Linux Programming
4. The File

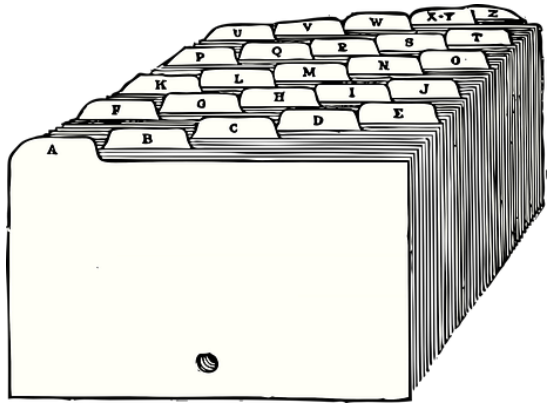
이선우

sunwool@inha.ac.kr

INHA UNIVERSITY

- ***Files and basic system calls***
- File sharing
- Standard I/O
- Standard I/O Libraries

File

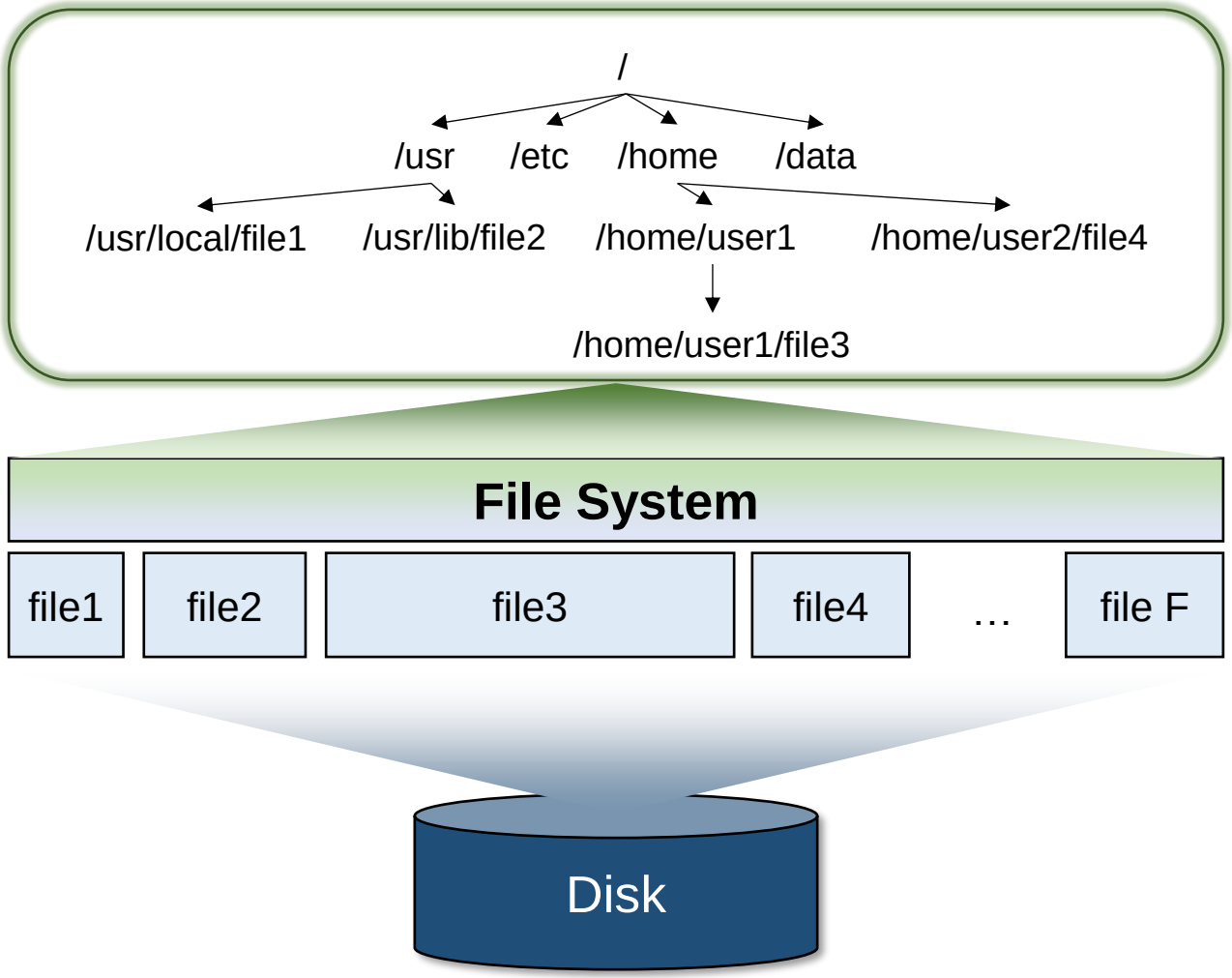


- Logically, a container of data
- Physically, a contiguous sequence of bytes
- There is no format imposed by OS.
- **Each byte is individually addressable in a disk file**.
- File is also a uniform interface to external devices.

File System (1/2)

- A software that organizes computer files and the data.
  - It provides users with useful interfaces to access files.
- A variety of file systems are available!
  - FAT
  - EXT series (2,3,4)
  - NTFS
  - XFS
  - HDFS
  - F2FS
  - And a lot more!

