EE309
Lecture 4: File I/O

INSU YUN (윤인수)

School of Electrical Engineering, KAIST

Today’s lecture

- Understand FILE I/O in advance!

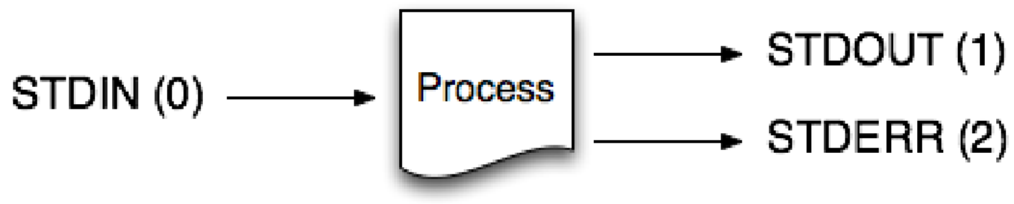
File descriptors

- To the kernel, all open files are referred to by file descriptors
- A file descriptor is a non-negative integer that is created
  - When we open an existing file or
  - When we create a new file

- When we want to read or write a file, we identify the file with the file descriptor

Standard in/out/error

- By convention, UNIX system shells associate file descriptors
  - 0: Standard input (stdin)
  - 1: Standard output (stdout)
  - 2: Standard error (stderr)



System-Level Functions for I/O

`int open(char *pathname, int flags, mode_t mode);`

- Opens the file `pathname` and returns a file descriptor

`int close(int fd);`

- Closes `fd`

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

- Reads up to `count` bytes from `fd` into the buffer at `buf`

`int write(int fd, void *buf, int count);`

- Writes up to `count` bytes into `fd` from the buffer at `buf`

`int lseek(int fd, int offset, int whence);`

- Assigns the file pointer of `fd` to a new value by applying an `offset`

`open()`

- Converts a path name into a file descriptor
  - `int open(const char *pathname, int flags, mode_t mode);`
- Arguments
  - `pathname`: name of the file
  - `flags`: bit flags for `O_RDONLY, O_WRONLY, O_RDWR`
  - `mode`: permissions to set if file must be created
- Returns
  - File descriptor (or -1 if error)
- Performs a variety of checks
  - e.g., whether the process is entitled to access the file (Later in details)

`close()`

- Close a file
  - `int close(int fd);`
- Arguments
  - `fd`: A file descriptor to close
- Returns
  - 0 if OK, -1 on error

- NOTE: When a process terminates, all of its open files are closed automatically by the kernel
  - But please try to do it explicitly for efficient resource management

`read()`

- Reads bytes from a file descriptor
  - `int read(int fd, void *buf, int count);`
- Arguments
  - File descriptor: integer descriptor returned by `open()`
  - Buffer: pointer to memory to store the bytes it reads
  - Count: maximum number of bytes to read
- Returns
  - Number of bytes read
    - Value of 0 if nothing more to read
    - Value of -1 if an error
- Performs a variety of checks
  - Whether file has been opened, whether reading is okay

`write()`

- Writes bytes from a file descriptor
  - `int write(int fd, void *buf, int count);`
- Arguments
  - File descriptor: integer descriptor returned by `open()`
  - Buffer: pointer to memory to write the bytes
  - Count: maximum number of bytes to write
- Returns
  - Number of bytes write
    - Usually equal to `count`
    - Value of -1 if an error
- Performs a variety of checks
  - Whether file has been opened, whether writing is okay

lseek()

- Assigns the file pointer of `fd` to a new value by applying an `offset`
  - `int lseek(int fd, off_t offset, int whence);`
- Arguments
  - File descriptor: integer descriptor returned by `open()`
  - If `whence` is `SEEK_SET`, the file's offset is set to offset bytes from the beginning of the file
  - If `whence` is `SEEK_CUR`, the file's offset is set to its current value plus the offset
  - If `whence` is `SEEK_END`, the file's offset is set to the size of the file plus the offset
- Returns
  - The current new file offset

```
// Get the current offset
off_t currpos;
currpos = lseek(fd, 0, SEEK_CUR);
```

