

Project 3: Design and Implementation of File System

Qinya Li

Department of Computer Science and Engineering

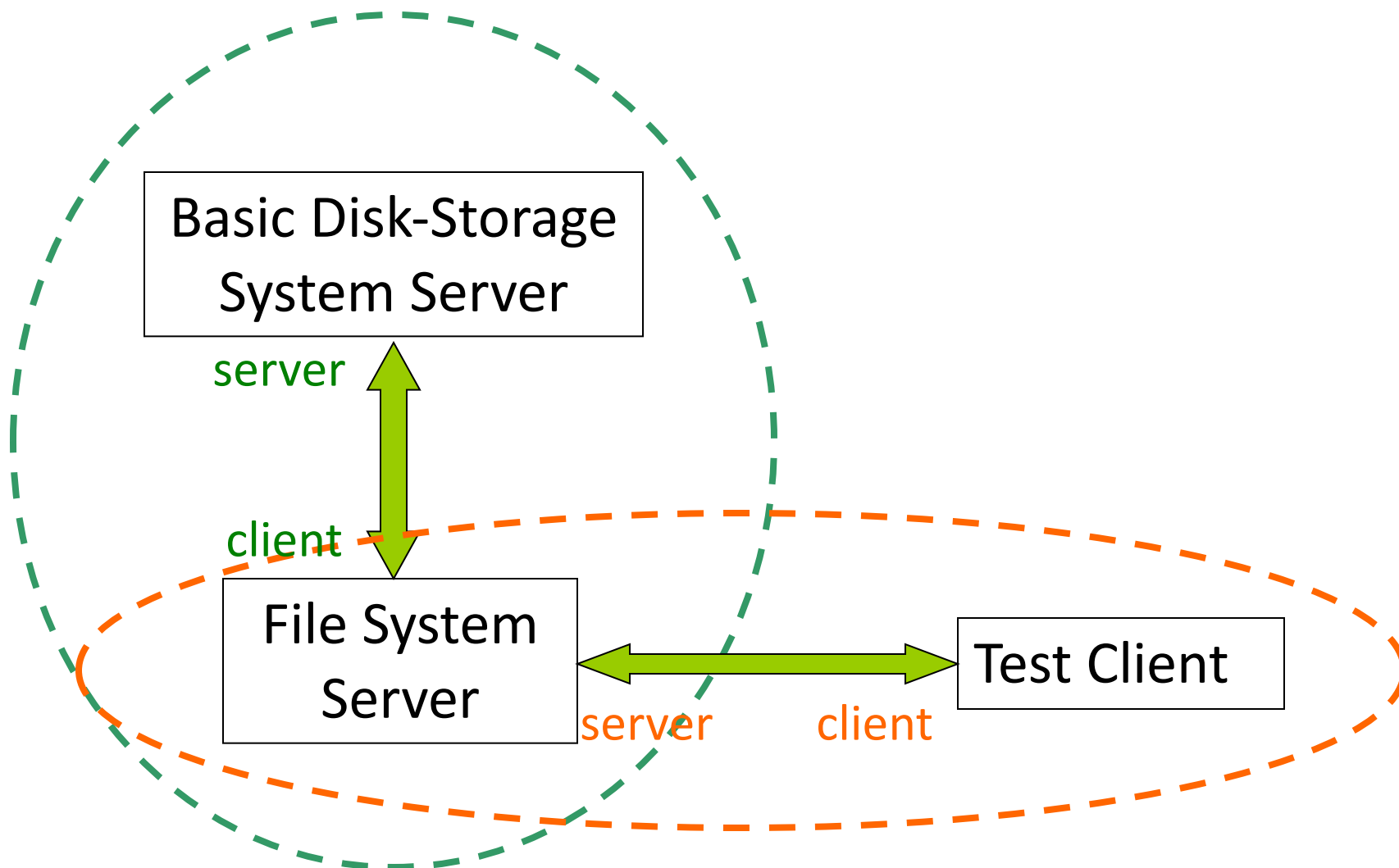
Shanghai Jiao Tong University

Spring 2023

Objectives

- Design and implement a basic disk-like secondary storage server.
- Design and implement a basic file system to act as a client using the disk services provided by the server designed above.
- Use socket API.

File System Architecture



Content

- Step 1: Design a basic disk-storage system
- Step 2: Design a basic file system
- Step 3: Work together

Enviroment

- VitrualBox/VMWare (optional)
- Ubuntu Linux (recommended)

What to Submit

- A “tar” file of your DIRECTORY, containing:
 - Your makefile
 - Any “.cc”, “.c”, and “.h” files
 - Any “readme” or “.pdf” files asked for in the project
 - A text file containing the runs of your programs for each of the project parts “typescript”
 - ▶ Do not submit ALL runs you have done, just the output required to demonstrate a successful (or unsuccessful) run
 - ▶ If you cannot get your program to work, submit a run of whatever you can get to work as you can get partial credit
 - ▶ Copy & paste, or use “>” to redirect output to a file
- DO NOT SUBMIT your object or executable files, remove them before you pack your directory

How to Submit

- Remove your “.o” files and executables

```
rm *.o
```

```
rm basic_server
```

```
rm basic_client
```

- Pack your entire directory (including the directory)

```
tar -cvf Prj3+StudentID.tar project3
```

- Send your **Prj3+StudentID.tar** file to Canvas

Project organized structure

■ Prj3+StudentID.tar

● step1

- ▶ disk.c
- ▶ makefile
- ▶ ...

