SWE2024

SUNG KYUN KWAN UNIVERSITY

File I/O

Prof. Joonwon Lee (joonwon@skku.edu)
TA – Jaehyung Park (jaeseanpark@gmail.com)
TA – Luke Albano (lukealbano@arcs.skku.edu)
Sungkyunkwan University

Contents

- File/directory concept
- Unix I/O
- Standard I/O
- Error handling for file I/O

File/Directory Concept

What is File?

- A Linux **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:
  - **/boot/vmlinux-5.4.0-42-generic** (kernel image)
  - **/dev/mem** (kernel memory image)
  - **/proc** (kernel data structures)

File Types

- Each file has a *type* indicating its role in the system
  - **Regular file**: contains arbitrary data
  - **Directory**: index for a related group of files
  - **Socket**: for communicating with a process on another machine

- Other file types beyond our scope
  - Named pipes (FIFOs)
  - Symbolic links
  - Character and block devices

Regular File

- A regular file contains arbitrary data

- **Applications** often distinguish between ***text files*** and ***binary files***
  - Text files are regular files with only ASCII or Unicode characters
  - Binary files are everything else
    - e.g. object files, JPEG images
  - **Kernel doesn't know the difference!**

- Text file is sequence of text lines
  - Text line is sequence of *chars* terminated by newline *char* ('**\n**')

Directory

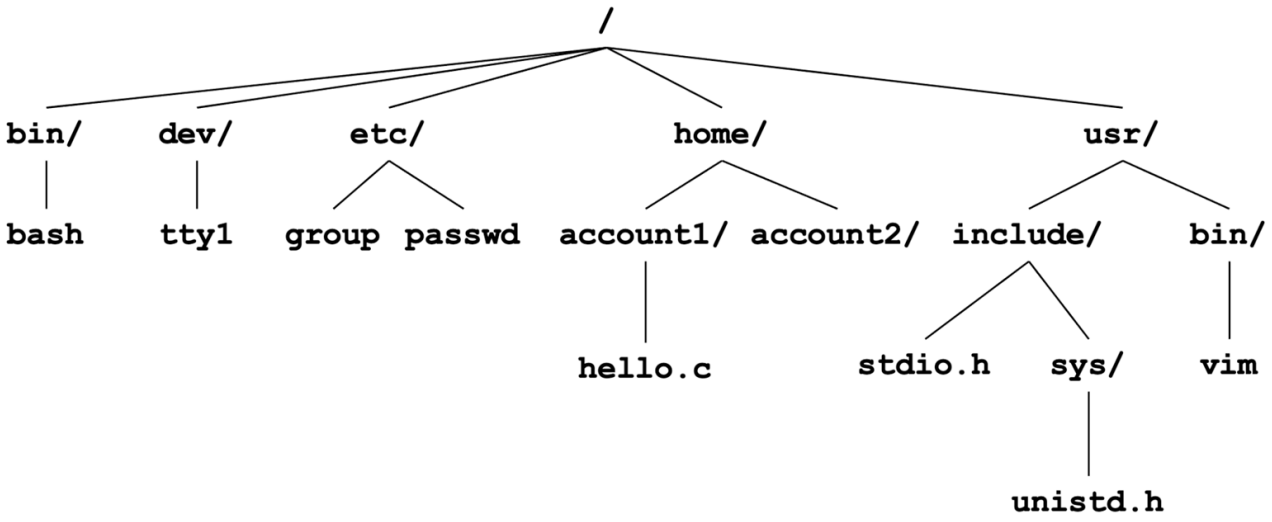
- Directory consists of an array of *links*
  - Each link maps a *filename* to a file

- Each directory contains at least two entries
  - `.` (dot) is a link to itself
  - `..` (dot dot, double dot) is a link to the parent directory in the directory hierarchy

- Commands for manipulating directories
  - **`mkdir`**: create empty directory
  - **`ls`** : view directory contents
  - **`rmdir`** : delete empty directory