TODO

- Add example about lseek()

I/O Efficiency

- Reads from standard input and writes to standard output
- The program doesn't close the input file or output file
  - Instead, the program uses the feature of the UNIX kernel that closes all open file descriptors in a process when that process terminates

```c
#include "apue.h"

#define BUFFSIZE    4096

int
main(void)
{
    int     n;
    char    buf[BUFFSIZE];

    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");

    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

`BUFFSIZE`	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	Number of loops
1	20.03	117.50	138.73	516,581,760
2	9.69	58.76	68.60	258,290,880
4	4.60	36.47	41.27	129,145,440
8	2.47	15.44	18.38	64,572,720
16	1.07	7.93	9.38	32,286,360
32	0.56	4.51	8.82	16,143,180
64	0.34	2.72	8.66	8,071,590
128	0.34	1.84	8.69	4,035,795
256	0.15	1.30	8.69	2,017,898
512	0.09	0.95	8.63	1,008,949
1,024	0.02	0.78	8.58	504,475
2,048	0.04	0.66	8.68	252,238
4,096	0.03	0.58	8.62	126,119
8,192	0.00	0.54	8.52	63,060
16,384	0.01	0.56	8.69	31,530
32,768	0.00	0.56	8.51	15,765
65,536	0.01	0.56	9.12	7,883
131,072	0.00	0.58	9.08	3,942
262,144	0.00	0.60	8.70	1,971
524,288	0.01	0.58	8.58	986

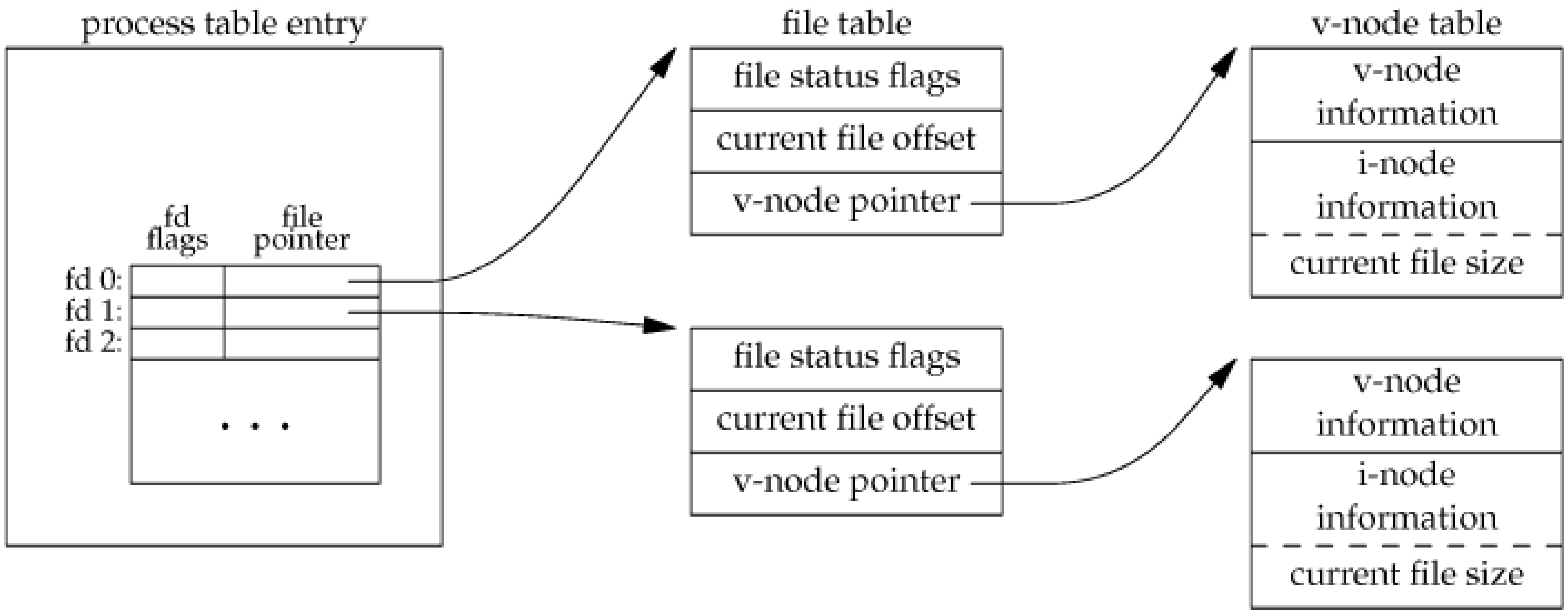
