

File I/O

Prof. Jin-Soo Kim(jinsookim@skku.edu)
TA – Taekyun Roh(tkroh0198@skku.edu)
Sewan Ha(hsewan2495@gmail.com)
Computer Systems Laboratory
Sungkyunkwan University
<http://csl.skku.edu>



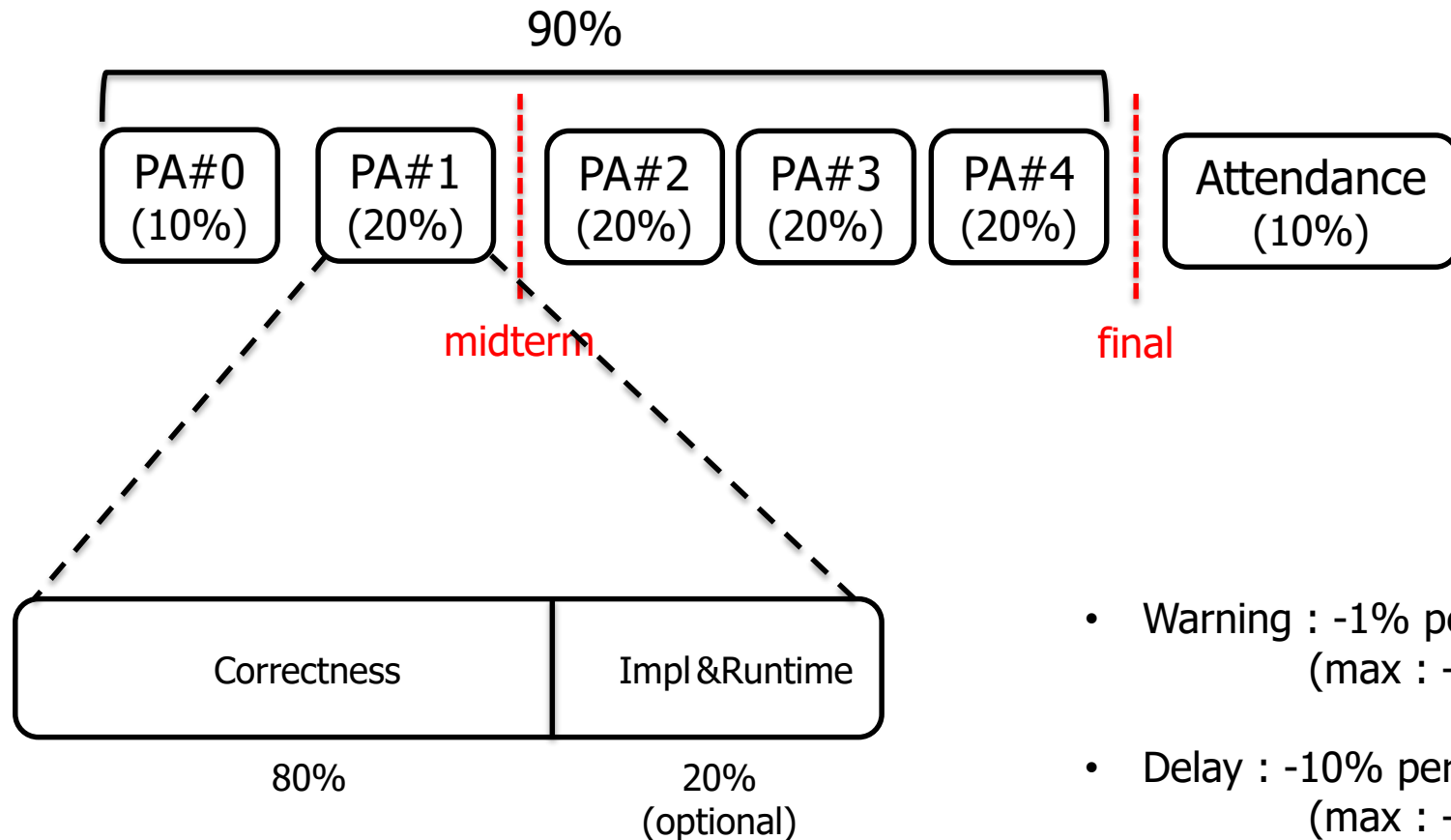
Announcement (1)



- **All the scoring jobs will be done by script**
 - Even if your code has some minor flaws like print format, that would not be allowed
 - Instead, you can fix your code and hand in late.
 - -10% per each day

Announcement (2)

■ Grading policy



- Warning : -1% per line
(max : -10%)
- Delay : -10% per day
(max : -60%)

