu saja Anda bisa mengerti dan merasakan bagaimana perasaan khawatir dan tertekan dari teman teman Anda disana. Tiba tiba terdengar sebuah kalimat

"Baiklah, mari kita coba sistem operasi buatanmu!" Yang ditunjukkan kepada Anda

Seketika Anda mulai mengucurkan keringat dingin mengingat sistem operasi yang Anda kerjakan belum sepenuhnya selesai, Anda bisa merasakan dinginnya seluruh tubuh Anda secara tiba tiba. Dalam hati Anda mulai berpikir "Apakah aku terkena tipes?". Secara perlahan lahan dan lesu Anda mulai menunjukkan repository Gitlab kepada Kaiba.

Sejauh ini Anda belum berhasil menyelesaikan spesifikasi yang telah diberikan, wajar saja. Interview ini sulit dan menantang, tapi itu bukan artinya untuk menyerah.

"Kegagalan tidak terjadi disaat Anda tidak menyelesaikan seluruh spesifikasi,
kegagalan terjadi disaat Anda menyerah dalam menyelesaikan seluruh spesifikasi."

Silahkan lanjut ke BAGIAN 2

BAGIAN 2

Duel disk yang menyala

Kaiba mulai mengamati dan mencoba sistem operasi yang telah dipilihnya secara acak tersebut, ia mulai menjalankan perintah clone pada repository git dan menjalankannya pada duel disk yang telah disiapkan oleh rekan kerja di sebelahnya. Tanpa disadari, duel disk tersebut menyala merah memancarkan cahaya, semakin lama cahaya tersebut semakin terang dan juga semakin panas. Hingga terdengar ledakan

*** DUA AARRR ***

Sekelebat cahaya dan suara yang keras memenuhi ruangan meet tersebut, Anda bisa melihat layar Kaiba menjadi putih sepenuhnya untuk beberapa saat. Setelah beberapa saat keadaan mulai terkendali, Anda mulai menyadari duel disk yang ia gunakan meledak beberapa saat setelah menjalankan OS tersebut. Tampak tubuh Kaiba yang terlentang di lantai, langsung saja rekan rekan kerjanya melarikannya ke rumah sakit di Korea dengan menggunakan jet pribadinya. Anda tidak dapat mempercayai apa saja yang baru terjadi, semua hal terjadi dengan sangat cepat dan tampak tidak nyata bagi Anda. Terjadi keheningan untuk beberapa saat. Tidak ada orang di meet ini yang tahu apa yang harus dilakukan.

Petinggi Kaiba Corp (SISTER)
"Jangan percaya, mereka semua hode"

Hingga akhirnya sekumpulan petinggi Kaiba Corp yang disebut dengan SISTER datang dalam meet dan menyatakan seluruh orang yang ada disini diterima untuk bekerja dan mengharapkan mereka untuk mengembangkan sistem operasi duel disk. Ternyata semua orang yang ada disini sudah diterima untuk bekerja sedari awal, ternyata interview ini hanyalah proses untuk menanamkan mentalitas baja kepada seluruh peserta. Mereka mulai mengeluarkan spesifikasi baru yang harus dikerjakan dengan rentang waktu 1 bulan. Para petinggi ini mengharapkan progress dan hasil dalam pengembangan sistem operasi ini. Selamat bagi mereka yang telah menyelesaikan spesifikasi dengan baik, dan bagi mereka yang belum menyelesaikannya silahkan diselesaikan dan melanjutkan milestone 2, untuk selanjutnya diharapkan OS tidak meledak saat dijalankan.

Sedikit pesan dari para petinggi:
Jika kalian tidak mengerti tentang mata kuliah ini, itu tidak masalah. Beberapa tahun lagi kalian akan melupakannya, tetapi ~~penderitaan~~ **daya juang** dan **mentalitas pantang menyerah** yang kalian bangun disini akan kalian kenang seumur hidup.

II. Deskripsi Tugas

Pada milestone ini, kalian akan melanjutkan sistem operasi yang sudah kalian buat dengan menambahkan *filesystem* yang akan mendukung adanya direktori dan sebuah *shell* sederhana. Hal-hal yang akan Anda lakukan pada milestone ini adalah

- Memahami ***filesystem*** sederhana yang fungsional.
- Membuat *syscall* **readSector**, **writeSector**, **read**, dan **write**.
- Membuat sebuah **shell** sederhana.
- Membuat utility kecil pada shell : **cd**, **ls**, **mkdir, cat**, **cp, mv**.

Untuk membantu proses pengerjaan milestone ini, kit milestone kedua akan disediakan oleh SISTER.

III. Langkah Pengerjaan

3.1. *Filesystem*

Filesystem dari sistem operasi yang akan dibuat terdiri atas 3 filesystem yang telah dialokasikan sejak awal, yaitu filesystem **map** pada sector offset 0x100, filesystem **node** pada sector offset 0x101-0x102, dan filesystem **sector** pada sector offset 0x103. Seluruh file akan diproses dengan partisi berukuran 1 sektor (1 *sektor* ekuivalen dengan 512 *bytes)*, file yang tidak dalam kelipatan 512 bytes akan dilakukan **pembulatan keatas** ke kelipatan 512 terdekat.

Bagian 3.1 akan digunakan sebagai penjelasan terhadap filesystem yang akan diimplementasikan. Lompat ke bagian 3.2 untuk bagian implementasi.

Kit milestone kedua akan menyediakan struktur data untuk filesystem. Struktur data terletak pada header *filesystem.h*. Berikut adalah ketiga deskripsi singkat terkait spesifikasi ketiga filesystem

- Filesystem **map** terdiri atas 512 bytes dimana masing-masing byte berfungsi seperti entri dan menandakan status sektor terisi atau belum terisi pada filesystem.
- Filesystem **node** terdiri atas 64 entri yang setiap entrinya memiliki ukuran 16 bytes.
- Filesystem **sector** terdiri dari 32 entri dan setiap entri pada filesystem **sector** berukuran 16 bytes.

3.1.1. Map

Filesystem **map** terdiri dari 512 bytes yang setiap byte akan berfungsi layaknya entri. Setiap byte akan menunjukan status keterisian *physical drive*. Indeks byte akan merepresentasikan status keterisian sektor dengan indeks yang sama, contohnya byte ke-73 menyimpan status keterisian sektor ke-73. Filesystem ini akan digunakan pada proses *writing* untuk mengetahui lokasi *empty space* dan lokasi *space* yang terisi.

Struktur dari filesystem **map**:

Filesystem **Map**

Byte Offset	0	1	2	3
0	Sektor 0	Sektor 1	Sektor 2	Sektor 3
4	Sektor 4	Sektor 5	Sektor 6	Sektor 7
8	Sektor 8	Sektor 9	Sektor 10	Sektor 11
12	Sektor 12	Sektor 13	Sektor 14	Sektor 15
...	...	...	...	...

Perhatikan bahwa bytes-bytes tersebut hanya digunakan untuk **status sektor terisi atau tidak**, bukan tempat penyimpanan fisik data dalam file. Nantinya setiap byte akan digunakan sebagai tipe data boolean.

Contoh Filesystem **Map** yang terisi

Byte Offset	0	1	2	3
0	true	true	true	true
4	true	true	true	true
8	true	true	true	true
12	true	true	true	true
16	true	false	true	true
20	false	false	true	false
...	...	...	...	...