File sharing

- UNIX system supports the sharing of open files among different processes

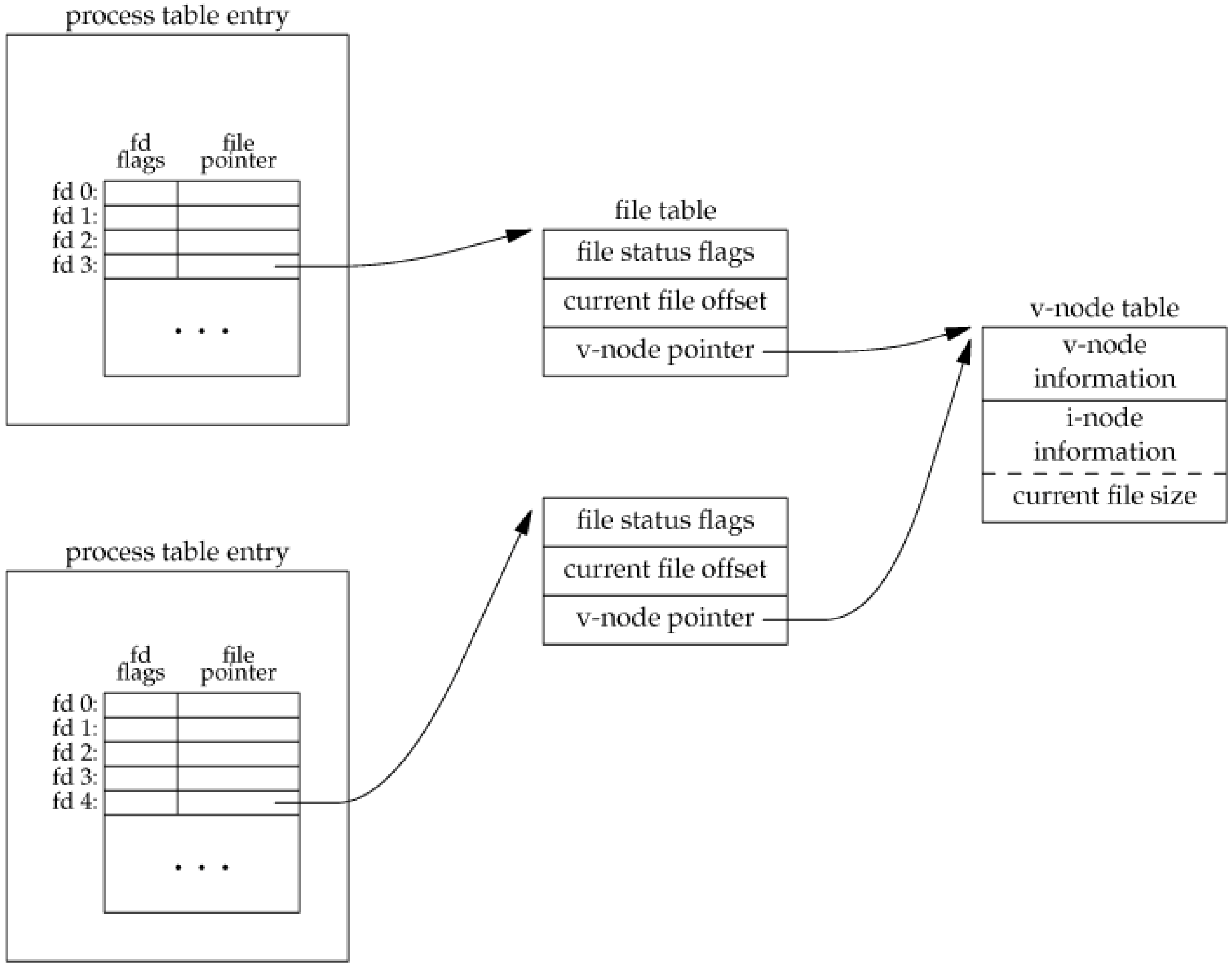
```
int fd1 = open("./hello.txt", O_RDONLY);
int fd2 = open("./hello.txt", O_WRONLY);
```

Data structures to represent open files

- Process table entry for every process
  - The file descriptor flags
  - A pointer to file table entry
- A file table for all open files
  - The file status flags (e.g., read, write, append, …)
  - The current file offset
  - A pointer to the v-node table entry for the file
- Each open file has a v-node structure
  - Type of file (e.g., a normal file, a directory, a device, …)
  - Pointers to functions that operate on the file
  - Pointer to i-node: the owner of the file, the size of file, data blocks, …



- NOTE: Linux has no v-node, but it has a generic i-node structure. Conceptually, this is same with v-node.


process table entry
fd flags
file pointer
fd 0:
fd 1:
fd 2:
fd 3:
file table
file status flags
current file offset
v-node pointer
v-node table
v-node information
i-node information
current file size
process table entry
fd flags
file pointer
fd 0:
fd 1:
fd 2:
fd 3:
fd 4:
file status flags
current file offset
v-node pointer