Contents



- **File in Unix**
- **System calls for File I/O**
- **Standard I/O functions**

Unix Files

- A Unix **file** is a sequence of m bytes:
 - $B_0, B_1, \dots, B_k, \dots, B_{m-1}$
- All I/O devices are represented as files:
 - `/dev/sda1` (hard disk partition)
 - `/dev/tty2` (terminal)
- Even the kernel is represented as a file:
 - `/dev/kmem` (kernel memory image)
 - `/proc` (kernel data structures)

Unix File Types

- **Regular file**
 - Binary or text file
 - Unix does not know the difference!
- **Directory file**
 - A file that contains the names and locations of other files.
- **Character special and block special files**
 - Terminals (character special) and disks (block special)
- **FIFO (named pipe)**
 - A file type used for inter-process communication
- **Socket**
 - A file type used for network communication between processes

Unix I/O



■ Characteristics

- The elegant mapping of files to devices allows kernel to export simple interface called Unix I/O.
- All input and output is handled in a consistent and uniform way ("**byte stream**")

■ Basic Unix I/O operations (system calls):

- Opening and closing files
 - **open()** and **close()**
- Changing the **current file position** (seek)
 - **lseek()**
- Reading and writing a file
 - **read()** and **write()**

Opening Files

- Opening a file informs the kernel that you are getting ready to access that file.

```
int fd;    /* file descriptor */
if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
    perror("open");
    exit(1);
}
```

- Returns a small identifying integer **file descriptor**
 - `fd == -1` indicates that an error occurred
- Each process created by a Unix shell begins life with **three open files associated with a terminal**:
 - 0: standard input
 - 1: standard output
 - 2: standard error

Closing Files

- Closing a file informs the kernel that you are finished accessing that file.

```
int fd;      /* file descriptor */
int retval; /* return value */

if ((retval = close(fd)) < 0) {
    perror("close");
    exit(1);
}
```

- Closing an already closed file is a recipe for disaster in threaded programs (more on this later)
- Moral: Always check return codes, even for seemingly benign functions such as close()

Reading Files

- Reading a file copies bytes from the current file position to memory, and then updates file position.

```
char buf[512];
int fd;      /* file descriptor */
int nbytes;  /* number of bytes read */

/* Open file fd ... */
/* Then read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
    perror("read");
    exit(1);
}
```

- Returns number of bytes read from file `fd` into `buf`
 - `nbytes < 0` indicates that an error occurred.
 - **short counts** (`nbytes < sizeof(buf)`) are possible and are not errors!

Writing Files

- Writing a file copies bytes from memory to the current file position, and then updates current file position.

```
char buf[512];
int fd;      /* file descriptor */
int nbytes;  /* number of bytes read */

/* Open the file fd ... */
/* Then write up to 512 bytes from buf to file fd */
if ((nbytes = write(fd, buf, sizeof(buf))) < 0) {
    perror("write");
    exit(1);
}
```

- Returns number of bytes written from buf to file fd.
 - **nbytes < 0** indicates that an error occurred.
 - As with reads, **short counts** are possible and are not errors!

File Offset

- An offset of an opened file can be set explicitly by calling `lseek()`, `lseek64()`

```
char buf[512];
int fd;          /* file descriptor */
off_t pos;       /* file offset */

/* Get current file offset */
pos = lseek(fd, 0, SEEK_CUR);
/* The file offset is incremented by written bytes */
write(fd, buf, sizeof(buf));
/* Set file position to the first byte of the file */
pos = lseek(fd, 0, SEEK_SET);
```

- Returns the new offset of the file `fd`.
 - `nbytes < 0` indicates that an error occurred.
 - An offset can be set beyond the end of the file.
 - If data is written at that point, a file "hole" is created.

Unix I/O Example

- Copying standard input to standard output one byte at a time.

```
int main(void)
{
    char c;

    while(read(0, &c, 1) != 0)
        write(1, &c, 1);
    exit(0);
}
```

Dealing with Short Counts

- **Short counts can occur in these situations:**
 - Encountering (end-of-file) EOF on reads.
 - Reading text lines from a terminal.
 - Reading and writing network sockets or Unix pipes.
- **Short counts does not occur in these situations:**
 - Reading from disk files (except for EOF)
 - Writing to disk files.
- **How should you deal with short counts in your code?**
 - One way is to use the RIO (Robust I/O) package.

