CS415: Systems Programming

File related System Calls

Most of the slides in this lecture are either from or adapted from the slides provided by Dr. Ahmad Barghash

Remember

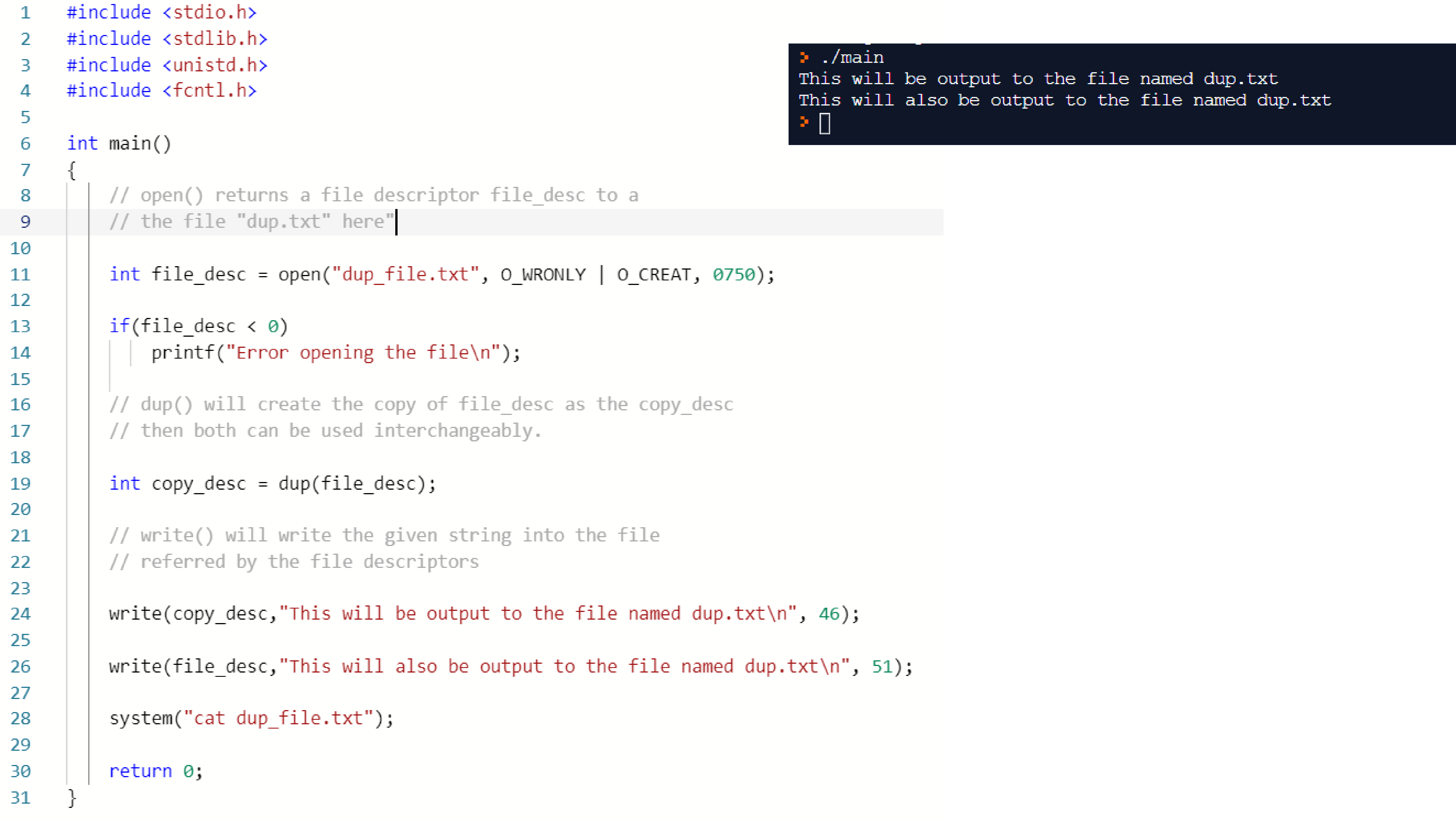
UNIX File I/O: Performed mostly using 7 commands

- open
- close
- read
- write
- lseek
- dup, dup2

Covered till now!

dup System Call

- `int dup(int oldfd);`
  - oldfd: old file descriptor whose copy is to be created.
  - Returns a new file descriptor.
- The `dup`() system call creates a copy of a file descriptor.
  - It uses **the lowest-numbered unused** descriptor for the new descriptor.
  - If the copy is successfully created, then the original and copy file descriptors may be used interchangeably.
  - They both refer to the same open file description and thus share file offset and file status flags.
- Include the header file `unistd.h` for using the `dup`() system call.



