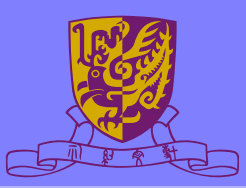


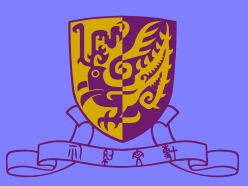
CSC/CEG 3150 Tutorial

Introduction to FAT32 and Assignment 3

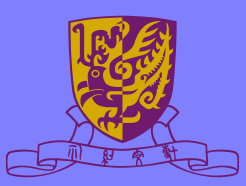
XIAO Zigang
zgxiao@cse.cuhk.edu.hk

Department of Computer Science and Engineering
The Chinese University of Hong Kong

December 17, 2008

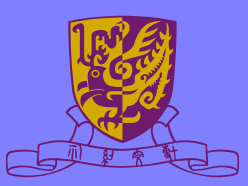


Outline



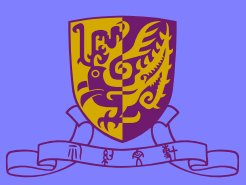
Sectors

- Basic unit of physical disk access
- Usually 512 bytes for magnetic disk
- Usually 2048 bytes for optical disk
- The size is NOT fixed in Assignment 3



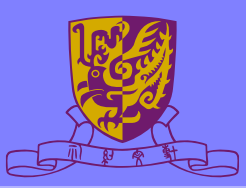
How are files stored in the disks?

- File size various
- Recall what we learned from Assignment 2
  - Like memory, hard disk can also be viewed as a flat space, using (logical) sector address(no longer C/H/S)
  - We use some simple data structure to manage the data
- Actually the idea is similar
- We may also use naive management strategy, e.g.
  - File stores in contiguous sectors
  - All files store in the same directory(i.e. no directory)
  - A header per file, stores some meta-info, e.g. file size
  - We group all the headers together like a database so as to locate the file (linear search)
- This really can make a file system,but...



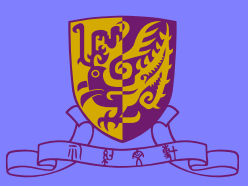
Further consideration

- Defect
  - When appending data to a file, we must reallocate space - Disk I/O is very slow!
  - When files are deleted, fragments are made, we must defragement the disk often
  - When the number of files is large, it takes too long to do the search
- Solution
  - Allow files to store non-consecutive
  - **Directory tree** is more convenient to group data into a hierarchical manner



Clusters and FAT

- Cluster is basic (logical) unit in FAT32
- File and directory content is stored in clusters
- A file occupies several clusters, rather than sectors. There is **time-space trade off**
- If the size of a file exceeds one cluster, use a **linked list** to find next block(cluster), which will form a **cluster chain**
- Note that the **cluster index starts from 2**

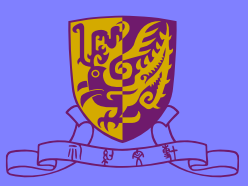


FAT

- The FAT located in FAT area is used for such purpose, like a static linked list.
- The basic idea of FAT is to use **File Allocation Table** to keep track of the file allocation status
- Each entry A in FAT points to another FAT entry B, which means the Cluster A is occupied by this file, and its next cluster is Cluster B
- Use a reserved number EOC to denote **End Of Cluster** (think of NULL)

Description	value
Unallocated	`0x?0000000`[1]
Allocated	`0x?0000002`∼`0x?FFFFFEF`
EOC	`0x?FFFFFF8`∼`0x?FFFFFFF`

[1] '?' means upper 4 bits are usually zero but are reserved and should be left untouched