File Metadata

- **Data about data, in this case file data.**
 - Maintained by kernel, accessed by users with the **stat** and **fstat** functions.

```
/* Metadata returned by the stat and fstat functions */
struct stat {
    dev_t      st_dev;      /* device */
    ino_t      st_ino;      /* inode */
    mode_t     st_mode;     /* protection and file type */
    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 type (if inode device) */
    off_t      st_size;     /* total size, in bytes */
    unsigned long st_blksize; /* blocksize for filesystem I/O */
    unsigned long st_blocks; /* number of blocks allocated */
    time_t     st_atime;     /* time of last file access */
    time_t     st_mtime;     /* time of last file modification */
    time_t     st_ctime;     /* time of last inode change */
}; /* statbuf.h included by sys/stat.h */
```

Accessing File Metadata

```
/* statcheck.c - Querying and manipulating a file's meta data */
```

```
int main (int argc, char **argv)
{
    struct stat st;
    char *type, *readok;

    stat(argv[1], &st);
    if (S_ISREG(st.st_mode)) /* file type */
        type = "regular";
    else if (S_ISDIR(st.st_mode))
        type = "directory";
    else
        type = "other";
    if ((st.st_mode & S_IRUSR)) /* OK to read? */
        readok = "yes";
    else
        readok = "no";

    printf("type: %s, read: %s\n", type, readok);
    exit(0);
}
```

```
bass> ./statcheck statcheck.c
type: regular, read: yes
bass> chmod 000 statcheck.c
bass> ./statcheck statcheck.c
type: regular, read: no
```


Standard I/O Functions

- The C standard library (libc.a) contains a collection of higher-level **standard I/O** functions
- Examples of standard I/O functions:
 - Opening and closing files (**fopen** and **fclose**)
 - Reading and writing bytes (**fread** and **fwrite**)
 - Reading and writing text lines (**fgets** and **fputs**)
 - Formatted reading and writing (**fscanf** and **fprintf**)

Standard I/O Streams

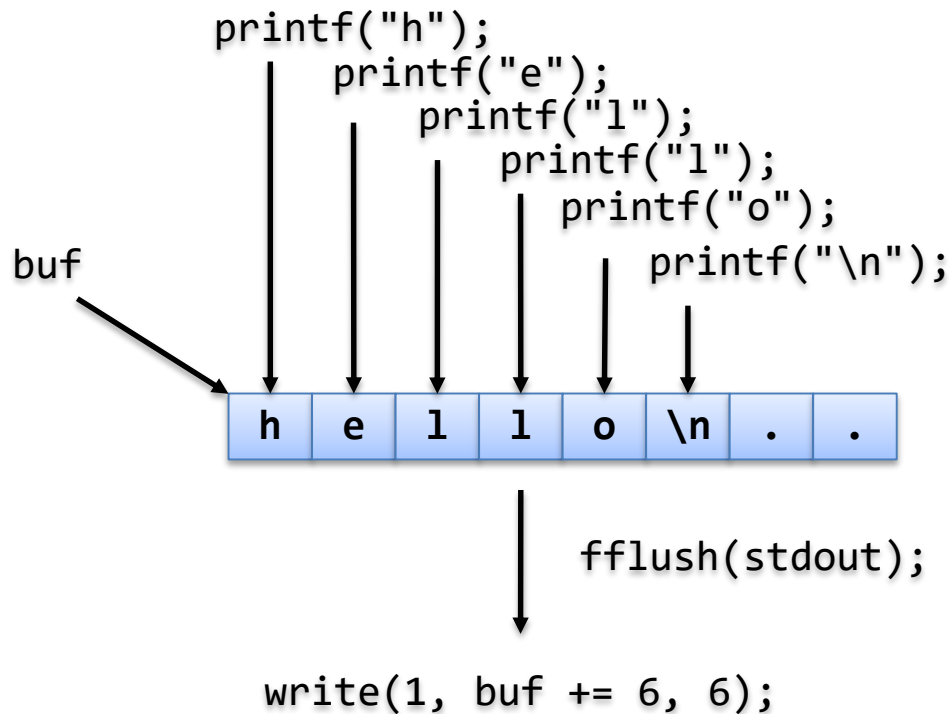
- **Standard I/O models open files as **streams****
 - Abstraction for a file descriptor and a buffer in memory
- **C programs begin life with three open streams (defined in `stdio.h`)**
 - **`stdin`** (standard input)
 - **`stdout`** (standard output)
 - **`stderr`** (standard error)

```
#include <stdio.h>
extern FILE *stdin; /* standard input (descriptor 0) */
extern FILE *stdout; /* standard output (descriptor 1) */
extern FILE *stderr; /* standard error (descriptor 2) */

int main() {
    fprintf(stdout, "Hello, world\n");
}
```