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main()
{
    // open() returns a file descriptor file_desc to a
    // the file "dup.txt" here"

    int file_desc = open("dup_file.txt", O_WRONLY | O_CREAT, 0750);

    if(file_desc < 0)
        printf("Error opening the file\n");

    // dup() will create the copy of file_desc as the copy_desc
    // then both can be used interchangeably.

    int copy_desc = dup(file_desc);

    // write() will write the given string into the file
    // referred by the file descriptors

    write(copy_desc,"This will be output to the file named dup.txt\n", 46);

    write(file_desc,"This will also be output to the file named dup.txt\n", 51);

    system("cat dup_file.txt");

    return 0;
}
```

```
> ./main
This will be output to the file named dup.txt
This will also be output to the file named dup.txt
> 
```

dup2 System Call

- `int dup2(int oldfd, int newfd);`
  - oldfd: an old file descriptor
  - newfd: the new file descriptor which is used by `dup2`() to create a copy.

- The `dup2`() system call is similar to `dup`() but the basic difference between them is that instead of using the lowest-numbered unused file descriptor, it uses the descriptor number specified by the program.
  - If the descriptor newfd was previously open, it is silently closed before being reused.
  - If oldfd is not a valid file descriptor, then the call fails, and newfd is not closed.
  - If oldfd is a valid file descriptor, and newfd has the same value as oldfd, then dup2() does nothing, and returns newfd.
- Include the header file `unistd.h` for using `dup2`() system call.

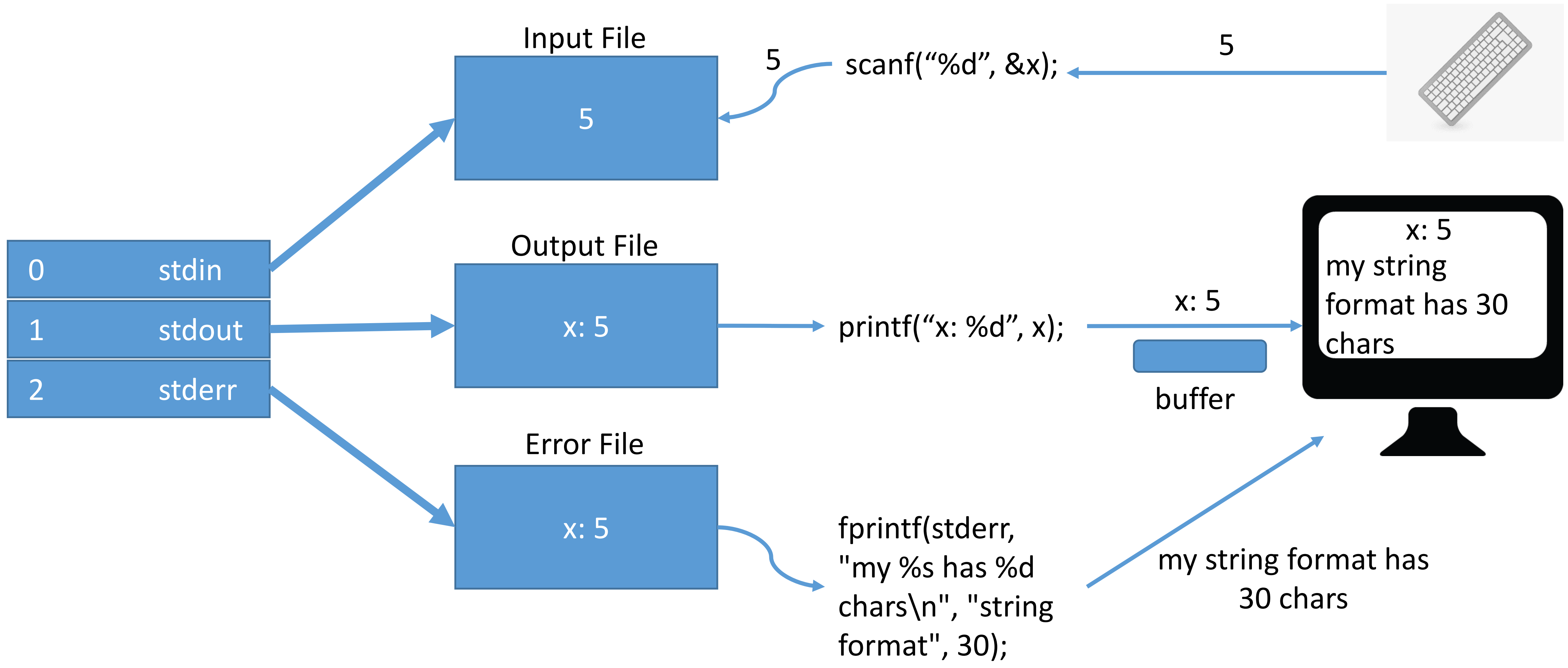
File Descriptor

- In UNIX, a file descriptor (FD) is an abstract indicator (handle) used to access a file or other input/output resource. A file descriptor is a `non-negative integer`, generally represented in the C programming language as the type int (negative values being reserved to indicate "no value" or an error condition).

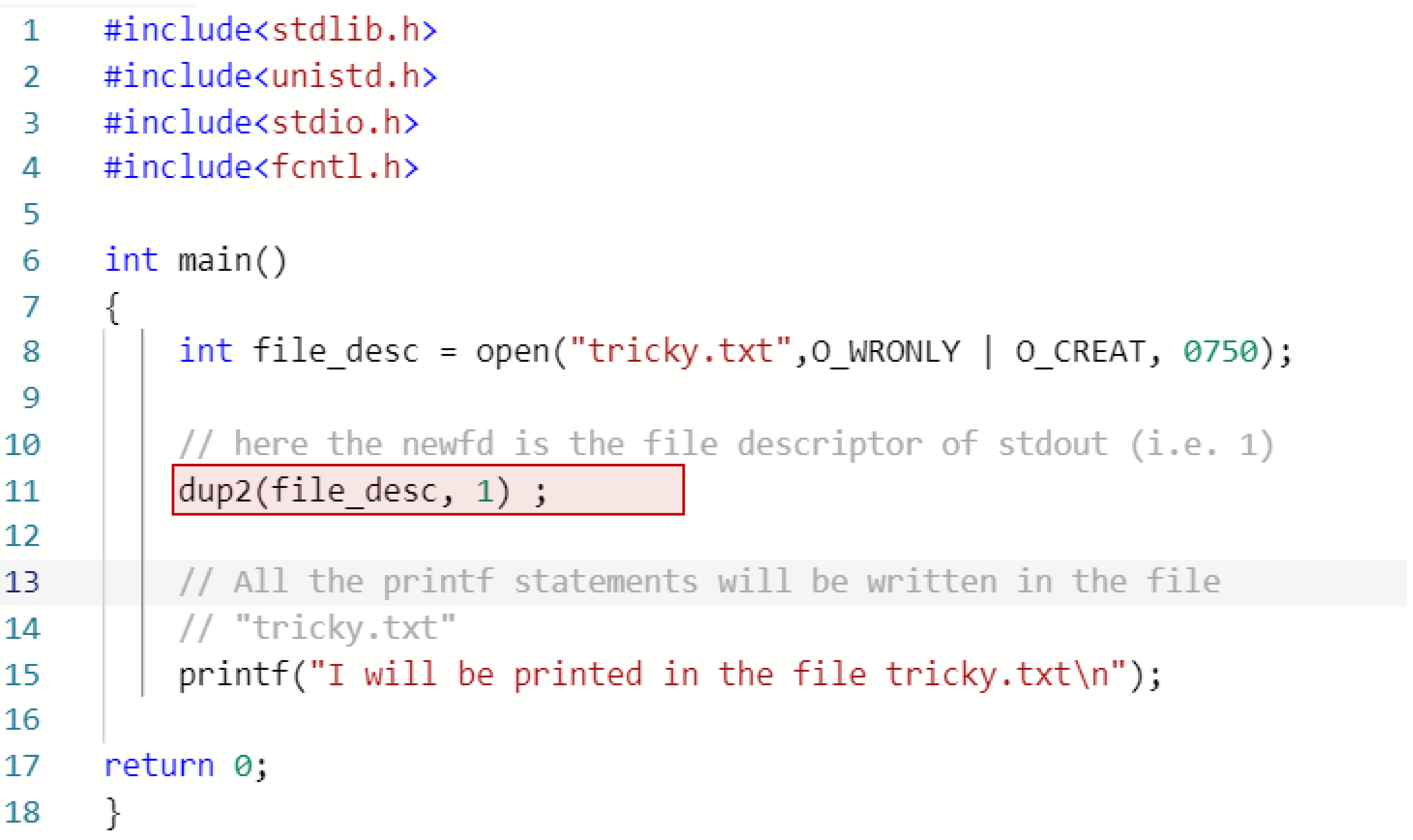
- Each UNIX process (except perhaps a daemon) should expect to have three standard file descriptors, corresponding to the three standard streams:

Integer Value	Name	File Stream
0	Standard Input	stdin
1	Standard Output	stdout
2	Standard Error	stderr

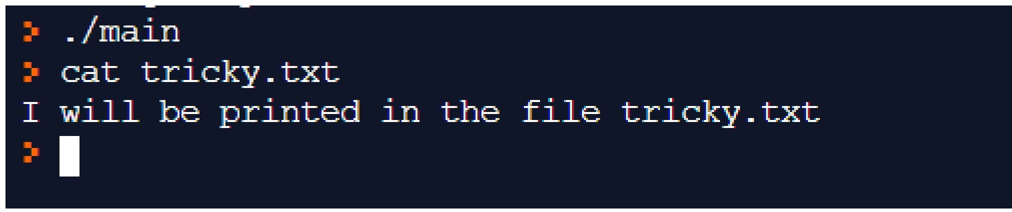
Each process has a default number of open files: **stdin**, **stdout**, and **stderr**



dup2 System Call - Example

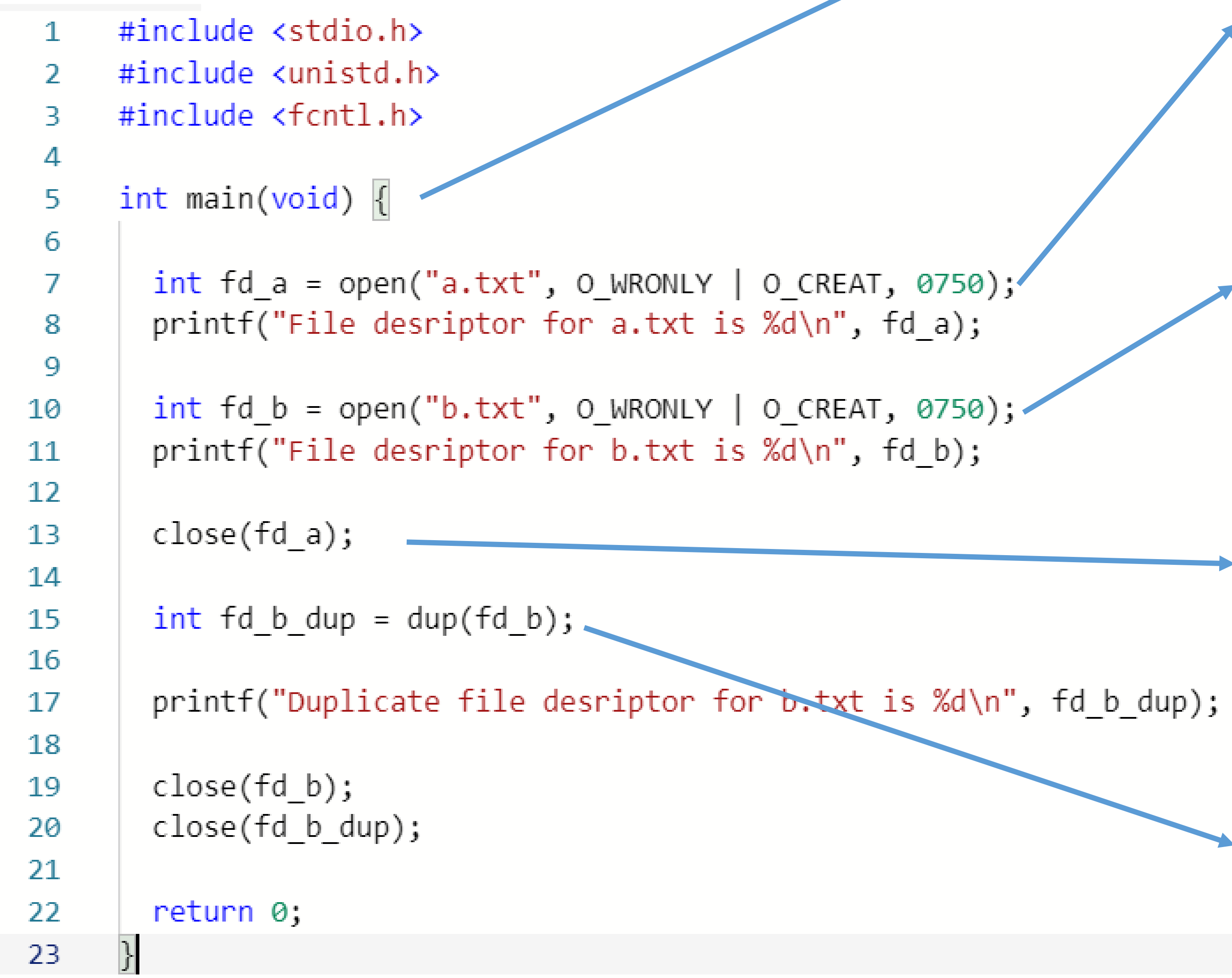