3.1.2. Node

Filesystem **node** terdiri atas 2 sektor berukuran 2x512 bytes yang akan menampung informasi dari file dan direktori yang ada pada filesystem kalian. Satu entry pada filesystem berukuran 16 bytes, yang terbagi seperti berikut

	idx	P	S	Nama Node												
	37	10	15	O	S	_	H	e	h	e	\0	\0	\0	\0	\0	\0	\0
Byte ke-		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- 1 byte pointer / penunjuk parent index directory (byte **P**)
- 1 byte pointer / penunjuk index entri pada filesystem **sector** (byte **S**)
- 14 bytes untuk nama file (13 karakter dengan menggunakan *null terminated string*)

Pada diagram contoh entri filesystem **node**, entri tersebut memiliki indeks **ke-37** pada filesystem **node**. Entri tersebut terletak pada suatu folder yang terletak pada indeks **ke-10** pada filesystem **node**. Entri ini adalah suatu file dikarenakan menunjuk pada indeks filesystem **sector** yang valid (rentang nilai valid 0 hingga 32), informasi partisi file ini terdapat pada indeks **ke-15** pada filesystem **sector**. File memiliki nama “OS_Hehe”, *null terminator* pada 7 byte sisa dapat diabaikan.

Filesystem ini akan memberi dukungan untuk folder. Entri folder pada filesystem **node** akan memiliki *flag* khusus yang menandai bahwa entri tersebut adalah sebuah folder. Perhatikan bahwa filesystem **sector** hanya memiliki ukuran 32 entri sehingga *flag* folder dapat ditaruh pada byte **S** dengan nilai yang tidak ada pada filesystem **sector**. Detail *flag* akan dibahas lebih lanjut pada bagian interaksi filesystem.

Struktur dari filesystem **node**:

Filesystem Node

Byte offset	idx	P	S	Nama Node
0	0	P_0	S_0	Nama node 0
16	1	P_1	S_1	Nama node 1
32	2	P_2	S_2	Nama node 2
...	...	...		
...	...	...		
992	62	P_{62}	S_{62}	Nama node 62
1008	63	P_{63}	S_{63}	Nama node 63

P_i : Indeks parent node dari node ke-i

S_i : Indeks entri filesystem **sector**

3.1.3. Sector

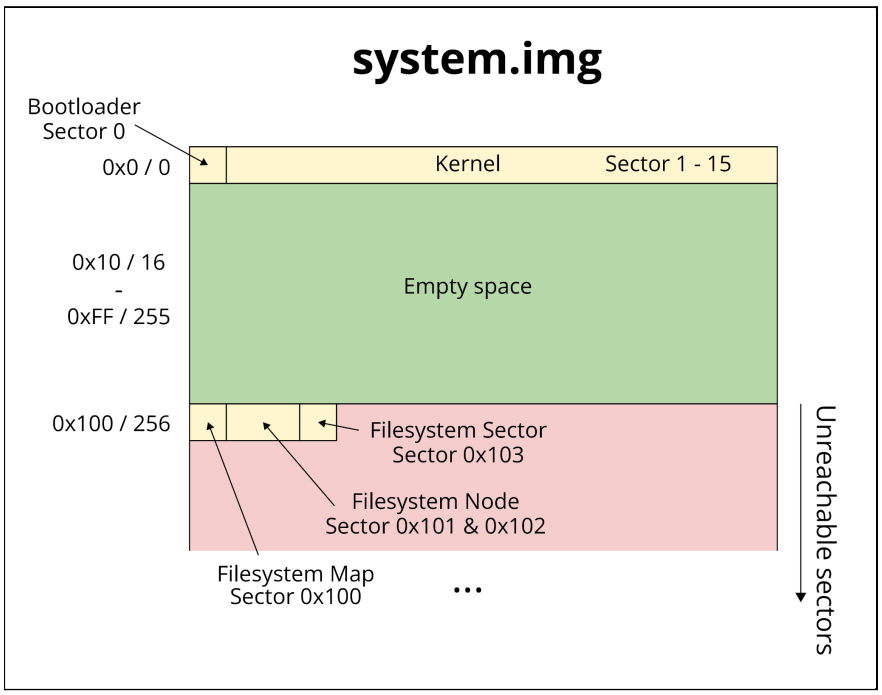
Filesystem **sector** berisi 512 bytes yang akan terbagi dengan entri berukuran 16 bytes. Setiap entri akan menyimpan informasi partisi suatu file. Setiap byte pada entri menunjukkan lokasi sektor dari partisi file. Berikut adalah contoh entri pada filesystem **sector**

idx	Entri sector															
6	23	42	32	12	13	14	0	0	0	0	0	0	0	0	0	0
Byte ke-	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pada contoh, entri memiliki indeks **ke-6** pada filesystem **sector**. File tersebut terpartisi menjadi 6 bagian yang setiap bagiannya ditunjuk oleh byte-byte pada entri. Partisi pertama terdapat

pada sektor **23**, partisi kedua pada sektor **42**, partisi ketiga pada sektor **32**, partisi keempat pada sektor **12**, partisi kelima pada sektor **13**, dan partisi terakhir pada sektor **14**.

Perhatikan bahwa satu byte hanya memiliki rentang nilai 0 hingga 255 (untuk *unsigned char*) sehingga filesystem ini **tidak dapat menunjuk** kepada sektor yang lebih dari 255. Pada kit, ukuran maksimum kernel diatur secara default 15 sektor dan 1 sektor disediakan khusus untuk bootloader. Bootloader terletak pada sektor **0**. Lokasi awal kernel terdapat pada sektor **1** sehingga **16 sektor awal pada system.img reserved untuk sistem operasi**. Berikut adalah illustrasi untuk system.img



Agar sistem operasi mengetahui lokasi sektor mana saja yang masih kosong, sistem operasi perlu menandai pada filesystem **map**, penandaan **map** akan dibahas lebih lanjut setelah implementasi operasi **readSector** dan **writeSector**.

Berdasarkan fakta-fakta sebelumnya, rentang nilai pada setiap byte entri **sector** yang valid adalah **16 hingga 255**.

Struktur dari filesystem **sector**:

Filesystem Sector

idx	Entri Sector
0	Partisi File 0
1	Partisi File 1

2	Partisi File 2
...	...
...	...
30	Partisi File 30
31	Partisi File 31

3.1.4. Interaksi antar *Filesystem*

Ketiga filesystem akan digunakan pada proses eksekusi operasi **write** dan **read**, kecuali filesystem **map** tidak digunakan pada proses **read**. Agar filesystem dapat menyimpan folder dan node dapat ditaruh pada root, diperlukan konvensi untuk flag. Berikut adalah akan konvensi yang digunakan

- Byte **P** = 0xFF : Node terletak pada root.
- Byte **S** = 0xFF : Entri node adalah suatu folder.

