# System Programming

## 2. File IO (1): Standard I/O Library

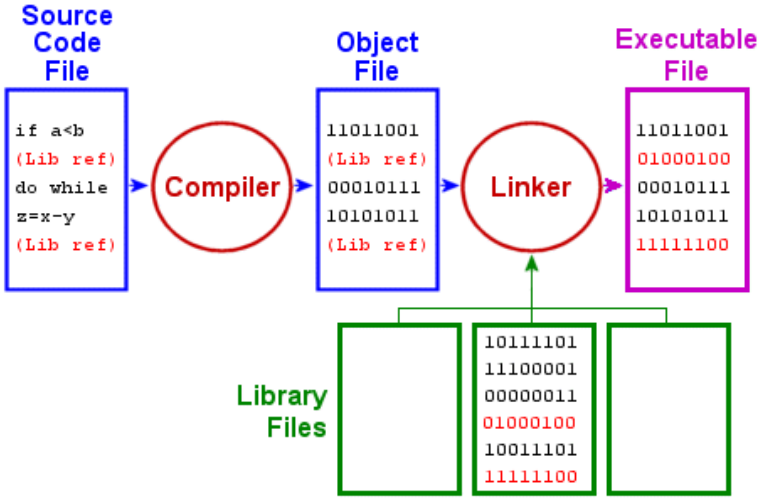
Yunseok Rhee

rheeys@hufs.ac.kr

Division of Computer Engineering

# Library (1)

- A set of compiled object functions for reuse
  - e.g.  Graphic Lib., Mathematical Lib., etc.
  - In Linux, generally located in  “/lib” or in “/usr/lib”.
  - Only necessary functions(objects) will be linked to the user program
- Compile & Linking (review)

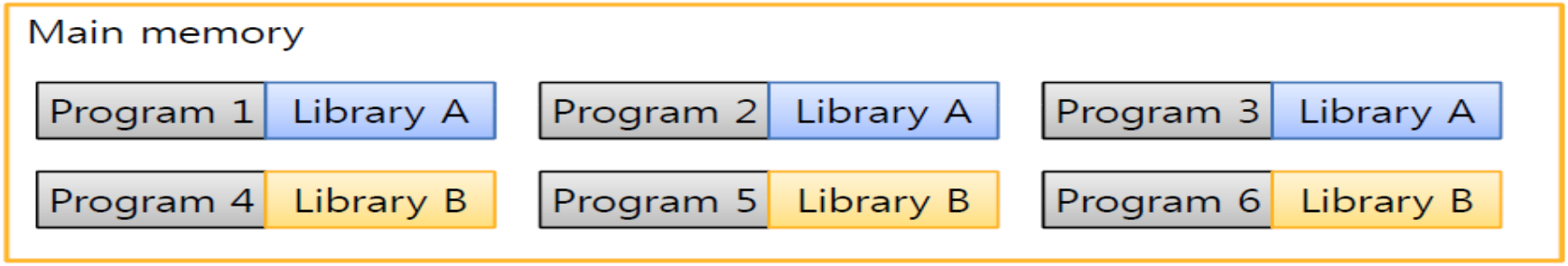


# Library (2)

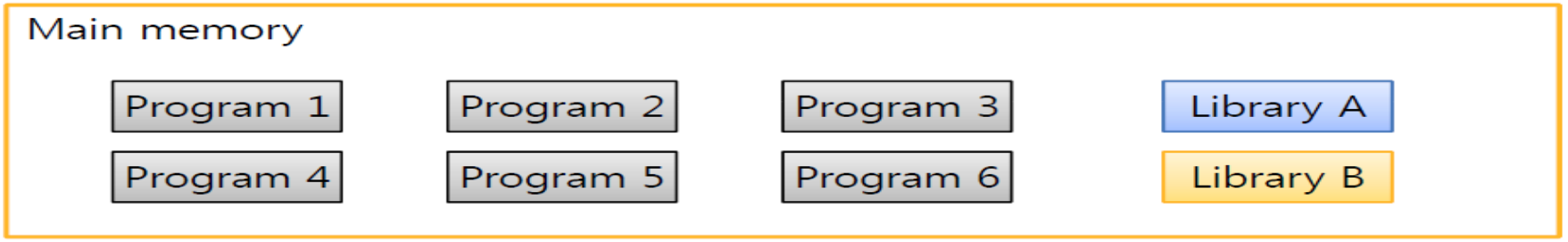
- Types of libraries.
  - Shared library (`*.so, *.dll`)
    - Only one copy of the function resides in the memory. The function will be shared between several processes. (memory saving)
    - The address of the function will be resolved at run-time. (called dynamic linking or binding)
    - A *symbol table* for the dynamic linking exists in memory. (memory overhead).
    - Useful for ***server systems***
  - Static library (`*.a`)
    - Necessary functions are added(linked) to each binary program.
    - So, several same copies of a function reside in memory. (overhead)
    - Useful for ***embedded systems***

