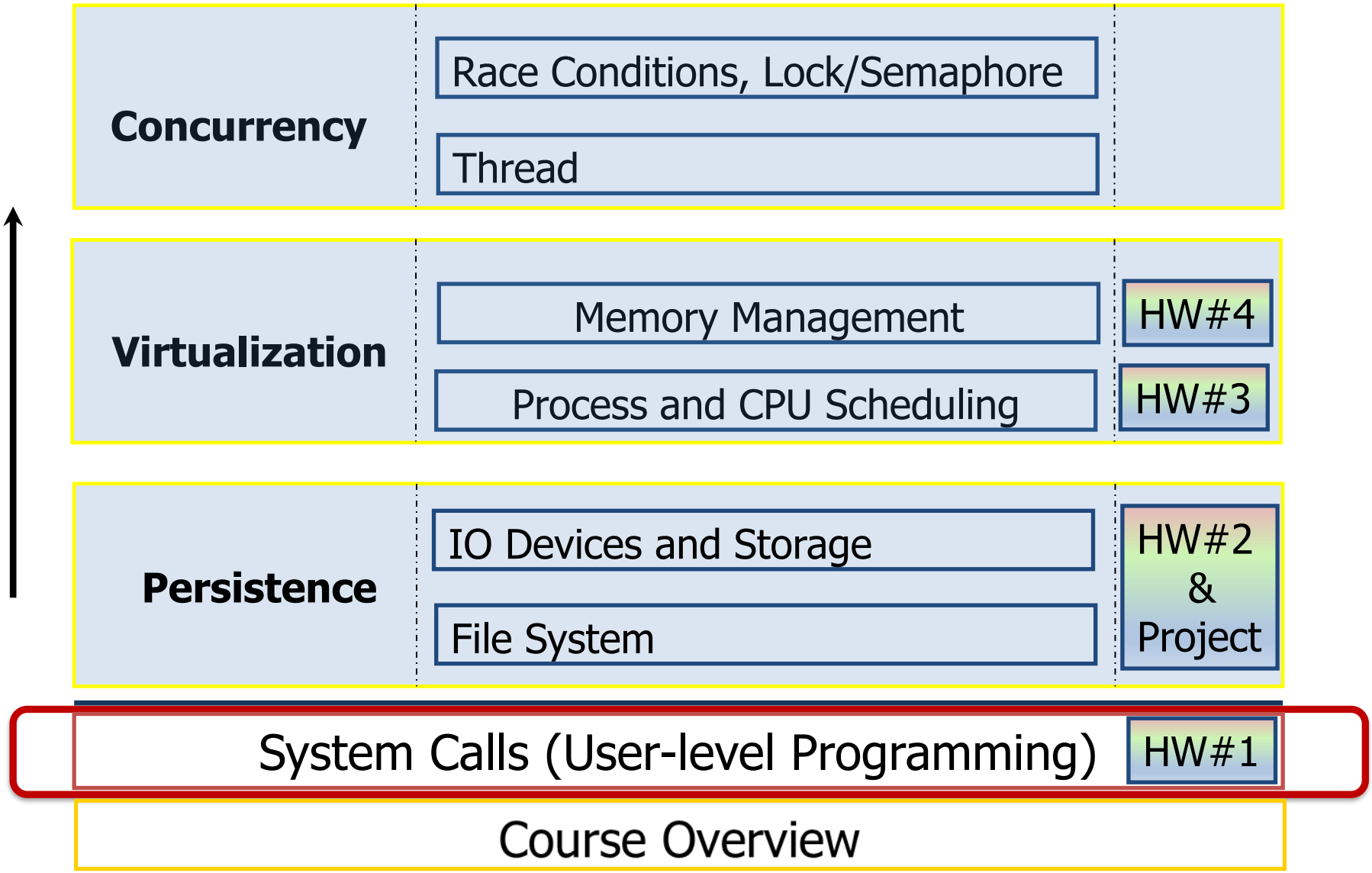
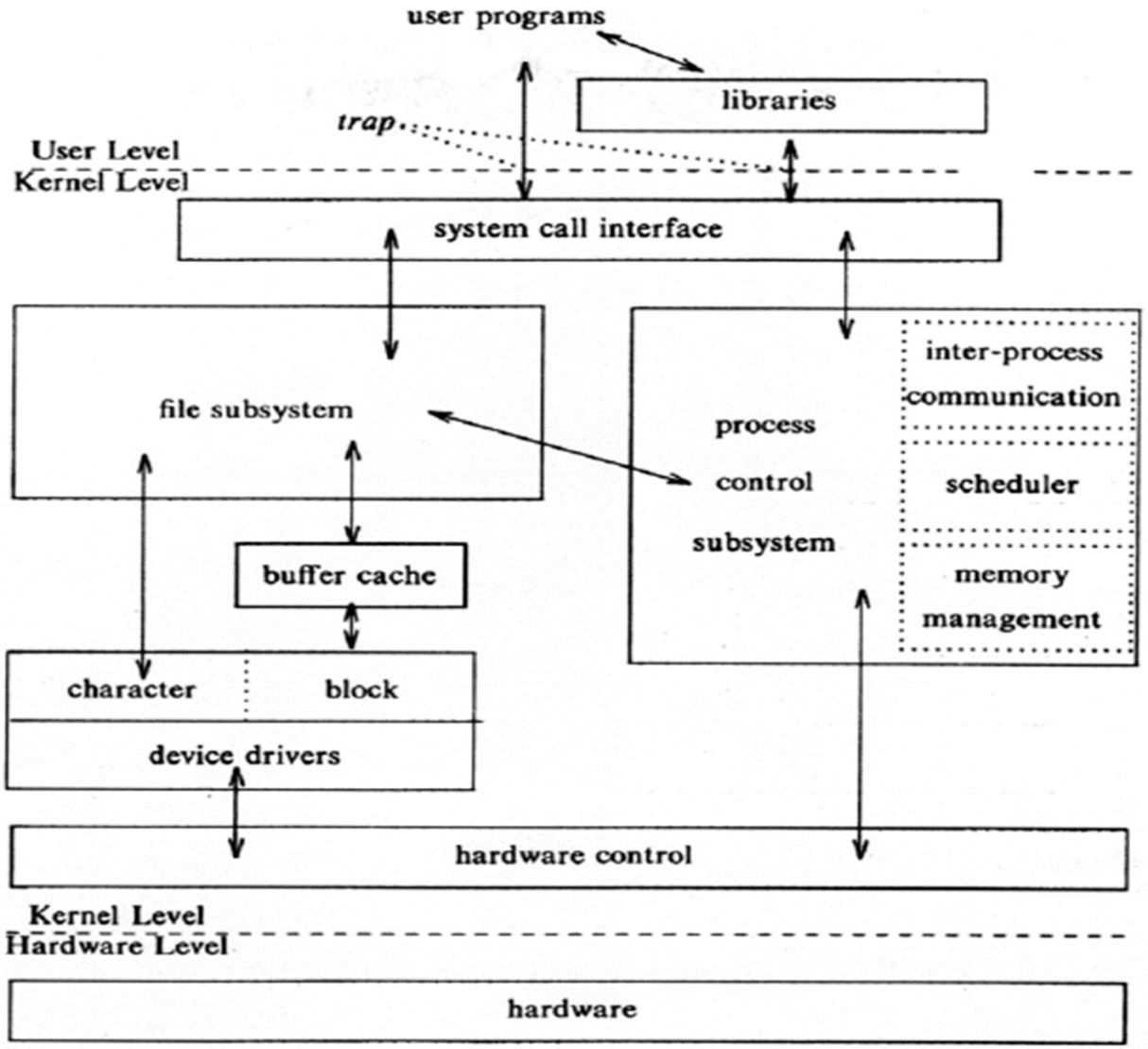
Lecture 4: User-level Programming via System Calls (File & Directory)

The Course Organization (Bottom-up)



System call

OS provides services via **System Call** (typically a few hundred) to run **process**, access memory/devices/files, etc.



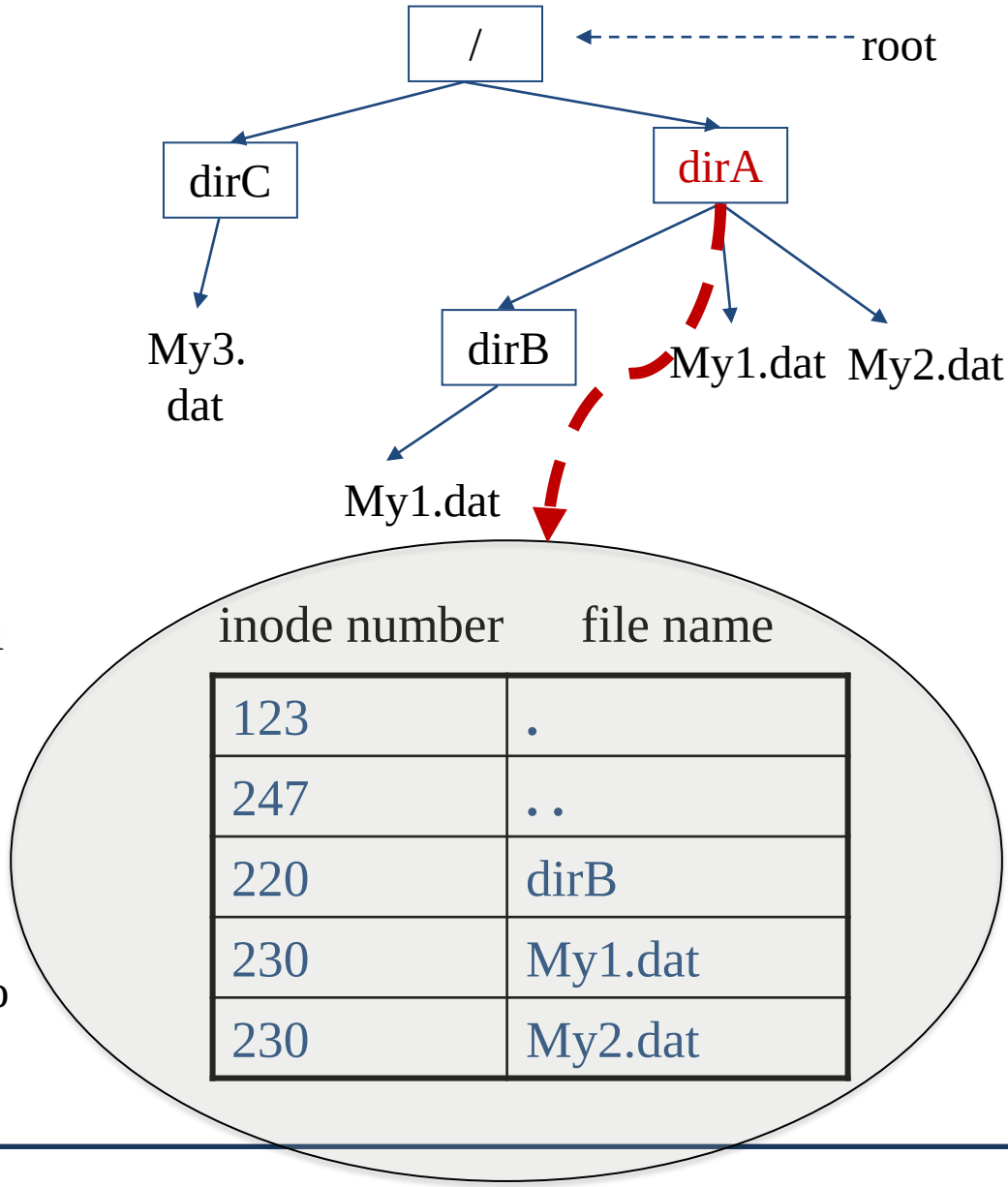
The Design Of The Unix Operating System (Maurice Bach, 1986)