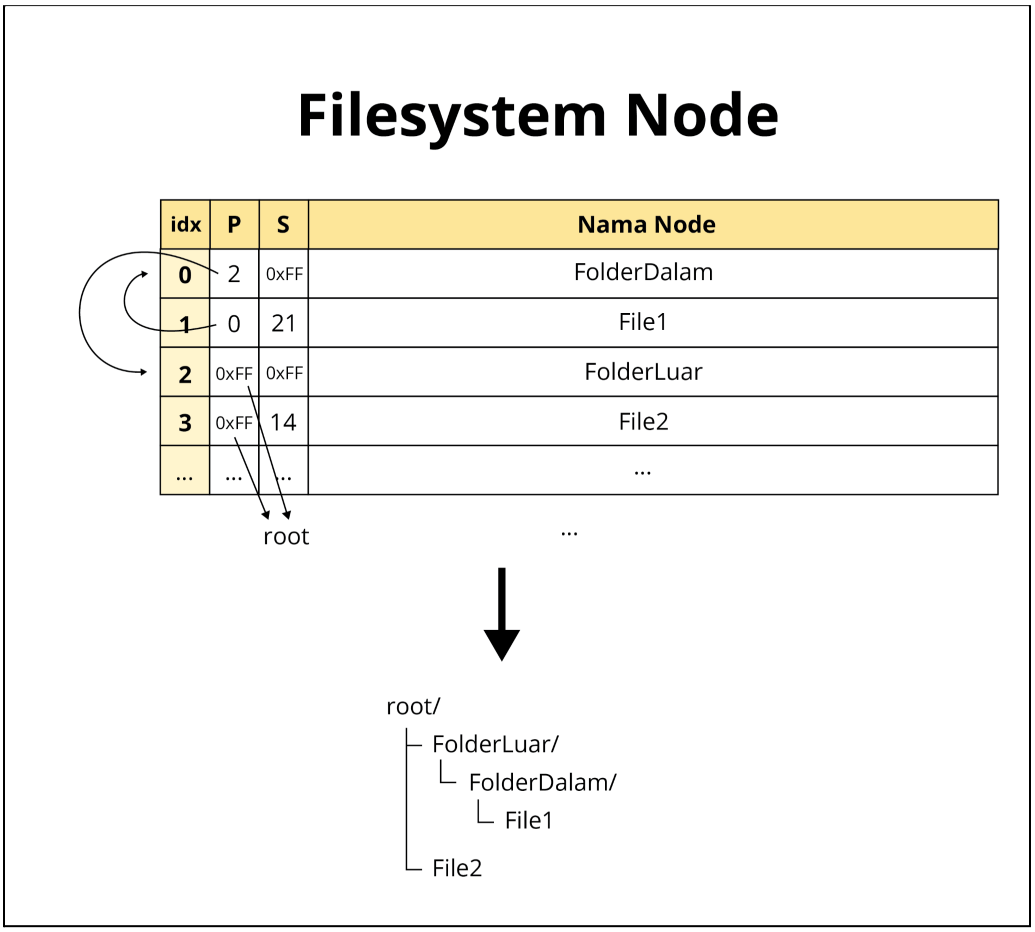
Kedua flag merupakan flag pada filesystem **node**. Selain konvensi flag untuk menandai hal khusus, diperlukan juga konvensi untuk entri kosong yang akan berguna untuk mengetahui bahwa suatu entri kosong atau entri yang valid. Berikut adalah konvensi entri kosong untuk ketiga filesystem

- Entry filesystem **map** tidak memiliki entri kosong, secara default seluruh entri bernilai **false**.
- Entry filesystem **node** kosong jika nama node adalah **string kosong** (panjang string 0).
- Entry filesystem **sector** kosong jika byte pertama (indeks 0) bernilai **0**.

Dengan konvensi entri kosong diatas, seluruh byte yang bernilai nol juga akan terkategorikan menjadi **filesystem yang seluruh entrinya kosong**. Pada awal pembuatan *image* sistem operasi seluruh byte *image* bernilai nol sehingga secara tidak langsung filesystem selain **map** telah terinisiasi oleh entri kosong. Hanya filesystem **map** yang perlu dilakukan inisiasi nilai awal untuk menjaga bootloader dan kernel tidak di-overwrite oleh operasi **write**. Inisiasi filesystem **map** akan dibahas setelah **readSector** dan **writeSector** telah diimplementasikan.

3.1.4.1. Illustrasi Direktori pada Filesystem Node

Agar lebih menggambarkan, berikut adalah beberapa illustrasi untuk interaksi antar filesystem yang dibuat. Berikut adalah illustrasi sebuah direktori yang ada pada filesystem **node**

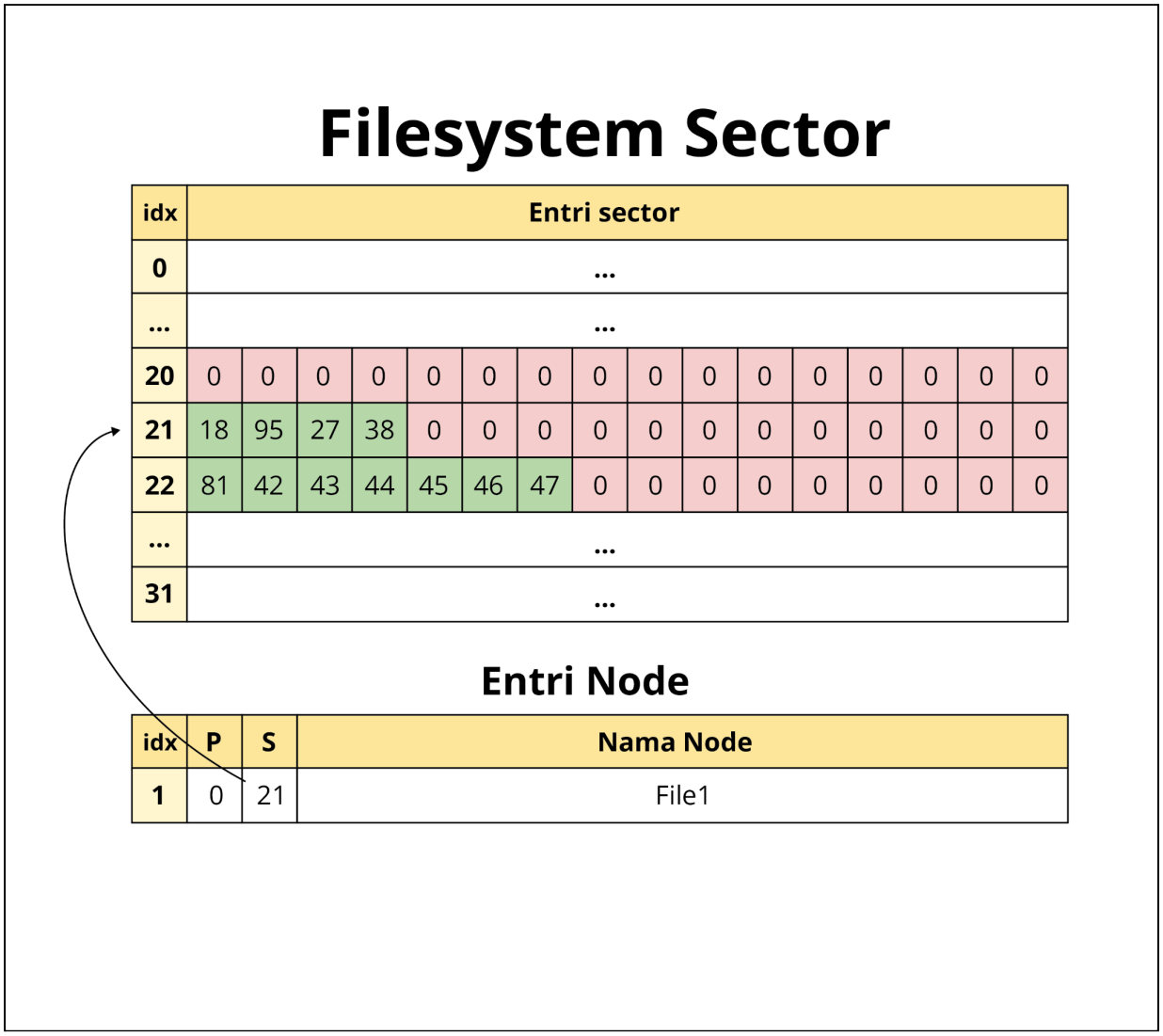


Pada illustrasi terdapat 2 file dan 2 folder pada filesystem **node**. *File2* terletak pada root sehingga indeks parent bernilai 0xFF. *FolderLuar* terletak root tetapi *FolderDalam* terletak didalam *FolderLuar*. Karena *FolderLuar* memiliki indeks ke-2 pada **node**, maka indeks parent *FolderDalam* bernilai **2**. Demikian juga *File1* yang terletak pada *FolderDalam* sehingga memiliki indeks parent bernilai **0**.