● step2

- ▶ fs.c
- ▶ makefile
- ▶ ...

● step3

- ▶ disk.c
- ▶ fs.c
- ▶ client.c
- ▶ makefile
- ▶ ...

● report.pdf

● Prj3README

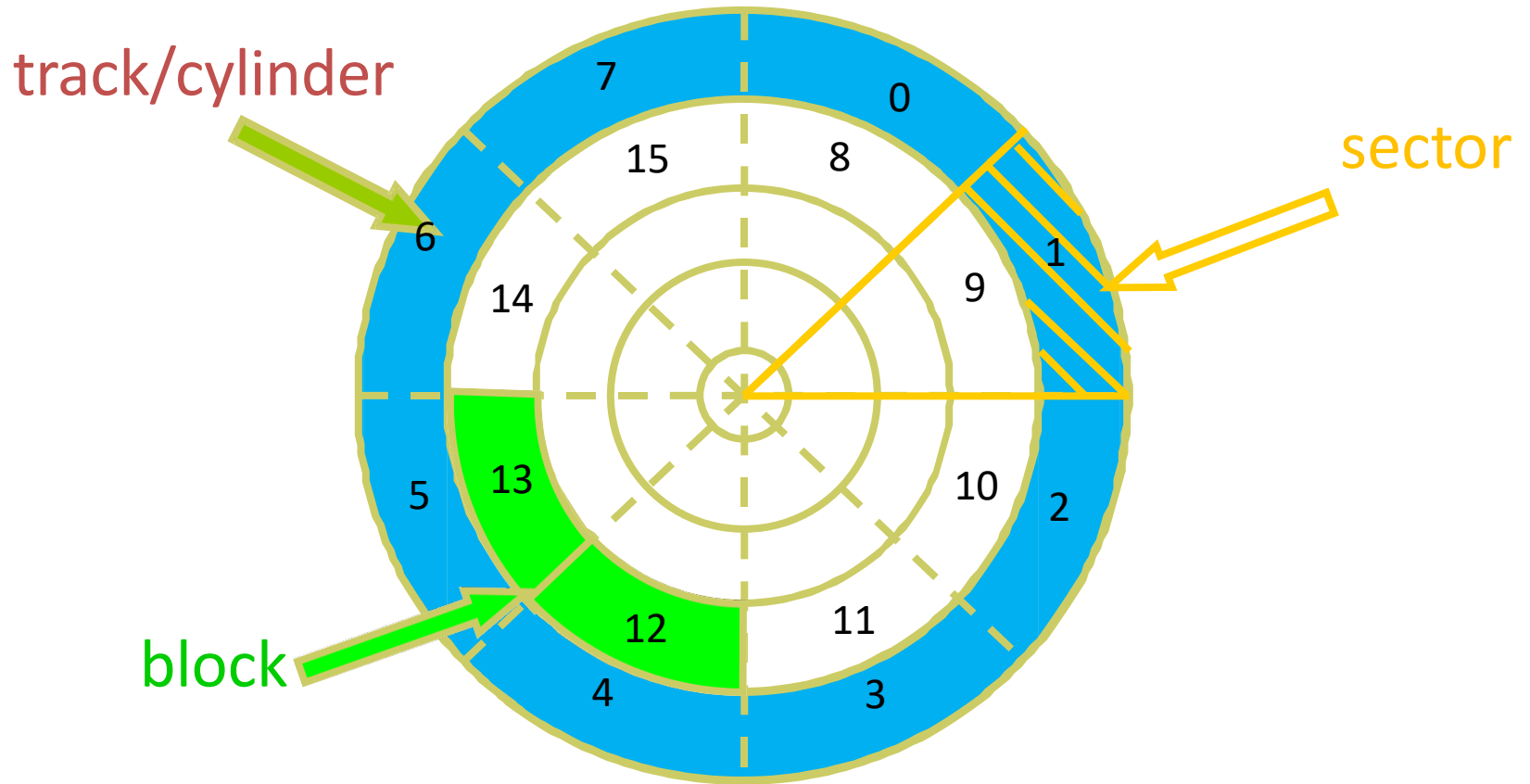
Demo and Presentation

- Deadline
 - May 26, 2023
- Demo slot
 - May 27-28, 2023, SEIEE 3-East 309
- Presentation (Optional)
 - May 30, 2023

Step 1: Overview

- Implement a simulation of a physical disk.
- cylinders and sectors
- track-to-track time
- sector size = 256bytes
- Store the actual data in a real disk file.