# Library (3)

- Executable using **static** library

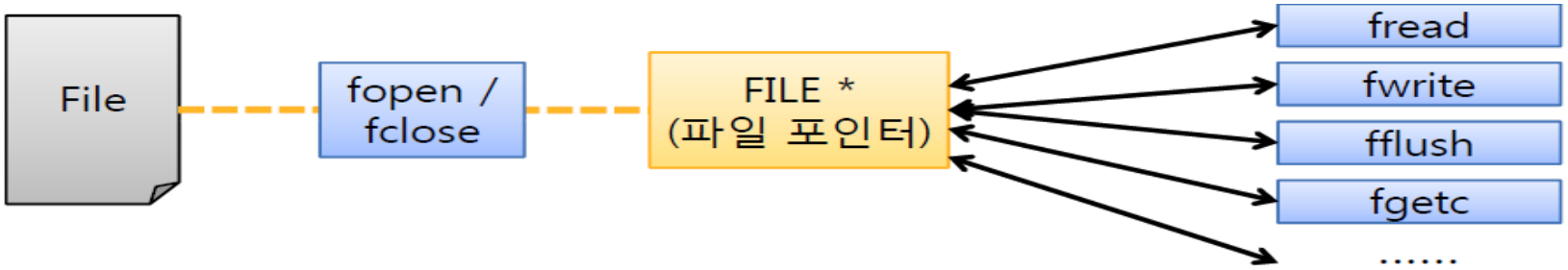



- Executable using using **shared** library



# Standard I/O Library

- <stdio.h>
  - a header file which defines symbols and APIs of the standard I/O library (usually for console and files)
- File I/O with the standard I/O library




  - I/O devices are mapped to special files
    - Console terminal: ***stdin, stdout, stderr***
    - Console files will be automatically open at run-time.

# FILE object in C

- I/O stream object created by standard I/O library
  - accessed by a pointer **FILE***
  - the file stream pointer is used to designate an open file.
  - a file pointer has several system information of an open file.
- stdin, stdout, stderr
  - file stream pointers for the three instances of a console
  - already be opened by the "**shell**" and they are inherited to a user program.

# File descriptor

- OS system calls for I/O
  - use file descriptors (NOT FILE*)
  - a file descriptor for an open file is an integer
  - descriptors `0,1,2` are assigned to stdin, stdout, stderr
  - for user open files, file descriptors are assigned from 3 in ascending order
    - usually, a user can open 1024 files at maximum
- A standard I/O library function will eventually call the appropriate system call.
  - printf, fprintf, puts, ..... → call **`write()`**
- Why use standard I/O library?
  - more convenient than simple system calls
  - formatting, library buffering, ...

# File stream & File Descriptor

- A file stream is 1:1 mapped to a file descriptor

- Thus, we can get each counterpart information by the following functions

```
#include<stdio.h>
int fileno(FILE *stream);
```