File System (2/2)



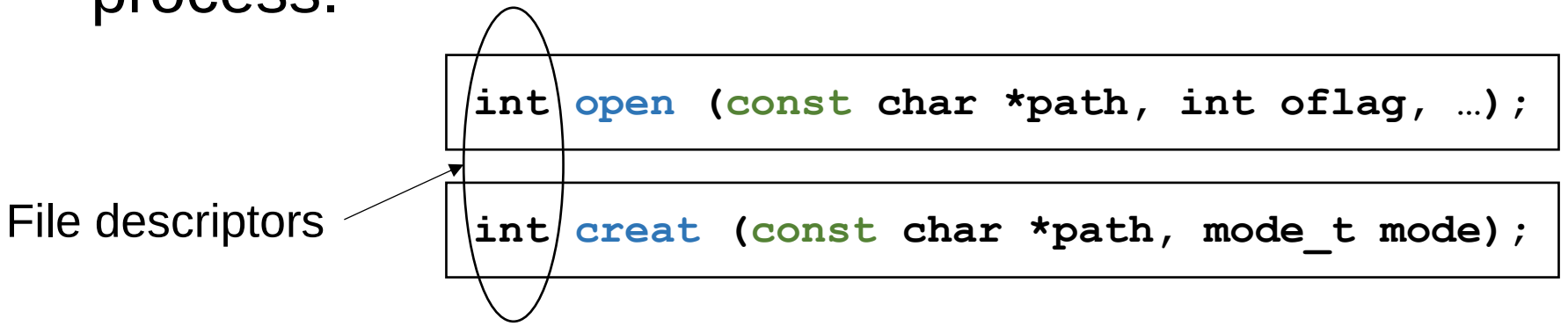
Unix File Access Primitives

- System Calls for file handling

Name	Description
`open`	Opens a file for reading or writing, or creates
`creat`	Creates an empty file
`close`	Closes a previously opened file
`read`	Extracts information from a file
`write`	Places information into a file
`lseek`	Moves to a specified byte in a file
`unlink`	Removes a file
`remove`	Alternative method to remove a file
`fcntl`	Controls attributes associated with a file

File Descriptor (1/2)

- All open files are referred to by file descriptors.
- A non-negative integer.
  - The new file descriptor will be the smallest unused integer.
- When opening an existing file (`open()`) or creating a new file (`creat()`), the kernel returns a file descriptor to the user process.



File Descriptor (2/2)

- When reading (`read()`) or writing (`write()`) a file, the kernel finds the target file using the input file descriptor.

```
ssize_t read (int fd, void *buf, size_t count);
```

```
ssize_t write (int fd, void *buf, size_t nbytes);
```

- Each process created by a shell begins life with three open files associated with a terminal.

File descriptor	Symbolic constant	Description
0	STDIN_FILENO	Standard input
1	STDOUT_FILENO	Standard output
2	STDERR_FILENO	Standard error

Example

```
/* a rudimentary example program */

/* these header files are discussed below */
#include <fcntl.h>
#include <unistd.h>

main()
{
    int fd;
    ssize_t nread;
    char buf[1024];

    /* open file “data” for reading */
    fd = open(“data”, O_RDONLY);

    /* read in the data */
    nread = read(fd, buf, 1024);

    /* close the file */
    close(fd);
}
```

- Primitive System Data Types
  - The data types ending with ‘_t’ are called the ***primitive system data type***.
  - They are defined in <sys/types.h>.
  - The purpose of having these additional definitions is to enable user programs to support different architectures easily.

System Call: open (1/2)

```
#include <fcntl.h>
int open (const char *path, int oflag, mode_t mode);
```

- A system call that opens a file based in the given ‘mode’.
  - `open()` returns the file descriptor on success and -1 on error.
- Arguments
  - path: file path
  - oflag: only one of the following settings is allowed at once.
    - O_RDONLY #0 Read-only
    - O_WRONLY #1 Write-only
    - O_RDWR #2 Read & Write

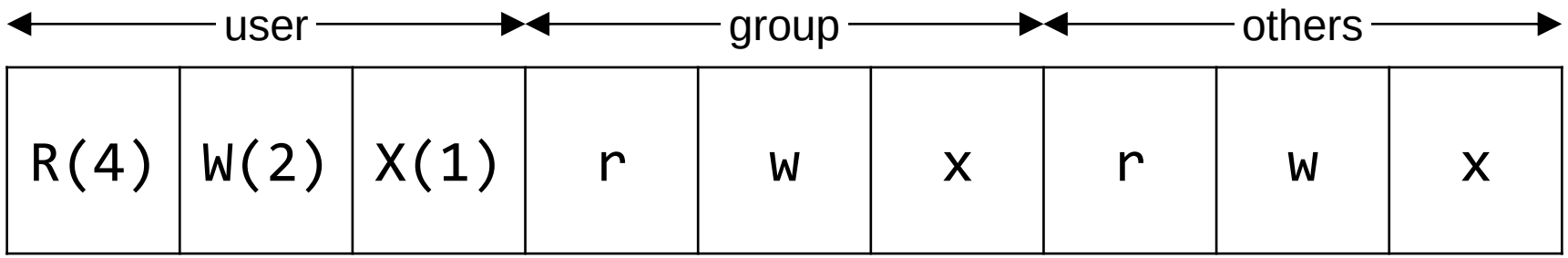
System Call: open (2/2)

- Optional flags
  - O_APPEND
  - O_CREAT
  - O_EXCL
  - O_TRUNC
  - O_NONBLOCK
- Mode
  - Only used when oflag is O_CREAT
  - Set the file permission

```
fd = open(“/tmp/newfile”, O_WRONLY|O_CREAT, 0644);
/* if file exists “file open” else “file create & open” */

fd = open(“/tmp/newfile”, O_WRONLY|O_CREAT|O_EXCL, 0644);
/* if file exists “open error” else “file create & open” */

fd = open(“/tmp/newfile”, O_WRONLY|O_CREAT|O_TRUNC, 0644);
/* if file exists “file truncate & open ” else “file create & open” */
```

File Permission

- In Unix, file permission is described as three-digit octal value.