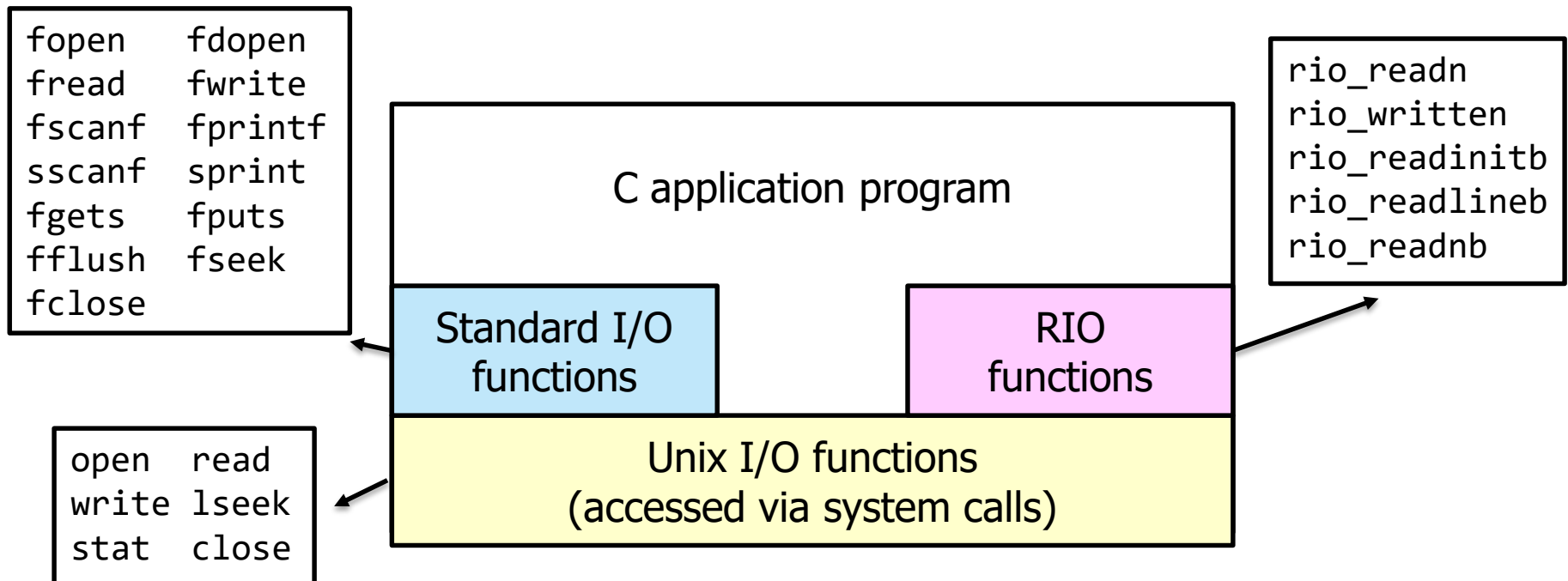
Buffering in Standard I/O

- Standard I/O functions use buffered I/O



Unix I/O vs. Standard I/O

- Standard I/O are implemented using low-level Unix I/O.



- Which ones should you use in your programs?

Pros/Cons of Unix I/O



■ Pros

- The most general and lowest overhead form of I/O.
 - All other I/O packages are implemented on top of Unix I/O functions.
- Unix I/O provides functions for accessing file metadata.

■ Cons

- System call overheads for small-sized I/O.
- Dealing with short counts is tricky and error prone.
- Efficient reading of text lines requires some form of buffering, also tricky and error prone.
- These issues are addressed by the standard I/O.

Pros/Cons of Standard I/O

■ Pros

- Buffering increases efficiency by decreasing the number of **read()** and **write()** system calls.
- Shout counts are handled automatically.

■ Cons

- Provides no function for accessing file metadata.
- Standard I/O is not appropriate for input and output on network sockets.
 - But there is a way using **fdopen()**

Summary

■ Unix file I/O

- `open()`, `read()`, `write()`, `close()`, ...
- A uniform way to access files, I/O devices, network sockets, kernel data structures, etc.

■ When to use standard I/O?

- When working with disk or terminal files.

■ When to use raw Unix I/O

- When you need to fetch file metadata.
- When you read or write network sockets or pipes.
- In rare cases when you need absolute highest performance.