Perhatikan bahwa untuk byte **S** folder bernilai 0xFF dan file bernilai antara 0 hingga 31. Interaksi byte **S** dan filesystem **sector** akan diperlihatkan pada bagian selanjutnya.

3.1.4.2. Illustrasi Interaksi Filesystem Node dan Sector

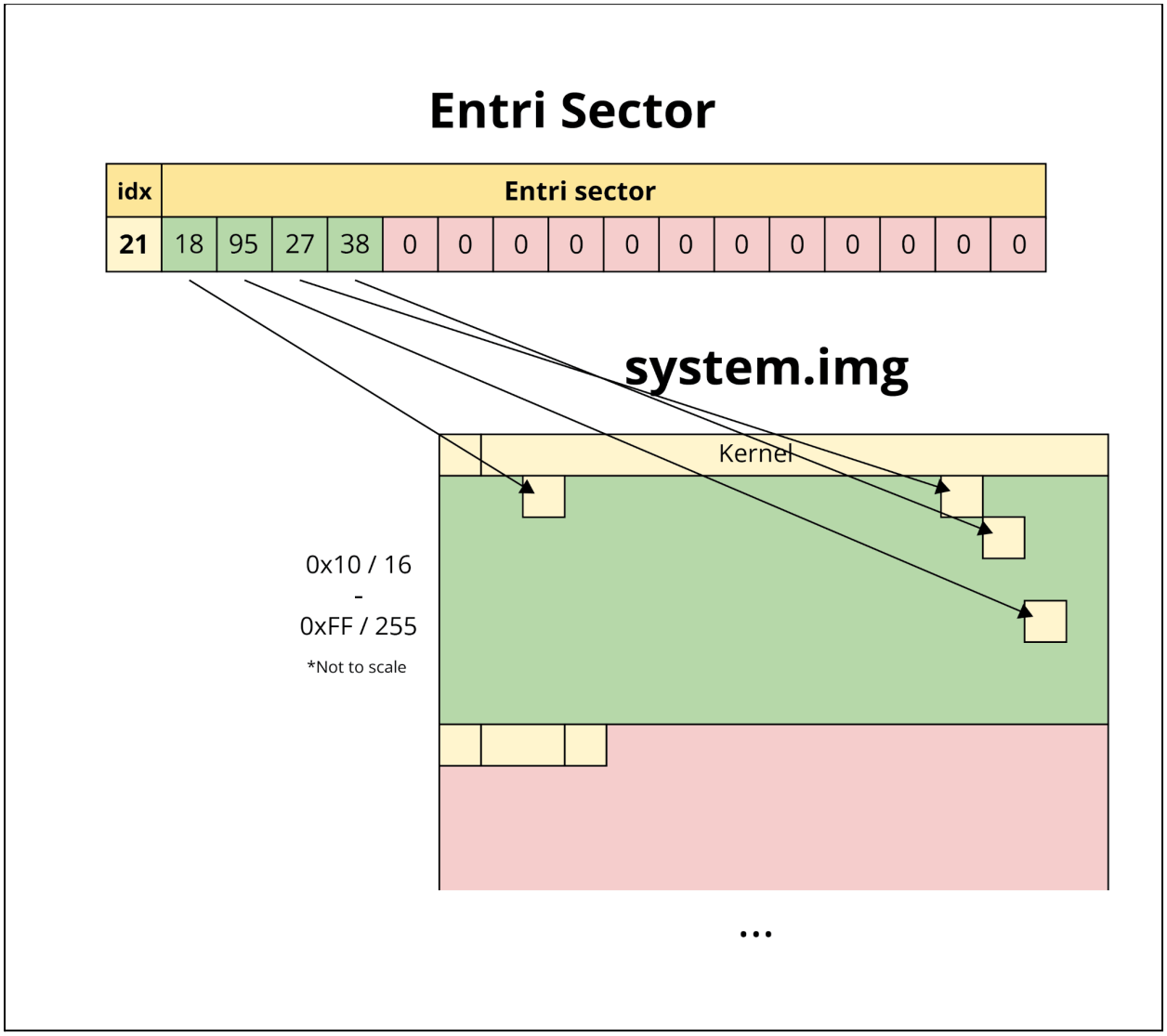
Berikut adalah illustrasi untuk menggambarkan interaksi byte **S** milik **node** dan filesystem **sector**



File1 memiliki indeks **sector** bernilai 21 sehingga informasi partisi file terdapat pada filesystem **sector** dengan indeks ke-21. Ingat, filesystem sector **tidak menyimpan isi file** tetapi hanya informasi indeks-indeks partisi sektor dari file. Informasi indeks-indeks tersebut akan digunakan untuk menunjuk kepada tempat penyimpanan, dalam kasus ini adalah **system.img**.

3.1.4.3. Illustrasi Interaksi Filesystem Sector dan system.img

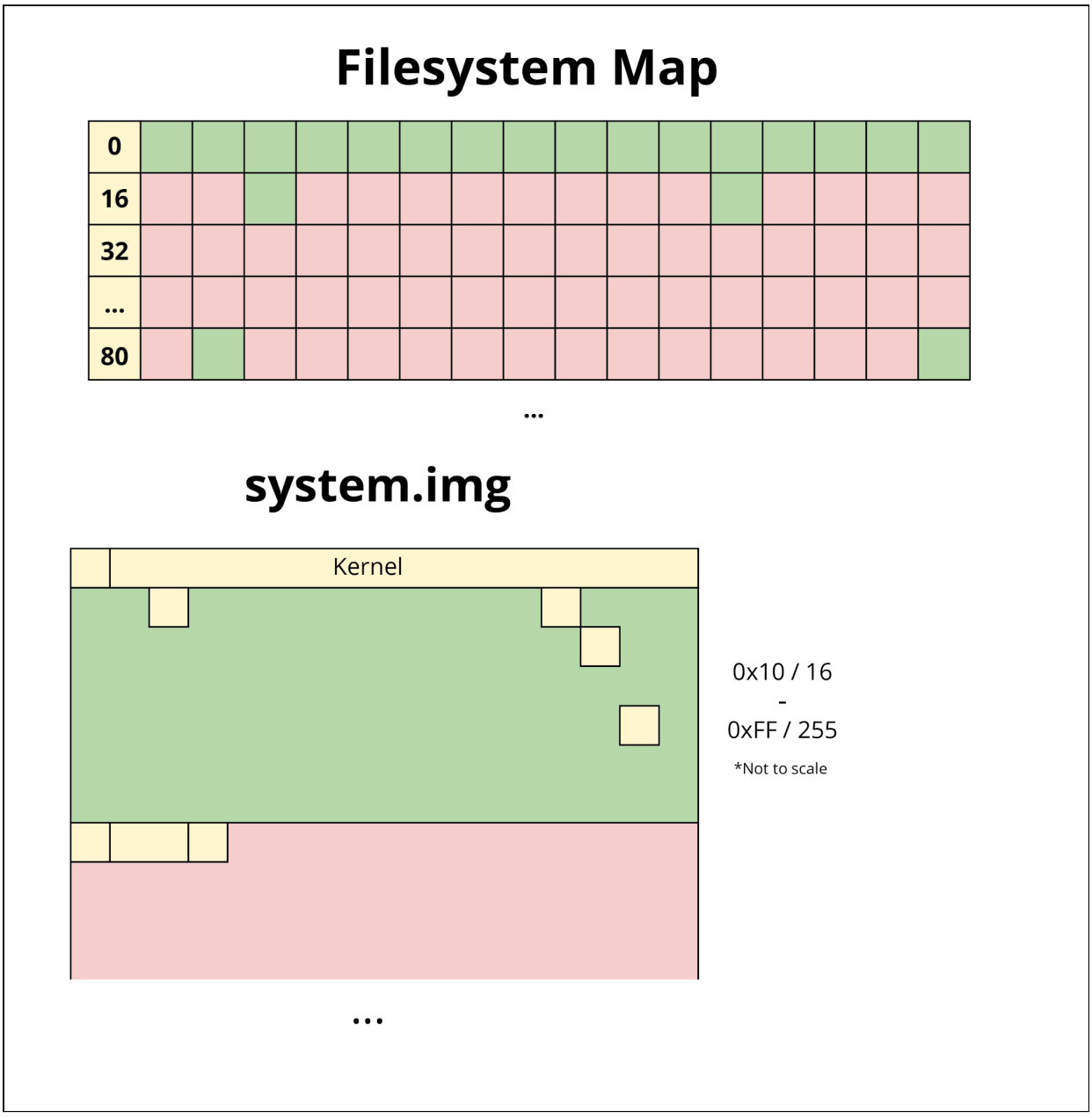
Berikut adalah illustrasi bagaimana indeks-indeks pada filesystem **sector** digunakan untuk mengambil kembali isi file