E.g., 644

1	1	0	1	0	0	1	0	0

Example: open()

```
#include <stdlib.h>          /* for exit() */
#include <fcntl.h>           /* for open() */
#include <stdio.h>           /* for printf() */

char *workfile = “junk”;

main() {
    int fd;
    if ((fd = open(workfile, O_RDWR)) == -1) {
        printf(“Couldn’t open %s\n”, workfile);
        exit(1);
    }

    exit(0);
}
```

- There is the maximum number of open files per process.
  - It was 20 in the past, but most Unix and LINUX systems today allow 1,024.
- The kernel itself also has the maximum number of open files: it varies depending on the OS (e.g., Ubuntu 18.04 has a limit of 52,751,403).

System Call: Create()

```
#include <fcntl.h>
int creat (const char *pathname, mode_t mode);
```

- An alternative way to create a new file.
  - `creat()` returns the file descriptor on success and -1 on error.
  - If the file already exists, the `mode` argument is ignored, and the file is opened with `O_WRONLY | O_TRUNC`.
- The following two system calls yield identical results:

```
fd = creat("/tmp/newfile", 0644);
fd = open("/tmp/newfile", O_WRONLY | O_CREAT | O_TRUNC, 0644);
```

Owner of a New File

- When creating a new file (regardless of which function is used):
  - You need write permission in the parent directory where the file belongs to.

- Who owns the file?
  - The owner and group are set to the effective user and group IDs of the current process.

System Call: Close()

```
#include <unistd.h>
int close (int filedes);
```

- An open file is closed by `close()`.
  - `close()` return 0 on success and -1 on error.
- To prevent total chaos, all open files are automatically closed when the program completes execution.

```
fd = open("/tmp/newfile", O_WRONLY | O_CREAT | O_TRUNC, 0644);
…
ret = close(fd);
```

System Call: Read() (1/2)

```
#include <unistd.h>
ssize_t read (int filedes, void *buffer, size_t n);
```

- `read()` system call reads `n` bytes from a file associated with `filedes` and copies the data into `buffer`.
- It returns the number of bytes successfully read, 0 at the end of the file (EOF), or -1 on error.

System Call: Read() (2/2)

- File position
  - Each file descriptor has its own file position.
  - `read()` updates the current file position after every call.

An example of reading a file of size 700 bytes.

```
int fd;
ssize_t n1, n2, n3;
char buf1[512], buf2[512], buf3[512];

if( (fd = open("foo", O_RDONLY)) == -1)
    return -1;

                                    /* f_position : 0 */
n1 = read(fd, buf1, 512); /* n1 : 512, f_position : 512 */
n2 = read(fd, buf2, 512); /* n2 : 188, f_position : 700 */
n3 = read(fd, buf3, 512); /* n3 : 0, when read EOF */
```

System Call: Write()

```
#include <unistd.h>
ssize_t write (int filedes, void *buffer, size_t n);
```

- `write()` system call writes `n` bytes from `buffer` to a file associated with `filedes`.
- It returns the number of bytes successfully written or -1 on error.
- If a program opens an existing file, it overwrites the data byte by byte.
- If **O_APPEND** option is specified in `open()`, the file position is set to the end of the file.

Example: read() and write()

```
int copyfile (const char *name1, const char *name2) {
    int infile, outfile;
    ssize_t nread;
    char buffer[BUFSIZE];          /* BUFSIZE : 512 */

    if ( (infile = open (name1, O_RDONLY ) )== -1)
         return(-1);               /* open name1 fail */
    if ( (outfile = open (name2, O_WRONLY | O_CREAT | O_TRUNC, 0644) )== -1) {
        close (infile);            /* open name2 fail */
        return (-2);
    }
    while ( (nread = read (infile, buffer, BUFSIZE) ) > 0) {
        if ( write(outfile, buffer, nread) < nread ) {
            close (infile);
            close (outfile);
            return (-3);           /* error on write */
        }
    }
    close (infile);
    close (outfile);
    if ( nread == -1)  return (-4) /* error on the last read */
    else   return (0);             /* success */
}
```

The textbook (Korean version), 29 page, copyfile().

Read and Write Efficiency Analysis (1/2)

- Timings measured with different buffer sizes.

BUFSIZE	Real time (s)	User time (s)	System time (s)
1	24.49	3.13	21.16
64	0.46	0.12	0.33
512	0.12	0.02	0.08
4096	0.07	0.00	0.05
8192	0.07	0.01	0.05

- The larger the buffer, the better the performance.
  - The best performance is achieved when the buffer size is divisible by the system's disk block size (4,096).
  - It also reduces the number of system calls (context switching cost!).

Read and Write Efficiency Analysis (2/2)

BUFSIZE	Real time (s)	User time (s)	System time (s)
1	24.49	3.13	21.16
64	0.46	0.12	0.33
512	0.12	0.02	0.08
4096	0.07	0.00	0.05
8192	0.07	0.01	0.05

- It seems like too fast! We are writing data into the disk! What happened?
  - `write()` does not directly write the data into the disk space but put them into the kernel buffer cache and then return the context (delayed writing).

System Call: lseek()

```
#include <unistd.h>
off_t lseek (int filedes, off_t offset, int start_flag);
```

- An open file's file position is explicitly set by `lseek()`.
  - File position is an offset in a regular file that marks where the next `read()` or `write()` will occur.
  - `lseek()` returns the new file position on success or -1 on error.
- The start_flag option:

String type option	Integer type option	Description
SEEK_SET	0	Beginning of the file
SEEK_CUR	1	The '`offset`' position
SEEK_END	2	End of the file.