Cluster chain

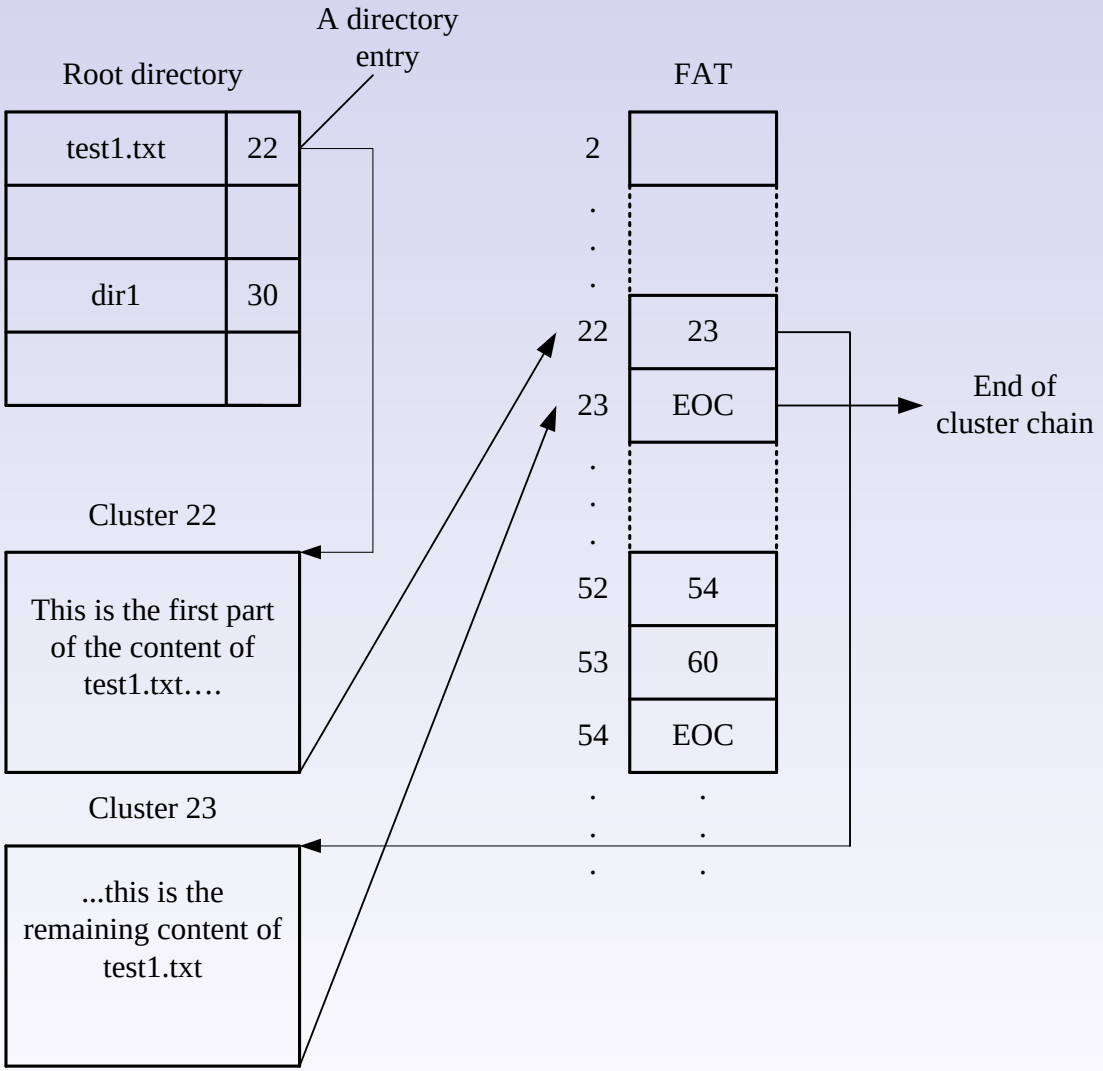
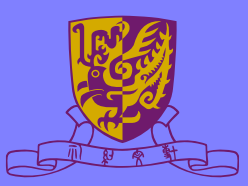


Figure: Cluster chain



FAT32 File System Layout

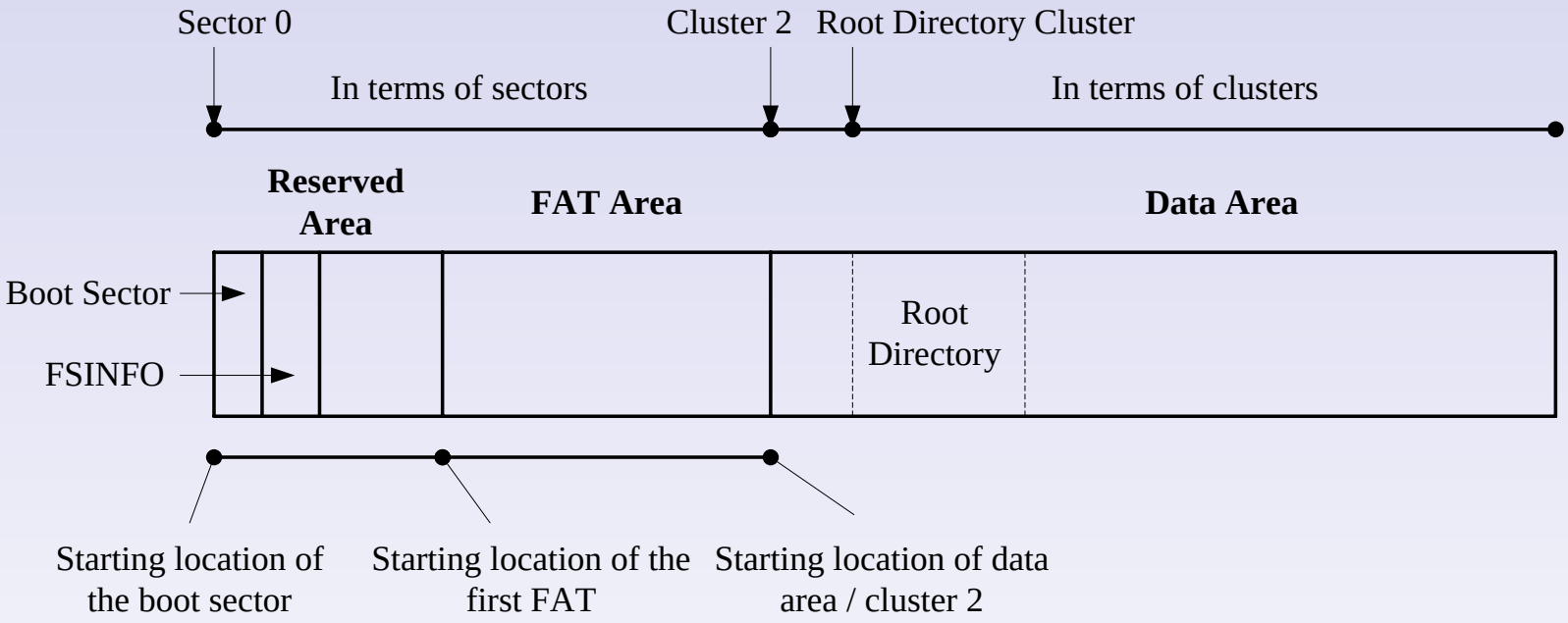
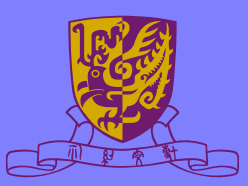
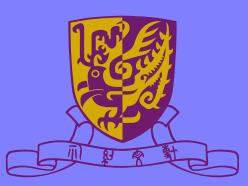