  - returns a file descriptor (number) for the open FILE stream

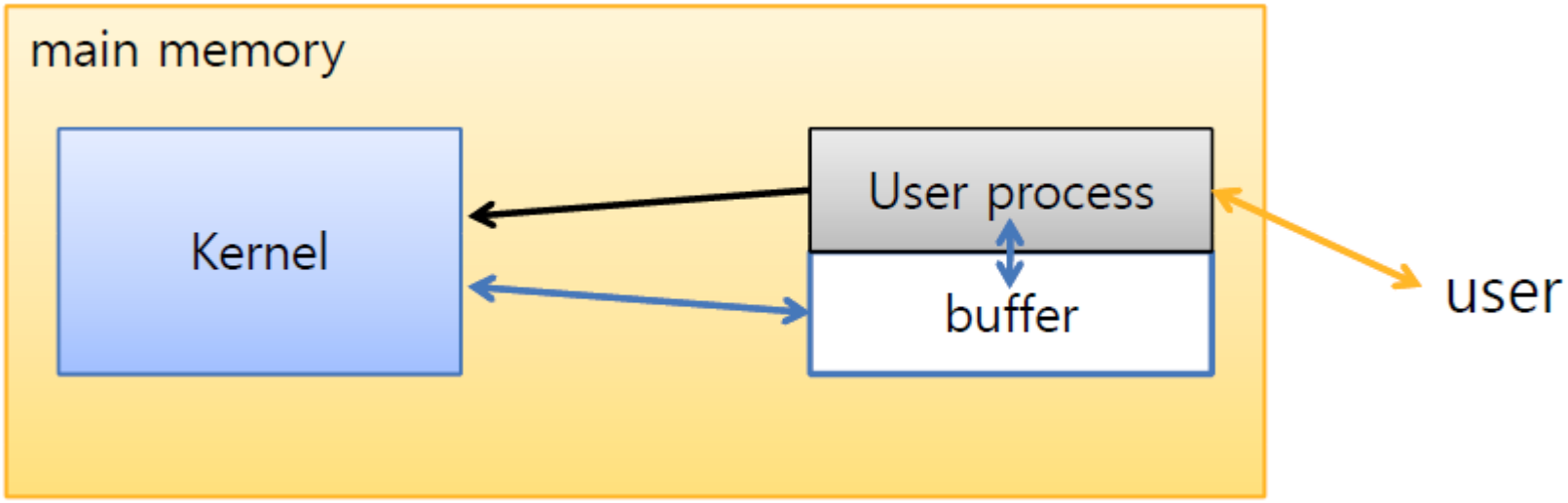
```
#include<stdio.h>
FILE * fdopen(int fildes, const char *mode);
```

  - using the file descriptor of an open file, creates and returns a FILE stream

# Library buffering (1)

- Library buffering
  - user-level buffering by library (i.e. user program)
  - reduce the number of system calls

    e.g. “DEL” key processing in keyboard input



# Library buffering (2)

- Full buffering
  - lib-level buffer for disk blocks (multiple KBs)
  - significantly reduce system calls.
  - For synchronization with the kernel , fflush() can be used.
- Line buffering
  - used for console I/O.
  - actual I/O happens when a “newline” (enter) appears
  - getchar() problem
    - a character is not delivered until entering a “newline”
- Unbuffering
  - no use of library buffer
  - direct delivery to syscalls
  - safe at a power failure.

# Library buffering (3)

- Linux library buffering
  - stderr: always **unbuffering**
  - stddin/stdout: always **line buffering**
  - anything else: always **full buffering** (by default)