Setiap byte pada entri **sector** merupakan angka sektor keberapa isi file ditulis. Untuk proses pembacaan, sektor-sektor tersebut dapat dibaca langsung dan dimasukkan ke dalam buffer.

3.1.4.4. Illustrasi Filesystem Map

Filesystem **map** akan digunakan untuk menandai keterisian suatu sektor. Berikut adalah illustrasi lanjutan untuk filesystem **map** yang terisi



Pada illustrasi **map**, blok yang berwarna merah berarti boolean pada lokasi tersebut bernilai **false** dan warna hijau menunjukkan nilai **true**. Pada illustrasi interaksi **sector** dan **node** sektor 18, 95, 27, 38, 42 hingga 47, dan 81 telah terisi sehingga **map** perlu ditandai dengan terisi. Terlihat pada illustrasi byte ke-18, 95, 27, 38, dan 81 telah ditandai true.

3.2. Pembuatan *syscall readSector, writeSector, read* dan *write*

Sebelum melanjutkan pada bagian ini, pastikan kit untuk struktur data filesystem (*filesystem.h*) telah dimasukkan kedalam sistem operasi. Berikut adalah list struktur data dan macro yang disediakan

- FS_<Filesystem>_SECTOR_NUMBER
  Menunjukkan lokasi sektor filesystem.
- FS_NODE_P_IDX_ROOT
  Isi byte **P** node yang terletak pada root.
- FS_NODE_S_IDX_FOLDER
  Isi byte **S** node yang merupakan folder.
- **struct** map_filesystem, **struct** node_filesystem, **struct** sector_filesystem
  Digunakan untuk merepresentasikan filesystem pada kode.
- **struct** node_entry, **struct** sector_entry
  Digunakan untuk merepresentasikan entri setiap filesystem terkait.
- **struct** file_metadata
  Menyimpan informasi terkait file untuk keperluan operasi **read** atau **write**.
- **enum** fs_retcode
  Sebagai return code untuk operasi filesystem sistem operasi.

Jika dibutuhkan, lakukan modifikasi pada *filesystem.h* sesuai dengan kebutuhan.

3.2.1. Implementasi *readSector* dan *writeSector*

Implementasikan `void readSector()` dengan kode berikut

```
void readSector(byte *buffer, int sector_number) {
    int sector_read_count = 0x01;
    int cylinder, sector;
    int head, drive;

    cylinder = div(sector_number, 36) << 8; // CH
    sector   = mod(sector_number, 18) + 1;  // CL

    head  = mod(div(sector_number, 18), 2) << 8; // DH
    drive = 0x00;                                // DL

    interrupt(
        0x13,                       // Interrupt number
        0x0200 | sector_read_count, // AX
```

```
        buffer,                     // BX
        cylinder | sector,          // CX
        head | drive                // DX
    );
}
```

Perhatikan bahwa kode diatas menggunakan operasi mod dan div pada *std_lib.c*. Fungsi ini harus kalian implementasikan sendiri. *Interrupt 0x13* dapat digunakan untuk berbagai macam hal, tetapi pada tugas ini hanya akan kita gunakan untuk membaca dan menulis sector. Interrupt 0x13 tersebut memiliki beberapa argumen yang diletakkan register **AX**, **BX**, **CX**, dan **DX**. Berikut adalah penjelasan singkat dari INT 13h untuk fungsi 02h

1. Register AX dibagi menjadi 2 bagian.

   AH digunakan untuk memilih fungsi yang akan digunakan, pada kasus ini fungsi untuk membaca sector adalah `0x02`.

   AL digunakan untuk jumlah sektor yang akan dibaca, pada kasus ini hanya diperlukan `0x01` sektor setiap pembacaan.

2. Register BX adalah pointer buffer untuk penyalinan isi sektor. Tambahan, lebih akuratnya register yang digunakan adalah **ES:BX**.

3. Register CX digunakan untuk *cylinder* dan *sector.*

   CH digunakan untuk *cylinder, cylinder* yang dimaksud adalah nomor *cylinder* pada satu *track.* Karena 1 *cylinder* memiliki 2 *head* yang masing-masing ada 18 *sector* dengan total 36 *sector* per *cylinder*, pada kode di atas *cylinder* diisi dengan `div(sector_number, 36) << 8`, yakni merujuk ke lokasi cylinder tempat sector yang dicari berada. *Left bitwise shift* sebanyak 8 digunakan untuk menggeser nilai ke register CH.

   CL digunakan untuk *sector*, *sector* yang dimaksud adalah nomor *sector* pada satu *track*. Karena pada floppy, 1 *track* memiliki 18 *sector*, maka pada kode di atas, *sector* diisi dengan `mod(sector_number, 18) + 1` (sector dimulai dari 1 jika mengikuti konvensi penghitungan BIOS INT 13h).

4. Register DX adalah *head* dan *drive*.

   AH digunakan untuk *head*. Nilai *head* adalah `mod(div(sector_number, 18), 2) << 8`. Operasi mod dilakukan karena *head* hanya dapat bernilai 0 atau 1 dengan masing-masing memiliki 18 *sector*.

   AL digunakan untuk *drive*. Untuk *drive* nya adalah `0x00` (drive A:).

Lalu implementasikan `void writeSector()` yang mirip seperti `void readSector()`, tetapi

fungsi yang digunakan adalah `0x03` (Argumen kedua interrupt menjadi `0x0301`).

Catatan : Pastikan buffer cukup untuk menampung data (512 bytes) dari operasi `writeSector()` atau `readSector()`.

Untuk mencegah operasi **write** menulis pada kernel dan menghitung *unreachable sector* sebagai *empty space*, sektor-sektor tersebut perlu ditandai pada filesystem **map** sebagai terisi. Tandai sektor **ke-0 hingga 15** dan sektor **ke-256 hingga 511** terisi. Pastikan untuk tidak merubah sektor selain yang disebutkan.

```
void fillMap() {
  struct map_filesystem map_fs_buffer;
  int i;

  // Load filesystem map
  readSector(&map_fs_buffer, FS_MAP_SECTOR_NUMBER);

  /*
    Edit filesystem map disini
                              */

  // Update filesystem map
  writeSector(&map_fs_buffer, FS_MAP_SECTOR_NUMBER);
}
```

Tambahkan juga kode berikut pada fungsi **main** kernel agar `fillMap()` selalu dieksekusi setiap sistem operasi berjalan.

```
int main() {
    fillMap();
    makeInterrupt21();
    clearScreen();

    while (true);
}
```

3.2.2. Implementasi *read*

Spesifikasi wajib syscall **read / write hanya pembacaan / penulisan file dengan baik dan return code yang sesuai**. Penjelasan berikut dapat digunakan sebagai referensi

pengerjaan, detail implementasi dibebaskan kepada *pemagang*.