Figure: FAT32 file system overview



Three Important Areas in FAT32

- Reserved Area
  - First several sectors(usually 32) reserved for special use
  - The first sector (sector 0) is the **boot sector**
    - Contains important information about the file system
      E.g. number of bytes per sector, number of sectors per cluster
    - Contains the boot code (if necessary)
      Pointed by an assembly jump instruction in the first 3 bytes of sector 0
- FAT Area
  - Contains a number of File Allocation Tables (FATs), usually 2 (it is not fixed,though)
  - Located just after the reserved area
- Data Area
  - Contains the root directory
  - Contains files / directories
  - Located just after the last FAT



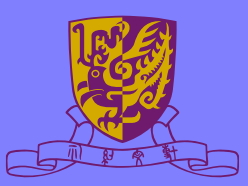
Boot Sector and DOS Boot Record

- Boot Sector is the first sector, i.e. sector 0
- DBR(DOS Boot Record) is 0-512B in Boot Sector
  - Contains IMPORTANT information of the file system
- The data structure is predefined
  - Declare a such data structure in your program
    - Be careful of the byte alignment in struct
  - Read and memcpy the boot sector into your data structure
  - You can now extract useful information from your data structure

```
0000000: eb3c 906d 6b64 6f73 6673 0000 0204 0100  .<.mkdosfs......
0000010: 0200 0200 a0f8 2800 2000 4000 0000 0000  ......(. .@.....
......
00001f0: 0000 0000 0000 0000 0000 0000 0000 55aa  ..............U.

OEM NAME = mkdosfs
bytes per sector = 0x0200 = 512
sector per cluster  = 0x04 = 4
```

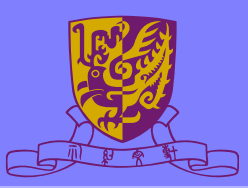
Listing 1: Boot Record example



0 - 35 Bytes in Boot Sector (FAT32)


```
#pragma pack(push,1)    /* BYTE align in memory (no padding) */
typedef struct BootEntry{
    unsigned char BS_jmpBoot[3];    /* Assembly instruction to jump to
                                       boot code */
    unsigned char BS_OEMName[8];    /* OEM Name in ASCII */
    unsigned short BPB_BytsPerSec;  /* Bytes per sector. Allowed values
                                       include 512,1024, 2048, and 4096 */
    unsigned char BPB_SecPerClus;   /* Sectors per cluster (data unit).
                                       Allowed values are powers of 2, but the
                                       cluster size must be 32KB or smaller */
    unsigned short BPB_RsvdSecCnt;  /* Size in sectors of the reserved area */
    unsigned char BPB_NumFATs;      /* Number of FATs */
    unsigned short BPB_RootEntCnt;  /* Maximum number of files in the root
                                       directory for  FAT12 and FAT16. This is
                                       0 for FAT32 */
    unsigned short BPB_TotSec16;    /* 16-bit value of number of sectors in
                                       file system */
    unsigned char BPB_Media;        /* Media type */
    unsigned short BPB_FATSz16;     /* 16-bit size in sectors of each FAT
                                       for FAT12 and  FAT16. For FAT32, this
                                       field is 0 */
    unsigned short BPB_SecPerTrk;   /* Sectors per track of storage device */
    unsigned short BPB_NumHeads;    /* Number of heads in storage device */
    unsigned long BPB_HiddSec;  /* Number of sectors before the start of partiti
    unsigned long BPB_TotSec32; /* 32-bit value of number of sectors in file sys
                    Either this value or the 16-bit value above must be  0 */
```

