System Programming

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

Seung-Ho Lim

Dept. of Computer & Electronic Systems Eng.

Linux System Calls

- File descriptor I/O
  - ***open(); close(); creat(), read(); write();***
  - ***seek();*** *// random access*
  - ***fcntl();*** *// for file/record locking*
- Process control
- Thread programming
- IPC
- Signal handling
- Memory management
- *Synchronization*
- *Time management*
- *Network socket API (TCP, UDP)*

System Calls & Library Calls for File I/O

- System Calls for File descriptor I/O
  - ***open(); close(); creat(), read(); write();***
  - ***seek();*** *// random access*
  - ***fcntl();*** *// for file/record locking*

- Library Calls for File I/O
  - ***fopen(); freopen(); fclose(); fread(); fwrite();***
  - ***fgetc(), fgetchar(); fputc, putchar(); …***
  - ***fseek(), fprintf(); fscanf();..***

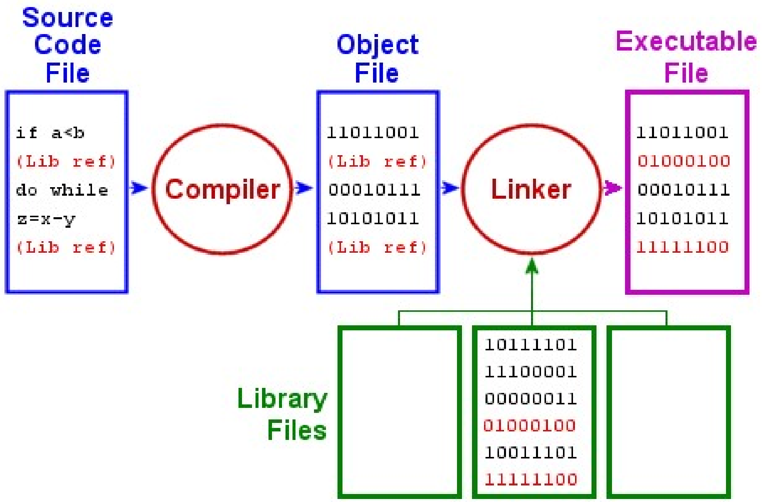
System Calls vs. Library Calls

- System Calls
  - they are entry points into kernel code where their functions are implemented.
  - documented in section 2 of the linux manual (e.g. `write`(2) or `man` 2 `write`)

- Library Calls
  - they are transfers to user code which performs the desired functions.
  - documented in section 3 of the linux manual (e.g. `printf`(3)).
  - also called *API*(application programming interfece)

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)

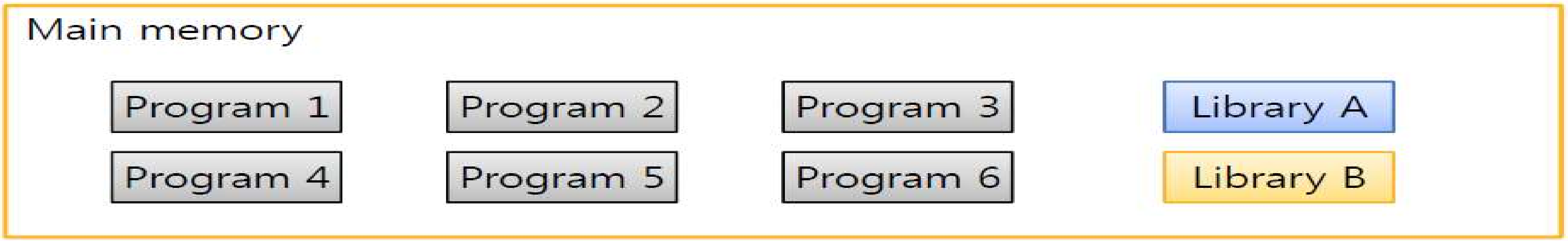


Library (2)

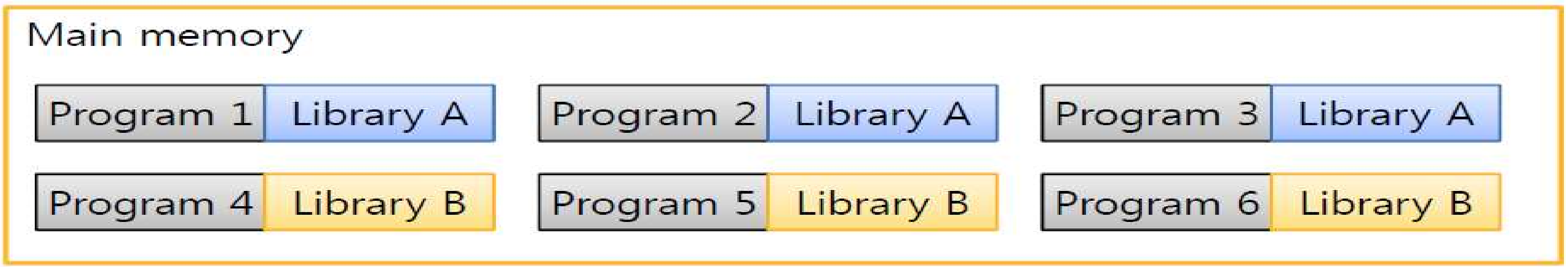
- Types of libraries.
  - Shared library (`*.so, *.dll`)
    - Only one copy of the function resides in the memory. The function will be shared between several processes. (memory saving)
    - The address of the function will be resolved at run-time. (called dynamic linking or binding)
    - A *symbol 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 **shared** library