# Set Buffering Type

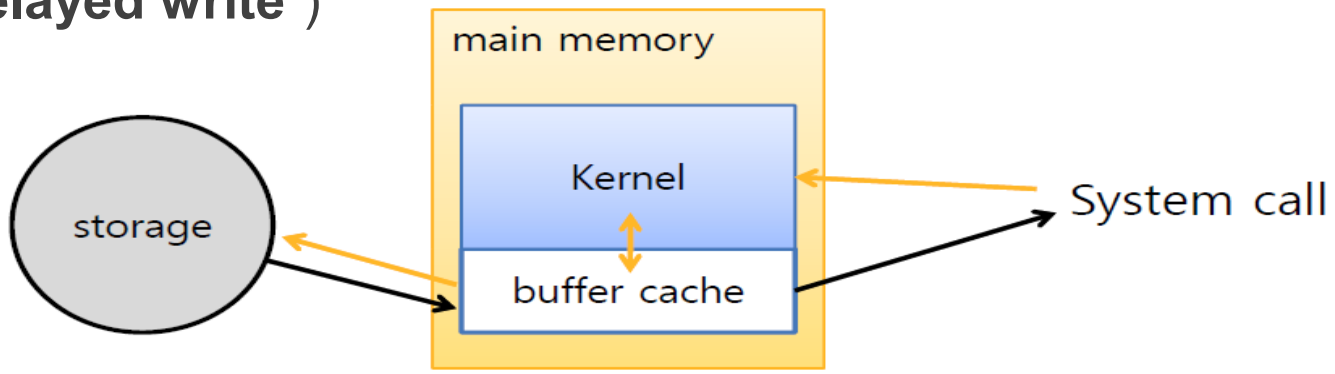
```
#include <stdio.h>
// set a buffer address that user provides
void setbuf (FILE *stream, char *buf);
            buf : non-NULL address for normal buffering
                  NULL if unbuffering
 return : none

// set a buffer address and buffering type
int setvbuf (FILE *stream, char *buf, int type, size_t size);
            buf : same as the above
            type : the type of buffering
            size : buffer size
  return  0 for success, or
            nonzero for an error
```

| *type* | *meaning* |
|---|---|
| _IOFBF | Full buffering |
| _IOLBF | Line buffering |
| _IONBF | Unbuffering |

# Kernel Buffering

- Kernel buffering
  - software caching by the kernel.
  - page cache (buffer cache): to reduce disk I/Os.

    e.g. frequently used disk blocks are kept in the kernel memory (page cache)
  - When reading from a disk
    - try page cache first, if fail do the disk I/O.
  - When writing to a disk
    - write the bytes into the cache, sync to the disk later. (called "**delayed write**")



# fflush

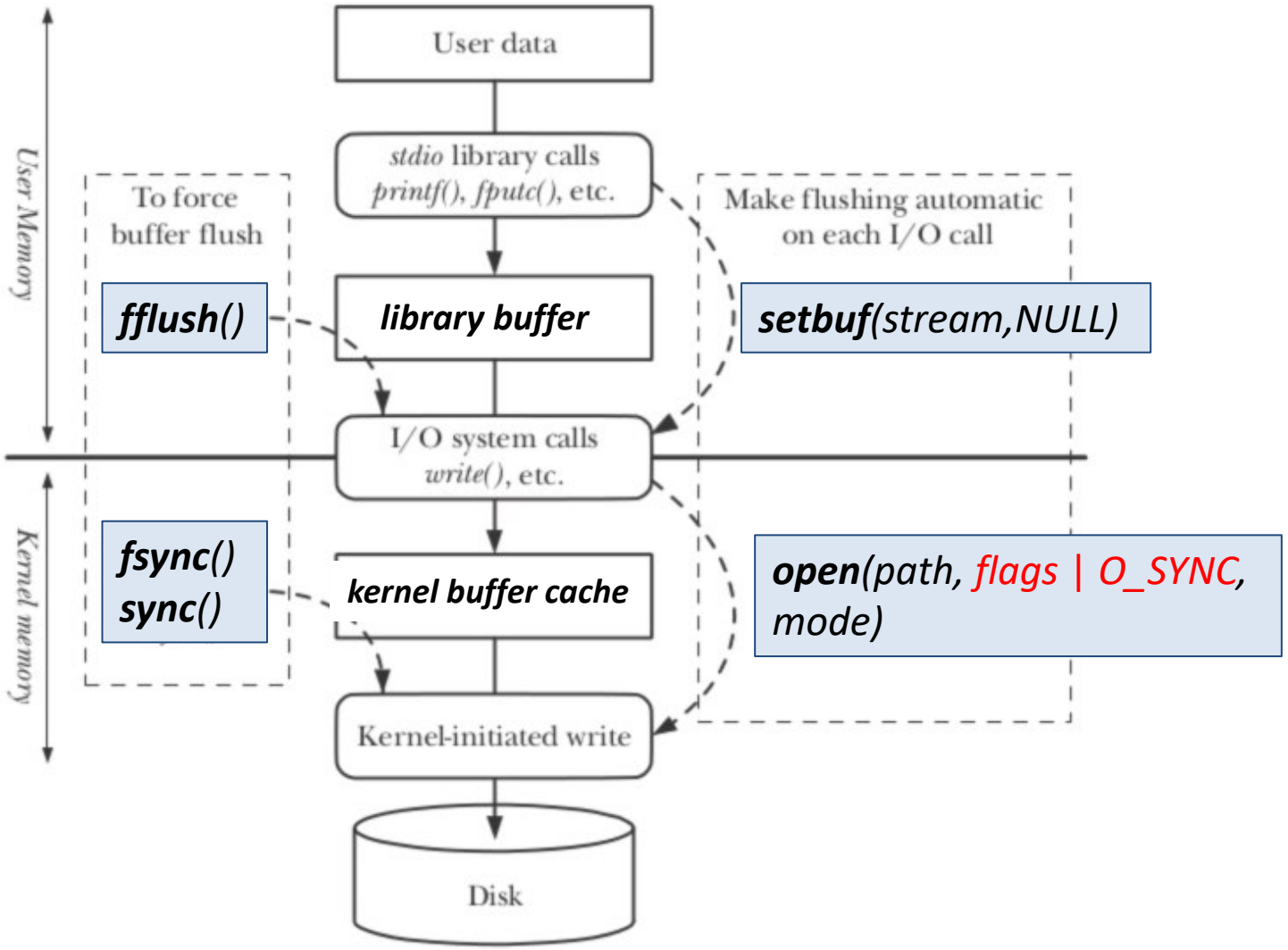
```
#include <stdio.h>
int fflush( FILE *stream);
  return  0 for normal
          EOF for error
```

