SSE2033

SUNG KYUN KWAN UNIVERSITY

Fall 2019

System Software Experiment 2

File I/O

Prof. Jinkyu Jeong (*jinkyu@skku.edu*)
TA – Gyusun Lee (*gyusun.lee@csi.skku.edu*)
TA – Jiwon Woo (*jiwon.woo@csi.skku.edu*)
Computer Systems and Intelligence Laboratory (*http://csi.skku.edu*)
Sungkyunkwan University

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:
  - $B_0, B_1, \ldots, B_k, \ldots, B_{m-1}$
- All I/O devices are represented as files:
  - **/dev/sda1** (hard disk partition)
  - **/dev/tty2** (terminal)
    - Ctrl + Alt + F1 ~ F7
- Even the kernel is represented as a file:
  - **/dev/mem** (kernel memory image)
  - **/proc** (kernel data structures)

Unix File Types

- Regular file
  - Contains arbitrary data
- 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);
}
```

- 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?

Dealing with Short Counts

```
ssize_t rio_readn(int fd, void *usrbuf, size_t n)
{
    size_t nleft = n;
    ssize_t nread;
    char *bufp = usrbuf;

    while (nleft > 0) {
         if ((nread = read(fd, bufp, nleft)) < 0) {
                 if (errno == EINTR) /* interrupted by sig handler return */
                         nread = 0; /* and call read() again */
                 else
                         return -1; /* errno set by read() */
         }
         else if (nread == 0)
                 break; /* EOF */
         nleft -= nread;
         bufp += nread;
     }
     return (n - nleft); /* return >= 0 */
}
```

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.so) 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**)

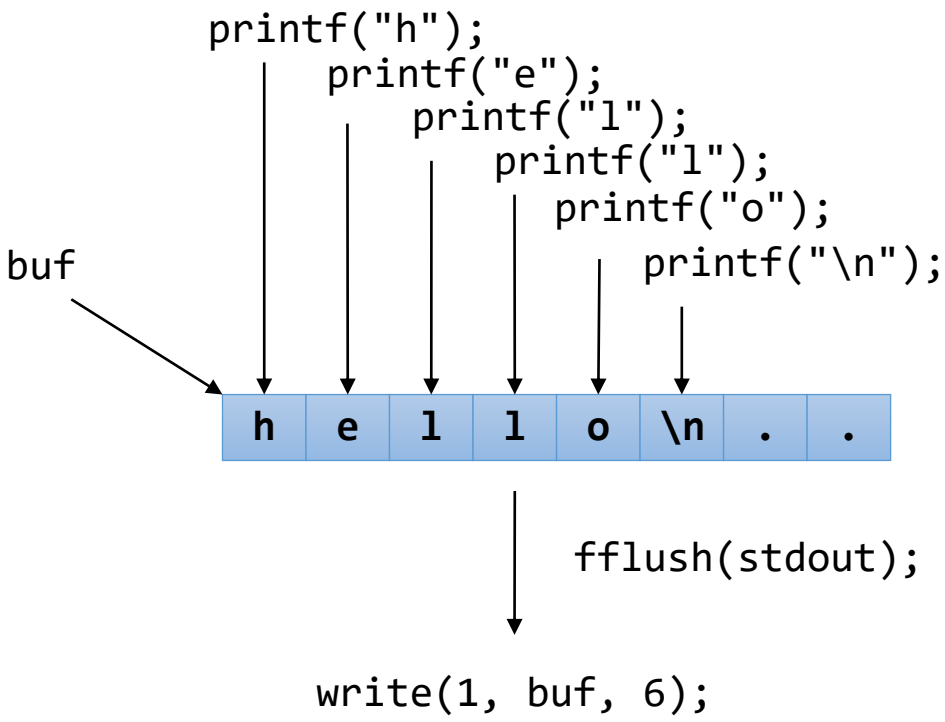
Standard I/O Streams

- 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



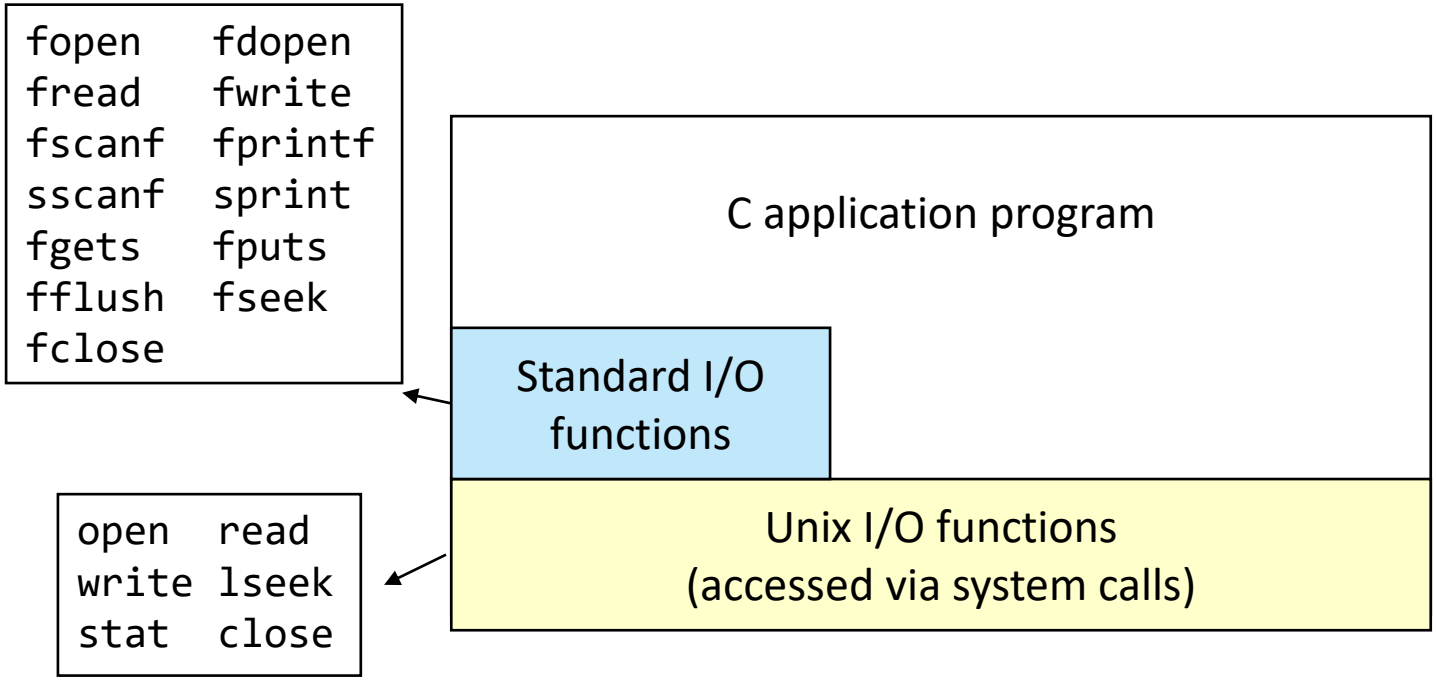
Buffering in Standard I/O