Directory Hierarchy

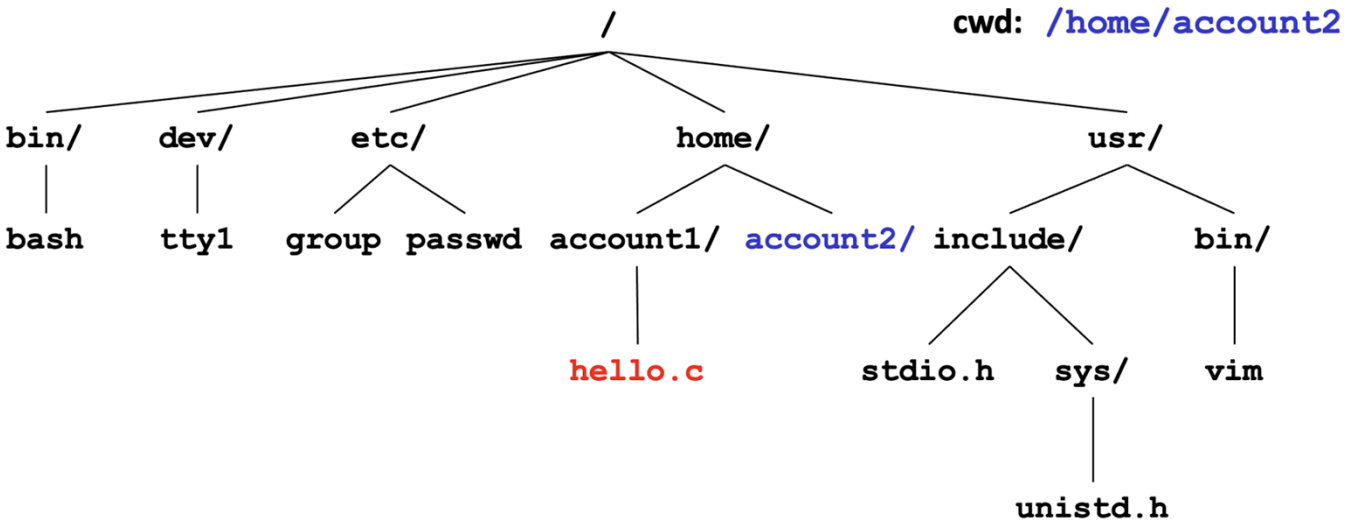
- All files are organized as a hierarchy anchored by root directory named '/' (slash)



- Kernel maintains *current working directory* (cwd) for each process
  - Modified using the **cd** command

Pathnames

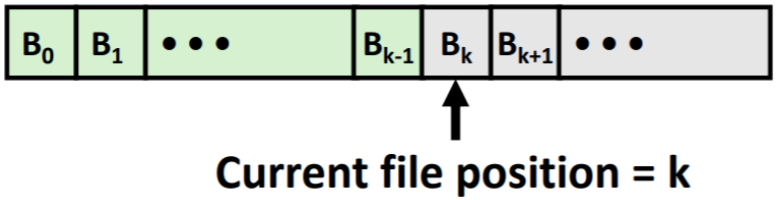
- Locations of files in the hierarchy denoted by *pathnames*
  - *Absolute pathname* starts with '/' and denotes path from root
    - `/home/account1/hello.c`
  - *Relative pathname* denotes path from current working directory
    - `../home/account1/hello.c`



Unix I/O

Unix I/O Overview

- Kernel exports a simple interface called *Unix I/O*
  - Opening and closing files
    - **`open()`** and **`close()`**
  - Reading and writing a file
    - **`read()`** and **`write()`**
  - Changing the current file position (seek)
    - Indicates next offset into file to read or write
    - **`lseek()`**



- All input and output is handled in a consistent and uniform way ("byte stream")

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 (stdin)
  - 1: standard output (stdout)
  - 2: standard error (stderr)

Opening File Flags

- Flags for `open()` function
  - O_RDONLY: open for reading file
  - O_WRONLY: open for writing file
  - O_RDWR: open for reading and writing
  - O_CREAT: create the file if it doesn't exist
  - O_APPEND: append to the end of file