Disk Structure



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	-------

Command

■ Command line

- `./disk <#cylinders> <#sector per cylinder> <track-to-track delay> <#disk-storage-filename>`

Protocol

- **I**: Information request. The disk returns two integers representing the disk geometry: the number of cylinders, and the number of sectors per cylinder.
- **R c s**: Read request for the contents of cylinder c sector s . The disk returns **Yes** followed by a writespace and those 256 bytes of information, or **No** if no such block exists.
- **W c s data**: Write request for cylinder c sector s . The disk returns **Yes** and writes the data to cylinder c sector s if it is a valid write request, or returns **No** otherwise.
- **E**: Exit the disk-storage system.

Calculations

■ Given:

- c: track/cylinder number (0, 1, 2, ...)
- s: sector number (0, 1, 2, ...)
- n: number of sectors per track/cylinder
- b: block size (256 bytes)

■ Block start location (offset) in the file

- $$l = b * (n * c + s)$$

Calculations (cont'd)

■ Given

- c1: current cylinder number
- c2: next cylinder number
- t: cylinder-to-cylinder / track-to-track time

■ Disk arm moving delay

- $d = t * \text{abs}(c1 - c2)$

■ Useful function

- usleep (unsigned long microseconds)
- E.g., usleep(100000);

Manipulating Storage Space

- file system call
 - `fseek()`, `fread()`, `fwrite()`, ...

```
int fseek( FILE *stream, long offset, int origin );
```

```
fseek(diskfile, (cylinder * sector_number + sector)*Sector_size, 0)
```


Manipulating Storage Space

- file system call

- `fseek()`, `fread()`, `fwrite()`, ...

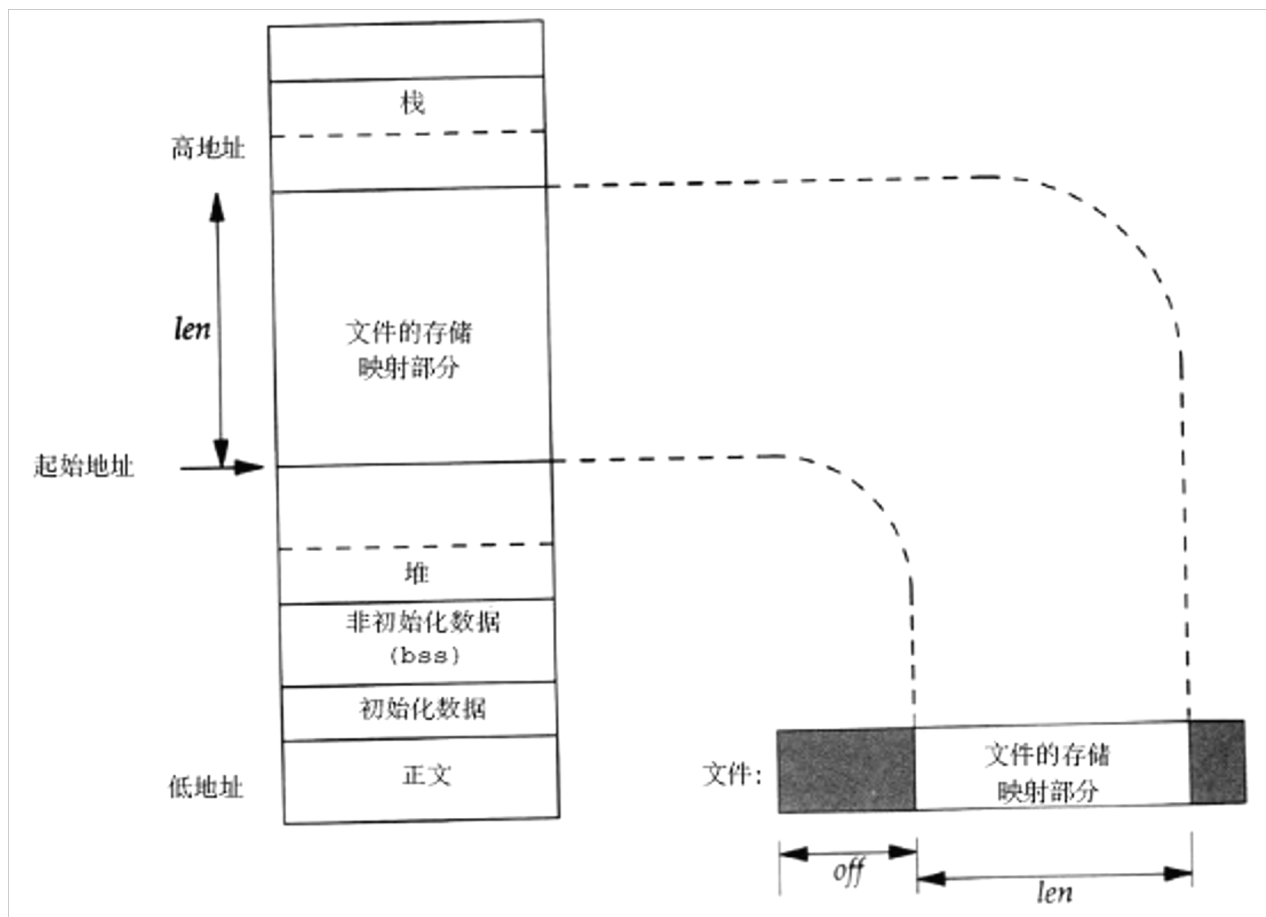
```
int fseek( FILE *stream, long offset, int origin );
```

```
fseek(diskfile, (cylinder * sector_number + sector)*Sector_size, 0)
```