Persistent Storage

- Keep a data **intact** even if there is a power loss.
  - Hard disk drive
  - Solid-state storage device
- Two key abstractions in the virtualization of storage
  - File
  - Directory

File and Directory

- File – A container to contain data of a file (a linear array of bytes)
  - Each file has low-level name (**inode number**)
- Directory – Implement directory tree (directory hierarchy)
  - Like a file, it also has a low-level name (**inode number**).
  - It contains a list of (**file name, inode number**) pairs.
  - Each entry in a directory refers to either *files* or other *directories*.



Creating Files

- Use `open()` system call with `O_CREAT` flag.

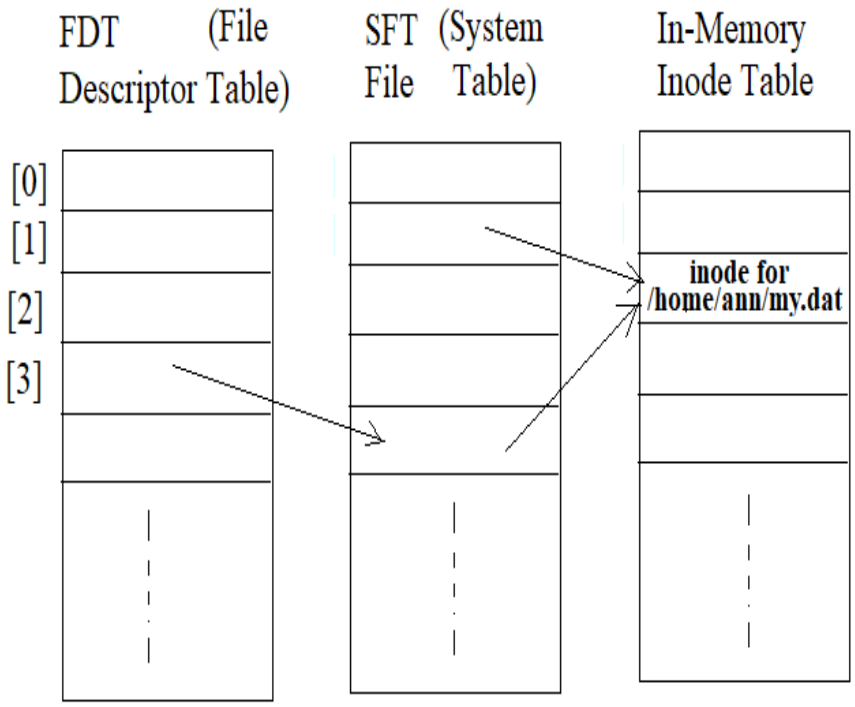
```
int myfd = open("/home/ann/my.dat", O_CREAT | O_WRONLY | O_TRUNC);
```

  - `O_CREAT` : create file.
  - `O_WRONLY` : only write to that file while opened.
  - `O_TRUNC` : make the file size zero (remove any existing content).

- `open()` system call returns **file descriptor**.
  - *File descriptor* is an integer, and is used to access files.
- `read(`file descriptor, buffer pointer, the number of bytes to read from`)`
  - Return the number of bytes it read
- `write(`file descriptor, buffer pointer, the number of bytes to write to`)`
  - Return the number of bytes it write

The Story behind open() (Unix System V)

```
int myfd = open("/home/ann/my.dat", O_CREAT | O_WRONLY | O_TRUNC);
```



myfd (the file descriptor for the newly-created file) is 3
When a process is created, file descriptor 0, 1, 2, are opened by default for standard input/ output/ error.

- A file descriptor specifies the index into file descriptor table (FDT) of the process.
- Entries of FDT contain pointers to entries in system file table (SFT).
- When a file is opened, an entry is created in both EDT and SFT.
- SFT entry contains information about whether a file is open for read or write, protection, and lock, the file offset, where the next data is read from or written to in the file, etc.
- Several entries in SFT may point to the same physical file

Reading And Writing, But Not Sequentially

- An open file has a **current offset**.
  - Determine **where** the next read or write will begin reading from or writing to within the file.
- Update the current offset
  - **Implicitly**: A read or write of `N` bytes takes place, `N` is added to the current offset.
  - **Explicitly**: `lseek()`

Reading And Writing, But Not Sequentially (Cont.)

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

- `fd` : File descriptor
- `offset` : Position the file offset to a particular location within the file
- `whence` : Determine how the seek is performed

From the man page:

```
If whence is SEEK_SET, the offset is set to offset bytes.
If whence is SEEK_CUR, the offset is set to its current
location plus offset bytes.
If whence is SEEK_END, the offset is set to the size of the
file plus offset bytes.
```

I/O redirection

- To access a file, a process uses a *file descriptor*, which is an index into the process *file descriptor table*, which in turn points to an entry in the *system file table*.
- *Redirection* means that the process modifies its file descriptor table entry so that it points to a different entry in the system file table.
  - Consider the command cat, which reads from a file and echoes to standard output.

    The following command redirects standard output to **my.file**

    ls -l > my.file

Use “dup()” to implement “redirection”

```
int dup(int fd);

DESCRIPTION
dup() is a “smart” function that can duplicate the file
descriptor, “fd”, to the lowest-numbered unused file
descriptor in the file descriptor table.


RETURN VALUE:
On success, return the new file descriptor.
On error, -1 is returned, and errno is set to indicate the
error.
```

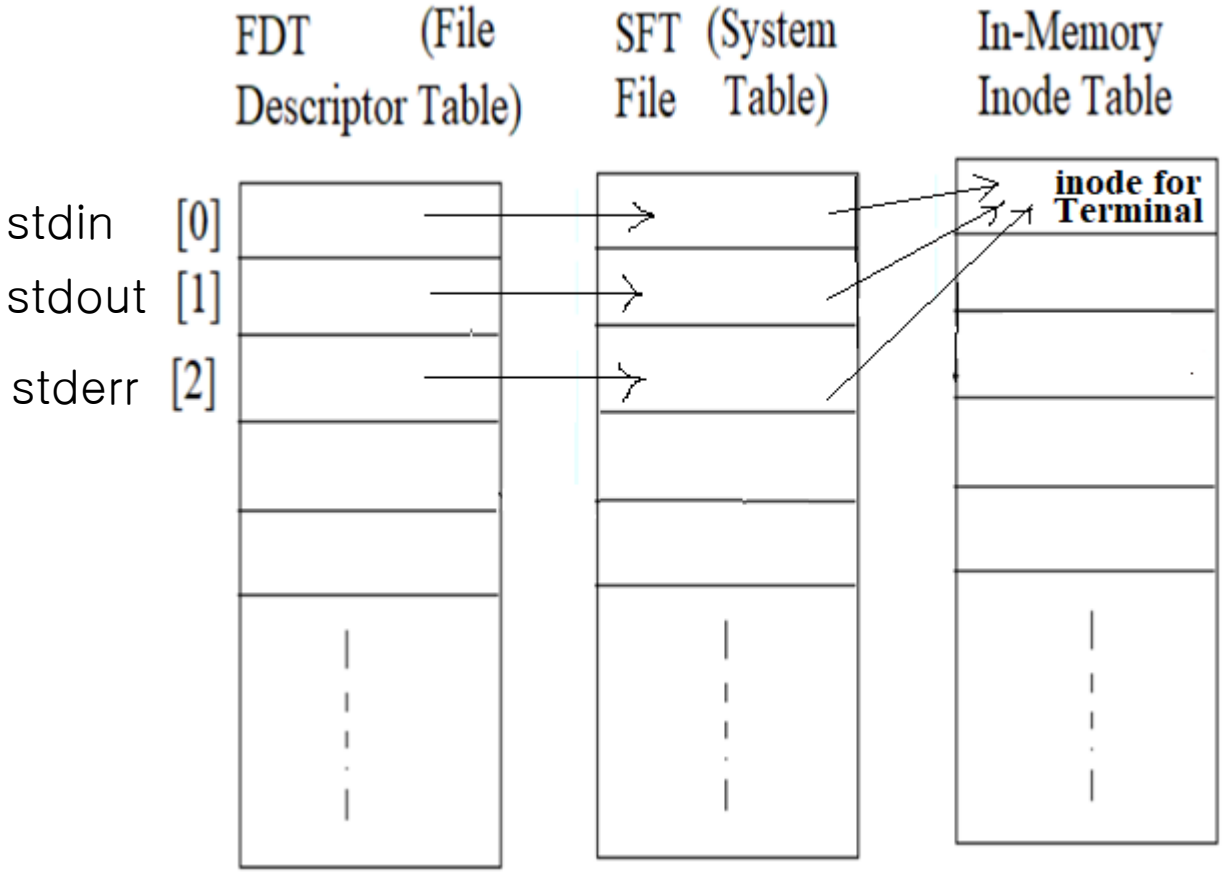
Use “dup()” to implement “redirection”

```
int dup(int fd)
```

- dup() is a “smart” function that can duplicate the file descriptor, “fd”, to the **lowest-numbered unused file descriptor** in the file descriptor table.