- Flags when create file with O_CREAT flag
  - ***mode*** flag locates third argument position in open system call
    - `int open(const char *pathname, int flags, mode_t mode)`
  - e.g. S_IRWXU, S_IRUSR, S_IRGRP, S_IROTH
    - 0764 equals `S_IRWXU | S_IRGRP | S_IWGRP | S_IROTH`

Closing Files

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

```c
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

```c
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

```c
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()`**

```c
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

Simple Unix I/O Example

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

```
int main(void) {
    char c;
    while(read(STDIN_FILENO, &c, 1) != 0)
        write(STDOUT_FILENO, &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
- Short counts does not occur in these situations:
  - Reading from disk files (except for EOF)
  - Writing to disk files
- Best practice is to always allow for short counts

Handling 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 signal 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 */
}
```

Accessing Directory

- Only recommended operation on a directory: read its ***entries***
  - **`dirent`** structure contains information about a directory entry

```
struct dirent {
   ino_t       d_ino;      /* inode number */
   off_t       d_off;      /* offset to the next dirent */
   unsigned short d_reclen;    /* length of this record */
   unsigned char  d_type;     /* type of file */
   char        d_name[256]; /* filename */
};
```

  - **`DIR`** structure contains information about directory while stepping through its entries (see example in the next slide)
  - **`opendir()`** and **`closedir()`**
    - Open and close directory stream
  - **`readdir()`**
    - Return a pointer to a **`dirent`** structure representing directory entry
    - Return **`NULL`** on reaching the end of the directory stream

Example of Accessing Directory

```c
#include <sys/types.h>
#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>

/* dircheck.c - Opening directory and reading its entries */
int main(int argc, char **argv) {
   DIR *directory;
   struct dirent *de;

   if (!(directory = opendir(argv[1]))) {
      perror("Failed to open directory");
   exit(1);
   }

   while (0 != (de = readdir(directory)))
      printf("Found: %s\n", de->d_name);

   closedir(directory);
   exit(0);
}
```

File Metadata

- **Metadata** is data about data, in this case file data
  - Per-file metadata 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 */
```

System Calls for File Metadata

- An information about a file can be obtained explicitly by calling **`stat()`, `fstat()`, `lstat()`**
  - Three header files must be included

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
```

  - **`int stat(const char *path, struct stat *buf)`**
    - Fills file metadata in buf pointed by path
  - **`int fstat(int fd, struct stat *buf)`**
    - Identical to **stat()**, except that the file is specified by the file descriptor fd
  - **`int lstat(const char *path, struct stat *buf)`**
    - Identical to **stat()**, except that if path is a symbolic link, fill link itself, not the file

Example of Accessing File Metadata

```c
/* 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);

    /* Determine file type */
    if (S_ISREG(st.st_mode)) type = "regular";
    else if (S_ISDIR(st.st_mode)) type = "directory";
    else type = "other";

    /* Check read access */
    if ((st.st_mode & S_IRUSR)) readok = "yes";
    else readok = "no";

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

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

Standard I/O

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)

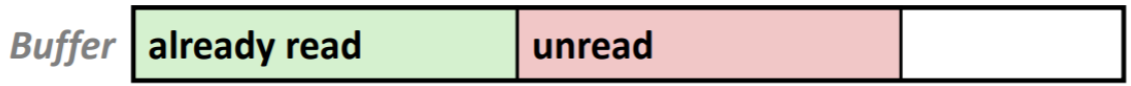
```c
#include <stdio.h>
extern FILE *stdin;  /* standard input  (descriptor 0) */
extern FILE *stdout; /* standard output (descriptor 1) */
extern FILE *stderr; /* standard error  (descriptor 2) */

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

Buffered I/O : Motivation

- Applications often read/write one character or line at a time
  - **`getc(), putc(), ungetc()`**
  - **`gets(), fgets()`**
    - Read line of text one character at a time, stopping at newline