Syscall **read** akan digunakan untuk melakukan pembacaan pada filesystem. **read** akan mengembalikan return code berdasarkan proses pembacaan, baik sukses, gagal, maupun alasan lain. Berikut adalah *signature* dari **read**

```
void read(struct file_metadata *metadata, enum fs_retcode *return_code);
```

Syscall ini dapat membaca file maupun folder, jika tujuan yang dibaca adalah folder akan dikembalikan return code FS_R_TYPE_IS_FOLDER. Jika pembacaan file berhasil akan dikembalikan return code FS_SUCCESS. Berkaitan dengan parameter **read**, berikut adalah definisi dari struktur data metadata

```
struct file_metadata {
    byte*        buffer;
    char*        node_name;
    byte         parent_index;
    unsigned int filesize;
};
```

Syscall **read** hanya menggunakan variabel `buffer`, `node_name`, dan `parent_index` untuk mencari suatu entri pada filesystem **node**. Variabel `filesize` digunakan untuk mengetahui ukuran hasil pembacaan pada syscall ini.

Berikut adalah pseudocode dari operasi **read**

```
void read(struct file_metadata *metadata, enum fs_retcode *return_code) {
  struct node_filesystem   node_fs_buffer;
  struct sector_filesystem sector_fs_buffer;
  // Tambahkan tipe data yang dibutuhkan

  // Masukkan filesystem dari storage ke memori buffer

  // 1. Cari node dengan nama dan lokasi yang sama pada filesystem.
  //    Jika ditemukan node yang cocok, lanjutkan ke langkah ke-2.
  //    Jika tidak ditemukan kecocokan, tuliskan retcode FS_R_NODE_NOT_FOUND
  //    dan keluar.

  // 2. Cek tipe node yang ditemukan
  //    Jika tipe node adalah file, lakukan proses pembacaan.
  //    Jika tipe node adalah folder, tuliskan retcode FS_R_TYPE_IS_FOLDER
  //    dan keluar.

  // Pembacaan
  // 1. memcpy() entry sector sesuai dengan byte S
  // 2. Lakukan iterasi proses berikut, i = 0..15
  // 3. Baca byte entry sector untuk mendapatkan sector number partisi file
  // 4. Jika byte bernilai 0, selesaikan iterasi
  // 5. Jika byte valid, lakukan readSector()
  //    dan masukkan kedalam buffer yang disediakan pada metadata
  // 6. Lompat ke iterasi selanjutnya hingga iterasi selesai
  // 7. Tulis retcode FS_SUCCESS dan ganti filesize
        pada akhir proses pembacaan.
}
```

Untuk mempermudah implementasi, berikut adalah contoh memasukkan data storage ke buffer filesystem dan buffer filesystem ke struct entry

```
struct node_filesystem   node_fs_buffer;
struct node_entry        node_buffer;
struct sector_filesystem sector_fs_buffer;
struct sector_entry      sector_entry_buffer;
// Asumsikan semua buffer filesystem diatas telah terinisiasi dengan baik

// Pembacaan storage ke buffer
readSector(sector_fs_buffer.sector_list, FS_SECTOR_SECTOR_NUMBER);
```

```
readSector(&(node_fs_buffer.nodes[0]),  FS_NODE_SECTOR_NUMBER);
readSector(&(node_fs_buffer.nodes[32]), FS_NODE_SECTOR_NUMBER + 1);

// Pengcopyan buffer ke entry
memcpy(&node_buffer, &(node_fs_buffer.nodes[i]), sizeof(struct node_entry));
memcpy(
       &sector_entry_buffer,
       &(sector_fs_buffer.sector_list[i]),
       sizeof(struct sector_entry)
);
```

3.2.3. Implementasi *write*

Syscall **write** akan digunakan untuk melakukan penulisan file atau folder ke filesystem. *Signature* dari **write** mirip dengan **read**. System call ini menggunakan `filesize` untuk menentukan tipe dan batas penulisan. Jika `filesize` bernilai 0, maka **write** akan membuat folder.

Berikut adalah pseudocode dari **write**

```
void write(struct file_metadata *metadata, enum fs_retcode *return_code) {
  struct node_filesystem   node_fs_buffer;
  struct sector_filesystem sector_fs_buffer;
  struct map_filesystem    map_fs_buffer;
  // Tambahkan tipe data yang dibutuhkan

  // Masukkan filesystem dari storage ke memori

  // 1. Cari node dengan nama dan lokasi parent yang sama pada node.
  //    Jika tidak ditemukan kecocokan, lakukan proses ke-2.
  //    Jika ditemukan node yang cocok, tuliskan retcode
  //    FS_W_FILE_ALREADY_EXIST dan keluar.

  // 2. Cari entri kosong pada filesystem node dan simpan indeks.
  //    Jika ada entry kosong, simpan indeks untuk penulisan.
  //    Jika tidak ada entry kosong, tuliskan FS_W_MAXIMUM_NODE_ENTRY
  //    dan keluar.

  // 3. Cek dan pastikan entry node pada indeks P adalah folder.
  //    root terdefinisi sebagai suatu folder.
  //    Jika pada indeks tersebut adalah file atau entri kosong,
  //    Tuliskan retcode FS_W_INVALID_FOLDER dan keluar.
```

```
// 4. Dengan informasi metadata filesize, hitung sektor-sektor
//    yang masih kosong pada filesystem map. Setiap byte map mewakili
//    satu sektor sehingga setiap byte mewakili 512 bytes pada storage.
//    Jika empty space tidak memenuhi, tuliskan retcode
//    FS_W_NOT_ENOUGH_STORAGE dan keluar.
//    Jika ukuran filesize melebihi 8192 bytes, tuliskan retcode
//    FS_W_NOT_ENOUGH_STORAGE dan keluar.
//    Jika tersedia empty space, lanjutkan langkah ke-5.

// 5. Cek pada filesystem sector apakah terdapat entry yang masih kosong.
//    Jika ada entry kosong dan akan menulis file, simpan indeks untuk
//    penulisan.
//    Jika tidak ada entry kosong dan akan menulis file, tuliskan
//    FS_W_MAXIMUM_SECTOR_ENTRY dan keluar.
//    Selain kondisi diatas, lanjutkan ke proses penulisan.

// Penulisan
// 1. Tuliskan metadata nama dan byte P ke node pada memori buffer
// 2. Jika menulis folder, tuliskan byte S dengan nilai
//    FS_NODE_S_IDX_FOLDER dan lompat ke langkah ke-8
// 3. Jika menulis file, tuliskan juga byte S sesuai indeks sector
// 4. Persiapkan variabel j = 0 untuk iterator entry sector yang kosong
// 5. Persiapkan variabel buffer untuk entry sector kosong
// 6. Lakukan iterasi berikut dengan kondisi perulangan
      (penulisan belum selesai && i = 0..255)
//    1. Cek apakah map[i] telah terisi atau tidak
//    2. Jika terisi, lanjutkan ke iterasi selanjutnya / continue
//    3. Tandai map[i] terisi
//    4. Ubah byte ke-j buffer entri sector dengan i
//    5. Tambah nilai j dengan 1
//    6. Lakukan writeSector() dengan file pointer buffer pada metadata
//       dan sektor tujuan i
//    7. Jika ukuran file yang telah tertulis lebih besar atau sama dengan
//       filesize pada metadata, penulisan selesai

// 7. Lakukan update dengan memcpy() buffer entri sector dengan
//    buffer filesystem sector
// 8. Lakukan penulisan seluruh filesystem (map, node, sector) ke storage
//    menggunakan writeSector() pada sektor yang sesuai
// 9. Kembalikan retcode FS_SUCCESS
```

```
}
```