Remind



▪ 6 System calls

- `open()`
- `close()`
- `read()`
- `write()`
- `lseek()`
- `stat()` / `fstat()`

Header



■ Header

- `stdio.h`
 - `Printf()`
- `string.h`
 - `Strcpy()`
- `fcntl.h`
 - `O_RDWR`
- `unistd.h`
 - `write()`
 - `close()`

Example #1 (1)

```
char filename[] = "hello-dos.txt";
int fd;
char buffer[16];
off_t pos = 0; // long long;

fd = open(filename, O_RDWR | O_CREAT, 0755);
read(fd, buffer, 6);
read(fd, buffer+6, 2);
```

```
lseek(fd, -2, SEEK_CUR);
buffer[0] = '\n';
write(fd, buffer, 1);
```

```
lseek(fd, 8, SEEK_SET);
strcpy(buffer, "How");
write(fd, buffer, 3);
```

```
close(fd);
```

```
fd = open(filename, O_WRONLY | O_CREAT | O_EXCL, 0755);
if (fd < 0)
    printf("errno : %d, error code - EEXIST : %d\n", errno, EEXIST);
```

File state (FD: 3)

path: "hello-dos.txt"
position: 0
size: 20


Permissions (octal)

0755

Submit

Permission bits

Special	User	Group	Other
✗ setuid is unset	✓ can read	✓ can read	✓ can read
✗ setgid is unset	✓ can write	✗ cannot write	✗ cannot write
✗ Sticky bit is unset	✓ can execute	✓ can execute	✓ can execute



H	e	l	l	o	.	\r	\n
W	h	o		a	r	e	
y	o	u	?	EOF			

Example #1 (2)

```
char filename[] = "hello-dos.txt";
int fd;
char buffer[16];
off_t pos = 0; // long long;

fd = open(filename, O_RDWR | O_CREAT, 0755);
read(fd, buffer, 6); // "Hello."
read(fd, buffer+6, 2);
```

File state (FD: 3)

path: "hello-dos.txt"
position: 6
size: 20

Header: unistd.h


Format: ssize_t read (**int** fd, **void** *buf, **size_t** nbytes)

```
lseek(fd, -2, SEEK_CUR);
buffer[0] = '\n';
write(fd, buffer, 1);
```

```
lseek(fd, 8, SEEK_SET);
strcpy(buffer, "How");
write(fd, buffer, 3);
```

```
close(fd);
```

```
fd = open(filename, O_WRONLY | O_CREAT | O_EXCL, 0755);
if (fd < 0)
    printf("errno : %d, error code - EEXIST : %d\n", errno, EEXIST);
```



H	e	l	l	o	.	\r	\n
W	h	o		a	r	e	
y	o	u	?	EOF			

Example #1 (3)

```
char filename[] = "hello-dos.txt";
int fd;
char buffer[16];
off_t pos = 0; // long long;

fd = open(filename, O_RDWR | O_CREAT, 0755);
read(fd, buffer, 6);
read(fd, buffer+6, 2); // "Hello.\r\n"
```

```
lseek(fd, -2, SEEK_CUR);
buffer[0] = '\n';
write(fd, buffer, 1);
```


```
lseek(fd, 8, SEEK_SET);
strcpy(buffer, "How");
write(fd, buffer, 3);
```

```
close(fd);
```

```
fd = open(filename, O_WRONLY | O_CREAT | O_EXCL, 0755);
if (fd < 0)
    printf("errno : %d, error code - EEXIST : %d\n", errno, EEXIST);
```

File state (FD: 3)

path: "hello-dos.txt"
position: 8
size: 20



.	e	l	l	o	.	\r	\n
W	h	o		a	r	e	
y	o	u	?	EOF			

Example #1 (4)

```
char filename[] = "hello-dos.txt";
int fd;
char buffer[16];
off_t pos = 0; // long long;

fd = open(filename, O_RDWR | O_CREAT, 0755);
read(fd, buffer, 6);
read(fd, buffer+6, 2);
```

File state (FD: 3)

path: "hello-dos.txt"
position: **6**
size: 20

```
lseek(fd, -2, SEEK_CUR);
buffer[0] = '\n';
write(fd, buffer, 1);
```


Header: unistd.h

Format: off_t lseek(**int** fd, **off_t** offset, **int** whence)

```
lseek(fd, 8, SEEK_SET);
strcpy(buffer, "How");
write(fd, buffer, 3);
```

```
close(fd);
```

```
fd = open(filename, O_WRONLY | O_CREAT | O_EXCL, 0755);
if (fd < 0)
    printf("errno : %d, error code - EEXIST : %d\n", errno, EEXIST);
```