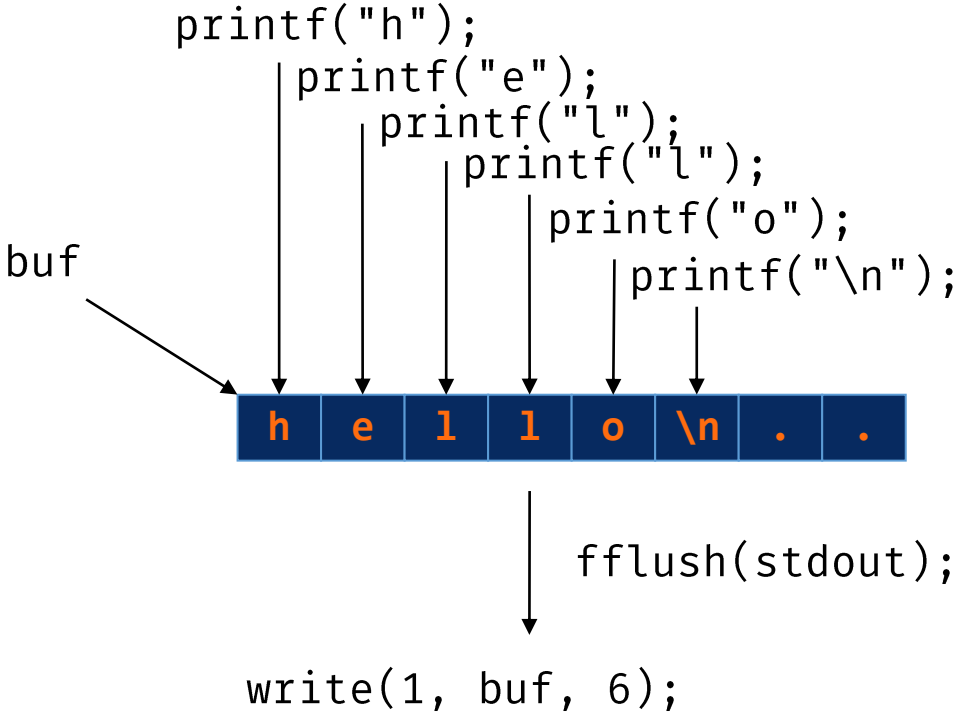
- Implementing as Unix I/O calls expensive
  - Each **`read()`** and **`write()`** require Unix kernel calls
    - > 10,000 clock cycles

- Solution : buffered read
  - Use Unix **`read()`** to grab block of bytes
  - User input functions take one byte at a time from buffer
    - Refill buffer when empty



Buffering in Standard I/O

- Standard I/O functions use buffered I/O



- Buffer flushed to output `fd` on "\n", call to **`fflush()`** or **`exit()`**, or return from main