```c
#include<stdlib.h>
#include<unistd.h>
#include<stdio.h>
#include<fcntl.h>

int main()
{
    int file_desc = open("tricky.txt",O_WRONLY | O_CREAT, 0750);

    // here the newfd is the file descriptor of stdout (i.e. 1)
    dup2(file_desc, 1) ;

    // All the printf statements will be written in the file
    // "tricky.txt"
    printf("I will be printed in the file tricky.txt\n");

return 0;
}
```



```
❯ ./main
❯ cat tricky.txt
I will be printed in the file tricky.txt
❯ 
```

A tricky use of dup2() system call: As in dup2(), in place of newfd any file descriptor can be put. Above is a C implementation in which the file descriptor of Standard output (stdout) is used. This will lead all the printf() statements to be written in the file referred by the old file descriptor.



```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main(void) {

  int fd_a = open("a.txt", O_WRONLY | O_CREAT, 0750);
  printf("File desriptor for a.txt is %d\n", fd_a);

  int fd_b = open("b.txt", O_WRONLY | O_CREAT, 0750);
  printf("File desriptor for b.txt is %d\n", fd_b);

  close(fd_a);

  int fd_b_dup = dup(fd_b);

  printf("Duplicate file desriptor for b.txt is %d\n", fd_b_dup);

  close(fd_b);
  close(fd_b_dup);

  return 0;
}
```

File Desc	Refers to
0	stdin
1	stdout
2	stderr

File Desc	Refers to
0	Stdin
1	Stdout
2	stderr
3	a.txt

File Desc	Refers to
0	stdin
1	stdout
2	stderr
3	a.Txt
4	b.txt

File Desc	Refers to
0	stdin
1	stdout
2	stderr
3	
4	b.txt

File Desc	Refers to
0	stdin
1	stdout
2	stderr
3	b.txt
4	b.txt

Remember that each process, there are 3 file descriptors (i.e., 0-2) are created for 3 default open files (i.e., stdin, stdout, stderr). Therefore, when opening a new file, the next file descriptor is 3!

```
File descriptor for a.txt is 3
File descriptor for b.txt is 4
```

Remember that dup uses the **lowest-numbered unused descriptor** for the new descriptor Therefore, when duplicating fd_b, **fd_b_dup** is assigned to the value “3”.

```
Duplicate file descriptor for b.txt is 3
```

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

int main()
{
  int fd1;
  char c1, c2;
  fd1 = open("test.txt", O_RDONLY, 0);
  read(fd1, &c1, 1);

  int pid = fork();
  if (pid>0){ /* Parent */
    wait(NULL);
    read(fd1, &c2, 1);
    printf("Parent: c1 = %c, c2 = %c\n", c1, c2);
  }
  else if(pid == 0){ /* Child */
    read(fd1, &c2, 1);
    printf("Child: c1 = %c, c2 = %c\n", c1, c2);
  }
  return 0;
}
```

test.txt

```
Hello
```

```
Child: c1 = H, c2 = e
Parent: c1 = H, c2 = l
```

What happens when a process opens a file?