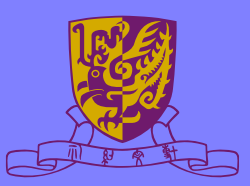
Listing 2: Boot Entry 0 - 35 Bytes



36 - 89 Bytes in Boot Sector (FAT32)


```
    unsigned long BPB_FATSz32;      /* 32-bit size in sectors of one FAT */
    unsigned short BPB_ExtFlags;    /* A flag for FAT */
    unsigned short BPB_FSVer;       /* The major and minor version number */
    unsigned long BPB_RootClus;             /* Cluster where the root directory
                                    be found */
    unsigned short BPB_FSInfo;      /* Sector where FSINFO structure can be
                                    found */
    unsigned short BPB_BkBootSec;   /* Sector where backup copy of boot
                                    sector is located */
    unsigned char BPB_Reserved[12]; /* Reserved */
    unsigned char BS_DrvNum;        /* BIOS INT13h drive number */
    unsigned char BS_Reserved1;     /* Not used */
    unsigned char BS_BootSig;       /* Extended boot signature to identify
                                    if the next three values are valid */
    unsigned long BS_VolID;         /* Volume serial number */
    unsigned char BS_VolLab[11];    /* Volume label in ASCII. User defines
                                     when creating the file system */
    unsigned char BS_FilSysType[8]; /* File system type label in ASCII */
}BootEntry;
#pragma pack(pop)                  /* BYTE align in memory (no padding)*/
```

Listing 3: Boot Entry 36 - 89 Bytes



Sample work flow

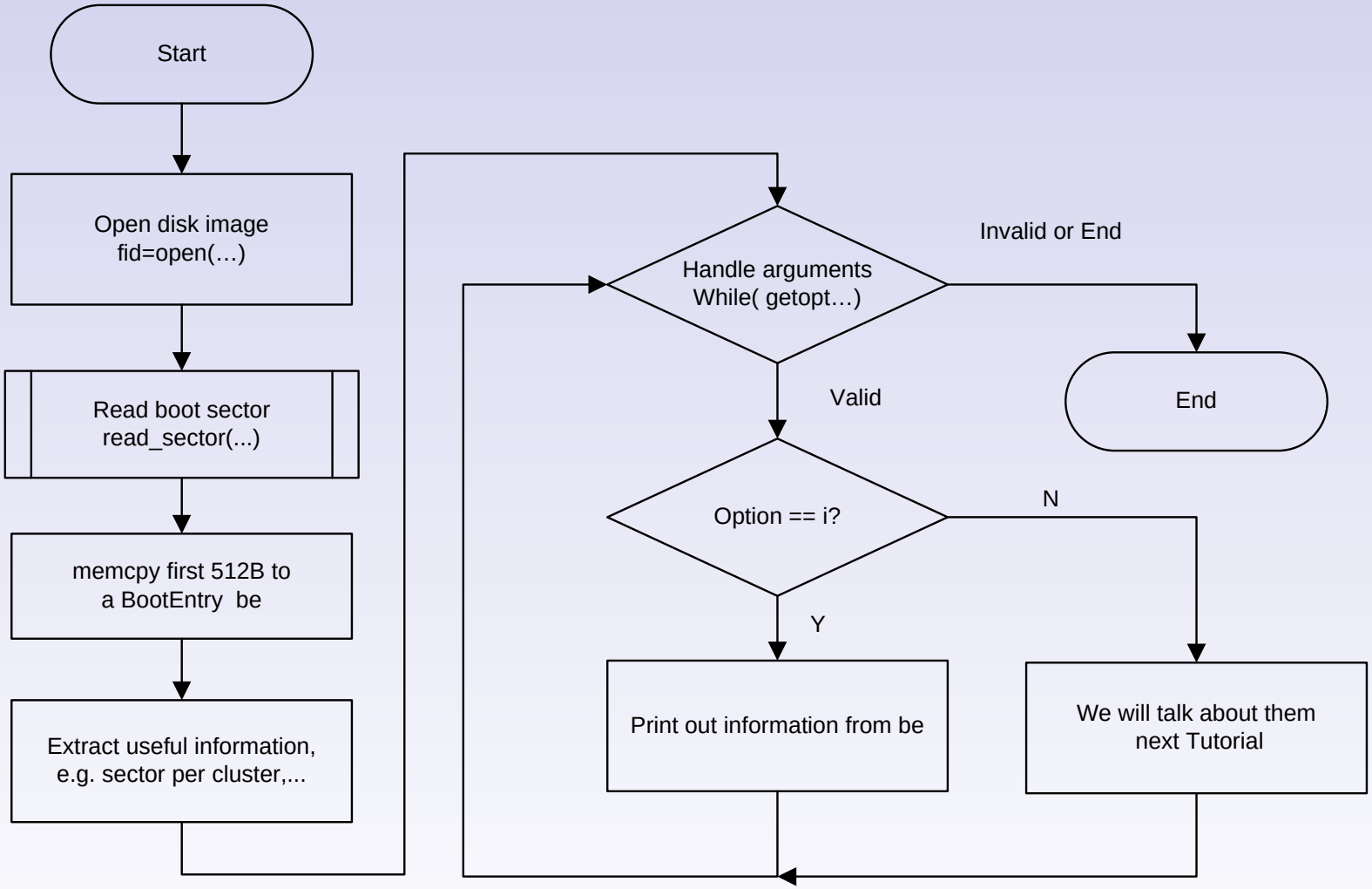
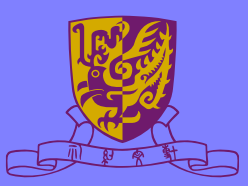