- `mmap()` system call

- create a mapping between a memory space and a file

mmap



Using mmap()

■ Header

- #include <sys/mman.h>

■ Open a file

```
int fd = open (filename, O_RDWR | O_CREAT, 0);  
if (fd < 0) {  
    printf("Error: Could not open file '%s'.\n", filename);  
    exit(-1);  
}
```

Using mmap() (cont'd)

- Stretch the file size to the size of the simulated disk

off_t **lseek**(int fildes, off_t offset, int whence);

```
long FILESIZE = BLOCKSIZE * SECTORS * CYLINDERS;  
int result = lseek (fd, FILESIZE-1, SEEK_SET);  
if (result == -1) {  
    perror ("Error calling lseek() to 'stretch' the file");  
    close (fd);  
    exit(-1);  
}
```

Using mmap() (cont'd)

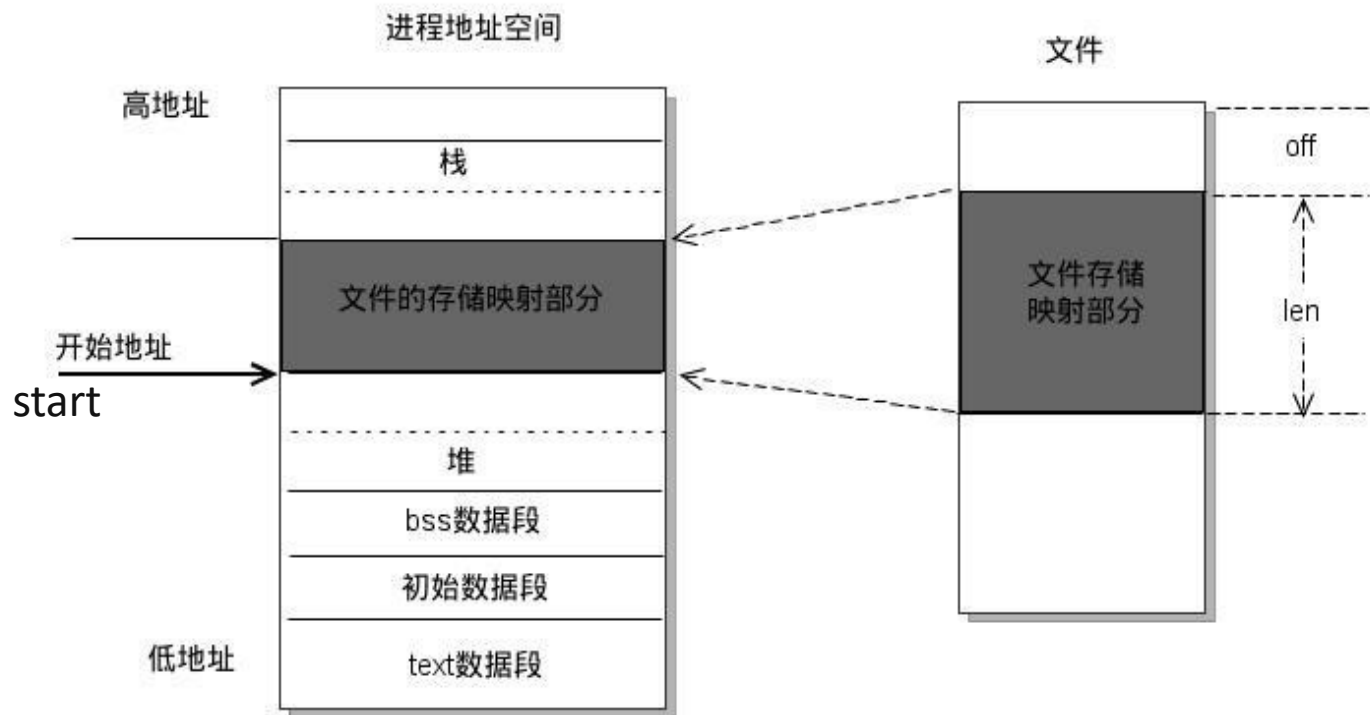
- Write something at the end of the file to ensure the file actually have the new size.

```
result = write (fd, "", 1);  
if (result != 1) {  
    perror("Error writing last byte of the file");  
    close(fd);  
    exit(-1);  
}
```

Using mmap() (cont'd)

■ Map the file

```
void *mmap(void *start, size_t len, int prot, int flags, int fd, off_t off);
```



Using mmap() (cont'd)

■ Map the file

```
void *mmap(void *start, size_t len, int prot, int flags, int fd, off_t off);
```

```
char* diskfile;
diskfile = (char *) mmap (NULL, FILESIZE,
                          PROT_READ | PROT_WRITE,
                          MAP_SHARED, fd, 0);
if (diskfile == MAP_FAILED){
    close(fd);
    printf("Error: Could not map file.\n");
    exit(-1);
}
```

Using mmap() (cont'd)

```
void *memcpy(void *dest, const void *src, size_t count);
```