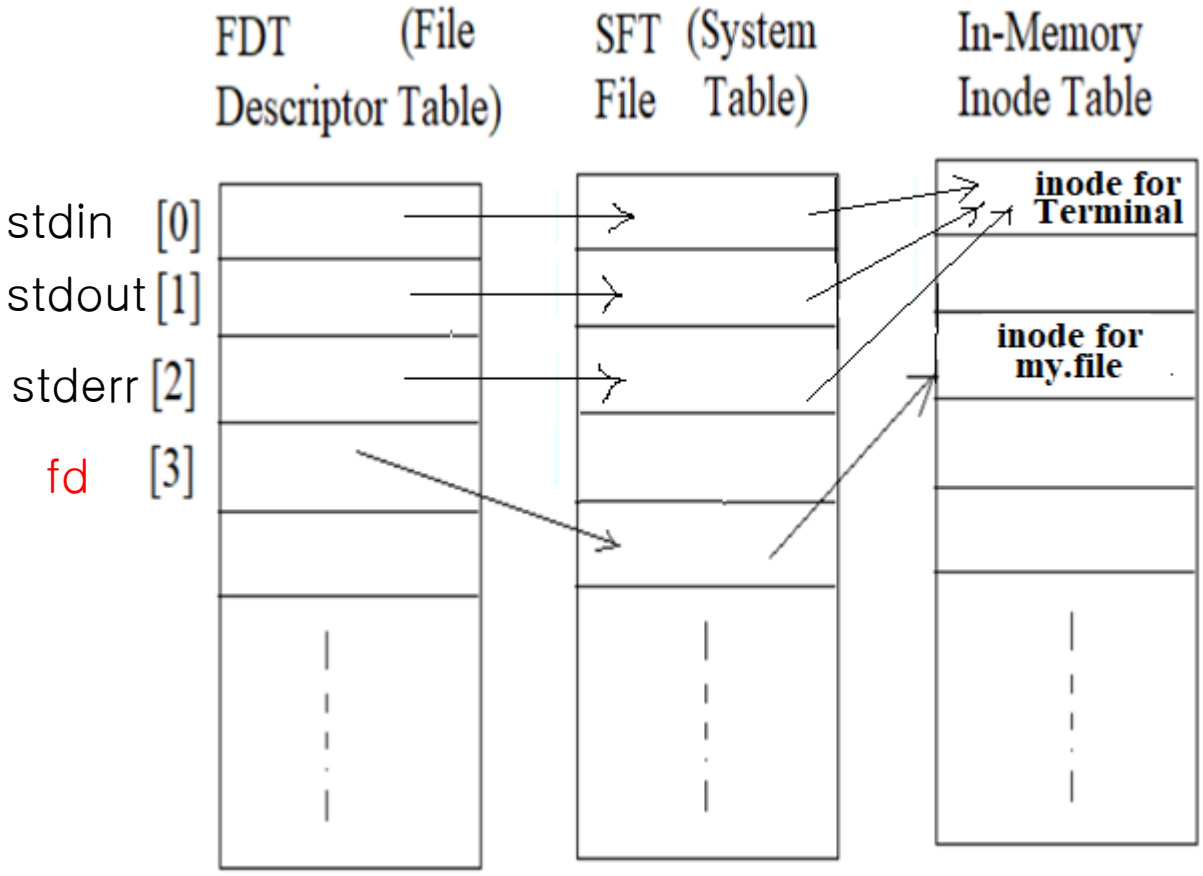
```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(void)
{
        int fd;

        fd = open("my.file", O_CREAT | O_TRUNC | O_WRONLY, S_IRUSR| S_IWUSR );
        close(1);
        dup(fd);
        close(fd);

        char *cmd = "ls";
        char *argv[3];
        argv[0] = "ls";    argv[1] = "-l";     argv[2] = NULL;

        execvp(cmd, argv);
}
```

I/O redirection




```
fd = open("my.file", O_CREAT, S_IRUSR| S_IWUSR);  /* Create the file – my.file */
close(1);                                          /* Close 1 (Standard output)*/
dup(fd);                                           /* Duplicate fd */
execvp(cmd, argv);                                 /* Execute  the command  ls -l  */
```

I/O redirection (continue)



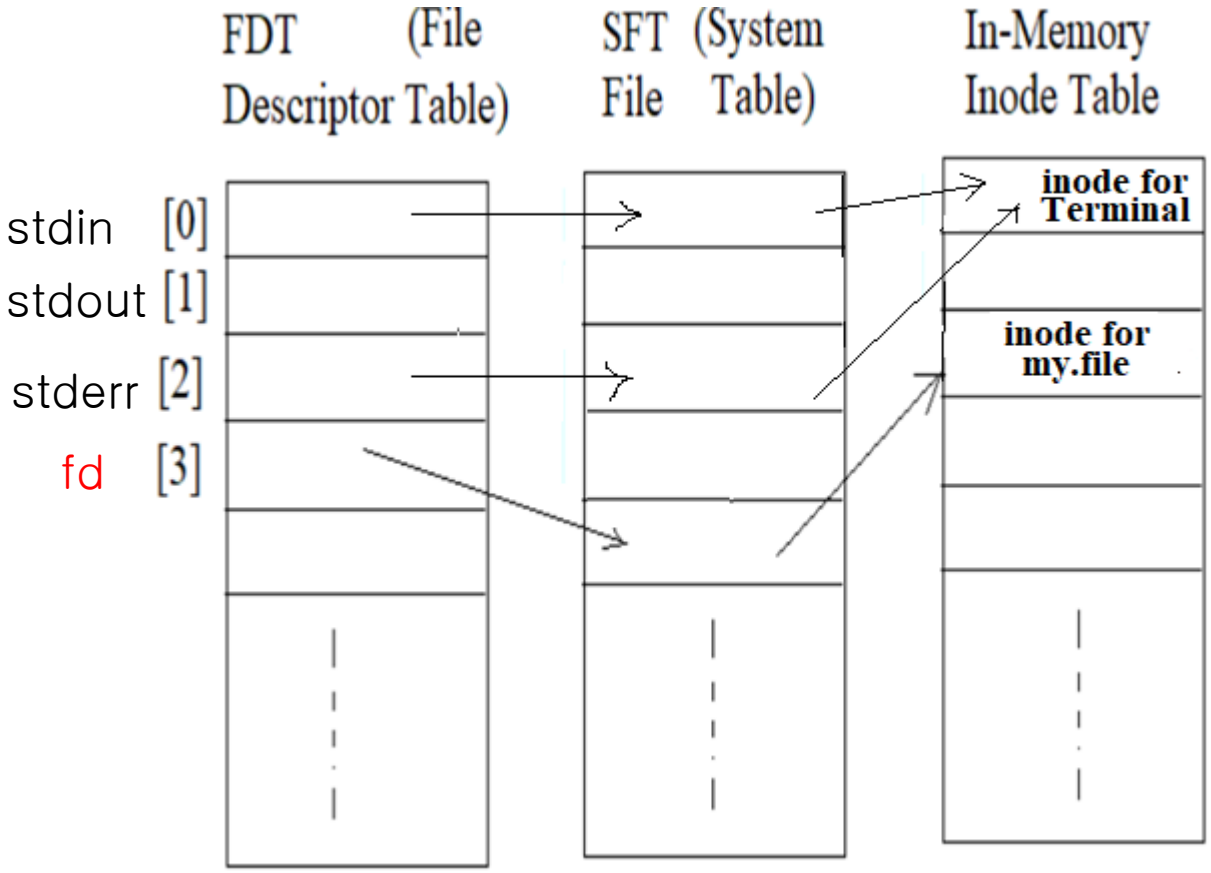
fd = open("my.file", O_CREAT, S_IRUSR| S_IWUSR); /* Create the file – my.file */

close(1); /* Close 1 (Standard output)*/

dup(fd); /* Duplicate fd */

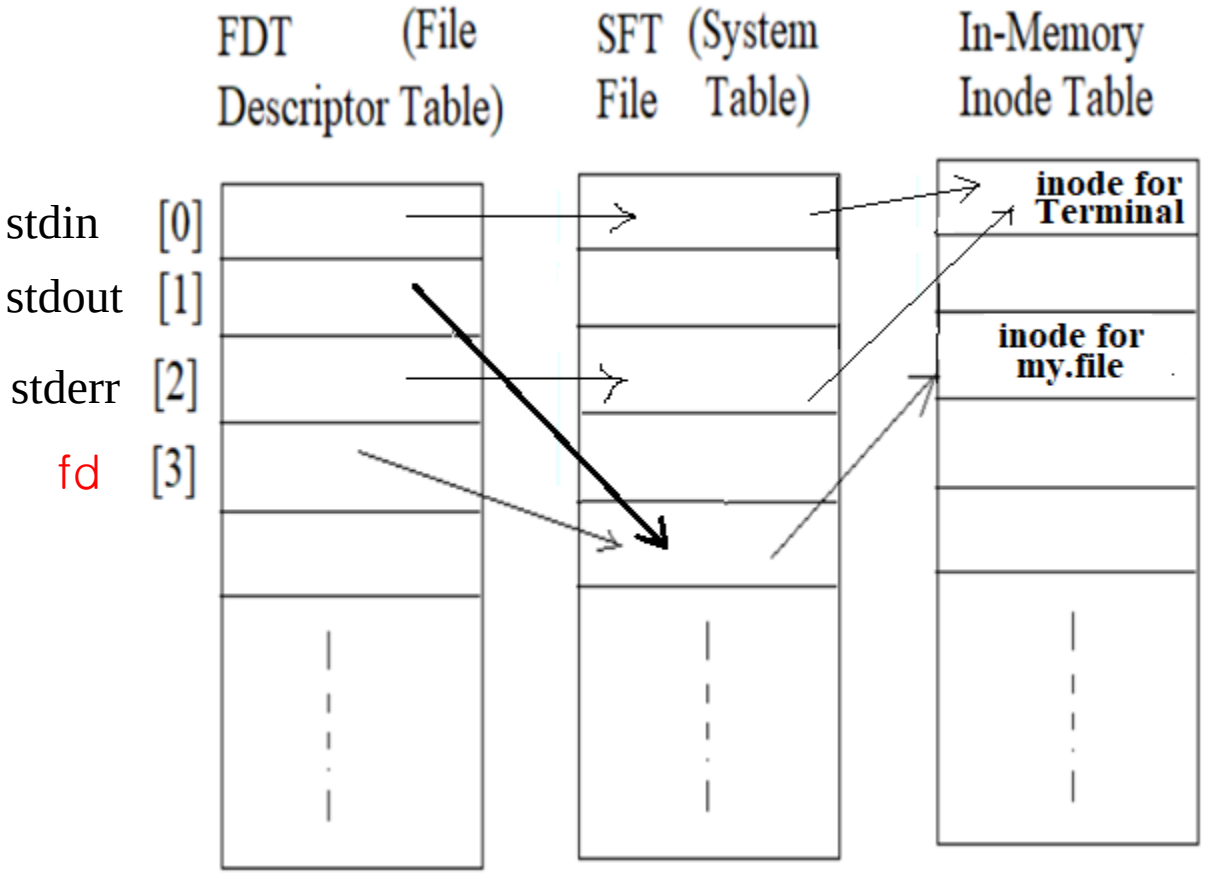
execvp(cmd, argv); /* Execute the command **ls -l** */