Figure: Sample work flow

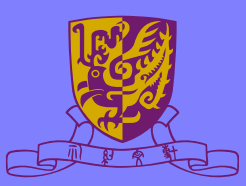


How to read data in terms of sectors?


```
/* go to the sector indicated by "sector_number",
 * then read n contiguous sectors (n = num_sectors),
 * and write the result into "buffer" */
/* Remember to clear the buffer first */
int read_sectors(int sector_number, unsigned char *buffer, int num_sectors)
{
        int dest, len;      /* used for error checking only */

        dest = lseek(fid, sector_number*bps, SEEK_SET);
        if(dest != sector_number*bps)
        {
                /* Error handling here */
        }
        if(bps*num_sectors > MAX)
        {
                /* Error handling here */
        }
        len = read(fid, buffer, bps*num_sectors);
        if(len != bps*num_sectors)
        {
                /* Error handling here */
        }
        return len;
}
```

Listing 4: read sector

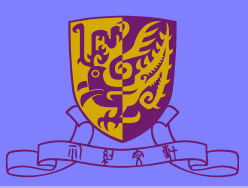


How to write data in terms of sectors?


```c
/* go to the sector indicated by "sector_number",
 * then write n contiguous sectors (n = num_sectors)
 * with the contents in "buffer" */
int write_sectors(int sector_number, unsigned char *buffer, int num_sectors)
{
        int dest, len;

        dest = lseek(fid, sector_number*bps, SEEK_SET);
        if (dest != sector_number*bps)
        {
                /* Error handling here */
        }
        len = write(fid, buffer, bps*num_sectors);
        if(len != bps*num_sectors)
        {
                /* Error handling here */
        }
        return len;
}
```

Listing 5: write sector



Relationship between Cluster and Sector

- cluster(C) to sector(S):
  - (C-2) * (sector per cluster) + (sector of cluster 2)
- sector(S) to cluster(C):
  - (S - sector of cluster 2) / (sector per cluster) +2
- sector of cluster 2 = ?
  - starting sector of data area

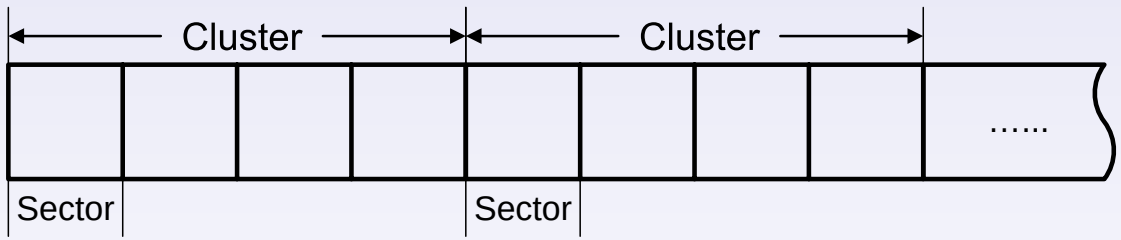
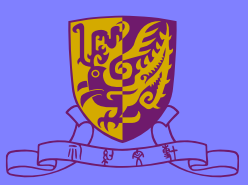


Figure: Example:sector per cluster = 4



Directory in FAT32

- In FAT32, directory is like a normal file
- Its contents are information of the files in it