■ Write to the file

```
memcpy (&diskfile[BLOCKSIZE * (c * SECTORS + s)]  
        , buf, len);
```

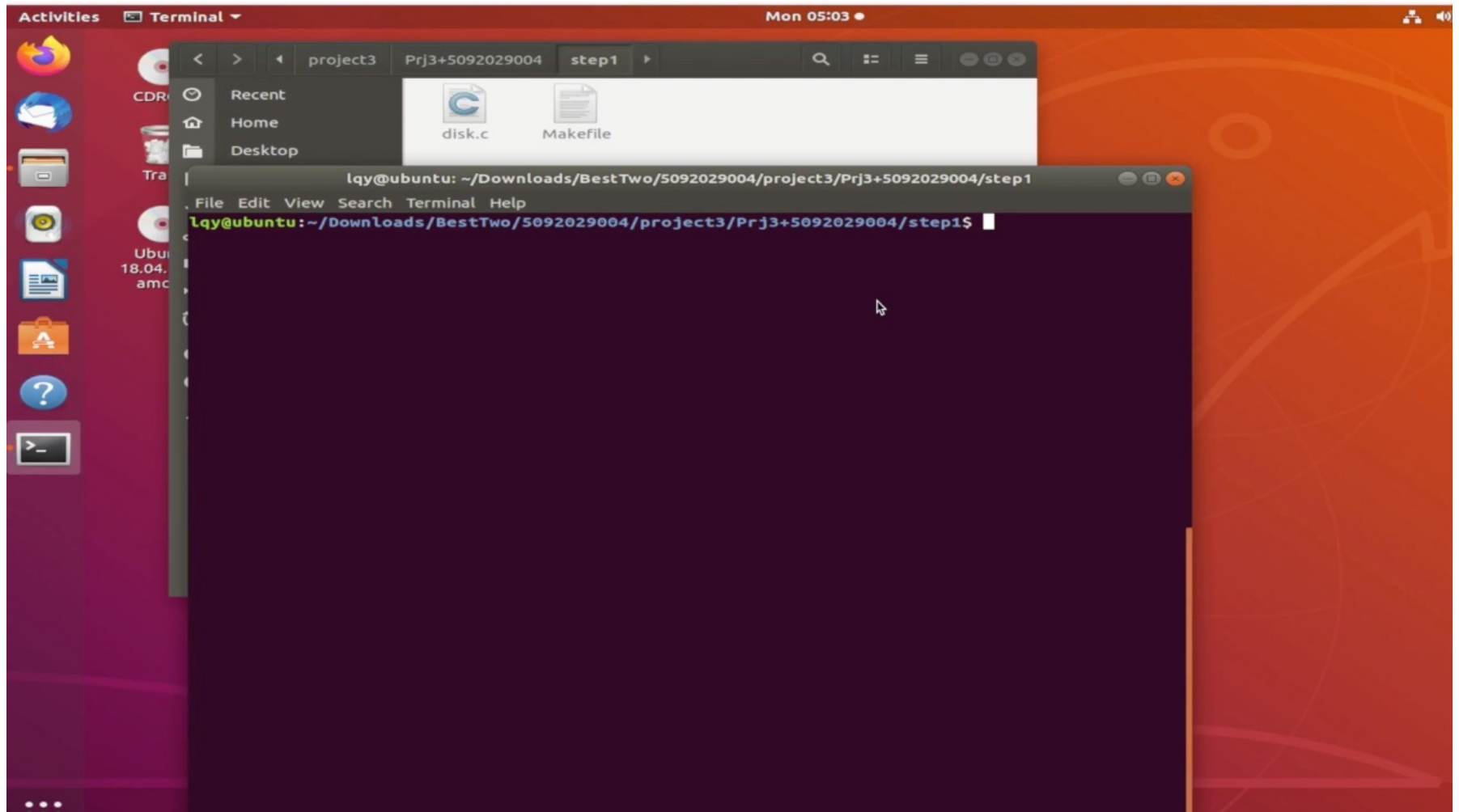
■ Read from the file

```
memcpy (buf, &diskfile[BLOCKSIZE * (c * SECTORS + s)]  
        , BLOCKSIZE);
```


Test your disk-storage system

- Read requests from STDIN and write a log file (disk.log)
- For each request, output one line.
- For **I**, output two integers, separate by a whitespace, which denote the number of cylinders and the number of sectors per cylinder.
- For **R c s**, output **No** if no such block exists, otherwise, output **Yes** followed by a whitespace and those 256 bytes of information.
- For **W c s data**, Write **Yes** or **No** to show whether it is a valid write request or not.
- For **E**, output **Goodbye!**.

Demo



Step 2: Overview

- Implement a file system
- files and directories
- store information in memory
- Command line
 - ./fs

Protocol I

- **f**: Format. This will format the file system on the disk, by initializing any/all of tables that the file system relies on.
- **mk f**: Create file. This will create a file named *f* in the file system.
- **mkdir d**: Create directory. This will create a subdirectory named *d* in the current directory.
- **rm f**: Delete file. This will delete the file named *f* from current directory.
- **cd path**: Change directory. This will change the current working directory to the path. The path is in the format in Linux, which can be relative path or absolute path. When the file system starts, the initial working path is /. You need to handle . and ..