H	e	l	l	o	.	\r	\n
W	h	o		a	r	e	
y	o	u	?	EOF			

Example #1 (5)

```
char filename[] = "hello-dos.txt";
int fd;
char buffer[16];
off_t pos = 0; // long long;

fd = open(filename, O_RDWR | O_CREAT, 0755);
read(fd, buffer, 6);
read(fd, buffer+6, 2);
```

```
lseek(fd, -2, SEEK_CUR);
buffer[0] = '\n';
write(fd, buffer, 1);
```


```
lseek(fd, 8, SEEK_SET);
strcpy(buffer, "How");
write(fd, buffer, 3);
```

```
close(fd);
```

```
fd = open(filename, O_WRONLY | O_CREAT | O_EXCL, 0755);
if (fd < 0)
    printf("errno : %d, error code - EEXIST : %d\n", errno, EEXIST);
```

File state (FD: 3)

path: "hello-dos.txt"
position: 7
size: 20



H	e	l	l	o	.	\n	\n
W	h	o		a	r	e	
y	o	u	?	EOF			

Example #1 (6)

```
char filename[] = "hello-dos.txt";
int fd;
char buffer[16];
off_t pos = 0; // long long;

fd = open(filename, O_RDWR | O_CREAT, 0755);
read(fd, buffer, 6);
read(fd, buffer+6, 2);
```

```
lseek(fd, -2, SEEK_CUR);
buffer[0] = '\n';
write(fd, buffer, 1);
```

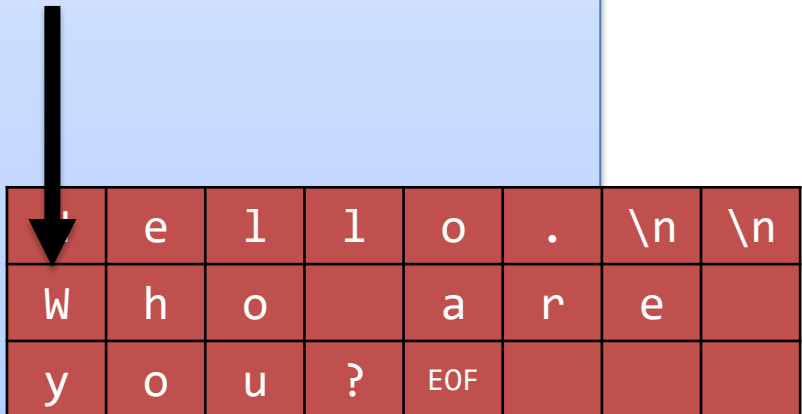
```
lseek(fd, 8, SEEK_SET);
strcpy(buffer, "How");
write(fd, buffer, 3);
```

```
close(fd);
```

```
fd = open(filename, O_WRONLY | O_CREAT | O_EXCL, 0755);
if (fd < 0)
    printf("errno : %d, error code - EEXIST : %d\n", errno, EEXIST);
```

File state (FD: 3)

path: "hello-dos.txt"
position: **8**
size: 20



h	e	l	l	o	.	\n	\n
W	h	o		a	r	e	
y	o	u	?	EOF			

Example #1 (7)

```
char filename[] = "hello-dos.txt";
int fd;
char buffer[16];
off_t pos = 0; // long long;

fd = open(filename, O_RDWR | O_CREAT, 0755);
read(fd, buffer, 6);
read(fd, buffer+6, 2);
```

```
lseek(fd, -2, SEEK_CUR);
buffer[0] = '\n';
write(fd, buffer, 1);
```


```
lseek(fd, 8, SEEK_SET);
strcpy(buffer, "How");
write(fd, buffer, 3);
```

```
close(fd);
```

```
fd = open(filename, O_WRONLY | O_CREAT | O_EXCL, 0755);
if (fd < 0)
    printf("errno : %d, error code - EEXIST : %d\n", errno, EEXIST);
```

File state (FD: 3)

path: "hello-dos.txt"
position: 11
size: 20



H	e	l		o	.	\n	\n
H	o	w		a	r	e	
y	o	u	?	EOF			

Example #1 (8)

```
char filename[] = "hello-dos.txt";  
int fd;  
char buffer[16];  
off_t pos = 0; // long long;
```

```
fd = open(filename, O_RDWR | O_CREAT, 0755);  
read(fd, buffer, 6);  
read(fd, buffer+6, 2);
```

```
lseek(fd, -2, SEEK_CUR);  
buffer[0] = '\n';  
write(fd, buffer, 1);
```

```
lseek(fd, 8, SEEK_SET);  
strcpy(buffer, "How");  
write(fd, buffer, 3);
```

```
close(fd);
```

```
fd = open(filename, O_WRONLY | O_CREAT | O_EXCL, 0755);  
if (fd < 0)  
    printf("errno : %d, error code - EEXIST : %d\n", errno, EEXIST);
```