Examples of lseek()

```
off_t filesize;
int filedes;

filesize = lseek(fd, (off_t)0, SEEK_END);
```

- This trick is popularly used to find the current size of the file.
- The below two examples are identical.

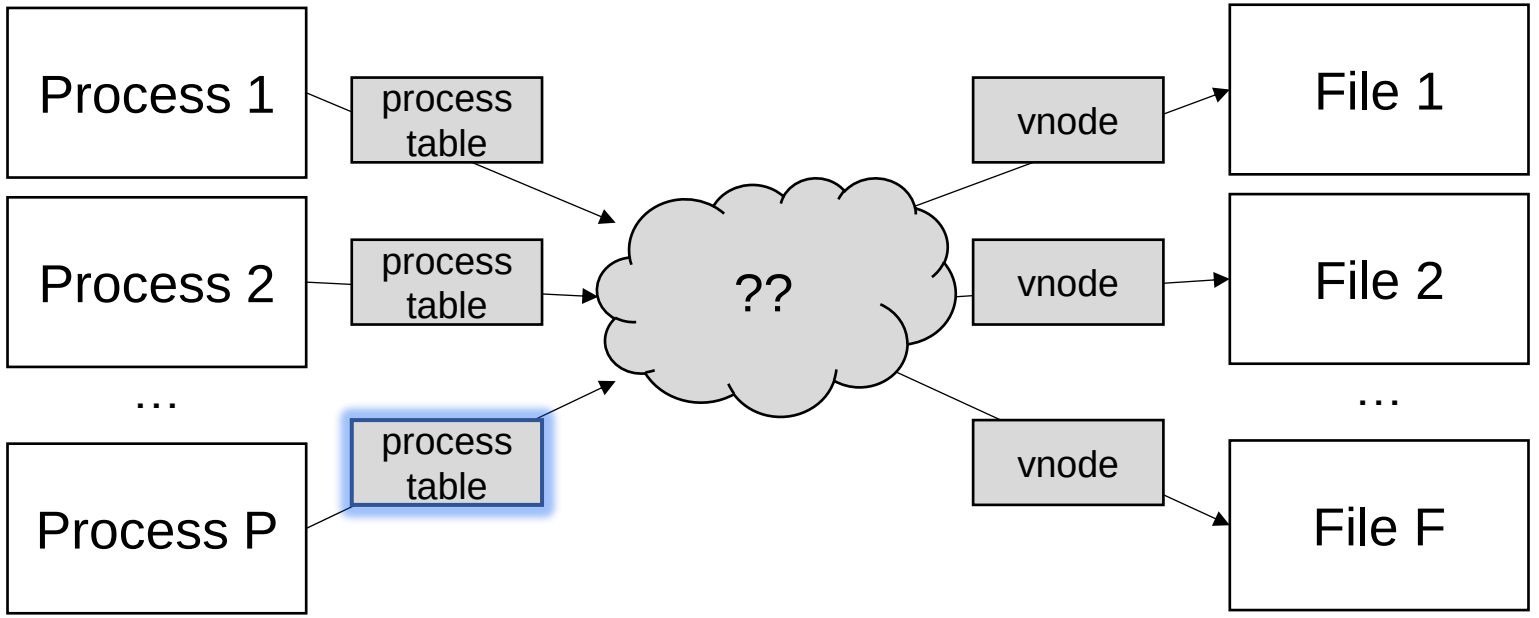
```
fd = open(fname, O_RDWR);
lseek(fd, (off_t)0, SEEK_END);
write(fd, outbuf, OBSIZE);
```

```
fd = open(fname, O_WRONLY|O_APPEND);
write(fd, outbuf, OBSIZE);
```

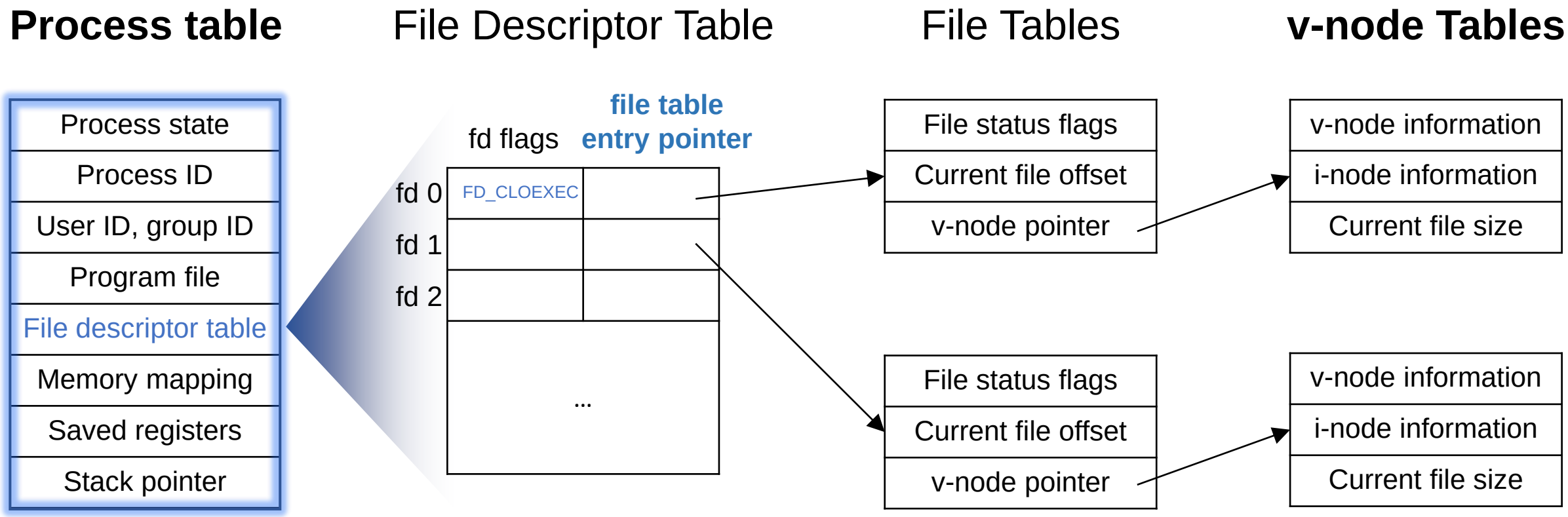
- Files and basic system calls
- ***File sharing***
- Standard I/O
- Standard I/O Libraries