- Executable using **static** 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

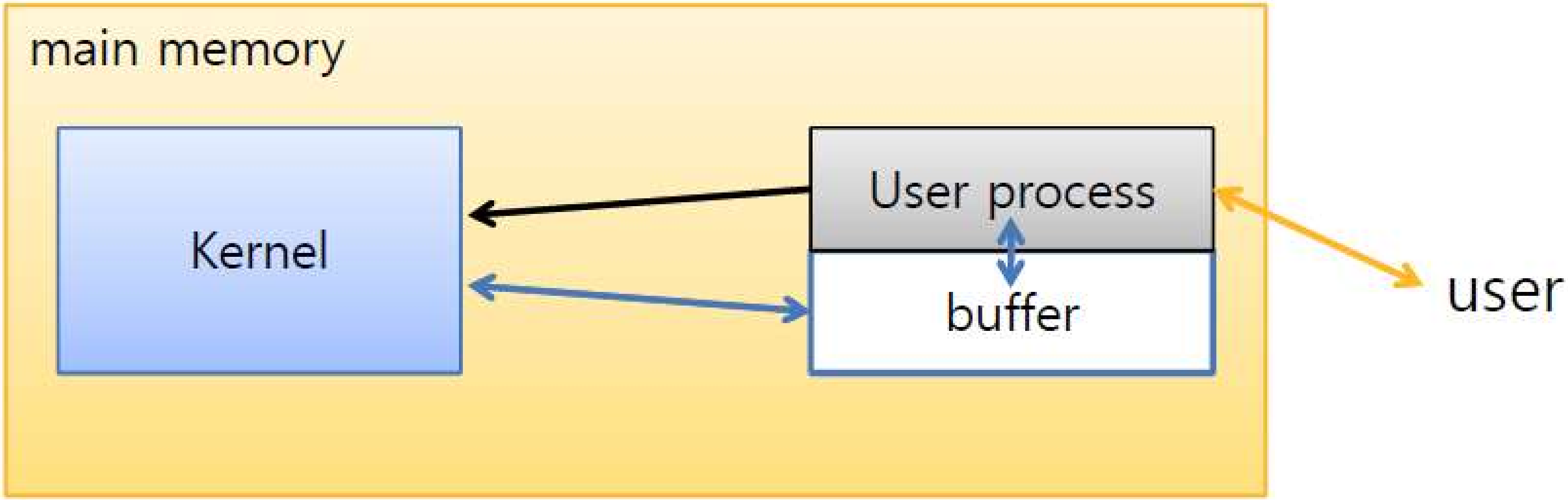
```
#include<stdio.h>
FILE * fdopen(int fildes, const char *mode);
```

  - 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

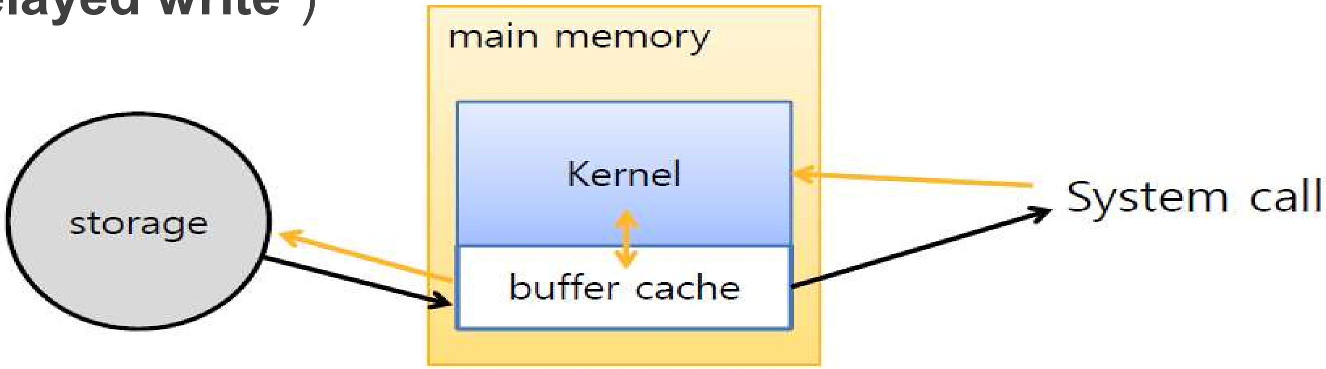
```
#include <stdio.h>
// set a buffer address that user provides
void setbuf (FILE *stream, char *buf);
              buf : non-NULL address for normal buffering
                      NULL if unbuffering
 return : none

// 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