Protocol II

- **rmdir *d***: Delete directory. This will delete the directory named *d* in the current directory.
- **ls**: Directory listing. This will return a listing of the files and directories in the current directory. You are also required to output some other information, such as file size, last update time, etc.
- **cat *f***: Catch file. This will read the file named *f*, and return the data that came from it.
- **w f l data**: Write file. This will overwrite the contents of the file named *f* with the *l* bytes of data. If the new data is longer than the data previously in the file, the file will be made longer. If the new data is shorter than the data previously in the file, the file will be truncated to the new length.

Protocol III

- **i f pos l data**: Insert to a file. This will insert to the file at the position before the pos^{th} character (0-indexed), with the l bytes of data. If the pos is larger than the size of the file. Just insert at the end of the file.
- **d f pos l**: Delete in the file. This will delete contents from the pos character (0-indexed), delete l bytes or till the end of the file.
- **e**: Exit the file system.

How to design a file system

■ How to store files?

● Choose

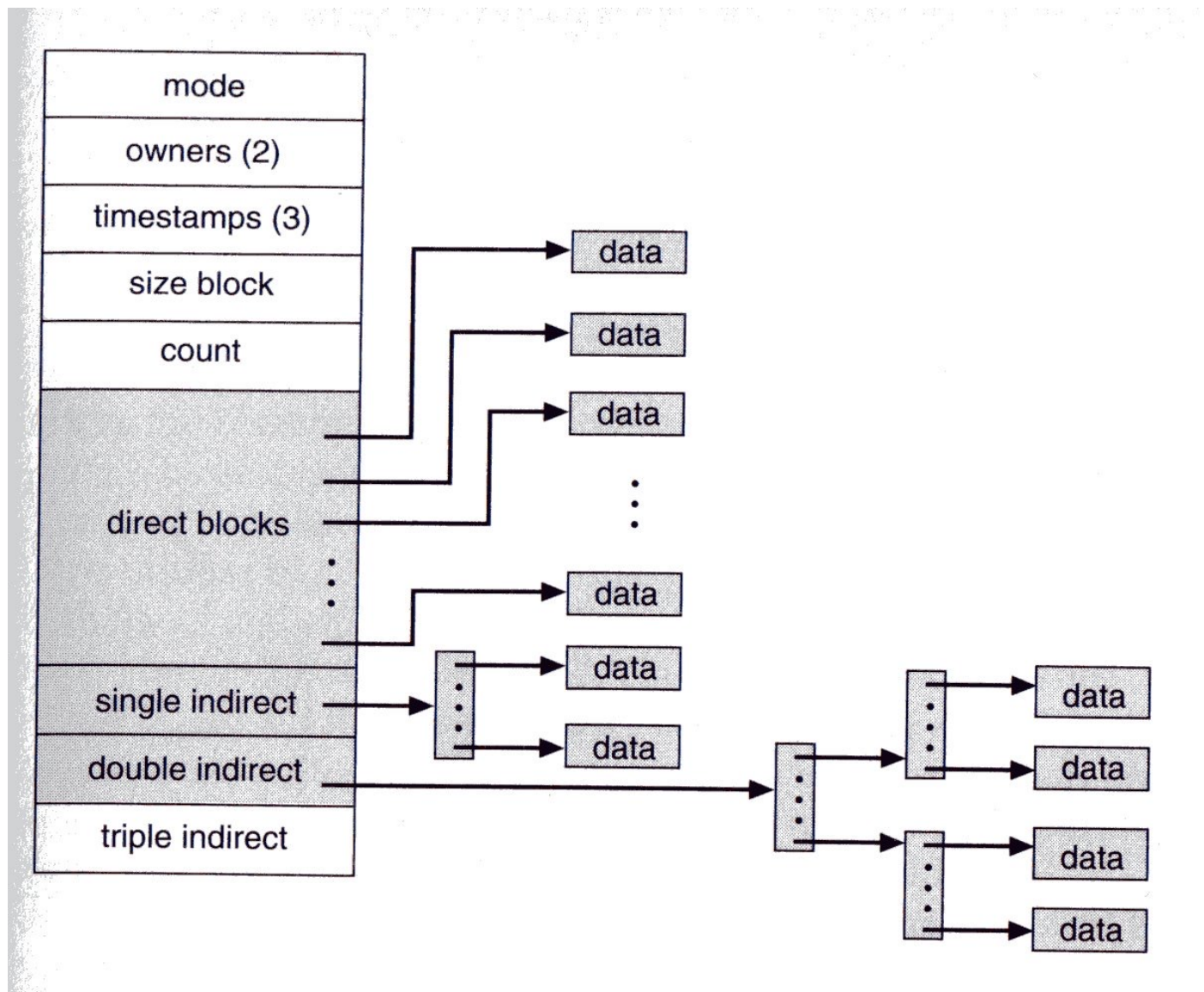
- ▶ FAT
- ▶ inode

File Allocation Table

Filename	Blocks (Cylinder, Sector)
file1	(1, 0)
file2	(1, 3), (1, 4)
file3	(2, 0), (2, 1), (2, 3)
...	...
fileN	(2, 2)