File Share

- Because multiple processes can share a single file, we need a system to organize such shared files and their information.

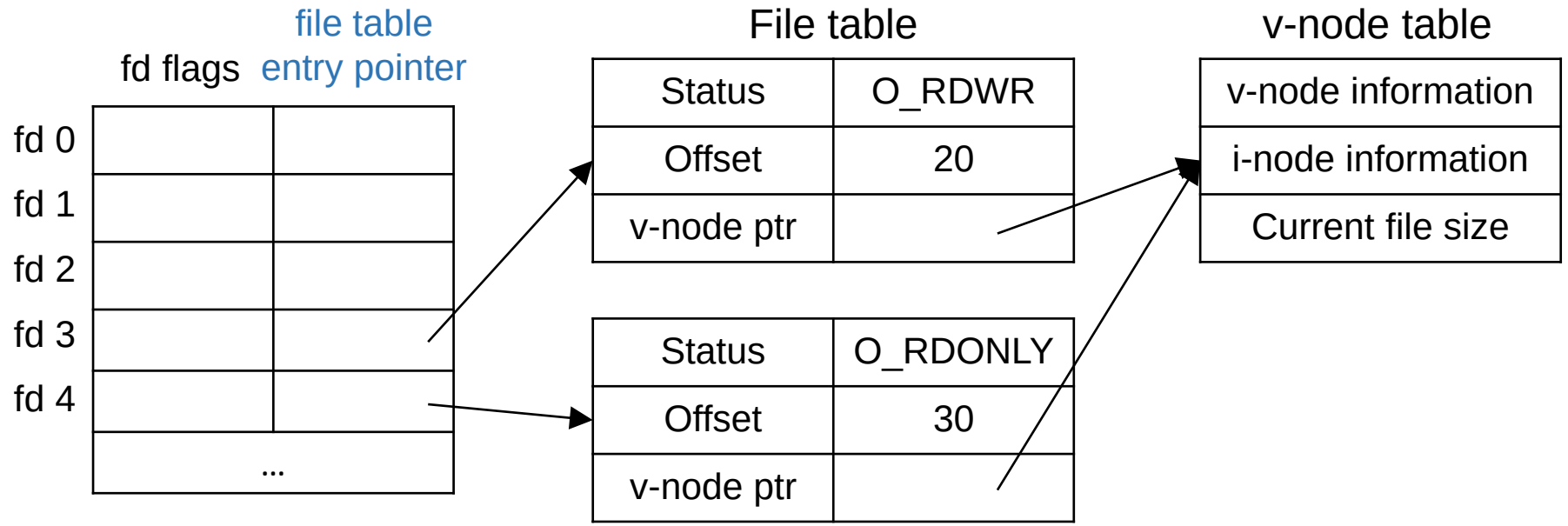


Hierarchical File Sharing



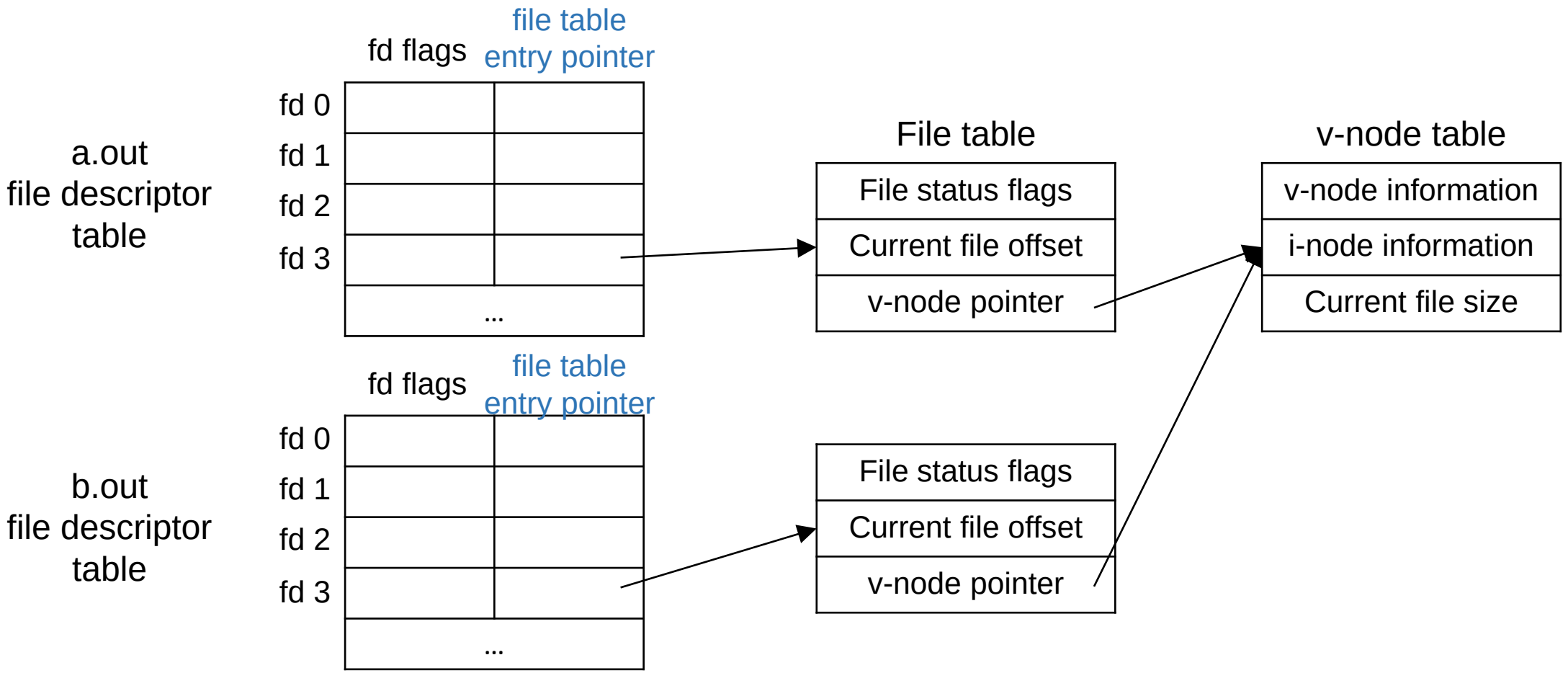
Example: File Sharing in a Process

```
int fd3, fd4; char buf[20];

fd3 = open("file", O_RDWR);
fd4 = open("file", O_RDONLY);

read(fd3, buf, 20);
read(fd4, buf, 30);

close(fd3); close(fd4);
```