I/O redirection (continue)



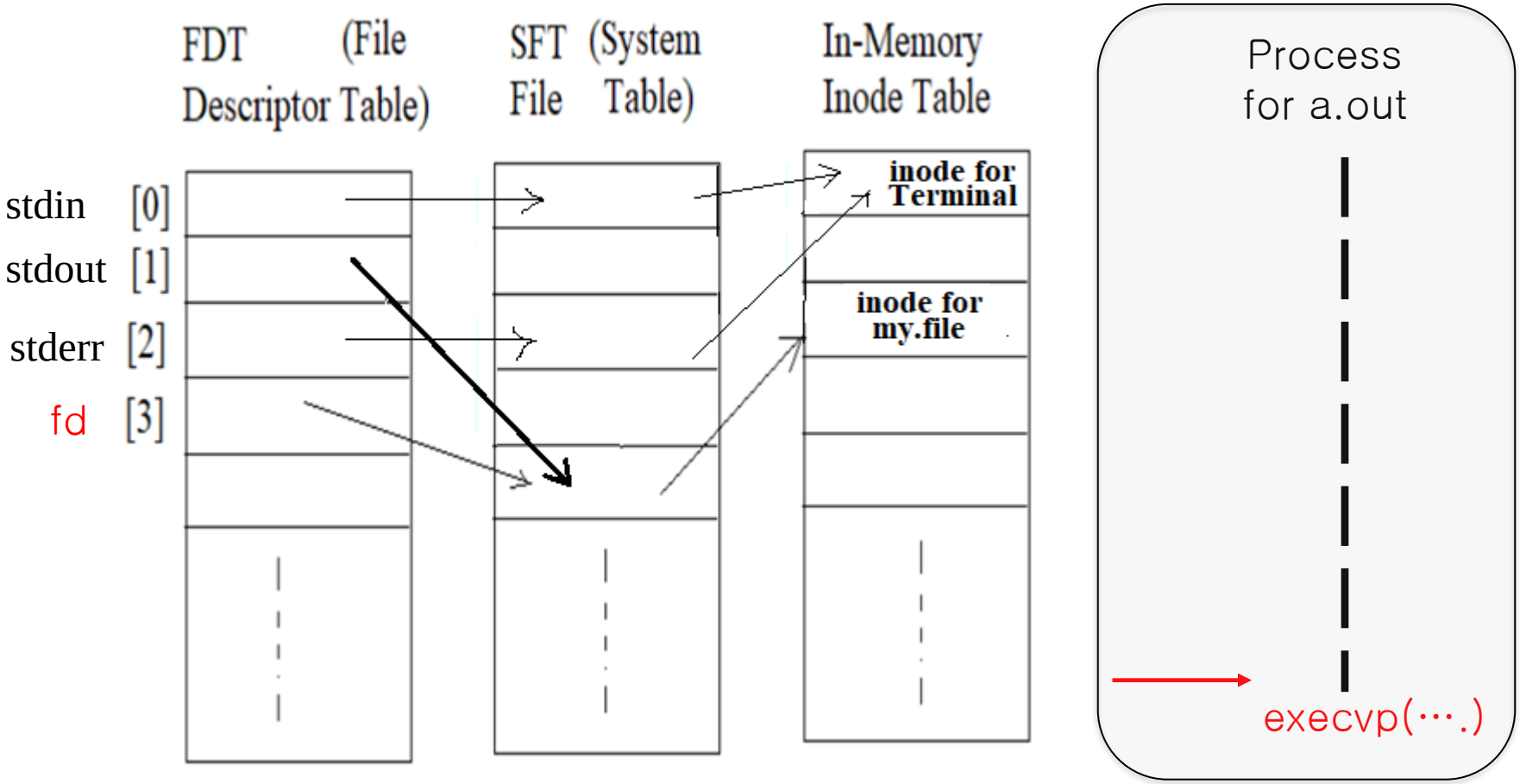

```
fd = open("my.file", O_CREAT, S_IRUSR| S_IWUSR);  /* Create the file – my.file */
close(1);                                          /* Close 1 (Standard output)*/
dup(fd);                                           /* Duplicate fd */
execvp(cmd, argv);                                 /* Execute  the command  ls -l */
```

I/O redirection (continue)



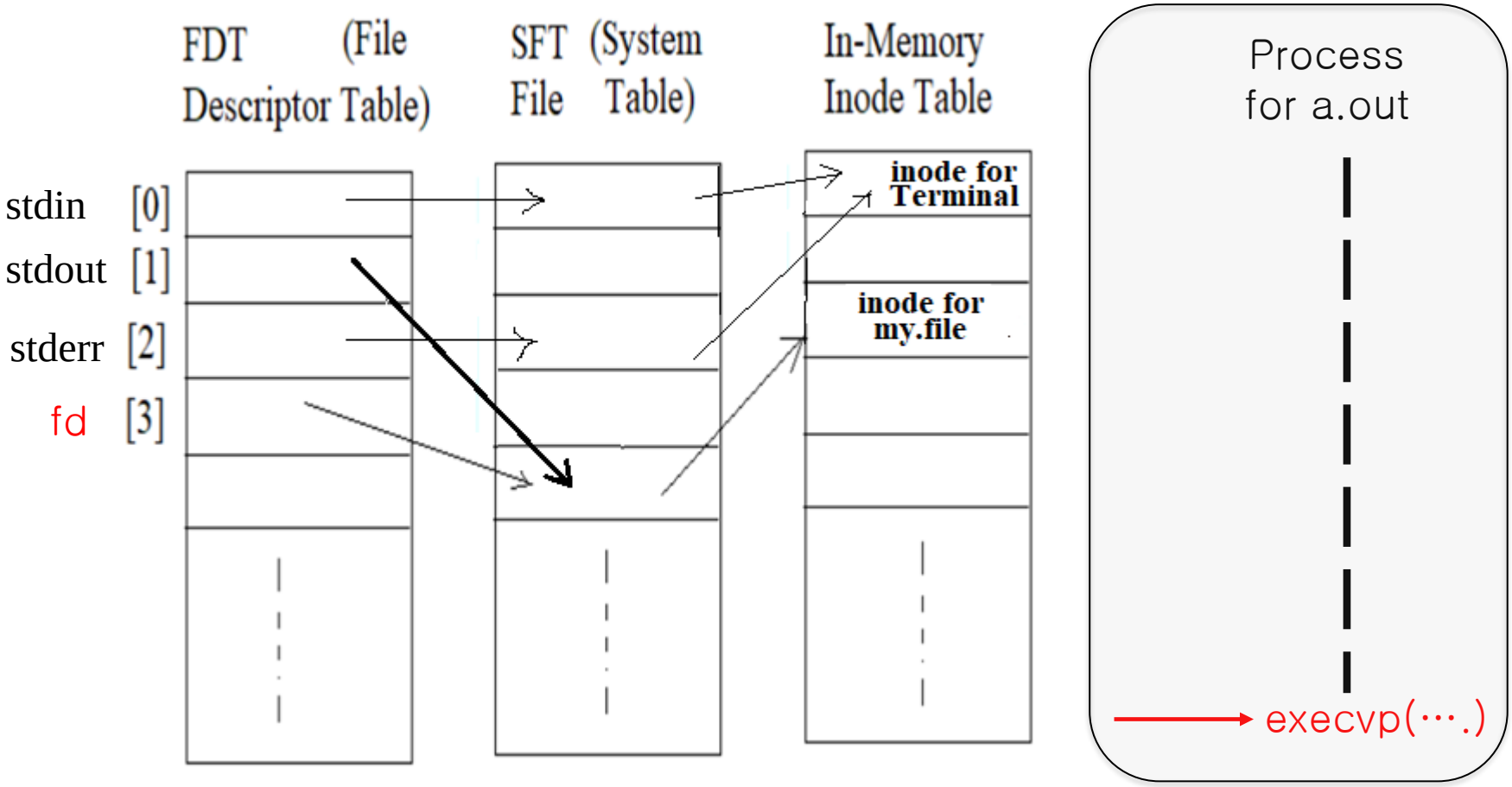

```
fd = open("my.file", O_CREAT, S_IRUSR| S_IWUSR);  /* Create the file – my.file */
close(1);                                         /* Close 1 (Standard output)*/
dup(fd);                                          /* Duplicate fd */
execvp(cmd, argv);                                /* Execute  the command  ls -l */
```

I/O redirection (continue; before execvp(…))