Standard I/O Buffering in Action

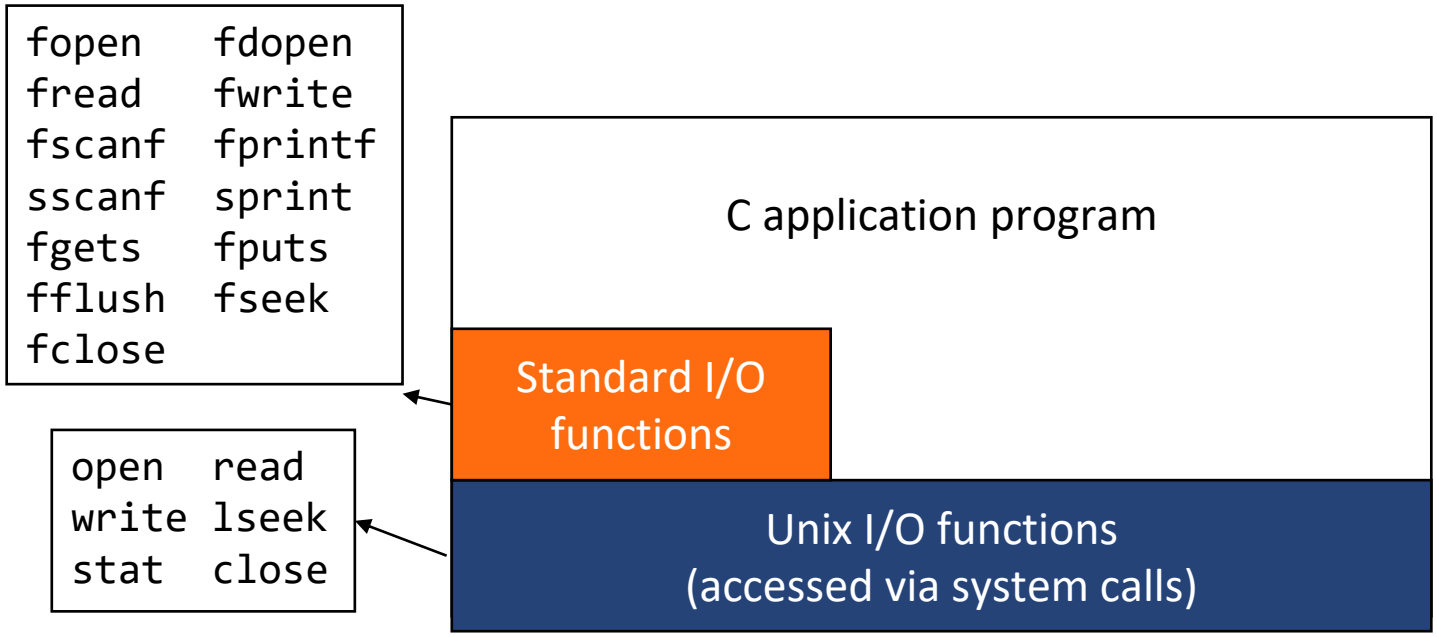
- You can see this buffering in action for yourself, by using Linux program, **`strace`**.

```c
#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 and Cons of Unix I/O

- Pros
  - The most general and lowest overhead form of I/O
    - All other I/O packages are implemented using 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 and 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()`**

Error Handling for File I/O

perror Function / *errno* Variable

- When a system call fails,
  - Returns -1 (or NULL for certain library functions)
  - The latest error information is stored in "*errno*"
- Variable *errno*
  - Stores int value indicating cause of error
  - int type extern global variable
  - Thread-safe!
- Function *perror*
  - Explain error information stored in *errno*
  - Print out the information through stderr stream.

errno.h

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

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

```
1   /* SPDX-License-Identifier: GPL-2.0 WITH Linux-syscall-note */
2   #ifndef _ASM_GENERIC_ERRNO_BASE_H
3   #define _ASM_GENERIC_ERRNO_BASE_H
4
5   #define EPERM        1  /* Operation not permitted */
6   #define ENOENT       2  /* No such file or directory */
7   #define ESRCH        3  /* No such process */
8   #define EINTR        4  /* Interrupted system call */
9   #define EIO      5  /* I/O error */
10  #define ENXIO        6  /* No such device or address */
11  #define E2BIG        7  /* Argument list too long */
12  #define ENOEXEC      8  /* Exec format error */
13  #define EBADF        9  /* Bad file number */
14  #define ECHILD      10  /* No child processes */
15  #define EAGAIN      11  /* Try again */
16  #define ENOMEM      12  /* Out of memory */
17  #define EACCES      13  /* Permission denied */
18  #define EFAULT      14  /* Bad address */
19  #define ENOTBLK     15  /* Block device required */
20  #define EBUSY       16  /* Device or resource busy */
21  #define EEXIST      17  /* File exists */
22  #define EXDEV       18  /* Cross-device link */
23  #define ENODEV      19  /* No such device */
24  #define ENOTDIR     20  /* Not a directory */
25  #define EISDIR      21  /* Is a directory */
26  #define EINVAL      22  /* Invalid argument */
27  #define ENFILE      23  /* File table overflow */
28  #define EMFILE      24  /* Too many open files */
29  #define ENOTTY      25  /* Not a typewriter */
30  #define ETXTBSY     26  /* Text file busy */
31  #define EFBIG       27  /* File too large */
32  #define ENOSPC      28  /* No space left on device */
33  #define ESPIPE      29  /* Illegal seek */
34  #define EROFS       30  /* Read-only file system */
35  #define EMLINK      31  /* Too many links */
36  #define EPIPE       32  /* Broken pipe */
37  #define EDOM        33  /* Math argument out of domain of func */
38  #define ERANGE      34  /* Math result not representable */
39
40  #endif
```

1) Defined as integers
2) Stored in **`errno`** when error occurs
3) Accessible through "**`man errno`**"

Handling Errors Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main() {
   FILE *fp;
   fp = fopen("file.txt", "r");
   if(fp == NULL){
      fprintf(stderr, "errno: %d\n", errno);
      perror("Error");
      exit(1);
   }
   fclose(fp);
   exit(0);
}
```

```
linux> ./test
linux> errno: 2
linux> Error: No such file or directory
```