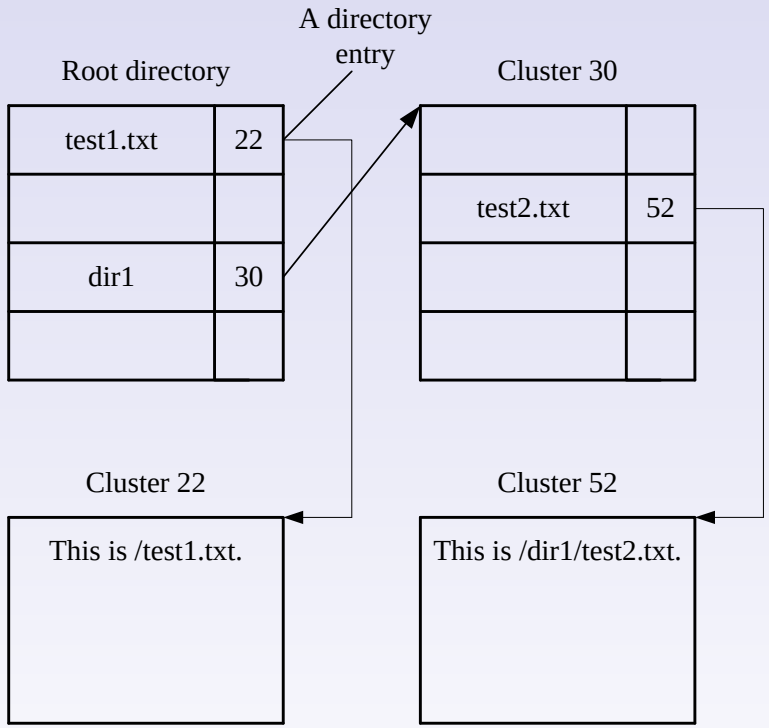
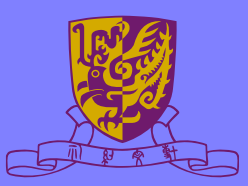


Figure: Directory entry



C struct of Directory Entry

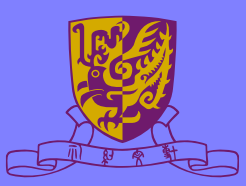

```
#pragma pack(push,1)
struct DirEntry
{
    /* Unallocated if first character is 0xe5 or 0x00 */
    unsigned char DIR_Name[11] ;      /* File name in 3.8 format */
    unsigned char DIR_Attr ;          /* File attributes */
    unsigned char DIR_NTRes ;         /* Reserved */
    unsigned char DIR_CrtTimeTenth ;  /* Created time */
    unsigned short DIR_CrtTime ;      /* Created time */
    unsigned short DIR_CrtDate ;      /* Created day */
    unsigned short DIR_LstAccDate ;   /* Accessed day */
    unsigned short DIR_FstClusHI ;    /* High 2 bytes of first cluster */
    unsigned short DIR_WrtTime ;      /* Written time */
    unsigned short DIR_WrtDate ;      /* Written day */
    unsigned short DIR_FstClusLO ;    /* Low 2 bytes of first cluster*/
    unsigned long DIR_FileSize ;      /* Size of file(0 for directories) */
};
#pragma pack(pop)
```

Listing 6: DirEntry struct

```
0000000: 5245 5355 4d45 2d31 5254 4620 00q3 347e  RESUME-1RTF ..4~
0000010: 4a30 8830 0000 4a33 7830 0900 f121 0000  .0.0.....0...!..

FILENAME = "RESUME-1.RTF"
SIZE = 0x0000 21f1
```

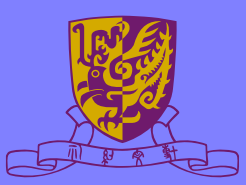
Listing 7: DirEntry example



File attributes, Starting cluster

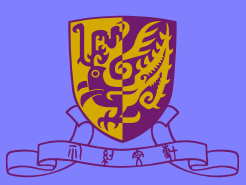
Flag Value	Description
0x01	Read only
0x02	Hidden file
0x04	System file
0x08	Volume label
0x0f	Long file name
0x10	Directory
0x20	Archive

- How to calculate 1st cluster of the file
  - Combine high 2 bytes and low 2 bytes to obtain
  - (Think about it... Hint: bit operation)
- How to set 1st cluster of the file to DirEntry
  - i.e. translate an 4 bytes address,e.g. 0x12345678 to high two bytes and low two bytes



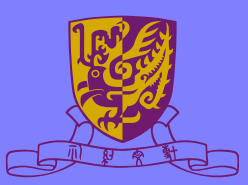
Introduction to assignment 3

- Three milestones in assignment 3
  1. List the contents – think about `ls`
  2. Reading an existing file – think about `cat/dd`
  3. Writing to an existing file – think about `echo`



List the contents of the root directory

- **Objective**:List the details of the files stored in the **root directory**
  1. Read Boot Entry to locate the starting cluster of root directory
  2. Follow the FAT to find out cluster chain of root directory
  3. Read file information in each cluster