```c
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>

#define MSG "Hello World"

int main() {
  // Assume hello.txt is an empty file
  char buf[sizeof(MSG)] = {};
  int status = 0;
  int fd1 = open("hello.txt", O_RDONLY);

  if (fork() == 0) {
    // child
    int fd2 = open("hello.txt", O_WRONLY);
    write(fd2, "Hello World", sizeof(MSG));
    return 0;
  }

  wait(&status);
  read(fd1, buf, sizeof(buf));
  printf("%s\n", buf);
}
```

```
$ cat hello.txt
$ ./sharing1
Hello World
```

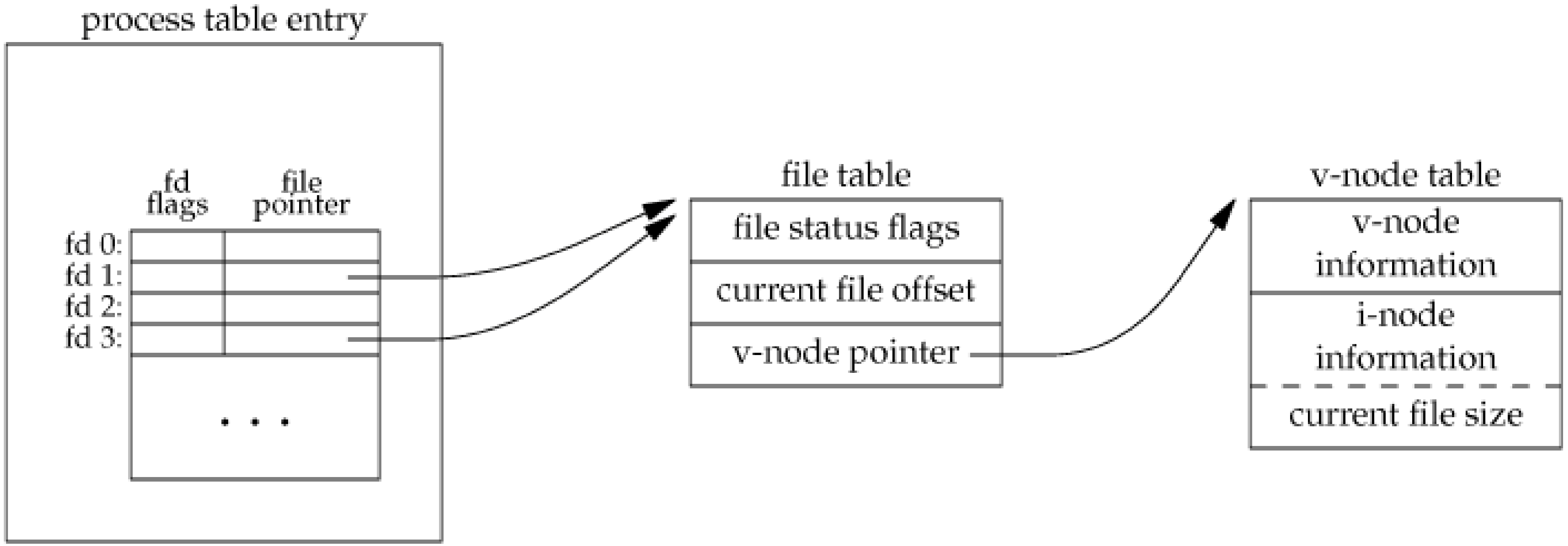
```c
int main() {
  // Assume hello.txt = "Hello World"
  char buf[sizeof(MSG)] = {};
  int status = 0;
  int fd1 = open("hello.txt", O_RDONLY);