Example: File Sharing across Processes

```
$ ./a.out          /* fd = open("test", O_RDONLY); */
$ ./b.out          /* fd = open("test", O_RDONLY); */
```



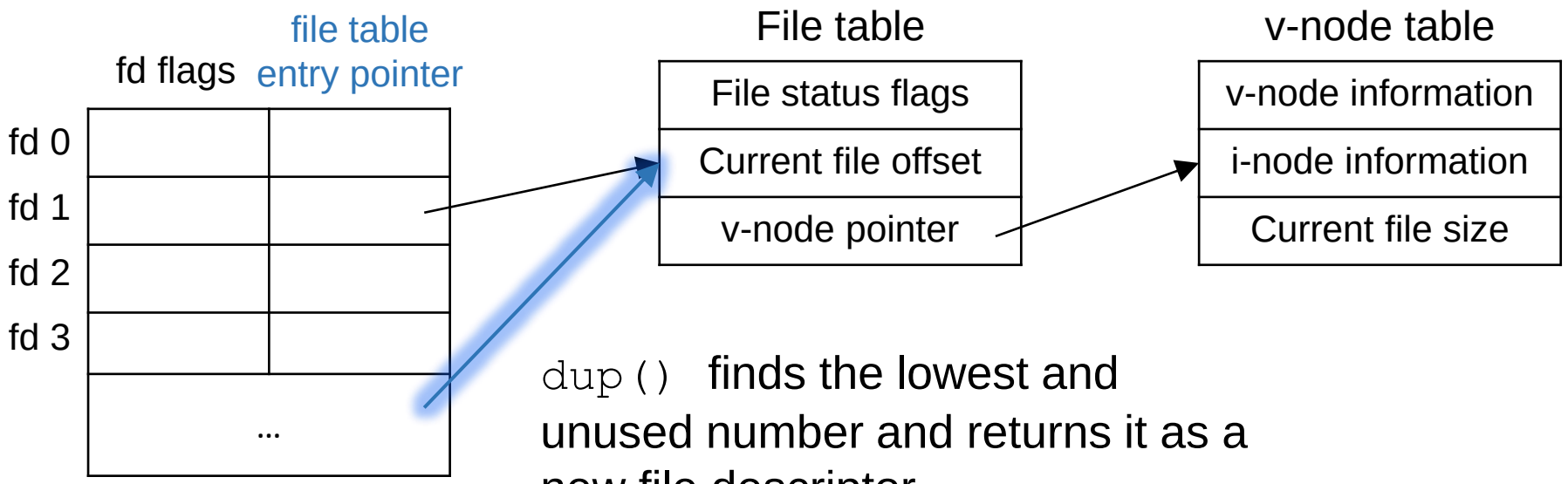
System Call: Dup() and Dup2()

```
#include <unistd.h>

int dup(int filedes);
int dup2(int filedes, int filedes2);
```

- `dup()` duplicates an existing file descriptor and returns the new file descriptor (lowest-number unused).
- `dup2()` duplicates an existing file descriptor and returns the new file descriptor (the user-provided value, `filedes2`).

Example: dup()