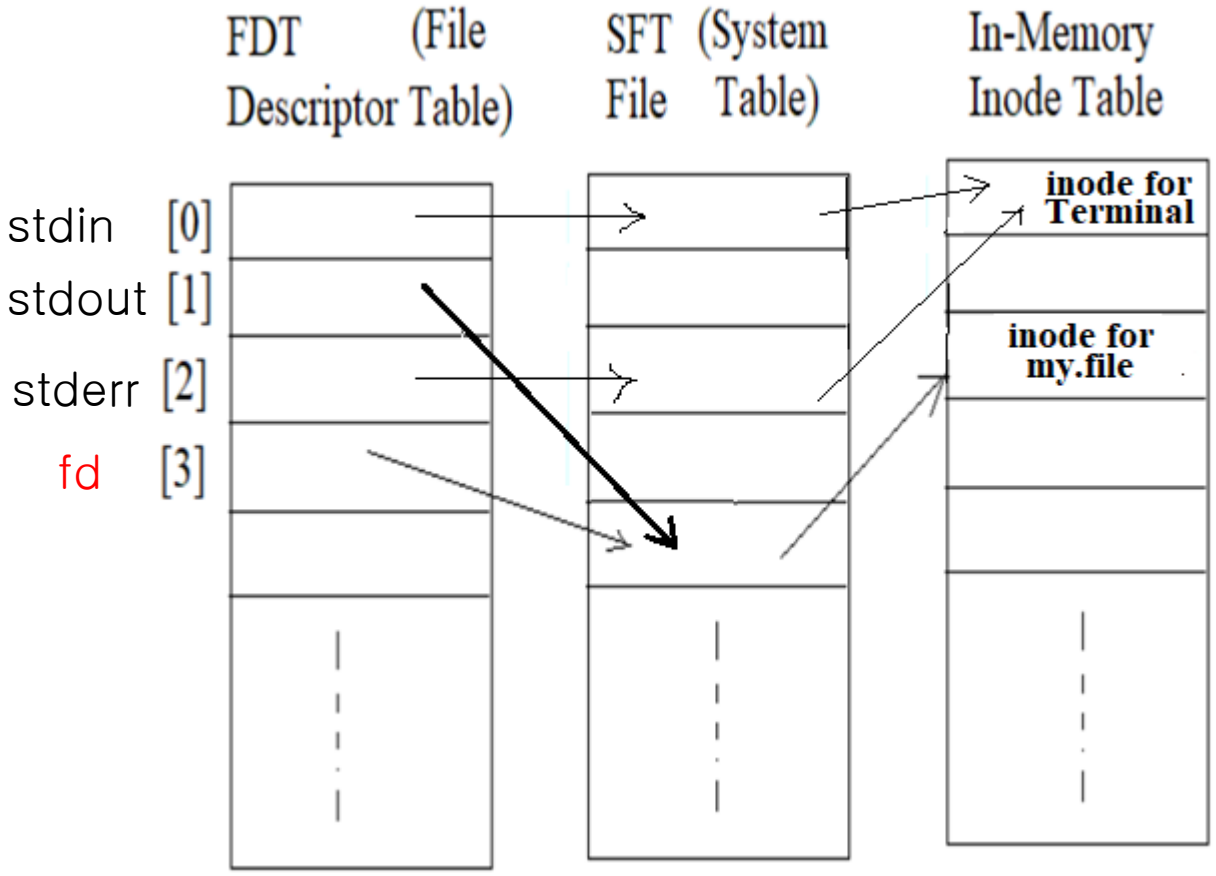

```
fd = open("my.file", O_CREAT, S_IRUSR| S_IWUSR);  /* Create the file – my.file */
close(1);                                          /* Close 1 (Standard output)*/
dup(fd);                                           /* Duplicate fd */
execvp(cmd, argv);                                  /* Execute  the command  ls -l */
```

I/O redirection (continue; executing execvp(…))




```
fd = open("my.file", O_CREAT, S_IRUSR| S_IWUSR);  /* Create the file – my.file */
close(1);                                          /* Close 1 (Standard output)*/
dup(fd);                                           /* Duplicate fd */
execvp(cmd, argv);                                  /* Execute  the command  ls –l */
```

I/O redirection (continue; after execvp(…))



Process for "ls -l" (totally replace the body for Process a.out)

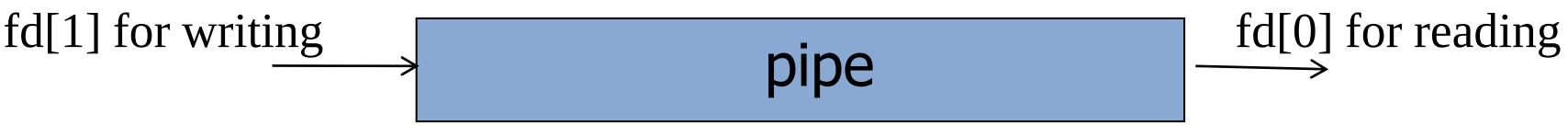
Note that stdout of Process for "ls -l" has been redirected to my.file

```
fd = open("my.file", O_CREAT, S_IRUSR| S_IWUSR);  /* Create the file – my.file */
close(1);                                          /* Close 1 (Standard output)*/
dup(fd);                                           /* Duplicate fd */
execvp(cmd, argv);                                 /* Execute  the command  ls –l */
```

Communication between parent/child processes via pipe

- System call pipe() returns two file descriptors by which we can access the input/output of a pipe (an I/O mechanism)

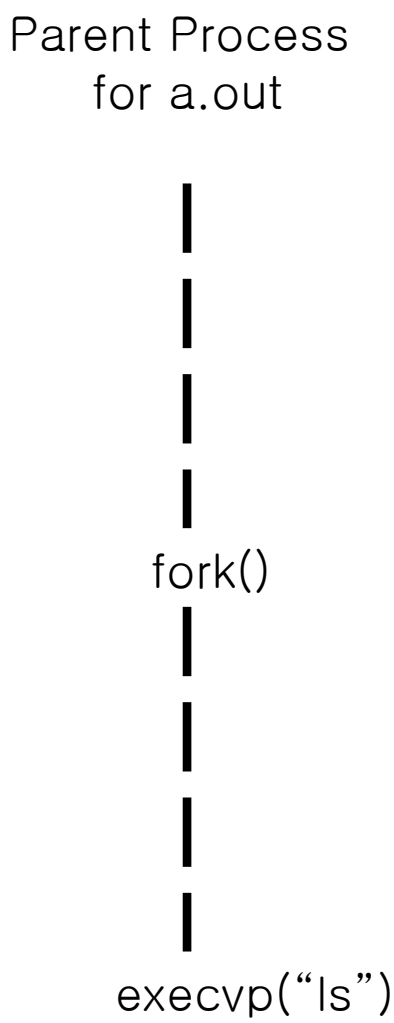
```
int fd[2];

int pipe(int fd[2]);
   return: 0 success; -1 error
```



Question: How to implement "***ls –l | wc -l***"?

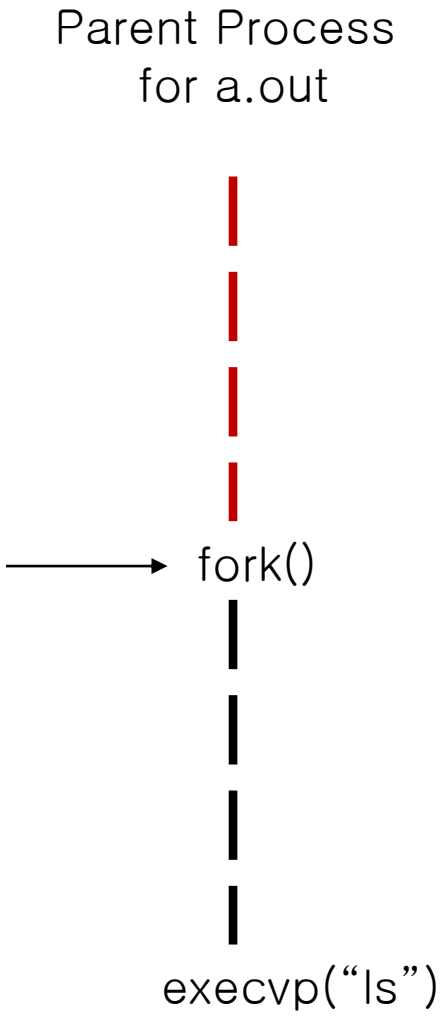
Implement "ls –l | wc –l"

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

main()
{
        int fd[2];
        int ret;
        char *cmd;
        char *argv[3];

        pipe(fd);
        if ( (ret=fork()) > 0 ){
                /* Parent process*/
                close(1);
                dup(fd[1]);   close(fd[0]);  close(fd[1]);

                cmd = "ls";  argv[0] = "ls";  argv[1] = "-l";  argv[2] = NULL;
                execvp(cmd, argv);
        }else if (ret == 0 ){
                /* Child process*/
                close(0);
                dup( fd[0]);  close( fd[0]);  close ( fd[1]);