- flush out the library buffer contents to the kernel. (synchronization)
- due to the buffering, printf (…) does not guarantee the actual output (why?)
- thus, for debugging, write a code as follows

```
printf("something");
fflush(stdout);
```

- for block device I/O (e.g. disk)
  - in block device, a transfer unit b/w disk and kernel is in KBs
  - fflush() moves the contents "lib. buffer" to "page cache"
  - thus, if we want a disk synchronization, use **sync()**
- When a file is closed, fflush() will be done automatically.

# I/O buffering & Sync



# File Open

```
#include <stdio.h>
FILE * fopen ( const char *filename, const char *type);
            type : access mode
  return  a file stream pointer to the open file if succeed, or
            NULL (error: failed to open)
```

- File access modes

| *Access modes* | *Description* |
|---|---|
| ***r*** | Read only |
| ***w*** | Truncate file to zero length or create a file for writing. |
| ***a*** | Append mode(EOF), write only. The file is created if it does not exist. |
| ***r+*** | Read/write |
| ***w+*** | Truncate file to zero length or create a file for reading & writing. |
| ***a+*** | Read/append mode(EOF). The file is created if it does not exist. |

- The number of open files has a limitation (by system configuration)

# File Reopen

```
#include <stdio.h>

FILE * freopen ( const char *filename, const char *type, FILE *stream);
            type : access mode
            stream : file pointer
 return  a file stream pointer to the open file
            NULL (error: failed to open)
```

- first, close a file linked to the input stream (3rd arg)
- and open a file with a given filename by reusing the old stream
- NOTE: the original file descriptor is also reused!

## Example

- guess what will happen in this code!

```
freopen("myfile.txt", "w", stdout);
printf("This sentence is redirected to a file.");
fclose(stdout);
```

# File Close

```
#include <stdio.h>
int fclose( FILE *stream);
  return 0 for normal, EOF for error
```

- When a process exits normally, all files are *automatically* closed.
- If a process is terminated without closing a file,
  - cannot check some errors that are reported by the fclose().
  - file data in the library buffer might be lost.