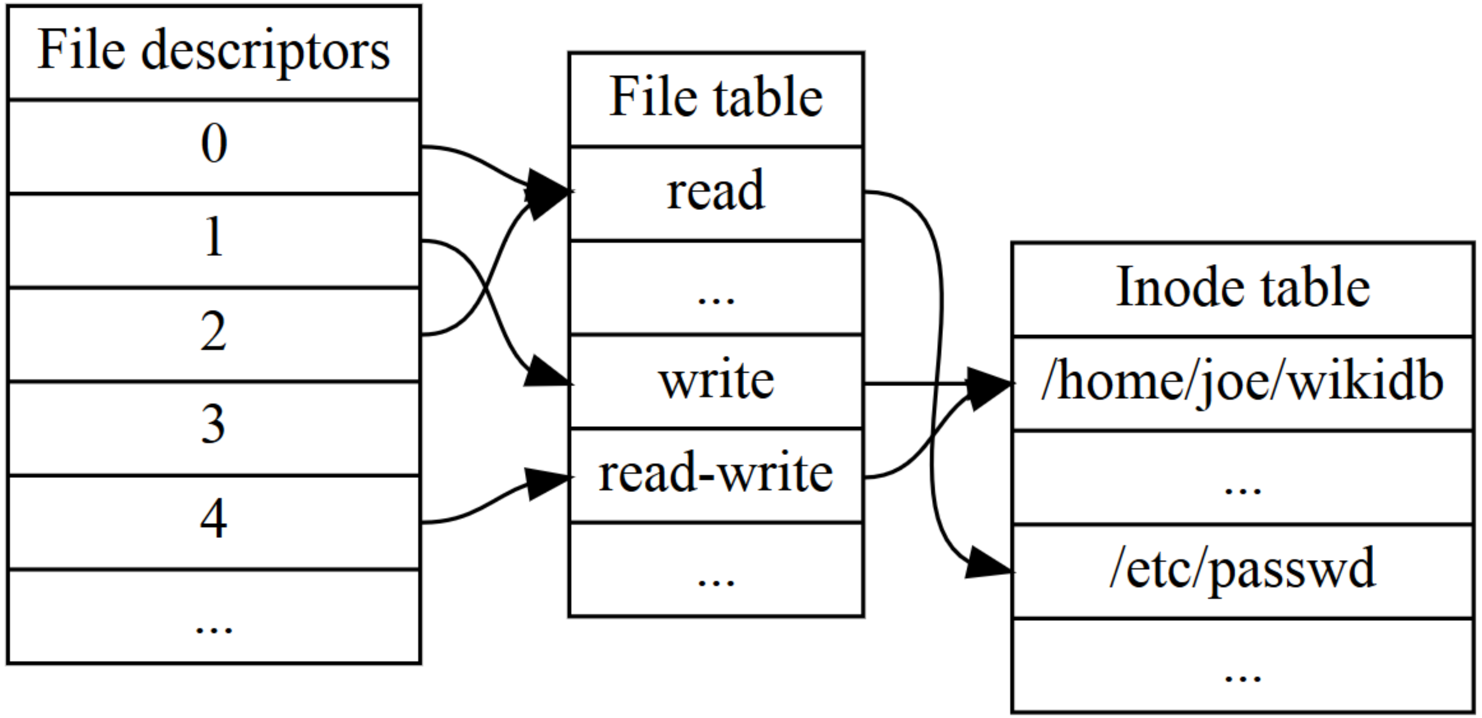
- A process might open several files to read from or write to. In each file, the next byte to be read/written must be known.
- Each process has an array to keep track of
  - Opened files
  - File status(open for read or write, ..etc)
  - Current offset (i.e., the cursor position) within a file
- For each opened file, the OS assigns a position in the ***file descriptor array***
- The position is filled with a **pointer** to a ***file table***
- *File table* handles the information in blue (file, file status, offset) for each opened file

file descriptor array ---> file table

What happens when a process opens a file?

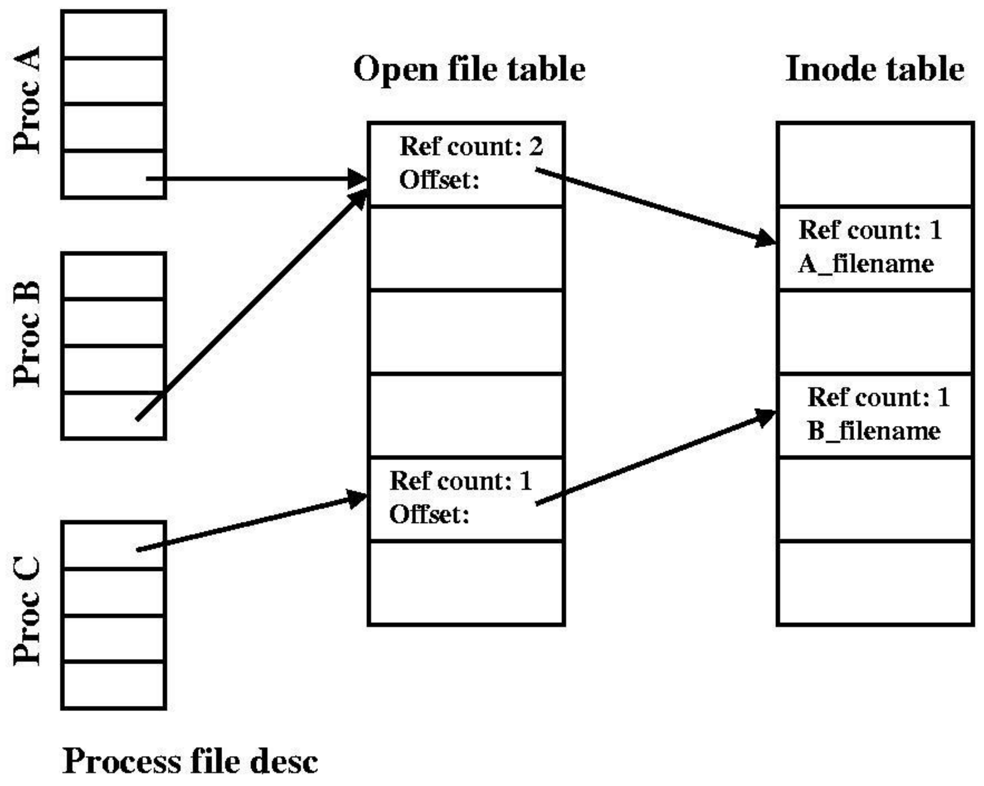
- The *file table* does not itself contain the file information, but instead has a pointer to another table (called the ***inode table***)
- *inode table* holds information about each file like what is the file location in disk.
- *inode table* that describes the actual underlying files