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

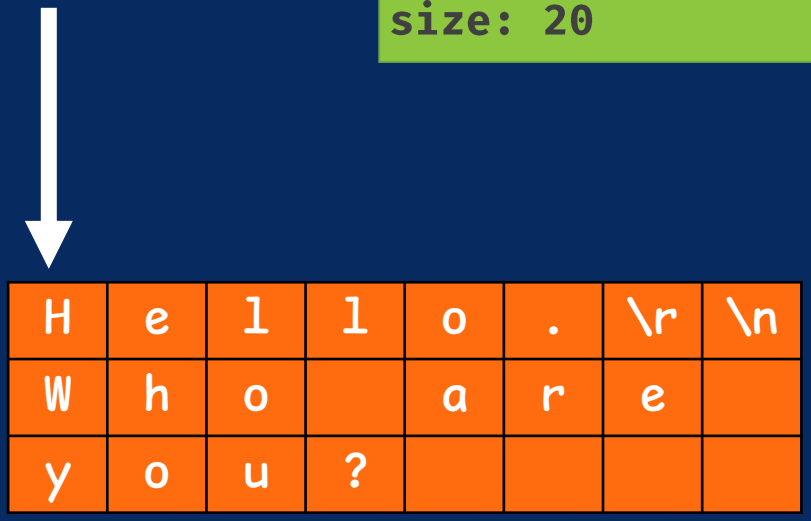
- 6 System calls
  - `open()`
  - `close()`
  - `read()`
  - `write()`
  - `lseek()`
  - `stat() / fstat()`

File I/O Example

```c
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

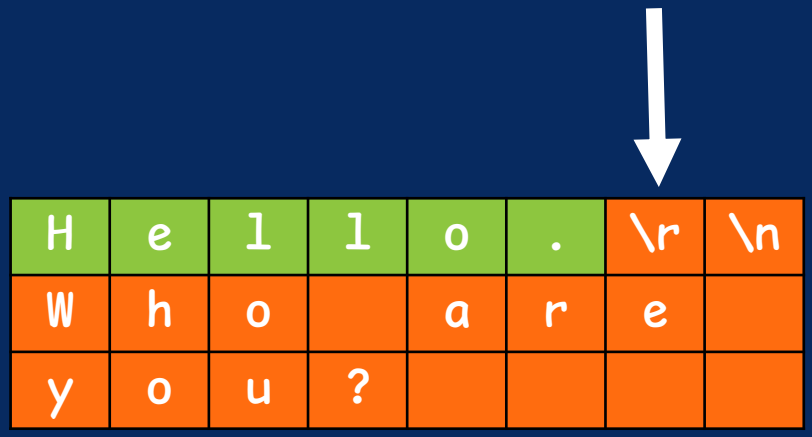


File I/O Example