Sample work flow

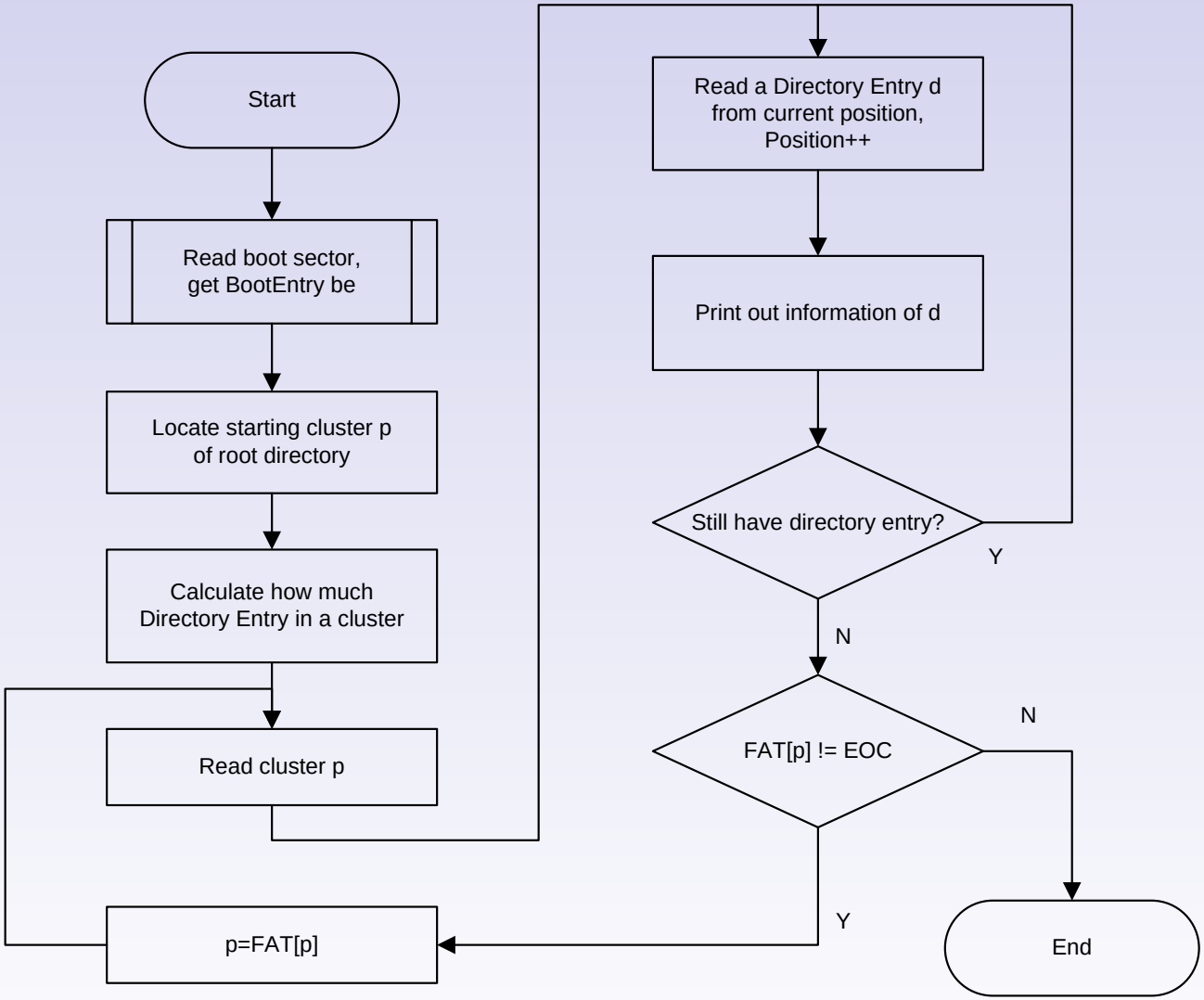
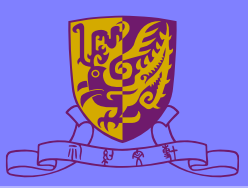


Figure: Sample work flow

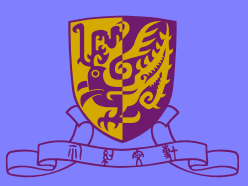


Sample output


```
$ ./a3 <arguments>
FILENAME        FILE SIZE       STARTING CLUSTER #
-------------------------------------------------
MAKEFILE        21              11
TEST.C          1023            12
HELLO.MP3       4194304         14

Total number of entries = 3
$ _
```

Listing 8: List content output



Reading an existing file

- **Objective**:Read and output contents from an existing file in **root directory**.
  1. Search the given file in directory, return fail if not found
  2. Reads meta information of the file(size, starting cluster)
  3. Output file content according to **cluster chain**
- The pathname is valid characters
  - 8.3 format
  - uppercase alphabets, digits
  - $ % ’ ‘ - { } ~ ! # ( ) & _ ∧
- The last cluster of the file **may not be fully filled**