  if (fork() == 0) {
    // child
    int fd2 = open("hello.txt", O_RDONLY);
    read(fd2, buf, sizeof(buf));
    return 0;
  }

  wait(&status);
  read(fd1, buf, sizeof(buf));
  printf("%s\n", buf);
}
```

```
$ cat hello.txt
Hello World
$ ./sharing2
Hello World
```

`dup()`

- Duplicate existing file descriptor
  - `int dup(int oldfd);`
- Arguments
  - `oldfd` : A file descriptor to duplicate
- Returns
  - New file descriptor if OK, -1 on error

- There is another version, `dup2(int oldfd, int newfd)`
  - This allows us to specify the new file descriptor to use


process table entry
fd flags
file pointer
fd 0:
fd 1:
fd 2:
fd 3:
. . .
file table
file status flags
current file offset
v-node pointer
v-node table
v-node information
i-node information
current file size

```c
#define MSG "Hello World"

int main() {
  // Assume hello.txt = "Hello World"
  char buf[sizeof(MSG)] = {};
  int status = 0;
  int fd1 = open("hello.txt", O_RDONLY);

  if (fork() == 0) {
    // child
    // int fd2 = open("hello.txt", O_RDONLY);
    int fd2 = dup(fd1);
    read(fd2, buf, sizeof(buf));
    return 0;
  }

  wait(&status);
  read(fd1, buf, sizeof(buf));
  printf("%s\n", buf);
}
```

```
$ cat hello.txt
Hello World
$ ./sharing3
```

Atomic operation

- *atomic operation* refers to an operation that might be composed of multiple steps.
  - If the operation is performed atomically, either all the steps are performed (on success) or none are performed (on failure).
  - It must not be possible for only a subset of the steps to be performed.

- ***If we deal with files that can be shared by multiple threads, we should be aware of such atomic operations***

Atomic Operation (1): Append a file

- Older versions of the UNIX System didn't support the O_APPEND option if a single process wants to append to the end of a file
  - The program would be:

```
if (lseek(fd, 0L, 2) < 0)           /* position to EOF */
   err_sys("lseek error");
if (write(fd, buf, 100) != 100)     /* and write */
   err_sys("write error");
```

  - Single-process → fine
  - But what if there are multiple processes and they are trying to touch the same file?
  - Thus, O_APPEND is introduced!

Atomic Operation (2): pread( ) and pwrite( )

- The Single UNIX Specification includes two functions that allow applications to seek and perform I/O atomically:
  - pread: equivalent to calling lseek followed by a call to read, with the following exceptions:
    - There is no way to interrupt the two operations that occur calling pread.
    - The current file offset is not updated.
  - pwrite: equivalent to calling lseek followed by a call to write, with similar exceptions to pread

