OPERATING SYSTEMS
TUTORIAL 9

Questions

- Contiguous allocation vs. linked allocation
  (FAT is better than the pure linked list)

- what the image file really is

- img file size does NOT change.

Review: How 3 Questions Related

Q1 Consider the superblock shown below:

```
0000000: 4353 4333 3630 4653 0200 0000 1400 0000  CSC360FS........
00
00
```

Q2 Consider the following block from the root directory:

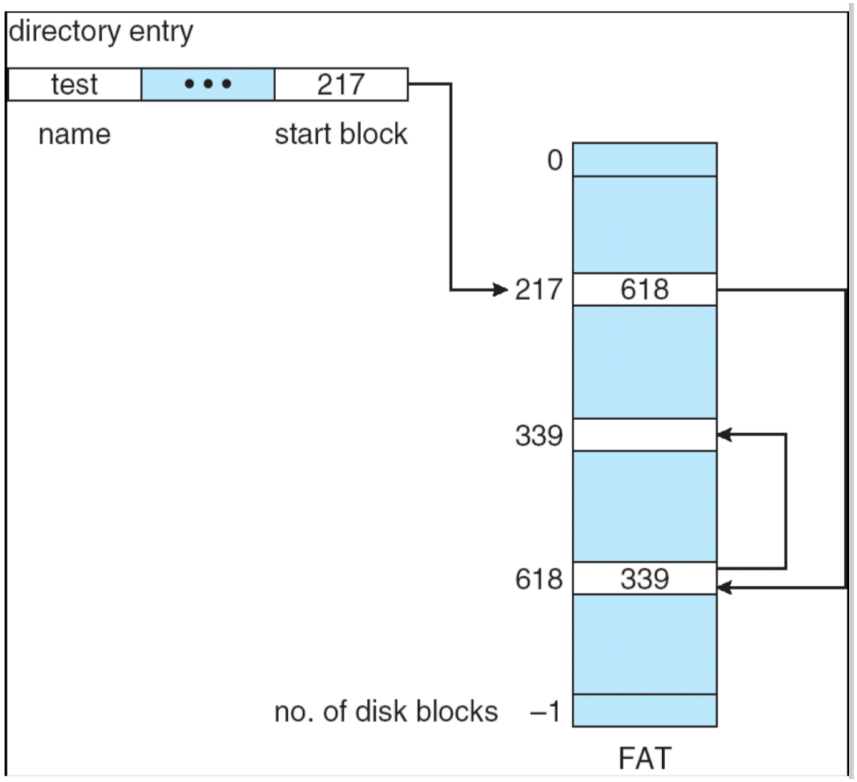
```
0005200: 0300 0000 3100 0000 0500 000a 0007 d50b  ....1...........
0005210: 0f0c 0000 07d5 0b0f 0c00 0066 6f6f 2e74  ...........foo.t
0005220: 7874 0000 0000 0000 0000 0000 0000 0000  xt..............
0005230:
0005240:
0005250:
0005260:
0005270:
0005280:
0005290:
00052a0:
00052b0:
```

Q3 Given the root directory information from the previous question and the

```
0000200: 0000 0001 0000 0001 0000 0001 0000 0001  ................
0000210: 0000 0001 0000 0001 0000 0001 0000 0001  ................
0000220: 0000 0001 0000 0001 0000 0001 0000 0001  ................
0000230: 0000 0001 0000 0001 0000 0001 0000 0001  ................
0000240: 0000 0001 0000 0001 0000 0001 0000 0001  ................
0000250: 0000 0001 0000 0001 0000 0001 0000 0001  ................
0000260: 0000 0001 0000 0001 0000 0001 0000 0001  ................
0000270: 0000 0001 0000 0001 0000 0001 0000 0001
```

Review: FDT vs. FAT

- FDT
  - –first block address
- FAT addresses
  - –linked list of addresses
  - –can be cached random access



diskinfo

./diskinfo test.img

```
Super block information:
Block size: 512
Block count: 5120
FAT starts: 1
FAT blocks: 40
Root directory start: 41
Root directory blocks: 8

FAT information:
Free Blocks: 5071
Reserved Blocks: 41
Allocated Blocks: 8
```

Super Block

Description	Size
File system identifier	8 bytes
Block Size	2 bytes
File system size (in blocks)	4 bytes
Block where FAT starts	4 bytes
Number of blocks in FAT	4 bytes
Block where root directory starts	4 bytes
Number of blocks in root dir	4 bytes

The first block (**512 B**) is reserved to contain information about the file system

Value of FAT entry

Value	Meaning
0x00000000	This block is available
0x00000001	This block is reserved
0x00000002– 0xFFFFFF00	Allocated blocks as part of files
0xFFFFFFFF	This is the last block in a file

SUPERBLOCK:

```
0000000: 4353 4333 3630 4653 0200 0000 1400 0000  CSC360FS........
0000010: 0001 0000 0028 0000 0029 0000 0008 0000  .....(...).......
0000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

disklist

./disklist test.img /subdir