What complexity!



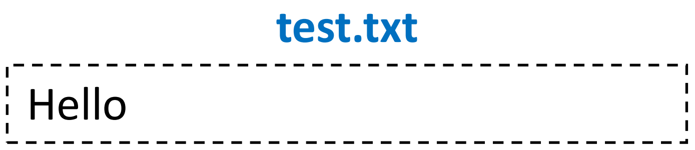
Actually, it turns out to be very flexible

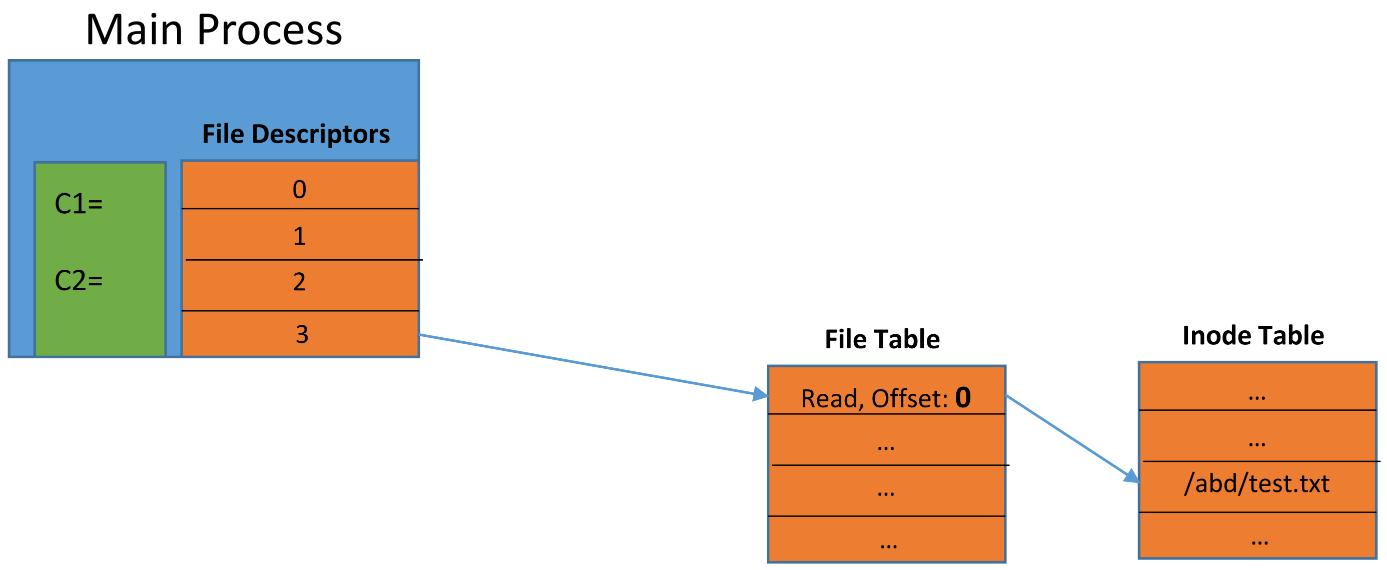
- Different processes can have file descriptors for the same file.



Revisit the previous program with visualization

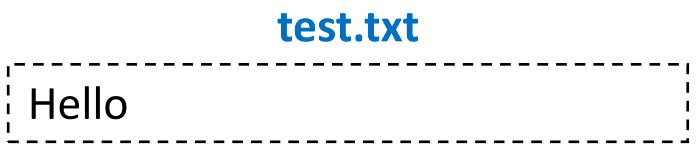
```
#include <stdio.h>

#include <sys/file.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

int main()

{

  int fd1;

  char c1, c2;

  fd1 = open("test.txt", O_RDONLY, 0);

  …












}
```