```
#include <unistd.h>

ssize_t pread(int fd, void *buf, size_t nbytes, off_t offset);

                    Returns: number of bytes read, 0 if end of file, –1 on error

ssize_t pwrite(int fd, const void *buf, size_t nbytes, off_t offset);

                    Returns: number of bytes written if OK, –1 on error
```

Atomic Operation (3) Creating a file

- When both of O_CREAT and O_EXCL options are specified, the open will fail if the file already exists.
  - The check for the existence of the file and the creation of the file was performed as an atomic operation.
- **Non-atomic operation**
  - If we didn't have this atomic operation, we might try:

```
if ((fd = open(path, O_WRONLY)) < 0) {
    if (errno == ENOENT) {
        if ((fd = creat(path, mode)) < 0)
            err_sys("creat error");
    } else {
        err_sys("open error");
    }
}
```

Atomic Operation (3) Creating a file

- The problem occurs if the file is created by another process between the open and the creat()
- If the file is created by another process between these two function calls, and if that other process writes something to the file, that data is erased when this creat( ) is executed.
- Combining the test for existence and the creation into a single atomic operation avoids this problem.

sync, fsync, and fdatasync

- Traditional implementations of the UNIX System have a buffer cache or page cache in the kernel through which most disk I/O passes.
- **Delayed write**
  - when we write data to a file, the data is normally copied by the kernel into one of its buffers and queued for writing to disk at some later time
- The kernel eventually writes all the delayed-write blocks to disk, normally when it needs to reuse the buffer for some other disk block.
- To ensure consistency of the file system on disk with the contents of the buffer cache, the sync, fsync, and fdatasync functions are provided.

sync, fsync, and fdatasync

```c
#include <unistd.h>

int fsync(int fd);
int fdatasync(int fd);

/* Returns: 0 if OK, -1 on error */

void sync(void);
```

- sync( )
  - queues all the modified block buffers for writing and returns. It does not wait for the disk writes to take place
  - sync is normally called periodically (usually every 30 seconds) from a system daemon, often called update, which guarantees regular flushing of the kernel's block buffers.
- fsync( )
  - applies to a single file specified by the file descriptor *fd*, and waits for the disk writes to complete before returning.
  - fsync also updates the file's attributes synchronously
- fdatasync( )
  - similar to fsync, but it affects only the data portions of a file

fflush() and fsync( )

- fflush() works on FILE*,
  - flushes the internal buffers in the FILE* of your application out to the OS.
- fsync works on a lower level,
  - tells the OS to flush its buffers to the physical media.

- Call fflush() may also invoke fsync(), but no guarantee

fcntl( )

- The fcntl function is used for five different purposes:
  - Duplicate an existing descriptor (*cmd* = F_DUPFD or F_DUPFD_CLOEXEC)
  - Get/set file descriptor flags (*cmd* = F_GETFD or F_SETFD)
  - Get/set file status flags (*cmd* = F_GETFL or F_SETFL)
  - Get/set asynchronous I/O ownership (*cmd* = F_GETOWN or F_SETOWN)
  - Get/set record locks (*cmd* = F_GETLK, F_SETLK, or F_SETLKW)

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* int arg */ );

/* Returns: depends on cmd if OK (see following), −1 on error */
```

File flags - (will be discussed later)

```
#include "apue.h"
#include <fcntl.h>

void
set_fl(int fd, int flags) /* flags are file status flags to turn on */
{
    int val;
    if ((val = fcntl(fd, F_GETFL, 0)) < 0)
        err_sys("fcntl F_GETFL error");
    val |= flags; /* turn on flags */
    if (fcntl(fd, F_SETFL, val) < 0)
        err_sys("fcntl F_SETFL error");
}
```