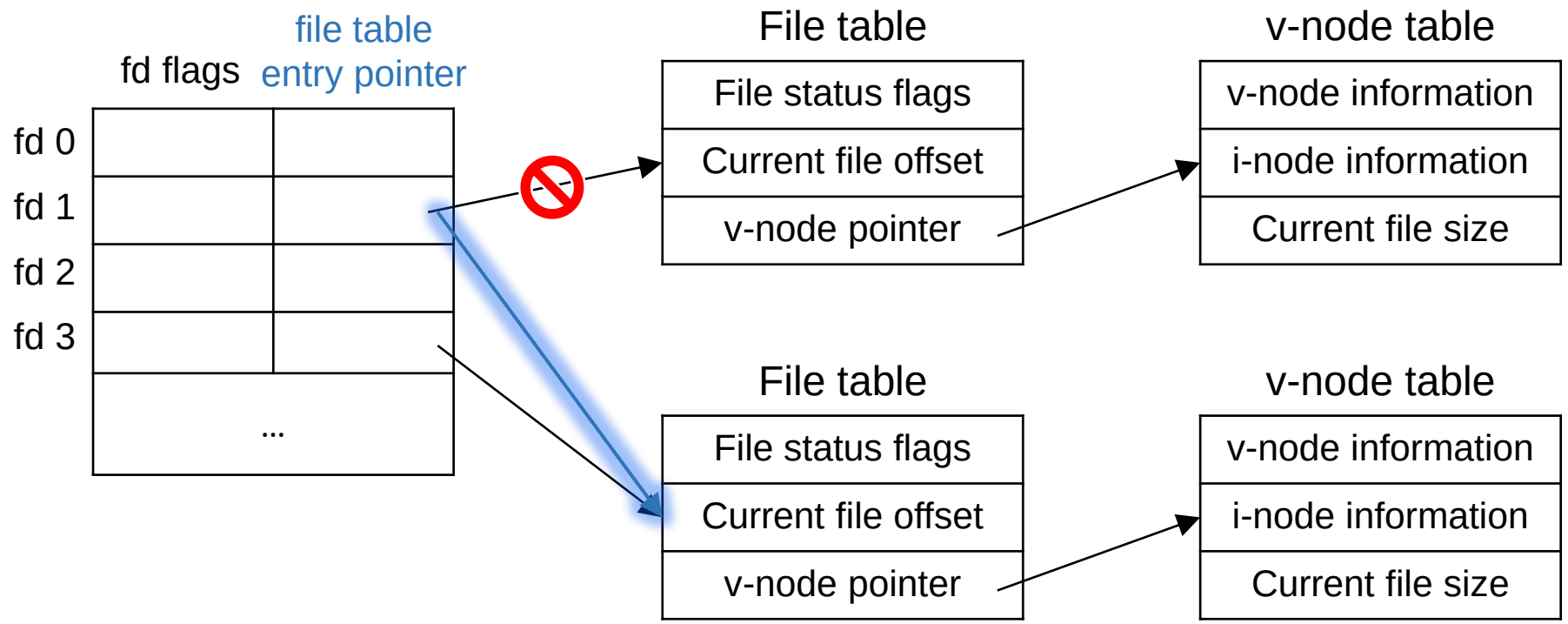
```
newfd = dup(1)
```



`dup()` finds the lowest and unused number and returns it as a new file descriptor.

Example: dup2()

```
fd3 = open("test", O_RDWR);
dup2(fd3, 1);
```



System Call: fcntl()

```
#include <fcntl.h>

int fcntl(int filedes, int cmd, …);
```

- `fcntl()` changes the properties of a file that is already opened.
  - The return value is dependent on `cmd` on success and -1 on error.
- Arguments
  - `filedes`: file descriptor of the target file.
  - `cmd`
    - F_DUPFD
    - F_GETFD / F_SETFD
    - F_GETFL / F_SETFL
    - F_GETOWN / F_SETOWN

} The flags related to file access (O_RDONLY, O_WRONLY, O_RDWR) and create (O_CREAT, O_EXCL, O_TRUNC) cannot be changed.

Example of fcntl()

```
#include <fcntl.h>

int filestatus(int filedes){
  int arg1;

  if (( arg1 = fcntl (filedes, F_GETFL)) == -1) {
    printf ("filestatus failed\n");
    return (-1);
  }

  /* file access mode flag test */
  switch ( arg1 & O_ACCMODE) {
    case O_WRONLY: printf ("write-only"); break;
    case O_RDWR:   printf ("read-write"); break;
    case O_RDONLY: printf ("read-only");  break;
    default: printf("No such mode");
  }
  if (arg1 & O_APPEND)
    printf (" -append flag set");

  printf ("\n");
  return (0);
}
```

arg1	1	1	1	1	1	1	0	0	1
O_ACCMODE	0	0	0	0	0	0	0	1	1
Result	0	0	0	0	0	0	0	0	1

After bit-wise and operation, the output directly show O_RDONLY, O_WRONLY, O_RDWR options!

- Files and basic system calls
- File sharing
- ***Standard I/O***
- Standard I/O Libraries

Standard I/O File Descriptors

- File descriptor ***0***: standard ***input***
- File descriptor ***1***: standard ***output***
- File descriptor ***2***: standard ***error***
- All the above three files are assigned to the I/O devices at the booting time.
  - 0 to the keyboard, 1 and 2 to the display.

Redirection (1/3)

```
$ program_name
```

- When typing the program name in shell, the `program_name` is read from the '***standard input***'.

```
$ program_name < input_file
```

- The `program_name` is read from the `input file`.
- The `<` is the input redirection operator.

```
newfd = open("input_file", O_RDONLY);
dup2(newfd, 0);
```

- The standard input descriptor copies the data from `newfd`, the input file.

Redirection (2/3)

```
$ program_name
```

- The output of the program is automatically written to '***standard output***'.

```
$ program_name > output_file
```

- The output of `program_name` is written to the `output file`.
- The `>` is the output redirection operator.

```
newfd = open("input_file", O_WRONLY);
dup2(newfd, 1);
```

Redirection (3/3)

```
$ program_name < input_file > output_file
```

- The `program_name` is read from the `input_file`, and its output is written to the `output_file`.

```
$ program1 | program2
```

- The input of `program2` is the output of `program1`.
- The `|` is pipe. We will study the pipe in chapter 7.