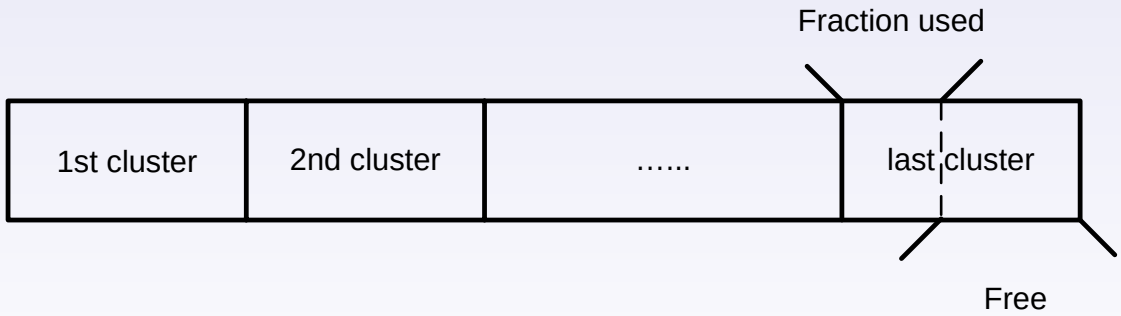
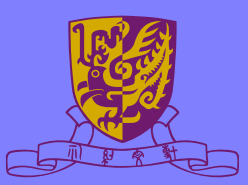


Figure: Example: Clusters occupied by a certain file



Sample work flow

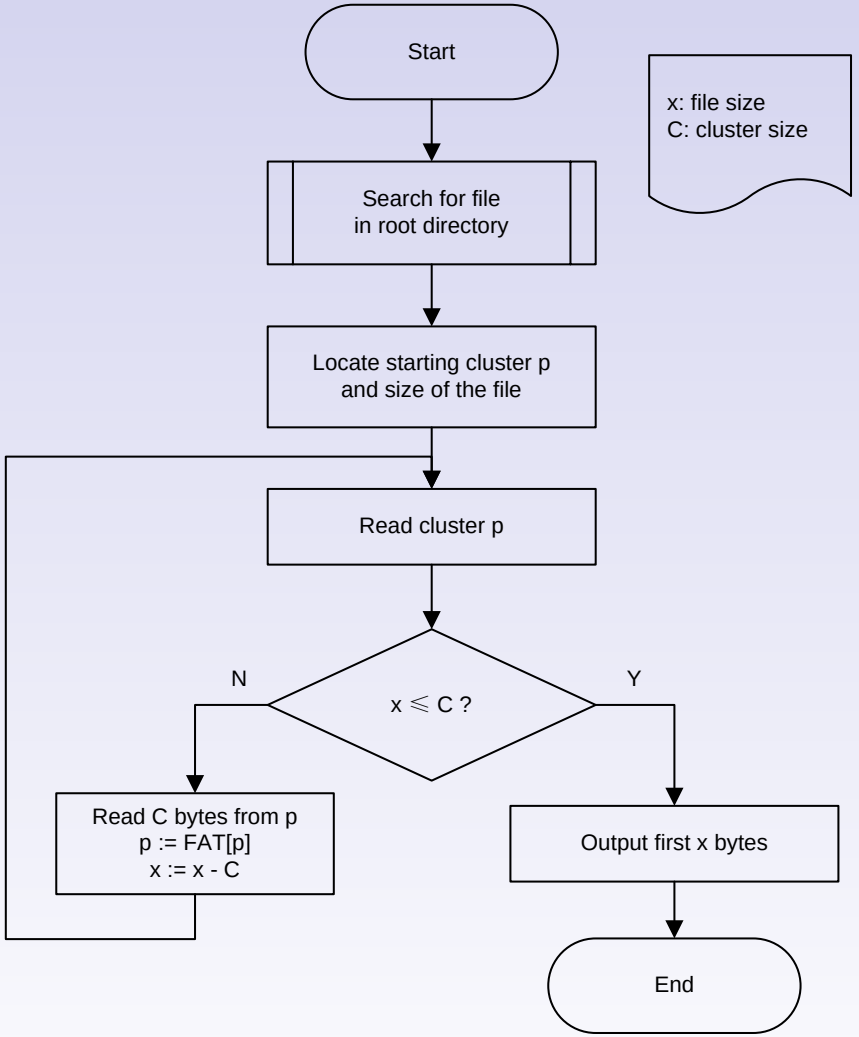
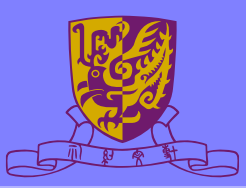


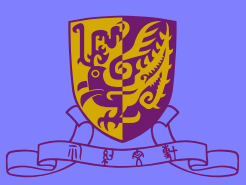
Figure: Sample work flow



Thank You

Now you should be able to complete Milestone 1 & 2

- See You Next Week -



Acknowledgement

- Some materials and pictures are from last year's tutorial notes made by *Mr.Cheong Chi Hong*
- The style of this slide is adapted from the template made by *HUANG Zheng-hua* in *Wuhan University*
- Google "vfs" to learn more about filesystem in Linux