```
F       2560    foo.txt     2005/11/15 12:00:00
F       5120    foo2.txt    2005/11/15 12:00:00
F      48127    makefs      2005/11/15 12:00:00
F          8    foo3.txt    2005/11/15 12:00:00
```

```
0005200: 0300 0000 3100 0000 0500 000a 0007 d50b  ....1...........
0005210: 0f0c 0000 07d5 0b0f 0c00 0066 6f6f 2e74  ...........foo.t
0005220: 7874 0000 0000 0000 0000 0000 0000 0000  xt..............
0005230: 0000 0000 0000 0000 0000 00ff ffff ffff  ................
0005240: 0300 0000 3600 0000 0a00 0014 0007 d50b  ....6...........
0005250: 0f0c 0000 07d5 0b0f 0c00 0066 6f6f 322e  ...........foo2.
0005260: 7478 7400 0000 0000 0000 0000 0000 0000  txt.............
0005270: 0000 0000 0000 0000 0000 00ff ffff ffff  ................
0005280: 0300 0000 4000 0000 5e00 00bb ff07 d50b  ....@...^.......
0005290: 0f0c 0000 07d5 0b0f 0c00 006d 616b 6566  ...........makef
00052a0: 7300 0000 0000 0000 0000 0000 0000 0000  s...............
00052b0: 0000 0000 0000 0000 0000 00ff ffff ffff  ................
00052c0: 0300 0000 9e00 0000 0100 0000 0807 d50b  ................
00052d0: 0f0c 0000 07d5 0b0f 0c00 0066 6f6f 332e  ...........foo3.
00052e0: 7478 7400 0000 0000 0000 0000 0000 0000  txt.............
00052f0: 0000 0000 0000 0000 0000 00ff ffff ffff  ................
```

Directory Entry

Description	Size
Status	1 byte
Starting Block	4 bytes
Number of Blocks	4 bytes
File Size (in bytes)	4 bytes
Create Time	7 bytes
Modify Time	7 bytes
File Name	31 bytes
unused (set to 0xFF)	6 bytes

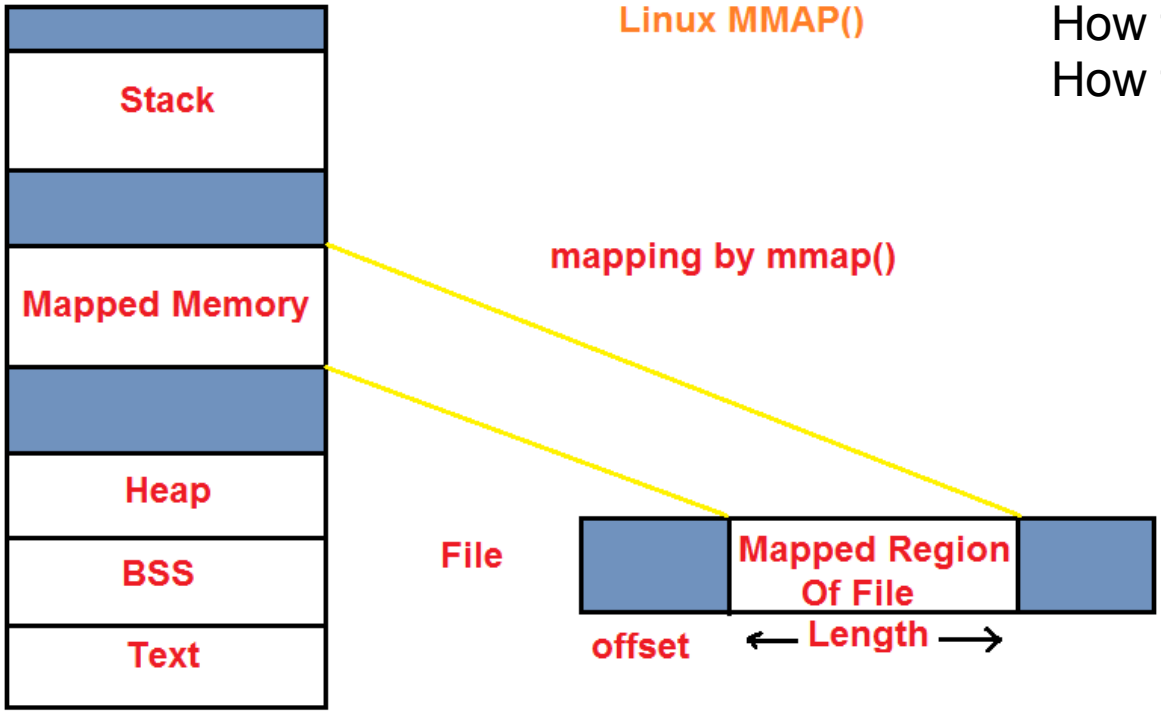
Takes up **64 B**, which implies there are 8 directory entries per **512 B block**

Hints on programming

mmap:

```
void *mmap(void *addr, size_t length, int prot,
int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

http://man7.org/linux/man-pages/man2/mmap.2.html



How to know the size of input file?
How to get file descriptor?

http://www.tutorialsdaddy.com/courses/linux-device-driver/lessons/mmap/

how to know the file size

fstat () int fstat (int fd, struct stat *buf);

```
struct stat {
        dev_t     st_dev;     /* ID of device containing file */
        ino_t     st_ino;     /* inode number */
        mode_t    st_mode;    /* protection */
        nlink_t   st_nlink;   /* number of hard links */
        uid_t     st_uid;     /* user ID of owner */
        gid_t     st_gid;     /* group ID of owner */
        dev_t     st_rdev;    /* device ID (if special file) */
        off_t     st_size;    /* total size, in bytes */
        blksize_t st_blksize; /* blocksize for file system I/O */
        blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
        time_t    st_atime;   /* time of last access */
        time_t    st_mtime;   /* time of last modification */
        time_t    st_ctime;   /* time of last status change */
    };
```

getting file descriptor

int open(const char *path, int oflags);

i.e., int fd = open("test.img", O_RONLY | O_WRONLY);

Value	Meaning
O_RDONLY	Open the file so that it is read only.
O_WRONLY	Open the file so that it is write only.
O_RDWR	Open the file so that it can be read from and written to.
O_APPEND	Append new information to the end of the file.
O_TRUNC	Initially clear all data from the file.
O_CREAT	If the file does not exist, create it. If the O_CREAT option is used, then you must include the third parameter.
O_EXCL	Combined with the O_CREAT option, it ensures that the caller *must* create the file. If the file already exists, the call will fail.

other APIs

fread() size_t fread (void *ptr, size_t size, size_t nmemb, FILE *stream)

ptr − This is the pointer to a block of memory with a minimum size of *size*nmemb* bytes.
size − This is the size in bytes of each element to be read.
nmemb − This is the number of elements, each one with a size of **size** bytes.
stream − This is the pointer to a FILE object that specifies an input stream.

fwrite() size_t fwrite (const void *ptr, size_t size, size_t nmemb, FILE *stream)

ptr − This is the pointer to the array of elements to be written.
size − This is the size in bytes of each element to be written.
nmemb − This is the number of elements, each one with a size of **size** bytes.
stream − This is the pointer to a FILE object that specifies an output stream.

fseek() int fseek (FILE *stream, long int offset, int whence)

stream − This is the pointer to a FILE object that identifies the stream.
offset − This is the number of bytes to offset from whence.
whence − This is the position from where offset is added.

It is specified by one of the following constants:
SEEK_SET, SEEK_CUR, SEEK_END

Byte Ordering

strings are Endian-Independent

- **htonl/htons/ntohl/ntohs ()**
  - –#include <arpa/inet.h>
  - –uint32_t htonl(uint32_t hostlong);
    - The htonl() function converts the unsigned integer **hostlong** from host byte order to network byte order.
  - –uint16_t htons(uint16_t hostshort);
    - The htons() function converts the unsigned short integer **hostshort** from host byte order to network byte order.
  - –uint32_t ntohl(uint32_t netlong);
    - The ntohl() function converts the unsigned integer netlong from network byte order to host byte order.
  - – uint16_t ntohs(uint16_t netshort);
    - The ntohs() function converts the unsigned short integer netshort from network byte order to host byte order.

PART-3

1. Search for file name from directories (sub-directories)
2. Obtain file size and starting block from the entry
3. Refer to FAT for finding the next block (may not be continuous)

PART-4

1. Get file attribute (size)
2. Find available blocks (FAT)
3. Update FAT and also dir entry
4. Copy (memcpy) file to the corresponding blocks.
   Need to refer to FAT when finding the next block.

FAT available blocks do NOT have to be continuous!!!
Recursively locating sub-dir!!!
You may want to modify in RAM and output to img eventually.
You may want to check if the file content is the same when you copy it out.

Useful Structures

```
// Super block
struct __attribute__((__packed__)) superblock_t {
  uint8_t   fs_id [8];
  uint16_t block_size;
  uint32_t file_system_block_count;
  uint32_t fat_start_block;
  uint32_t fat_block_count;
  uint32_t root_dir_start_block;
  uint32_t root_dir_block_count;
};
```

```
// Time and date entry
struct __attribute__((__packed__)) dir_entry_timedate_t {
  uint16_t year;
  uint8_t  month;
  uint8_t  day;
  uint8_t  hour;
  uint8_t  minute;
  uint8_t  second;
};
```

```
// Directory entry
struct __attribute__((__packed__)) dir_entry_t {
  uint8_t                    status;
  uint32_t                   starting_block;
  uint32_t                   block_count;
  uint32_t                   size;
  struct   dir_entry_timedate_t modify_time;
  struct   dir_entry_timedate_t create_time;
  uint8_t                    filename[31];
  uint8_t                    unused[6];
};
```

“__attribute__((__packed__))” is important and needed, otherwise, compiler optimizes for byte alignment