Example: Standard I/O

```
#include <stdlib.h>
#include <unistd.h>

#define SIZE 512

main()
{
  ssize_t nread;
  char buf[SIZE];

  while ( (nread = read (0, buf, SIZE)) > 0)
     write (1, buf, nread);
  exit (0);
}
```

```
$ ./io
This is line 1 (return)
This is line 1
This is line 2 (return)
This is line 2
<Ctrl-D>
$
```

- Each line is printed when the `return` key is pressed.
  - `read()` accepts data from a terminal after each newline character.

- Files and basic system calls
- File sharing
- Standard I/O
- ***Standard I/O Libraries***

The Standard I/O Library (1/3)

- Unix (Linux) I/O system call
  - It handles data only in the form of a simple sequence of bytes.
  - It leaves everything else up to the programmer.
  - Efficiency consideration also falls into the lap of the developer.

- Standard I/O (ANSI C)
  - Automatic buffering
  - more programmer-friendly interfaces
  - stdio.h

The Standard I/O Library (2/3)

- Difference between Standard I/O and Unix (LINUX) I/O
  - FILE * vs. file descriptor
- Standard I/O routines are written around the system call primitives.

```
#include <stdlib.h>
#include <unistd.h>

void main(){
    FILE *file_stream;
    if ((file_stream = fopen("junk", "r")) == NULL){
        printf("Could not open the junk file!\n");
        exit(1);
    }
}
```

The Standard I/O Library (3/3)

- The C standard library contains a collection of high-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 Library: fopen()

```
#include <stdio.h>

FILE *fopen(const char *restrict pathname, const char *restrict type);
```

- `type`
  - r or rb : open a file for reading (b for binary mode).
  - w or wb : truncate to 0 length or create a new file.
  - a or ab : append; open for writing at the end of the file or create a new file.
  - r+ / r+b / rb+ : open for reading and writing.
  - w+ / w+b / wb+ : truncate to 0 length or create a new file for reading and writing.
  - a+ / a+b / ab+ : open or create for reading and writing at the end of the file.

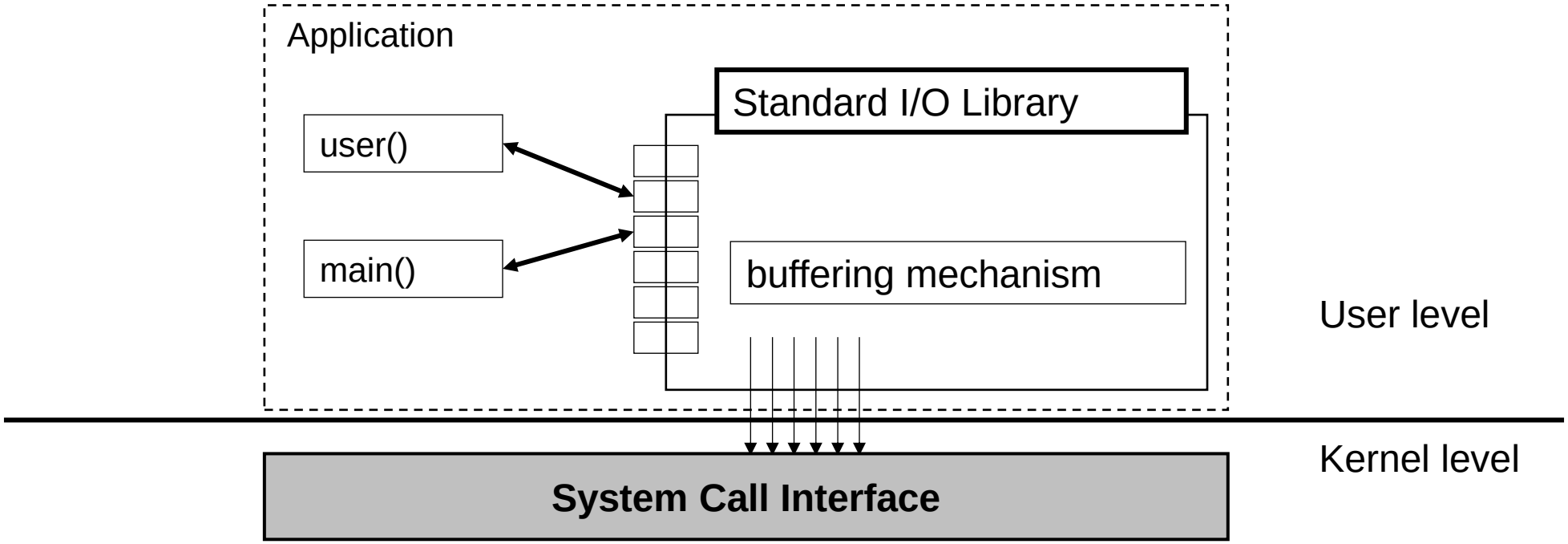
Standard I/O Library: getc() and putc()

```
#include <stdio.h>

int getc(FILE *istream);
int putc(int c, FILE *ostream);
```

- `getc()` returns the next character on success and EOF on the end of the file or error.
- `putc()` returns c on success and EOF on error.

Buffering

- Standard I/O avoid inefficiency by an elegant buffering mechanism.
- The data is internally buffered in the heap space, and the I/O is performed at bulk.
  - `getc()` and `putc()` are efficiently implemented based on the internal buffering!



Standard I/O Library: fprintf()

```
#include <stdio.h>

int fprintf(FILE *restrict fp, const char *restrict format, …);
```

- `fprintf()` returns the number of characters output on success and negative value on output errors.
- `restrict` pointer:
  - Let the C compiler know that the pointer does not point to the same memory space as other pointers.
  - Skipping a few low-level instructions (performance improvement).
  - Users are expected to guarantee that `fp` and `format` do not point to the same memory space.

Any Questions?