Setelah keempat *syscall* diimplementasikan, lakukan perubahan pada *interrupt handler* 0x21 seperti berikut

```
void handleInterrupt21(int AX, int BX, int CX, int DX) {
    switch (AX) {
        case 0x0:
            printString(BX);
            break;
        case 0x1:
            readString(BX);
            break;
        case 0x2:
            readSector(BX, CX);
            break;
        case 0x3:
            writeSector(BX, CX);
            break;
        case 0x4:
            read(BX, CX);
            break;
        case 0x5:
            write(BX, CX);
            break;
        default:
            printString("Invalid Interrupt");
    }
}
```

3.3. Membuat *shell* sederhana

Agar sistem operasi dapat digunakan pengguna, diperlukan sebuah interface yang disebut *shell*. Shell memberikan user interface dari layanan-layanan *syscall* sistem operasi. User interface dapat berupa sebuah *command line interface* atau *graphical user interface*. Untuk menghindari ke-keos-an, spesifikasi wajib hanya mencakup *command line interface*.

3.3.1. Implementasi *shell*

Shell akan menggunakan *REPL* atau *read-eval-print-loop* untuk memberikan command interface melalui keyboard pengguna. Berikut adalah contoh kode dasar dari shell

```
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
      printString("Unknown command\r\n");

  }
}
```

Kode tersebut akan menuliskan dan meminta input keyboard seperti command prompt distro Linux. Selain menampilkan tulisan prompt, shell juga perlu untuk menampilkan *current working directory*.

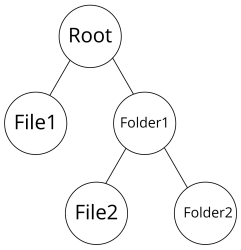
Untuk memudahkan pengerjaan pembuatan utility program, disarankan untuk membuat fungsi atau implementasi langsung pada shell *argument splitter* yang memproses buffer input shell dan merubah menjadi *array of string* yang berisi string-string yang dipisahkan oleh spasi. Asumsikan shell hanya **menerima** input command dengan spasi seperti berikut

```
cd folder1
```

```
mv folder1 folder2/renamed_f1
```

3.3.2. Membuat printCWD

Filesystem yang dibuat menggunakan sistem hierarki yang dapat direpresentasikan dengan sebuah *tree*. Pohon direktori tersebut berasal dari *root*, *internal node* tree adalah sebuah *folder*, dan daun dari pohon merupakan *file* atau *folder*.



Namun sedikit berbeda dengan biasanya dimana struktur data *tree* pada C menggunakan *pointer* untuk menunjukkan hubungan parent, pada filesystem ini hubungan parent direpresentasikan menggunakan indeks angka biasa yaitu byte **P** pada filesystem **node**. Penggunaan angka biasa akan memudahkan proses serialisasi struktur data sebelum penulisan ke media penyimpanan.

Untuk menampilkan *current working directory*, fungsi printCWD membutuhkan setidaknya pointer ke output string dan *byte current working directory*. Spesifikasi adalah ***shell*** dapat menampilkan *current working directory* (ex. `OS@IF2230:/folder1/$`) Detail implementasi dibebaskan, diperbolehkan implementasi langsung pada shell.

Tips : Perlakukan filesystem seperti *tree* dan aplikasikan algoritma pada *tree*.

3.4. Membuat *utility program*

Setelah memiliki shell sederhana untuk menerima interaksi, sistem operasi perlu ditambahkan *utility program* untuk melakukan aksi dari interaksi yang diterima. Berikut adalah spesifikasi beberapa *utility* yang wajib ditambahkan

- **cd**

  Utility cd merupakan alat dasar untuk melakukan navigasi pada filesystem.
  - *cd* dapat memindah *current working directory* ke folder tujuan
  - *cd* dapat naik satu tingkat dengan argumen “..”
  - *cd* dapat langsung kembali ke root dengan argumen “/”
  - Untuk *relative pathing* tidak diwajibkan

- **ls**

  Utility ls memperlihatkan konten yang ada pada *current working directory*.
  - *ls* dapat memperlihatkan isi pada *current working directory*
  - *ls* dapat memperlihatkan isi folder yang berada pada *current working directory* (Catatan : Hanya tampilkan isi dalam folder tersebut, tidak perlu ditampilkan seluruhnya secara rekursif, ex. “ls folder1” mengoutputkan ke layar <isi folder1>).

- **mkdir**

  Utility mkdir digunakan untuk membuat suatu folder baru.
  - *mkdir* dapat membuat folder baru pada *current working directory*

- **cat**

  Utility cat dapat digunakan untuk menampilkan isi dari suatu file.
  - *cat* dapat menampilkan isi dari file sebagai *text file*

- **cp**

  Utility cp digunakan untuk melakukan copy file.
  - *cp* dapat melakukan copy file dari *current working directory* ke *current working directory*
  - *Relative pathing* dan peng-copyan rekursif folder tidak diwajibkan

- **mv**

  Utility mv dapat digunakan untuk melakukan operasi *rename* atau memindahkan file maupun folder.
  - *mv* dapat memindahkan file dan folder ke root dengan "/<nama tujuan>"
  - *mv* dapat memindahkan file dan folder ke dalam *parent folder current working directory* dengan "../<nama tujuan>"
  - *mv* dapat memasukkan file dan folder ke folder yang berada pada *current working directory*.

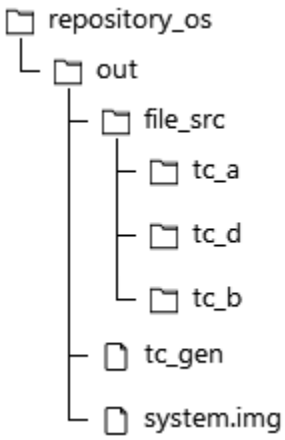
Untuk mempermudah milestone selanjutnya, disarankan untuk memisah *shell* dan *utility program* ke *file source code* yang terpisah.

3.5. Testing *syscall* dan *utility*

Pada kit diberikan source code program bernama *tc_gen.c* dan *tc_lib*. Program ini digunakan untuk membuat struktur direktori untuk filesystem yang telah pemagang buat sebelumnya. Program akan membaca *system.img* yang berada pada direktori yang sama dan melakukan pengeditan pada filesystem. Program dapat dicompile dan link menggunakan **gcc** seperti berikut

```
gcc tc_gen.c tc_lib -o tc_gen
```

Jika ingin membuat test case sendiri, lakukan modifikasi pada source code *tc_gen.c* sesuai dengan kebutuhan. Disediakan beberapa *test case* yang dapat digunakan untuk melakukan testing fungsionalitas system call pada sistem operasi.