```
#include <stdio.h>

int main()
{
    printf("h");
    printf("e");
    printf("l");
    printf("l");
    printf("o");
    printf("\n");
    fflush(stdout);
    exit(0);
}
```

```
linux> strace ./hello
execve("./hello", ["hello"], [/* ... */]).
...
write(1, "hello\n", 6)                = 6
...
exit_group(0)                         = ?
```

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
  - Buffuring 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()**

perror(), errno

- When a system call fails,
  - Returns -1 (or NULL for certain library functions)
  - The latest error information is stored in “errno”.

- perror()
  - Explain error information stored in errno.
  - Print out the information through stderr stream.
- errno
  - Stores int value indicating cause of error.
  - int type extern global variable.
  - Thread-safe!

errno.h

- /usr/include/asm-generic/errno-base.h

```
#ifndef _ASM_GENERIC_ERRNO_BASE_H
#define _ASM_GENERIC_ERRNO_BASE_H

#define EPERM        1  /* Operation not permitted */
#define ENOENT       2  /* No such file or directory */
#define ESRCH        3  /* No such process */
#define EINTR        4  /* Interrupted system call */
#define EIO      5  /* I/O error */
#define ENXIO        6  /* No such device or address */
#define E2BIG        7  /* Argument list too long */
#define ENOEXEC      8  /* Exec format error */
#define EBADF        9  /* Bad file number */
#define ECHILD      10  /* No child processes */
#define EAGAIN      11  /* Try again */
#define ENOMEM      12  /* Out of memory */
#define EACCES      13  /* Permission denied */
#define EFAULT      14  /* Bad address */
#define ENOTBLK     15  /* Block device required */
#define EBUSY       16  /* Device or resource busy */
#define EEXIST      17  /* File exists */
#define EXDEV       18  /* Cross-device link */
#define ENODEV      19  /* No such device */
#define ENOTDIR     20  /* Not a directory */
#define EISDIR      21  /* Is a directory */
#define EINVAL      22  /* Invalid argument */
#define ENFILE      23  /* File table overflow */
#define EMFILE      24  /* Too many open files */
#define ENOTTY      25  /* Not a typewriter */
#define ETXTBSY     26  /* Text file busy */
#define EFBIG       27  /* File too large */
#define ENOSPC      28  /* No space left on device */
#define ESPIPE      29  /* Illegal seek */
#define EROFS       30  /* Read-only file system */
#define EMLINK      31  /* Too many links */
#define EPIPE       32  /* Broken pipe */
#define EDOM        33  /* Math argument out of domain of func */
#define ERANGE      34  /* Math result not representable */

#endif
```

Defined as integers.
Stored in errno when error occurs.
Also accessible through “man errno”

Handling system call errors

```
#include <stdio.h>

int main()
{
        FILE *fp;
        fp = fopen(“file.txt”, “r”);
        if(fp == NULL){
                perror(“Error”);
                return (-1);
        }
        fclose(fp);
        return 0;
}
```

```
linux> ./test
linux> Error: No such file or directory
```

Handling system call errors

```
#include <stdio.h>

int main()
{
        FILE *fp;
        printf("Hello world!\n");
        fp = fopen("file.txt", "r");
        if(fp == NULL){
                perror("Error");
                return (-1);
        }
        fclose(fp);
        return 0;
}
```

stdout

stderr

Send stdout only to result.txt

```
linux> ./test > result.txt
linux> Error: No such file or directory
```

```
Hello world!
```

"re.txt"

Handling system call errors

```
#include <stdio.h>

int main()
{
        FILE *fp;
        printf("Hello world!\n");
        fp = fopen("file.txt", "r");
        if(fp == NULL){
                perror("Error");
                return (-1);
        }
        fclose(fp);
        return 0;
}
```

stdout

stderr

Send stderr to stdout's dest

```
linux> ./test > result.txt 2>&1
linux>
```

```
Error: No such file or directory
Hello world!


"re.txt"
```

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()`

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
```

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

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);

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: 6
size: 20
```

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

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	?				

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);

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: 6
size: 20
```

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

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	?				

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);
```

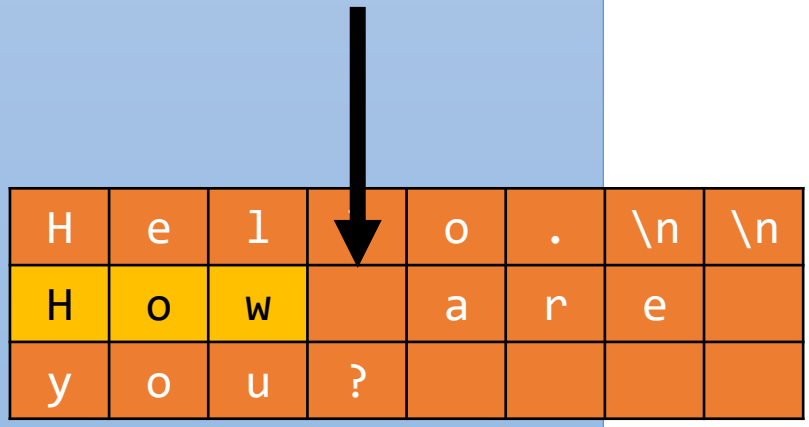
```
File state (FD: 3)

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

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

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
```



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);
```

Exercise

- Lab exercise #1:
  - Let's make ***xtar*** utility
  - ***tar -cf img.tar 1.jpg 2.jpg***
  - ***tar -xf img.tar***
  - It should work same as "tar"

- Download test files on web.
  - $wget http://csl.skku.edu/uploads/SSE2033F18/lab2.tar.gz
  - $tar –xvzf lab2.tar.gz
  - You can use 2 jpg files for testing your own tar program.

Exercise

- Handling error cases
  - Not enough input parameters
    - *xtar –cf img.tar*
  - Omitting operation
    - *xtar img.tar 1.jpg 2.jpg*
  - Wrong access
    - *xtar –cf img.tar No_file.jpg 1.jpg 2.jpg*
  - etc..

  - Return error code.
  - Print error code & proper error message for each situation.