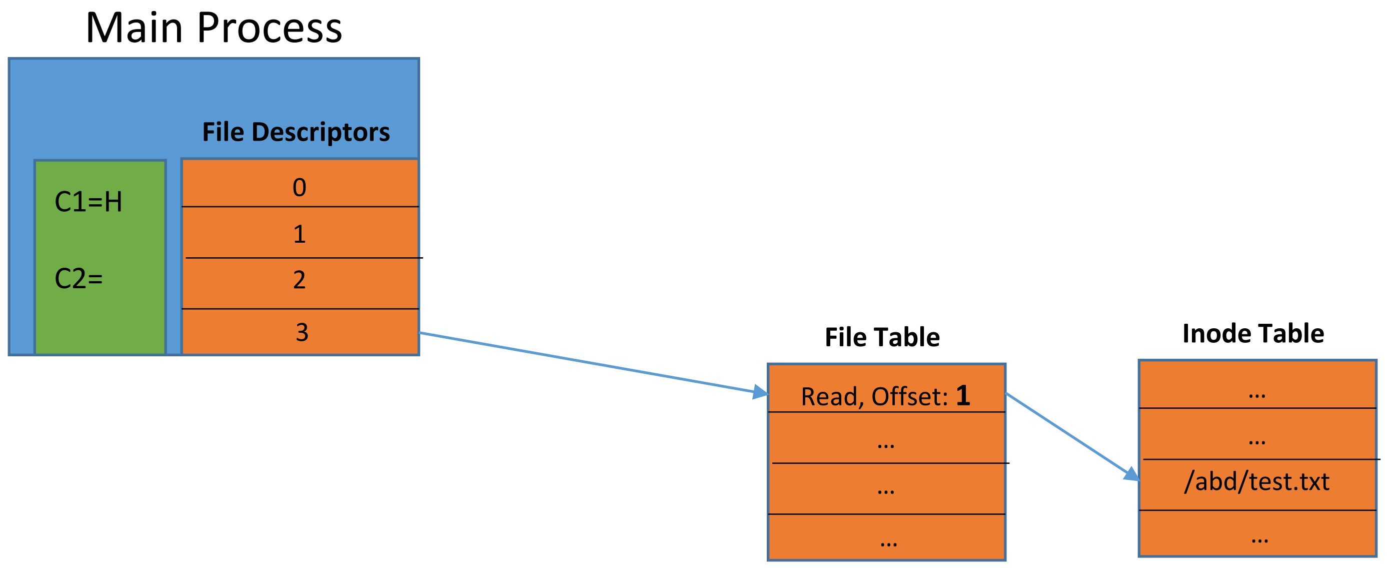




Revisit the previous program with visualization

```
#include <stdio.h>

#include <sys/file.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>


int main()

{

  int fd1;

  char c1, c2;

  fd1 = open("test.txt", O_RDONLY, 0);

  read(fd1, &c1, 1);

  …












}
```