$N \leq \text{\#Cylinder} \times \text{\#Sector}$

inode



inode

■ stat filename

```
lqy@ubuntu:~/Downloads/BestTwo/5092029004/project3/Prj3+5092029004/step2
$ stat Makefile
  File: Makefile
  Size: 155          Blocks: 8          IO Block: 4096   regular file
Device: 801h/2049d  Inode: 918292       Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1000/      lqy)   Gid: ( 1000/      lqy)
Access: 2023-02-05 04:56:01.000000000 -0800
Modify: 2012-06-17 07:45:54.000000000 -0700
Change: 2023-03-26 01:58:56.392531368 -0700
 Birth: -
```

Data structure

- inode
 - links count
 - links
 - next available block
- directory
 - parent
 - records (id , length , name)
 - Pointer that points to next part
 - Next available block

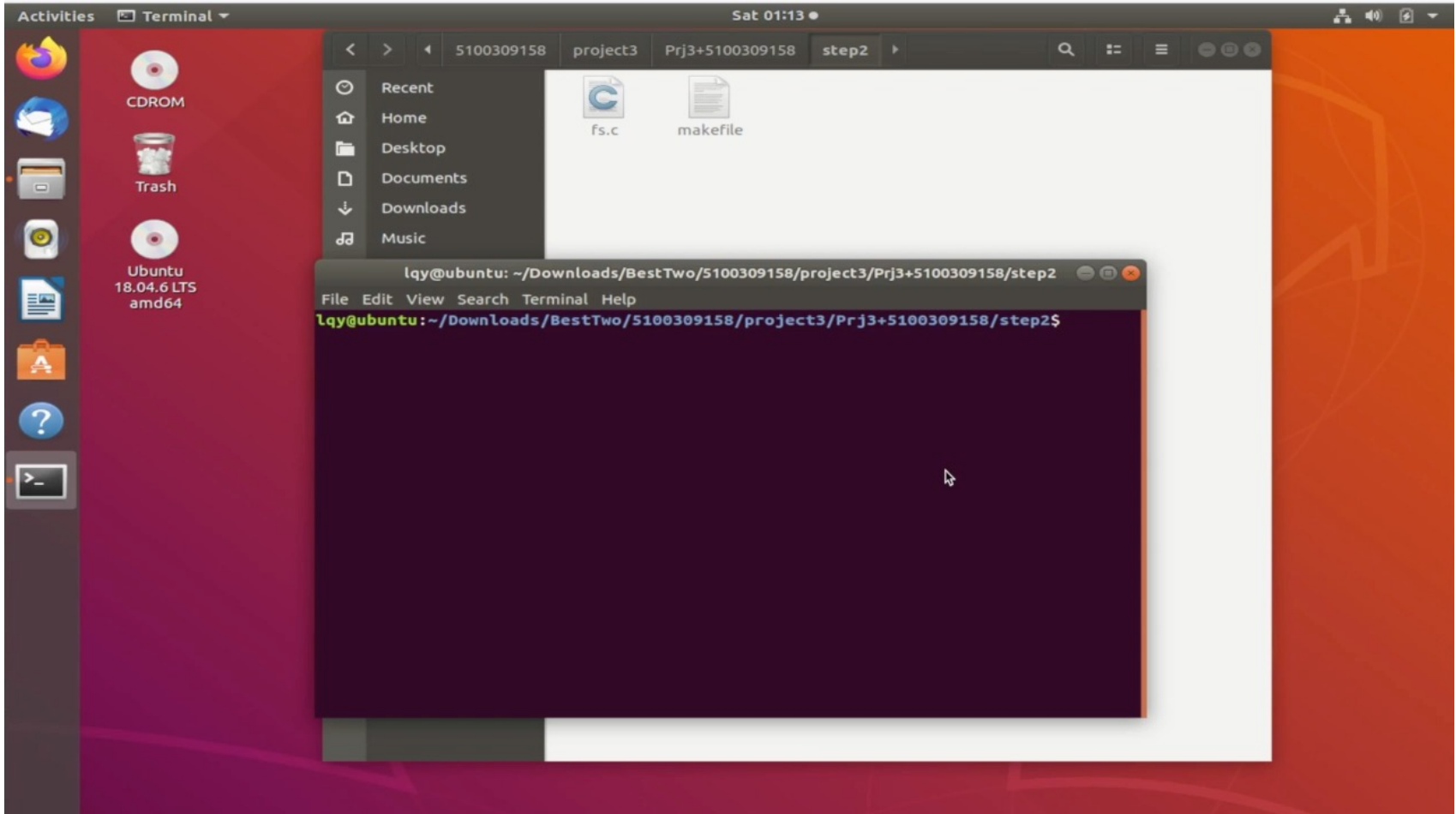
Something you may use

- read()
- write()
- <string.h>
 - strcpy()
 - strcat()
 - strlen()
 - memcpy()