                cmd = "wc";   argv[0] = "wc";  argv[1] = "-l";   argv[2] = NULL;
                execvp(cmd, argv);
        } else{
                /* Error in fork()*/
                printf("Error occurs when executing fork().\n");
                exit(-1);
        }
}
```



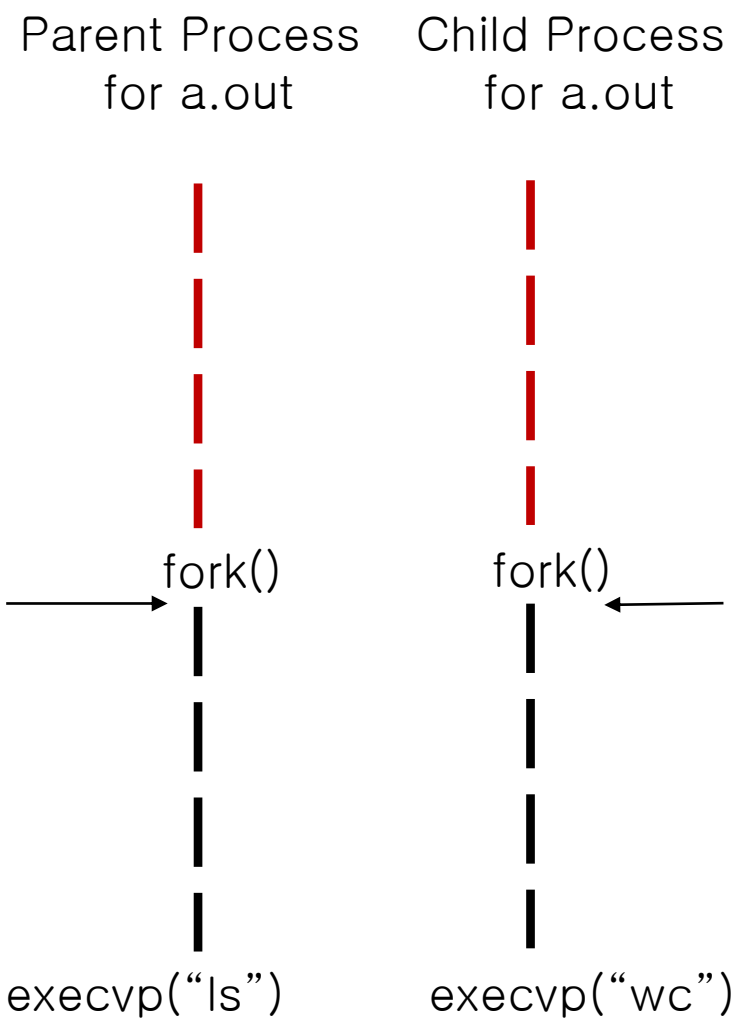
Implement "ls –l | wc –l"

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

main()
{
        int fd[2];
        int ret;
        char *cmd;
        char *argv[3];

        pipe(fd);
        if ( (ret=fork()) > 0 ){
                /* Parent process*/
                close(1);
                dup(fd[1]);   close(fd[0]);  close(fd[1]);

                cmd = "ls";  argv[0] = "ls";  argv[1] = "-l";  argv[2] = NULL;
                execvp(cmd, argv);
        }else if (ret == 0 ){
                /* Child process*/
                close(0);
                dup( fd[0]);  close( fd[0]);  close ( fd[1]);

                cmd = "wc";   argv[0] = "wc";  argv[1] = "-l";   argv[2] = NULL;
                execvp(cmd, argv);
        } else{
                /* Error in fork()*/
                printf("Error occurs when executing fork().\n");
                exit(-1);
        }
}
```



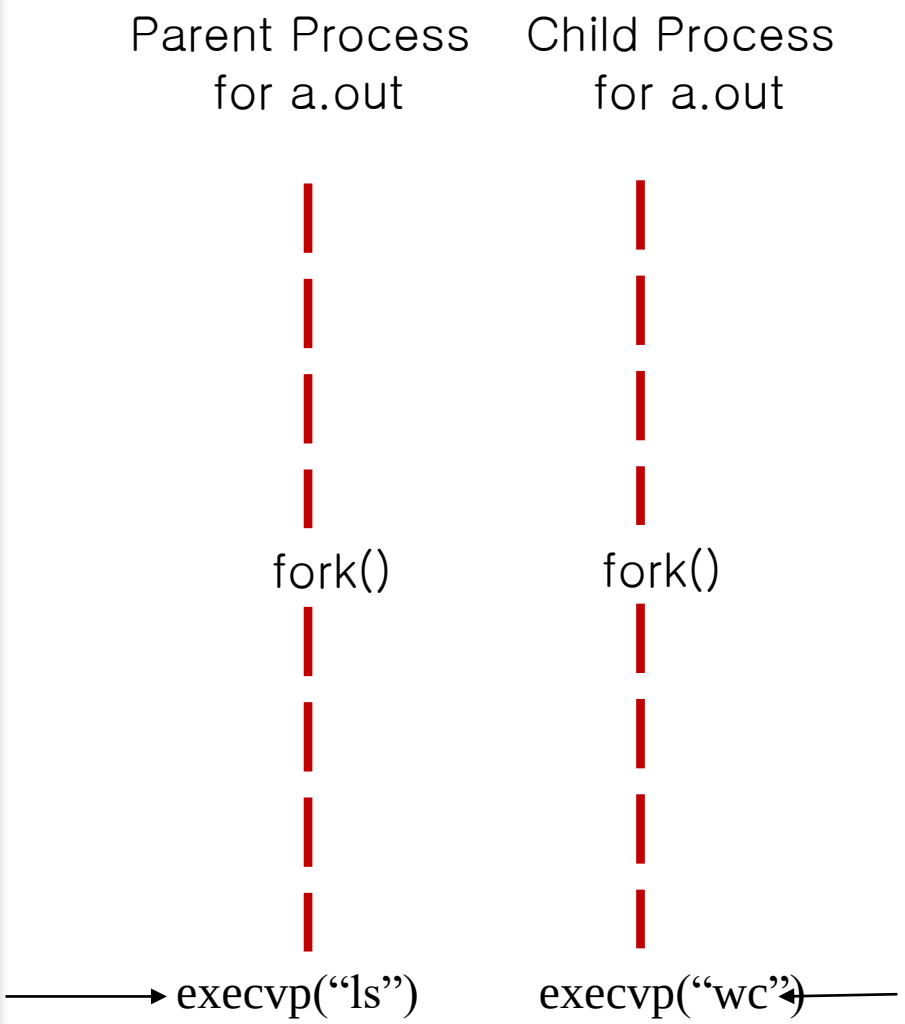
Implement "ls –l | wc –l"

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

main()
{
        int fd[2];
        int ret;
        char *cmd;
        char *argv[3];

        pipe(fd);
        if ( (ret=fork()) > 0 ){
                /* Parent process*/
                close(1);
                dup(fd[1]);   close(fd[0]);  close(fd[1]);

                cmd = "ls";  argv[0] = "ls";  argv[1] = "-l";  argv[2] = NULL;
                execvp(cmd, argv);
        }else if (ret == 0 ){
                /* Child process*/
                close(0);
                dup( fd[0]);  close( fd[0]);  close ( fd[1]);

                cmd = "wc";   argv[0] = "wc";  argv[1] = "-l";   argv[2] = NULL;
                execvp(cmd, argv);
        } else{
                /* Error in fork()*/
                printf("Error occurs when executing fork().\n");
                exit(-1);
        }
}
```



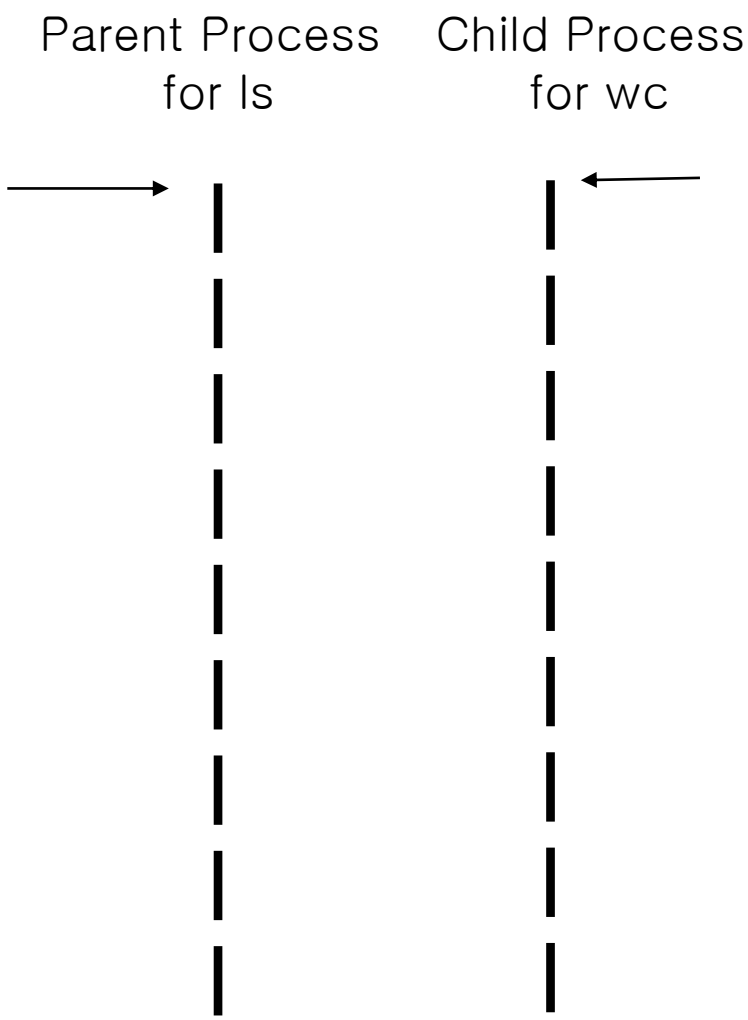
Implement "ls –l | wc –l"

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

main()
{
        int fd[2];
        int ret;
        char *cmd;
        char *argv[3];

        pipe(fd);
        if ( (ret=fork()) > 0 ){
                /* Parent process*/
                close(1);
                dup(fd[1]);   close(fd[0]);  close(fd[1]);

                cmd = "ls";  argv[0] = "ls";  argv[1] = "-l";  argv[2] = NULL;
                execvp(cmd, argv);
        }else if (ret == 0 ){
                /* Child process*/
                close(0);
                dup( fd[0]);  close( fd[0]);  close ( fd[1]);

                cmd = "wc";   argv[0] = "wc";  argv[1] = "-l";   argv[2] = NULL;
                execvp(cmd, argv);
        } else{
                /* Error in fork()*/
                printf("Error occurs when executing fork().\n");
                exit(-1);
        }
}
```