File state (FD: 3)
:CLOSED

Example #1 (9)

```
char filename[] = "hello-dos.txt";
int fd;
char buffer[16];
off_t pos = 0; // long long;

fd = open(filename, O_RDWR | O_CREAT, 0755);
read(fd, buffer, 6);
read(fd, buffer+6, 2);

lseek(fd, -2, SEEK_CUR);
buffer[0] = '\n';
write(fd, buffer, 1);

lseek(fd, 8, SEEK_SET);
strcpy(buffer, "How");
write(fd, buffer, 3);

close(fd);

fd = open(filename, O_WRONLY | O_CREAT | O_EXCL, 0755);
if (fd < 0)
    printf("errno : %d, error code - EEXIST : %d\n", errno, EEXIST);
```

Example #2 (1)

```
char filename[] = "hello-dos.txt";
int fd;
char buffer[16];
off_t pos = 0;


fd = open(filename, O_WRONLY);
lseek(fd, 1, SEEK_END);

strcpy(buffer, "!\n");
write(fd, buffer, 2);

close(fd);
```

File state (FD: 3)

path: "hello-dos.txt"
type: Regular
mode: Write only
position: 0
size: 20



H	e	l	l	o	.	\r	\n
W	h	o		a	r	e	
y	o	u	?	EOF			

Example #2 (2)

```
char filename[] = "hello-dos.txt";  
int fd;  
char buffer[16];  
off_t pos = 0;
```

```
fd = open(filename, O_WRONLY);  
lseek(fd, 1, SEEK_END);
```

```
strcpy(buffer, "!\n");  
write(fd, buffer, 2);
```

```
close(fd);
```

File state (FD: 3)

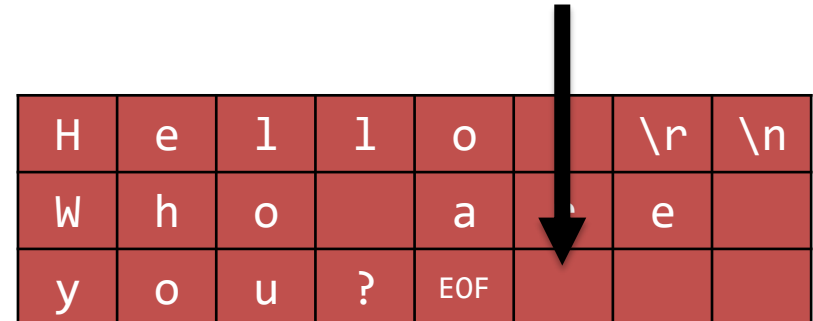
path: "hello-dos.txt"

type: Regular

mode: Write only

position: 21

size: 20



H	e	l	l	o		\r	\n
W	h	o		a		e	
y	o	u	?	EOF			

Example #2 (3)

```
char filename[] = "hello-dos.txt";
int fd;
char buffer[16];
off_t pos = 0;

fd = open(filename, O_WRONLY);
lseek(fd, 1, SEEK_END);

strcpy(buffer, "!\n");
write(fd, buffer, 2);

close(fd);
```

File state (FD: 3)

path: "hello-dos.txt"
type: Regular
mode: Write only
position: 23
size: 23

H	e	l	l	o	.	\r	\n
W	h	o		a	r	e	
y	o	u	?	\0	!	\n	EOF

↑
Data hole

Example #2 (4)

```
char filename[] = "hello-dos.txt";
int fd;
char buffer[16];
off_t pos = 0;

fd = open(filename, O_WRONLY);
lseek(fd, 1, SEEK_END);

strcpy(buffer, "!\n");
write(fd, buffer, 2);

close(fd);
```