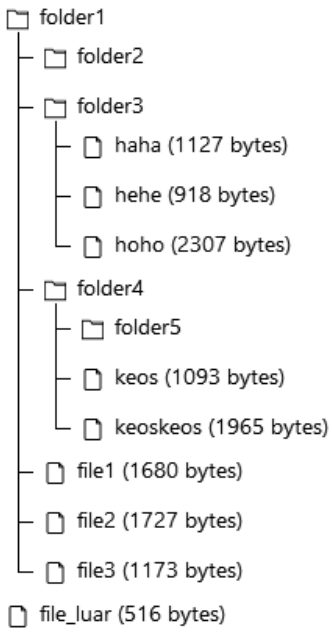


Letakkan program dan file sumber pada direktori yang sama dengan system.img seperti contoh diatas, cara pencarian file system.img dapat dicek pada source code *tc_gen.c*. Jalankan tc_gen setelah sistem operasi di *make build-run* agar **map** terinisiasi terlebih dahulu, tutup sistem operasi dan jalankan program seperti berikut

```
make build-run
./tc_gen A
```

Berikut adalah test case yang tersedia

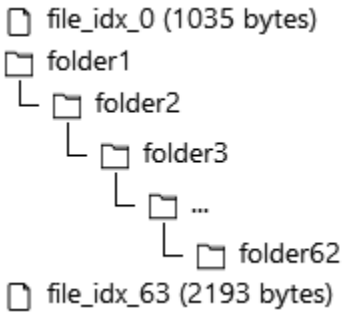
1. Test case cd, ls, read/write sector



Gunakan *utility cd* untuk masuk kedalam *folder1 → folder4 → folder5*, setelah itu gunakan "*cd ..*"

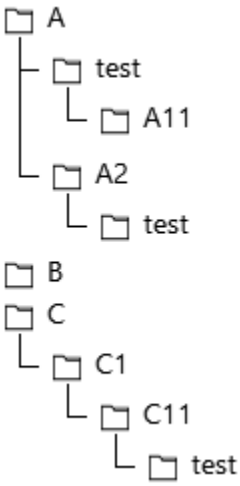
dan *ls*. Test case benar jika *ls* menampilkan *folder5*, *keos*, dan *keoskeos*.

2. Test case mv, cat, dan pembacaan syscall read dengan file



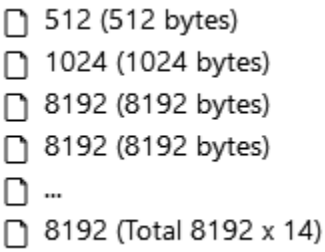
Jalankan utility *cat* pada *file_idx_0* dan *file_idx_63*. Setelah itu gunakan *mv* untuk memindahkan kedua file kedalam *folder1*. Test case benar jika *cat* menuliskan kedua file ke layar dan kedua file dipindahkan kedalam *folder1*. Kedua file adalah text file berisikan lirik.

3. Test case mkdir



Jalankan *mkdir test, mkdir A2, mkdir C11* pada root. Test case benar jika ketiga command *mkdir* berjalan dengan baik.

4. Test case cp, dan syscall read & write



Jalankan utility cp pada file *8192* dan *512*. Test case benar jika cp gagal pada file *8192* dan berhasil pada *512*.

Catatan tambahan : Semua *file* yang diberikan pada *test case* adalah *text file* dengan encoding ASCII dan *end line* bertipe *Line Feed*.

Apabila ingin melihat konsep dari file system, bisa merujuk ke **Chapter 10 Operating System Concepts, 8th edition oleh Silberschatz, Galvin, Gagne**.

IV. Penilaian

1. **Bagian *I/O sector* dan *read***
    - Syscall readSector dan writeSector berjalan dengan baik (20)
    - Syscall read dapat membaca file dan mengembalikan return code dengan baik (20)

2. **Bagian *write***
    - Syscall write (30)
        - Membuat folder (15)
        - Menulis file (15)

3. **Bagian *shell* dan *utility***
    - *Shell* (15)
        - Dapat menampilkan *current working directory* (10)
        - Loop input berjalan dengan baik (5)
    - *Utility* sesuai spesifikasi (15)
        - cd (2.5)
        - ls (2.5)
        - mv (2.5)
        - mkdir (2.5)
        - cat (2.5)
        - cp (2.5)

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. Lakukanlah *commit* yang wajar (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, 16 Maret 2022, 16.00 WIB waktu server.
   Deadline Selasa, 5 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.2.0.0** 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 2.**

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*.

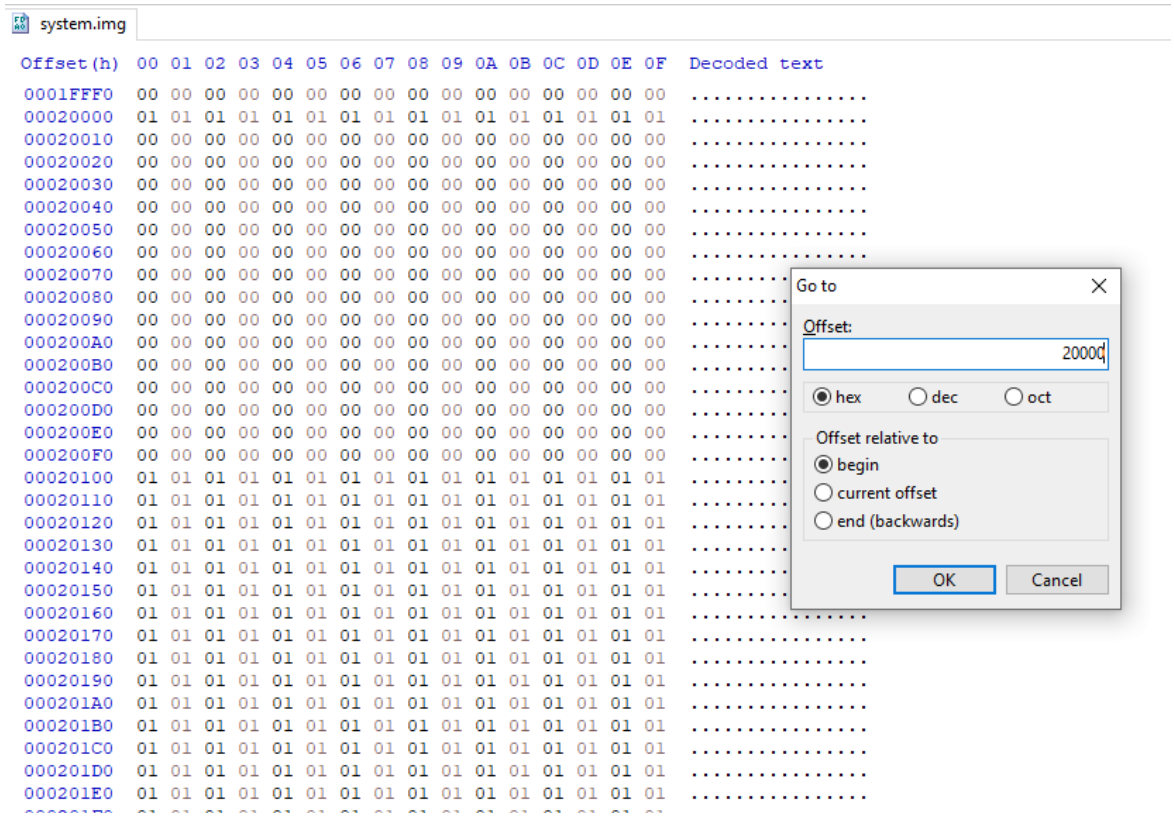
2. Pastikan flag -d selalu ada ketika menjalankan perintah ld86. Flag tersebut membuat ld86 menghapus header pada output file.

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*.

VII. Referensi

1. Spesifikasi Tugas Besar IF2230 - Sistem Operasi - Milestone 1
2. https://en.wikipedia.org/wiki/INT_13H
3. https://wiki.osdev.org/Interrupts
4. https://linux.die.net/man/1/hexedit
5. http://hilite.me/ (Pretty print untuk source code)
6. https://ascii-tree-generator.com/ (Pembuatan ilustrasi direktori)