```c
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

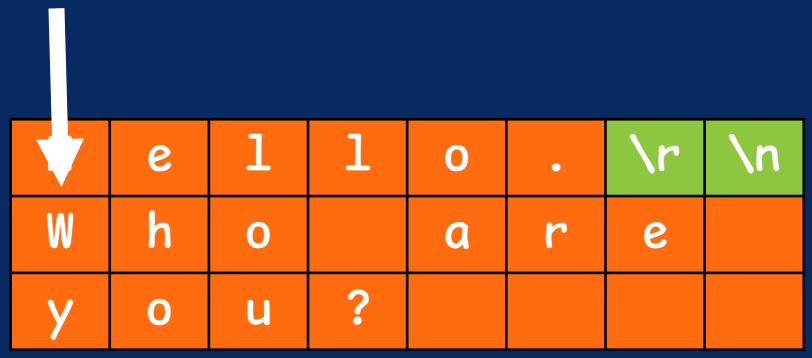


File I/O Example

```c
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

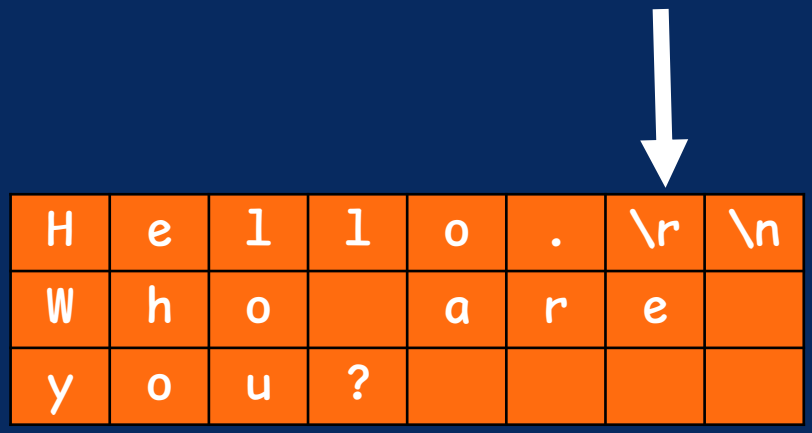


File I/O Example

```c
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

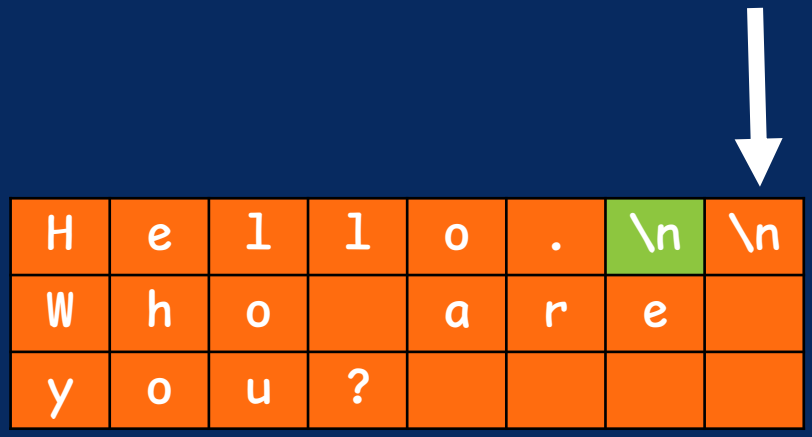


File I/O Example

```c
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)

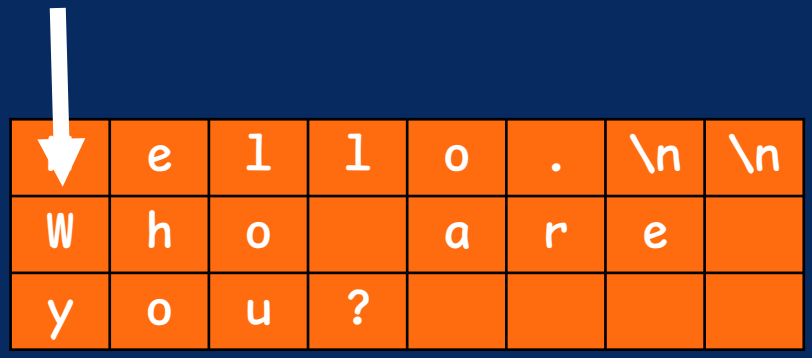
path: "hello-dos.txt"
position: 7
size: 20



File I/O Example