# File I/O functions

| *Function Prototypes* | *Input Arg.* | *Return* | |
|---|---|---|---|
| | | *normal* | *error* |
| *size_t* ***fread*** *(void *ptr, size_t size, size_t nitems, FILE *stream)* | - *ptr*: destination buffer address<br>- *size*: # of bytes of the object unit<br>- *nitems*: # of objects<br>- *stream*: file pointer | # of objects read | 0 |
| *size_t* ***fwrite*** *(void *ptr, size_t size, size_t nitems, FILE *stream)* | - *ptr*: source buffer address<br>- *size*: # of bytes of the object unit<br>- *nitems*: # of objects<br>- *stream*: file pointer | # of objects written | 0 |

# Character Input

| *Function Prototypes* | *Description* | *Return* | |
|---|---|---|---|
| | | *normal* | *error* |
| *int **getc** (FILE *stream), int **fgetc** (FILE *stream)* | Get a character from file. | char in integer type | EOF |
| *int **getchar**(void)* | Get a character from *stdin.* | char in integer type | EOF |
| *char ***fgets** (char *s, int size, FILE *stream)* | Get a NULL (“\0”) terminated string. Read until a newline or EOF. Max string size = size -1. | char string address | NULL |
| *char ***gets** (char *s)* | Get a NULL (“\0”) terminated string from *stdin*. Read until a newline or EOF. | char string address | NULL |
| *int **ungetc** (int c, FILE *stream)* | Put the character *c* into the file to enable rereading. | c | EOF |

# Character Output

| *Function Prototypes* | *Description* | *Return* | |
|---|---|---|---|
| | | *normal* | *error* |
| *int **putc** (int c, FILE *stream),*<br>*int **fputc** (int c, FILE *stream)* | Write a character to file. | char in integer type | EOF |
| *int **putchar** (int c)* | Write a character to *stdout.* | char in integer type | EOF |
| *int ***fputs** (const char *s, FILE *stream)* | Write a string without its trailing "\0". | # of chars | EOF |
| *char ***puts** (const char *s)* | Write a string and a trailing newline to *stdout.* | # of chars | EOF |

# File I/O example (1)

*fileio-ex.c*

```
$ ./fileio-ex firstFile secondFile
```

```
argc=3
argv[0]="./fileio-ex"
argv[1]="firstFile"
argv[2]="secondFile"
```

```
#include <stdio.h>

int main( int argc, char *argv[])
{
        int c;
        FILE *fpin, *fpout;

        if( argc != 3) {
                perror( argv[0]);
                exit(1);
        }
        if(( fpin = fopen( argv[1], "r")) == NULL) {
                perror( argv[1]);
                exit(2);
        }
```

# File I/O example (2)

```
        if(( fpout = fopen( argv[2], "a")) == NULL) {
                perror( argv[2];
                exit(3);
        }
        setbuf( fpin, NULL);   // unbuffered I/O
        setbuf( fpout, NULL); // unbuffered I/O

        while(( c = getc( fpin)) != EOF)
        putc( c, fpout);

        fclose( fpin);
        fclose( fpout);
        exit(0);
}
```

# File I/O example (3)

- Execution

```
$ cat test1.txt
Hello, world (1)

$ cat test2.txt
Hello, world (2)

$ ./a.out  test1.txt   test2.txt

$ cat test1.txt
Hello, world (1)

$ cat test2.txt
Hello, world (2)
Hello, world (1)

$
```

# Line I/O example (1)

*lineio-ex.c*

```
#include <stdio.h>

#define BUFFER_SIZE 100

int main(int argc, char *argv[])
{
        char ubuf[BUFFER_SIZE], line[BUFFER_SIZE];
        FILE *fpin, *fpout;

        if(argc != 3) {
                perror(argv[0]);
                return 1;
        }
        if ((fpin = fopen(argv[1], "r")) == NULL) {
                perror(argv[1]);
                return 2;
        }
```

# Line I/O example (2)

```
        if ((fpout = fopen(argv[2], "a")) == NULL) {
                perror(argv[2]);
                return 3;
        }
        if (setvbuf (fpin, ubuf, _IOLBF, BUFFER_SIZE) != 0) { // line buffering
                perror("setvbuf(fpin)");
                return 4;
        }
        if (setvbuf (fpout, ubuf, _IOLBF, BUFFER_SIZE) != 0) {
                perror("setvbuf(fpout) ");
                return 5;
        }

        while ( fgets (line, BUFFER_SIZE, fpin) != NULL)
                fputs (line, fpout);

        fclose(fpin);    fclose(fpout);
        return 0;
}
```