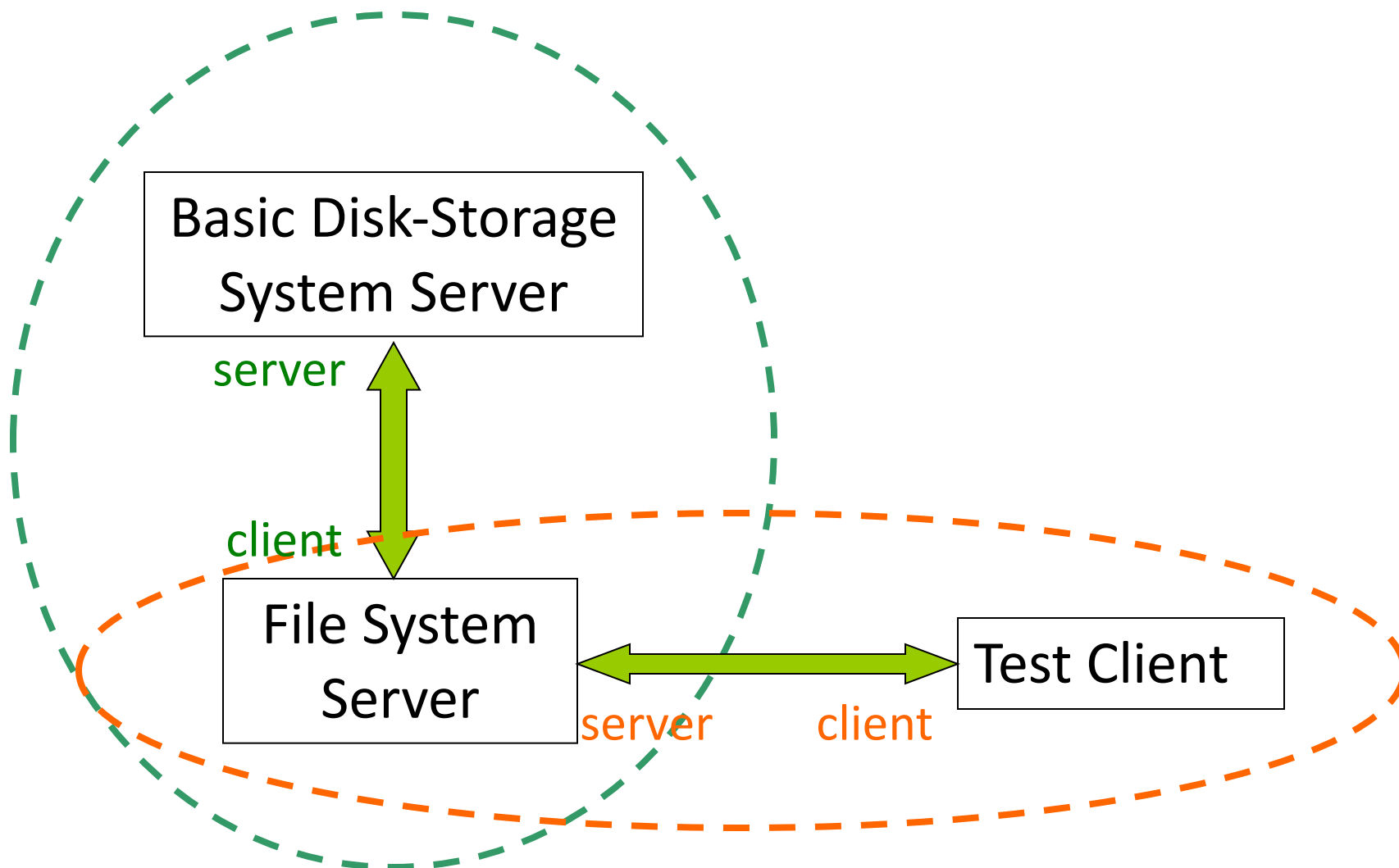
Test your file system

- For each request, output one line.
- For **f**, output **Done**.
- For **mk f**, **mkdir d**, **rm f**, **rmdir d**, **w f l data**, **i f pos l data**, **cd path**, and **d f pos l**, output **Yes**, if successful, or **No** otherwise.
- For **cat f**, if the file **f** exists, output its contents, otherwise output **No**.
- For **ls**, output all the files and directories in current directory. First output all the files, separate by a space, in lexicography order. Then output all the directories, separate by a space, in lexicography order. Output an '&' between the files and directories.
- For **e**, output **Goodbye!**.

demo



Step 3: Overview



Step 3: Overview

■ Command line

- `./disk <#cylinders> <#sectors per cylinder> <track-to-track delay> <#disk-storage-filename> <DiskPort>`
- `./fs <DiskPort> <FSPort>`
- `./client <FSPort>`

Step 3: Connecting a client to a server

```
int sockfd;  
sockfd = socket(AF_INET, SOCK_STREAM, 0);  
    // error check  
struct sockaddr_in serv_addr;  
struct hostent *host;  
serv_addr.sin_family = AF_INET;  
host = gethostbyname(argv[1]);  
    // error check  
memcpy(&serv_addr.sin_addr.s_addr, host->h_addr,  
    host->h_length);  
serv_addr.sin_port = htons(BASIC_SERVER_PORT);  
connect(sockfd, (sockaddr *) &serv_addr, sizeof(serv_addr))  
    // error check
```

demo