```c
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
```

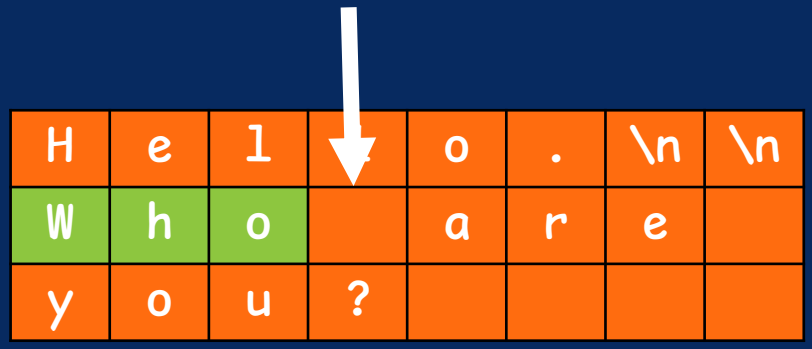


File I/O Example

```c
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



File I/O Example

```c
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

File I/O Example

```c
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);
```

Lab Exercise

- Add line numbers
  - Create “p4” directory in your home directory
  - Go to the directory and download “Aladdin.txt”
    - `wget http://www.fpx.de/fp/Disney/Scripts/Aladdin.txt`
  - Create “Aladdin_num.txt”, which has line numbers in front of each line of text
  - Only use **`open()`, `read()`, `write()`**
  - Don't use other functions such as, **`scanf()`, `printf()`, `sprintf()`, `fscanf()`, `fprintf()`, and `fopen()`**
    - Otherwise, zero

Exercise

```
ALADDIN:  THE COMPLETE SCRIPT
COMPILED BY BEN SCRIPPS <34RQNPQ@CMUVM.CSV.CMICH.EDU>
(Portions Copyright (c) 1992 The Walt Disney Company)

PEDDLER:    Oh I come from a land
    From a faraway place
    Where the caravan camels roam
    Where they cut off your ear /Where it's flat and immense
    If they don't like your face /And the heat is intense
    It's barbaric, but hey--it's home!
    When the wind's at your back
    And the sun's from the west
    And the sand in the glass is right
    Come on down,
    Stop on by
    Hop a carpet and fly
    To another Arabian night!
```

Aladdin.txt

```
1 | ALADDIN:  THE COMPLETE SCRIPT
2 | COMPILED BY BEN SCRIPPS <34RQNPQ@CMUVM.CSV.CMICH.EDU>
3 | (Portions Copyright (c) 1992 The Walt Disney Company)
4 |
5 | PEDDLER:    Oh I come from a land
6 |     From a faraway place
7 |     Where the caravan camels roam
8 |     Where they cut off your ear /Where it's flat and immense
9 |     If they don't like your face /And the heat is intense
10 |     It's barbaric, but hey--it's home!
11 |     When the wind's at your back
12 |     And the sun's from the west
13 |     And the sand in the glass is right
14 |     Come on down,
15 |     Stop on by
16 |     Hop a carpet and fly
17 |     To another Arabian night!
```

Aladdin_num.txt

```
linux> ls
ex4
linux> wget http://www.fpx.de/fp/Disney/Scripts/Aladdin.txt
linux> ls
Aladdin.txt ex4
linux> ./ex4 Aladdin.txt
linux> ls
Aladdin_num.txt Aladdin.txt ex4
```

Exercise

- Submit your exercise source code and Makefile
  - InUiYeJi cluster
  - Submit the folder into p4
  - ~swe2024-41_23s/bin/submit p4 p4
  - We will compile by using command *make*
    - If compilation fails, your points for this exercise will be zero

./p4

- Makefile
- main.c
- Other files you need to build executable
- **Remove “Aladdin.txt” and “Aladdin_num.txt”**

Summary Report

- Summary report about man command result of
  - **`open()`**
  - **`read()`**
  - **`write()`**
  - **`close()`**

- Submission form
  - A4 size PDF format (No page limitation)
  - [SWE2024 Report-3] studentID_name
  - Ex) [SWE2024 Report-3] 2022XXXXXX_홍길동
  - Submit via iCampus
  - Due: until Friday, 24 March 2023, 23:59