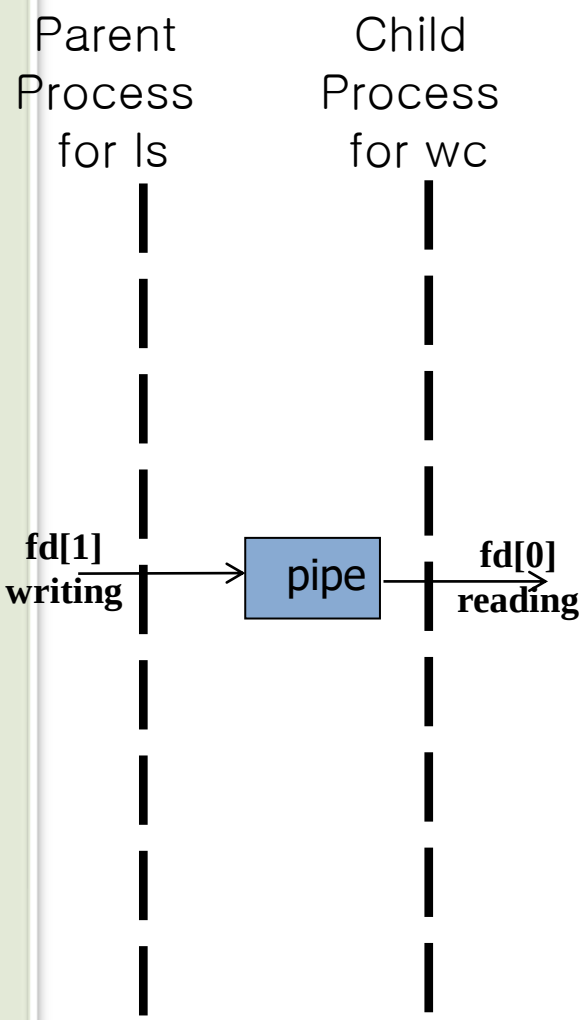
Implement "ls –l | wc –l"

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

main()
{
        int fd[2];
        int ret;
        char *cmd;
        char *argv[3];

        pipe(fd);
        if ( (ret=fork()) > 0 ){
                /* Parent process*/
                close(1);
                dup(fd[1]);   close(fd[0]);  close(fd[1]);

                cmd = "ls";  argv[0] = "ls";  argv[1] = "-l";  argv[2] = NULL;
                execvp(cmd, argv);
        }else if (ret == 0 ){
                /* Child process*/
                close(0);
                dup( fd[0]);  close( fd[0]);  close ( fd[1]);

                cmd = "wc";   argv[0] = "wc";  argv[1] = "-l";   argv[2] = NULL;
                execvp(cmd, argv);
        } else{
                /* Error in fork()*/
                printf("Error occurs when executing fork().\n");
                exit(-1);
        }
}
```



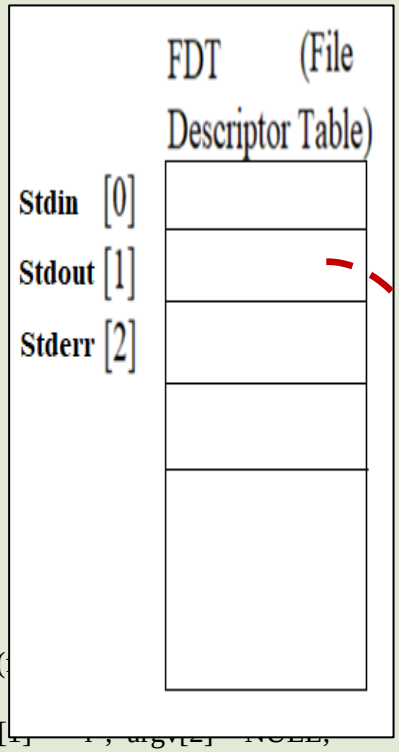
Implement "ls –l | wc –l"

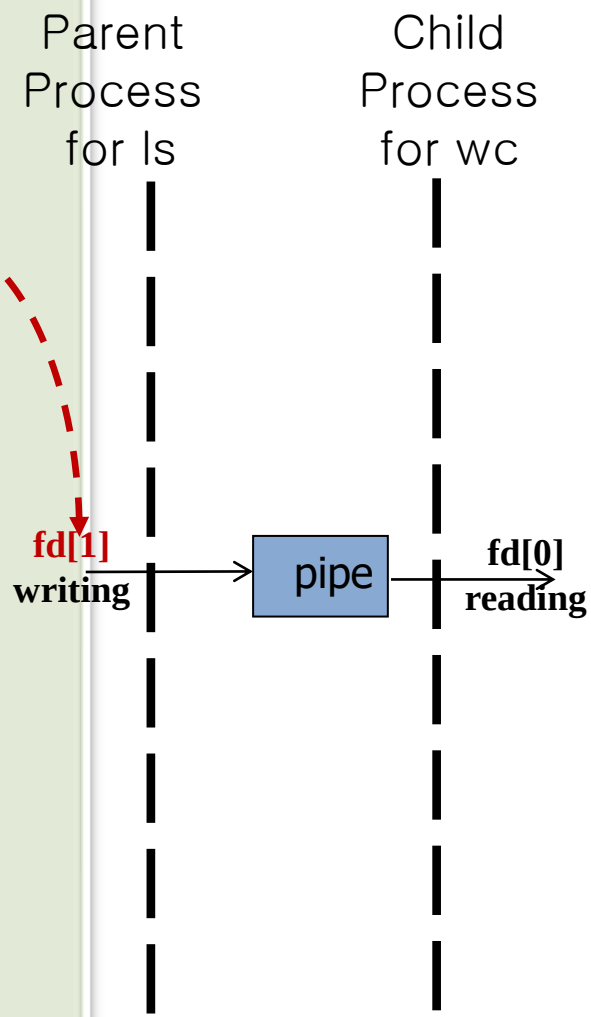
```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>

main()
{
        int fd[2];
        int ret;
        char *cmd;
        char *argv[3];

        pipe(fd);
        if ( (ret=fork()) > 0 ){
                /* Parent process*/
                close(1);
                dup(fd[1]);   close(fd[0]);  close(fd[1]);

                cmd = "ls";  argv[0] = "ls";  argv[1] = "-l";  argv[2] = NULL;
                execvp(cmd, argv);
        }else if (ret == 0 ){
                /* Child process*/
                close(0);
                dup( fd[0]);  close( fd[0]);  close ( fd[1]);

                cmd = "wc";   argv[0] = "wc";  argv[1] = "-l";   argv[2] = NULL;
                execvp(cmd, argv);
        } else{
                /* Error in fork()*/
                printf("Error occurs when executing fork().\n");
                exit(-1);
        }
}
```



Implement "ls –l | wc –l"

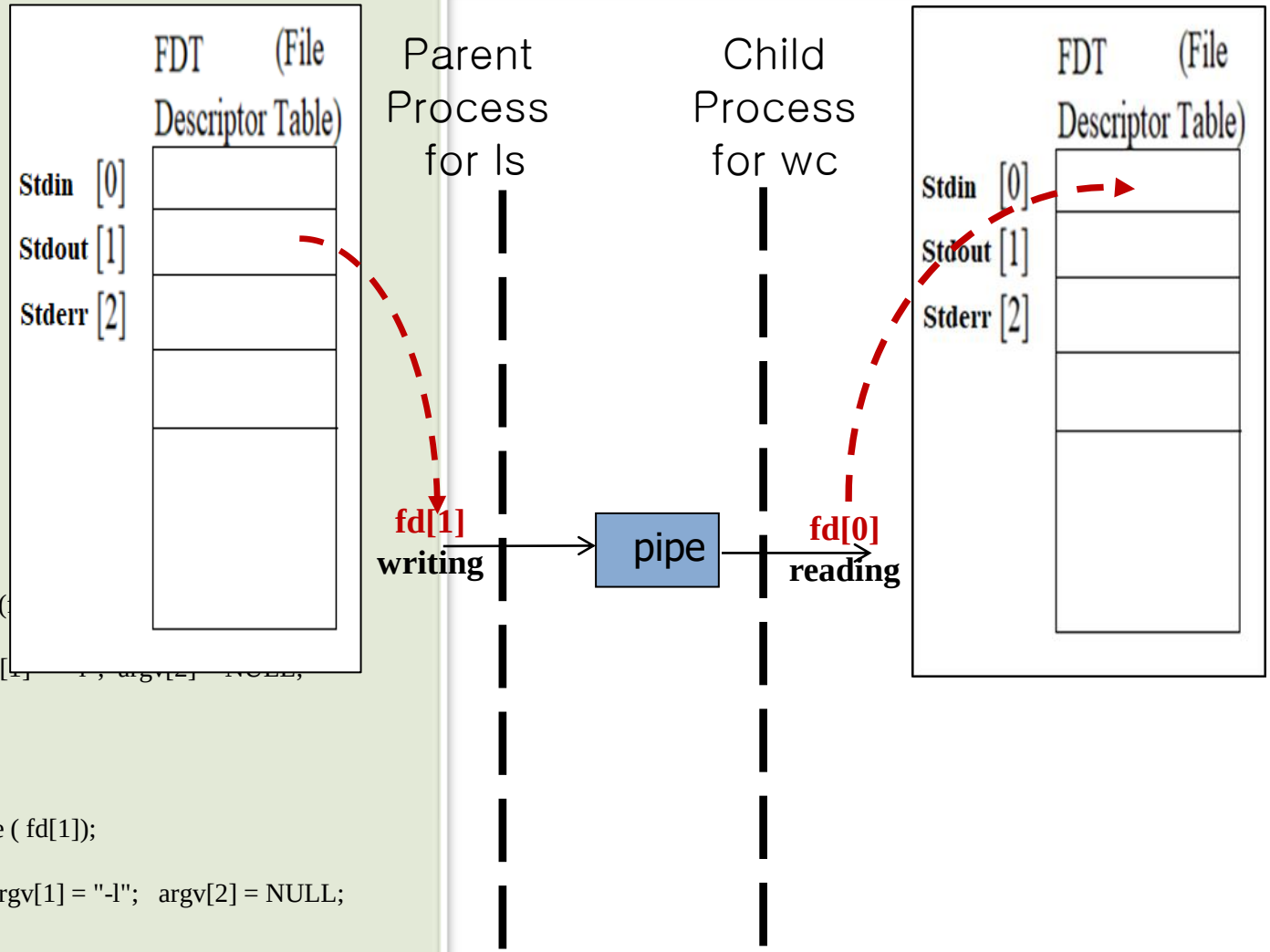
```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>

main()
{
        int fd[2];
        int ret;
        char *cmd;
        char *argv[3];

        pipe(fd);
        if ( (ret=fork()) > 0 ){
                /* Parent process*/
                close(1);
                dup(fd[1]);   close(fd[0]);  close(
                cmd = "ls";  argv[0] = "ls";  argv[
                execvp(cmd, argv);
        }else if (ret == 0 ){
                /* Child process*/
                close(0);
                dup( fd[0]);  close( fd[0]);  close ( fd[1]);

                cmd = "wc";   argv[0] = "wc";  argv[1] = "-l";   argv[2] = NULL;
                execvp(cmd, argv);
        } else{
                /* Error in fork()*/
                printf("Error occurs when executing fork().\n");
                exit(-1);
        }
}
```





Implement "ls –l | wc –l"

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

main()
{
        int fd[2];
        int ret;
        char *cmd;
        char *argv[3];

        pipe(fd);
        if ( (ret=fork()) > 0 ){
                /* Parent process*/
                close(1);
                dup(fd[1]);   close(fd[0]);  close(
                cmd = "ls";  argv[0] = "ls";  argv[
                execvp(cmd, argv);
        }else if (ret == 0 ){
                /* Child process*/
                close(0);
                dup( fd[0]);  close( fd[0]);  close ( fd[1]);

                cmd = "wc";   argv[0] = "wc";  argv[1] = "-l";   argv[2] = NULL;
                execvp(cmd, argv);
        } else{
                /* Error in fork()*/
                printf("Error occurs when executing fork().\n");
                exit(-1);
        }
}
```