File state (FD: 3)
:CLOSED

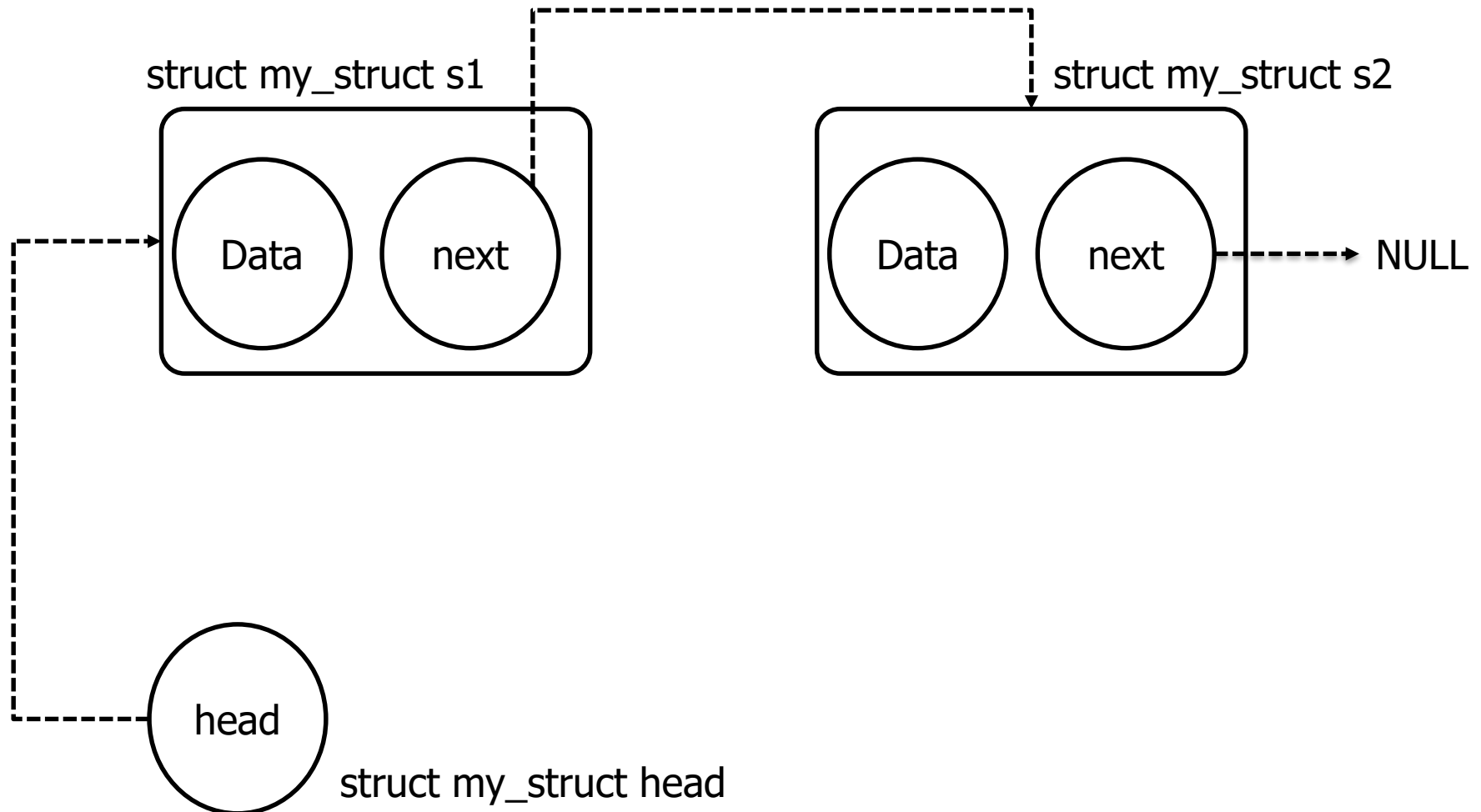
Exercise

- **Lab exercise #1:**
 - Let's make *xcat*, *xcp* utilities using "buffering".
 - *xcat* prints a file on the standard output.
 - *xcp* copies contents of a file into a new file.
 - Basically, executing *xcat* and *xcp* will be same as executing *cat* and *cp* without any options, respectively.
- **Your job is to make *xcat* and *xcp* by using system calls provided by Linux.**

Malloc / Free

- Both need `<stdlib.h>`
- Malloc
 - `struct my_struct *p = (struct my_struct *)malloc(sizeof(my_struct));`
- Free
 - `free(p)`

Linked List



Exercise

- Lab exercise #2:
 - Let's make *xsort* utility using linked list
 - *cat in.txt | ./xsort > out.txt*
 - It should work same as “sort”
- Your job is to make *xsort* by using system calls provided by Linux and `<stdlib>`.

Test of xsort

- We have `genesis.txt`
- `diff -bsq sort.out xsort.out`
- **Option**
 - -b: Ignore consecutive spaces
 - -s: Check the two files are the same
 - -q: Output only differences between two files