# Array I/O example

```
#define ARRAY_SIZE 10

        ......
        int i;
        int sample_arry[ARRAY_SIZE];
        FLLE *stream;

        if ((stream = fopen(argv[1], "w")) == NULL) {
                perror(argv[1]);
                return 1;
        }

        if (fwrite (sample_array, sizeof(int), ARRAY_SIZE, stream) != ARRAY_SIZE) {
                perror("fwrite error");
                return 2;
        }
        ......
```

# Struct I/O example

```
struct {
        short count;
        char sample;
        long total;
        float numeric[LENGTH];
} object;

FILE *stream;
…

if (fwrite (&object, sizeof(object), 1, stream) !=1)
        perror("fwrite error");
```

# File copy with Full buffering (1)

*filecopy.c*

```
# include <stdio.h>

#define BUFFER_SIZE 1024

int main(int argc, char *argv[])
{
        char ubuf[BUFFER_SIZE], fbuf[BUFFER_SIZE];
        int n;
        FILE *fpin, *fpout;

        if(argc != 3) {
                perror(argv[0]);
                return 1;
        }
        if((fpin = fopen(argv[1], "r")) == NULL) {
                perror(argv[1]);
                return 2;
        }
```

# File copy with Full buffering (2)

```
        if((fpout = fopen(argv[2], "w")) == NULL) {
                perror(argv[2]);
                return 3;
        }
        if (setvbuf(fpin, ubuf, _IOFBF, BUFFER_SIZE) != 0) { // full buffering
                perror("setvbuf(fpin)");
                return 4;
        }
        if (setvbuf (fout, ubuf, _IOFBF, BUFFER_SIZE) != 0 {
                perror("setvbuf(fpout)");
                return 5;
        }

        while ( n= fread(fbuf, sizeof(char), BUFFER_SIZE, fpin ) > 0)
                fwrite (fbuf, sizeof(char), n, fpout);

        fclose(fpin);
        fclose(fpout);
        return 0;
}
```

# File Offset

- Every open file has a (r/w) offset which indicates the next access position in the file
  - when a file is opened for reading/writing, the offset is set to the beginning of the file
  - when a file is opened for appending, the offset is set to the end of the file
  - While reading/writing, the offset automatically advances

# File Access Methods

- Sequential access
  - sequential access by following the r/w offset
- Random access
  - moves the r/w offset to a wanted access position by calling **fseek()** library function
    - or by **lseek()** system call,
  - mainly used for record processing.

cf. Keyed access

- Access a record of a DB by a key,
- A internal index tree of a DB is necessary.

# R/W offset related functions

| *Function Prototypes* | *Input Arg.* | *Return* | |
|---|---|---|---|
| | | *normal* | *error* |
| *int **fseek** (FILE *stream, long offset, int sopt)* | - *stream*: file pointer<br>- ***offset***: distance relative to SEEK option position<br>- *sopt*: SEEK option | 0 | -1 |
| *void **rewind** (FILE *stream)* | stream: file pointer | none | none |
| *long **ftell** (FILE *stream)* | stream: file pointer | current offset | -1 |

- SEEK options
  - SEEK_SET: new r/w offset = offset
  - SEEK_CUR: new r/w offset = current_offset + offset
  - SEEK_END: new r/w offset = EOF + offset

# Random access example (1)

*frandom-ex.c*

```
#include <stdio.h>
int main(int argc, char *argv[])
{
        FILE *fp;
        char buf[256];
        int rspn;
        long pos;

        if((fp = fopen(argv[1], "r")) == NULL) {
                perror(argv[1]);
                return 1;
        }
        rspn = fseek(fp, 8L, SEEK_SET);
        pos = ftell(fp);
```

# Random access example (2)

```
        fgets(buf, 256, fp);
        printf("Position : %ld\n", pos);
        printf("%s\n", buf);

        rewind(fp);
        pos = ftell(fp);

        fgets(buf, 256, fp);
        fclose(fp);
        return 0;
}
```

# Random access example (3)

- Execution

```
$ cat test.dat
This is a test data.

$ ./a.out test.dat
Position : 8
a test data.
Postion : 0
This is a test data.
$
```

# I/O Types

- Unformatted I/O (Binary I/O)
  - I/O in binary format (memory representation).
    integer : 4 byte, signed two's complement.
    float: 4 bytes, "sign + exp(8-bit) + mantissa(23-bit)".
    double: 8 bytes, "sign + exp(11-bit) + mantissa(52-bit)".
  - a user's viewer program must be supported.
- Formatted I/O
  - output: integer, float, double → output in an ASCII string
  - input: ASCII string input → integer, float, double (scan conversion)
  - e.g.
    %5d: integer to decimal ASCII string (5 digits)
    %f: 12.43
  - file contents can be seen by "cat file".