More about Pipe (https://man7.org/linux/man-pages/man7/pipe.7.html)

- Pipes provide a unidirectional interprocess communication channel. A pipe has a read end and a write end. Data written to the write end of a pipe can be read from the read end of the pipe.
- If a process attempts to read from an empty pipe, then read() will block until data is available.
- The communication channel provided by a pipe is a byte stream: there is no concept of message boundaries.
- If all file descriptors referring to the write end of a pipe have been closed, then an attempt to read() from the pipe will see end-of-file (read() will return 0). If all file descriptors referring to the read end of a pipe have been closed, then a write() will cause a SIGPIPE signal to be generated for the calling process. If the calling process is ignoring this signal, then write() fails with the error EPIPE.

Writing Immediately with `fsync()`

- The file system will **buffer** writes in memory for some time (for performance reasons), e.g. 5 or 30 seconds.
- At that later point in time, the write(s) will **actually be issued** to the storage device.
  - Writes seem to complete quickly; but data can be lost (e.g., machine crashes).
  - However, some applications require persistence guarantee, e.g. DBMS requires force writes to disk from time to time (your bank transactions).

```
int fsync(int fd)
```

- force all dirty (i.e., not yet written) data written to disk
- fsync() returns once all of theses writes are complete.

Writing Immediately with fsync() (Cont.)

```
#include <unistd.h>

int fsync(int fd)
```

On success, these system calls return zero. On error, -1 is returned, and errno is set appropriately.

- An Example of `fsync()`.

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
assert (fd > -1)
int rc = write(fd, buffer, size);
assert (rc == size);
rc = fsync(fd);
assert (rc == 0);
```

  - In some cases, this code needs to `fsync()` the directory that contains the file `foo`.

Renaming Files

- `rename(char* old, char *new)`
  - Rename a file to different name.
  - It implemented as an **atomic call**.
    - e.g. Change from `foo` to `bar`:

```
prompt> mv foo bar          // mv uses the system call rename()
```

    - How to update a file atomically:

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC, S_IRWXU);
char buffer[20]= "hello";
write(fd, buffer, sizeof(buffer)); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

Getting Information About Files

- `stat()`, `fstat()`: Show the file metadata
  - **Metadata** is information about each file.
    - Size, Low-level name, Permission, …
  - `stat` structure is below:

```
struct stat {
    dev_t st_dev;         /* ID of device containing file */
    ino_t st_ino;         /* inode number */
    mode_t st_mode;       /* protection */
    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 ID (if special file) */
    off_t st_size;        /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t st_blocks;   /* number of blocks allocated */
    time_t st_atime;      /* time of last access */
    time_t st_mtime;      /* time of last modification */
    time_t st_ctime;      /* time of last status change */
};
```

Getting Information About Files (Cont.)

- To see stat information, you can use the command line tool `stat`.

```
prompt> echo hello > file
prompt> stat file

File: 'file'
Size: 6 Blocks: 8 IO Block: 4096 regular file
Device: 811h/2065d Inode: 67158084 Links: 1
Access: (0640/-rw-r-----) Uid: (30686/ root) Gid: (30686/ remzi)
Access: 2011-05-03 15:50:20.157594748 -0500
Modify: 2011-05-03 15:50:20.157594748 -0500
Change: 2011-05-03 15:50:20.157594748 -0500
```

  - File system keeps this type of information in an `inode` structure.

Removing Files

- `rm` is Linux command to remove a file
  - `rm` call `unlink()` to remove a file.

```
prompt> strace rm foo
…
unlink("foo")                = 0      // return 0 upon success
…
prompt>
```

Why it calls `unlink()` not "`remove` or `delete`"?
We can get the answer later.

Making Directories

- `mkdir()`: Make a directory

```
prompt> strace mkdir foo
…
mkdir("foo", 0777)                      = 0
prompt>
```

- When a directory is created, it is empty.
- Empty directory have two entries: . (itself), .. (parent)

```
prompt> ls -a
./      ../
prompt> ls -al
total 8
drwxr-x---  2 remzi remzi    6 Apr 30 16:17 ./
drwxr-x--- 26 remzi remzi 4096 Apr 30 16:17 ../
```

Reading Directories

- A sample code to read directory entries (like `ls`).

```
#include <sys/stat.h>
#include <assert.h>
#include <stdio.h>
#include <dirent.h>

int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");              // open current directory
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL)    // read one directory entry
    {
        // print outthe name and inode number of each file
        printf("%d %s\n", (int) d->d_ino, d->d_name);
    }
    closedir(dp);                        // close current directory
    return 0;
}
```

  - The information available within `struct dirent`

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

Deleting Directories

- `rmdir()`: Delete a directory.
  - Require that the directory be **empty**
    - i.e. Only has "." and ".." entries.
  - If you call `rmdir()` to a non-empty directory, it will fail.

Hard Links

- `link(old pathname, new one)`
  - **Link** a new file name to an old one
  - Create another way to refer to *the same file*
  - The command-line link program : `ln`

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2  // create a hard link, link file to file2
prompt> cat file2
hello
```

Hard Links (Cont.)

- The way `link` works:
  - **Create** another name in the directory.
  - **Refer** it to the same inode number of the original file.
    - The file is not copied in any way.
  - Then, we now just have two human names (`file` and `file2`) that both refer to the same file.

Hard Links (Cont.)

- The result of `link()`

```
prompt> ls -i file file2
67158084 file  /* inode value is 67158084 */
67158084 file2 /* inode value is 67158084 */
prompt>
```

  - Two files have **same inode** number, but two file name (file, file2).
  - There is **no difference** between file and file2.
    - Both just link to the underlying metadata about the file.

Hard Links (Cont.)

- Thus, to remove a file, we call `unlink()`.

```
prompt> rm file
removed 'file'
prompt> cat file2          // Still access the file
hello
```

  - ***reference count***
    - Track how many different file names have been linked to this inode.
    - When `unlink()` is called, the reference count decrements.
    - If the reference count reaches zero, the files system free the inode and related data blocks.

      → truly "delete" the file

Hard Links (Cont.)

- The result of `unlink()`
  - `stat()` shows the reference count of a file.

```
prompt> echo hello > file            /* create file*/
prompt> stat file
... Inode: 67158084 Links: 1 ...     /* Link count is 1 */
prompt> ln file file2                /* hard link file2 */
prompt> stat file
... Inode: 67158084 Links: 2 ...     /* Link count is 2 */
prompt> stat file2
... Inode: 67158084 Links: 2 ...     /* Link count is 2 */
prompt> ln file2 file3               /* hard link file3 */
prompt> stat file
... Inode: 67158084 Links: 3 ...     /* Link count is 3 */
prompt> rm file                      /* remove file */
prompt> stat file2
... Inode: 67158084 Links: 2 ...     /* Link count is 2 */
prompt> rm file2                     /* remove file2 */
prompt> stat file3
... Inode: 67158084 Links: 1 ...     /* Link count is 1 */
prompt> rm file3
```

Symbolic Links (Soft Link)

- Symbolic link is more **useful** than Hard link.
  - Hard Link cannot be created for a directory.
  - Hard Link cannot create to a file to other partition.
    - Because inode numbers are only unique within a file system.

- Create a symbolic link: `ln -s`

```
prompt> echo hello > file
prompt> ln -s file file2  /* option -s : create a symbolic link, */
prompt> cat file2
hello
```

Symbolic Links (Cont.)

- What is different between *Symbolic link* and *Hard Link*?
  - Symbolic links are **a third type** the file system knows about.

```
prompt> stat file
  ... regular file ...
prompt> stat file2
  ... symbolic link ...        // Actually a file itself of a different type
```

  - The size of symbolic link (`file2`) is **4 bytes**.

```
prompt> ls -al
drwxr-x---  2 remzi remzi   29 May 3 19:10 ./
drwxr-x--- 27 remzi remzi 4096 May 3 15:14 ../              // directory
-rw-r-----  1 remzi remzi    6 May 3 19:10 file             // regular file
lrwxrwxrwx  1 remzi remzi    4 May 3 19:10 file2 -> file    // symbolic link
```

    - A symbolic link holds the pathname of the linked-to file as the data of the link file.

Symbolic Links (Cont.)

- If we link to a longer pathname, our link file would be bigger.

```
prompt> echo hello > alongerfilename
prompt> ln -s alongerfilename file3
prompt> ls -al alongerfilename file3
-rw-r----- 1 remzi remzi  6 May 3 19:17 alongerfilename
lrwxrwxrwx 1 remzi remzi 15 May 3 19:17 file3 -> alongerfilename
```

Symbolic Links (Cont.)

- **Dangling reference**
  - When remove an original file, symbolic link points nothing.

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt> rm file            // remove the original file
prompt> cat file2
cat: file2: No such file or directory
```

Summary

- System calls related to file & directory
  - open(), read(), write(), lseek()
  - dup(), pipe(), fsync(), rename(), stat()
  - mkdir(), rmdir(), opendir(), readdir()
  - unlink(), hard/symbolic link
- Next: File System Implementation
  - Chapter 40