# Formatted Output

| *Function Prototypes* | *Description* | *Return* | |
|---|---|---|---|
| | | *normal* | *error* |
| *int **printf** (const char *format, /* args */ ... )* | to the console | output length | negative integer |
| *int **fprintf** (FILE *stream, const char *format, /* args */ ... )* | to a file | | |
| *int **sprintf** (char *s, const char *format, /* args */ ... )* | to a string | | |

# Formatted Input

| *Function Prototypes* | *Description* | *Return* | |
|---|---|---|---|
| | | *normal* | *error* |
| *int **scanf** (const char *format, ... )* | from the console | input length | EOF |
| *int **fscanf** (FILE *stream, const char *format, ... )* | from a file | | |
| *int **sscanf** (char *s, const char *format, ... )* | from a string | | |

# Formatted I/O example (1)

*stdio-ex.c*

```
#include <stdio.h>

int main(int argc, char argv[])
{
        FILE *fp;
        char buf[256];
        int num, Nnum;
        char str[30], Nstr[30];

        scanf("%d %s", &num, str);
        if((fp = fopen("test.dat", "w")) == NULL) {
                perror(test.dat);
                return 1;
        }
```

# Formatted I/O example (2)

```
        fprintf(fp, "%d %s\n", num, str);

        if((fp = freopen("test.dat", "r", fp)) == NULL) {
                perror("test.dat");
                return 1;
        }
        fscanf(fp, "%d %s\n", &Nnum, Nstr);
        printf("%d %s\n", Nnum, Nstr);

        fclose(fp);
        return 0;
}
```

# File error check

| *Function Prototypes* | *Return* | |
|---|---|---|
| | *error value* | *when no error* |
| *int **ferror** (FILE *stream)* | nonzero value | 0 |
| *int **feof** (FILE *stream)* | nonzero value | 0 |
| *void **clearerr** (FILE *stream)* | none | none |

# File error check example (1)

*ferror-ex.c*

```
#include <stdio.h>

int main(void)
{       int ret;
        FILE *fp;

        fp = fopen("test.dat", "r");
        putc('?', fp);
        if(ret = ferror(fp))
                printf("ferror() return %d\n", ret);
        clearerr(fp);
        printf("ferror() return %d\n", ferror(fp));
        fclose(fp);
        return 0;
}
```

# File error check example (2)

- Execution

```
$ cat test.dat
1234 abcd
$ ./a.out
ferror() returned 1
ferror() returned 0
$
```

# EOF check example (1)

*feof-ex.c*

```
#include <stdio.h>
int main()
{
        int stat = 0;
        FILE *fp;
        char buf[256];

        fp = fopen("test.dat", "r");
        while(!stat)
                if(fgets(buf, 256, fp) printf("%s\n", buf)
                else stat = feof(fp);
        printf("feof returned %d\n", stat);
        fclose(fp);
        return 0;
}
```

# EOF check example (2)

- Execution

```
$ ./a.out
1234 abcd
feof returned 1
$
```

# Error handling

- Important ANSI C Features:
  - function prototypes
  - generic pointers (`void *`)
  - abstract data types (e.g. `pid_t, size_t`)
- Error Handling:
  - meaningful return values
  - *errno* variable
    - must include <errno.h>
  - look up constant error values via two functions:

```
#include <string.h>
char *strerror(int errnum) // returns pointer to message string

 #include <stdio.h>
void perror(const char *msg) // print the last error with the msg
```

# Homework

- Write a short text file using *vim* editor
- Write and run a file copy program (in the lecture note)
- At the next Quiz, you will be tested if you really did the work for yourself or not.