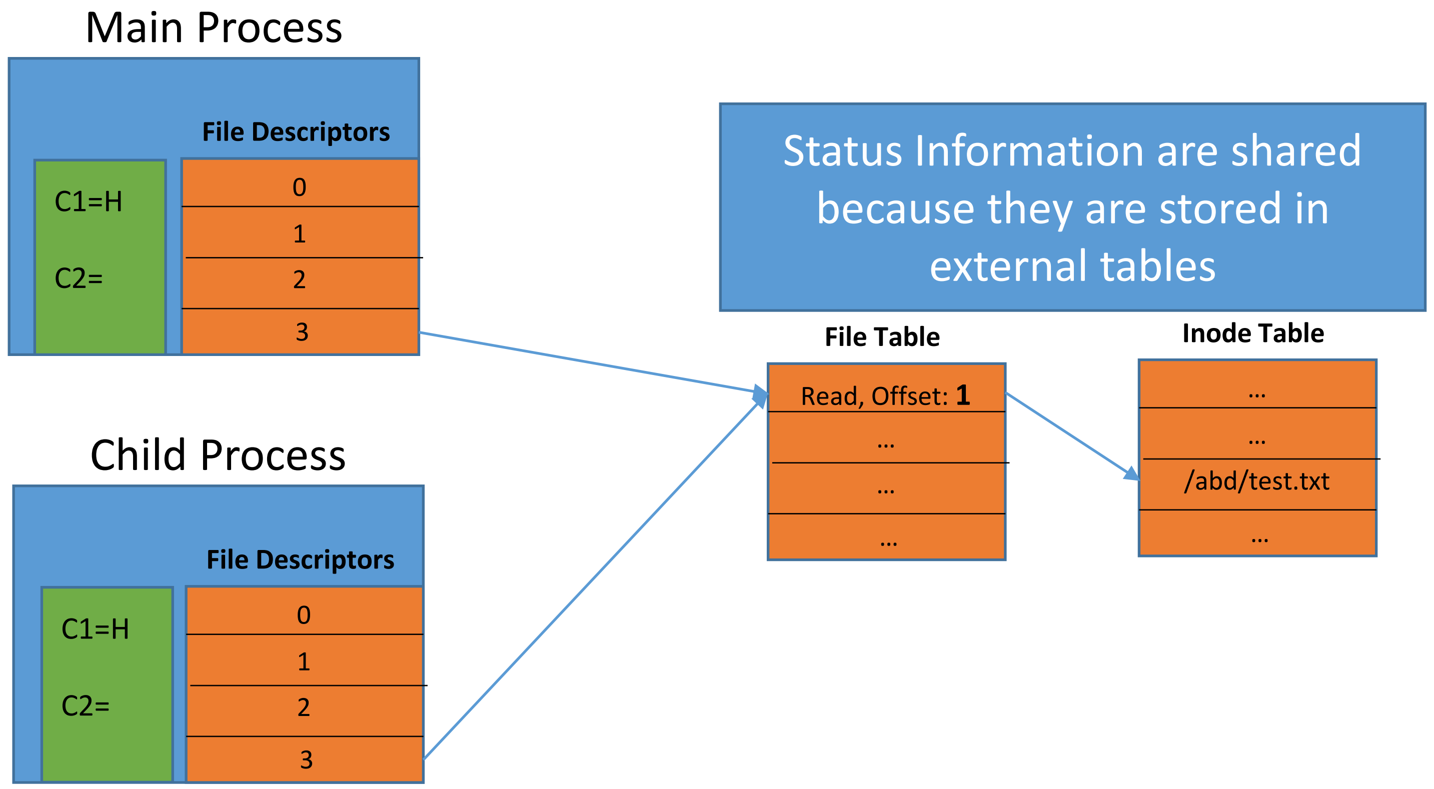


Revisit the previous program with visualization

```
#include <stdio.h>

#include <sys/file.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>


int main()

{

  int fd1;

  char c1, c2;

  fd1 = open("test.txt", O_RDONLY, 0);

  read(fd1, &c1, 1);

  int pid = fork();

  …












}
```

test.txt

Hello



Status Information are shared because they are stored in external tables

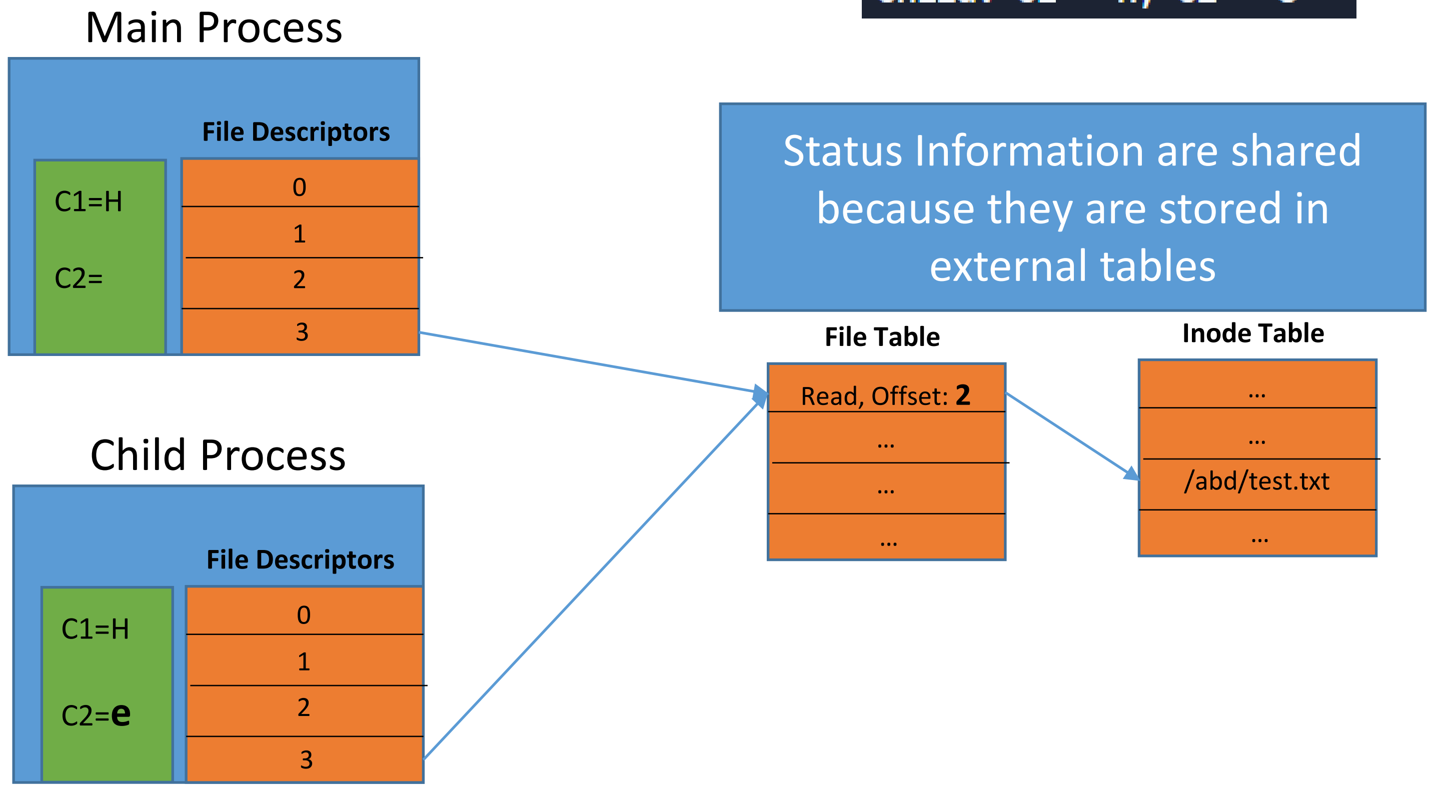
Revisit the previous program with visualization

```
#include <stdio.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main()
{
  int fd1;
  char c1, c2;
  fd1 = open("test.txt", O_RDONLY, 0);
  read(fd1, &c1, 1);
  int pid = fork();
  if (pid>0){ /* Parent */
    wait(NULL);
     …
  }
  else if(pid == 0){ /* Child */
    read(fd1, &c2, 1);
    printf("Child: c1 = %c, c2 = %c\n", c1, c2);
  }
  …
}
```

test.txt

Hello

```
Child: c1 = H, c2 = e
```



Revisit the previous program with visualization

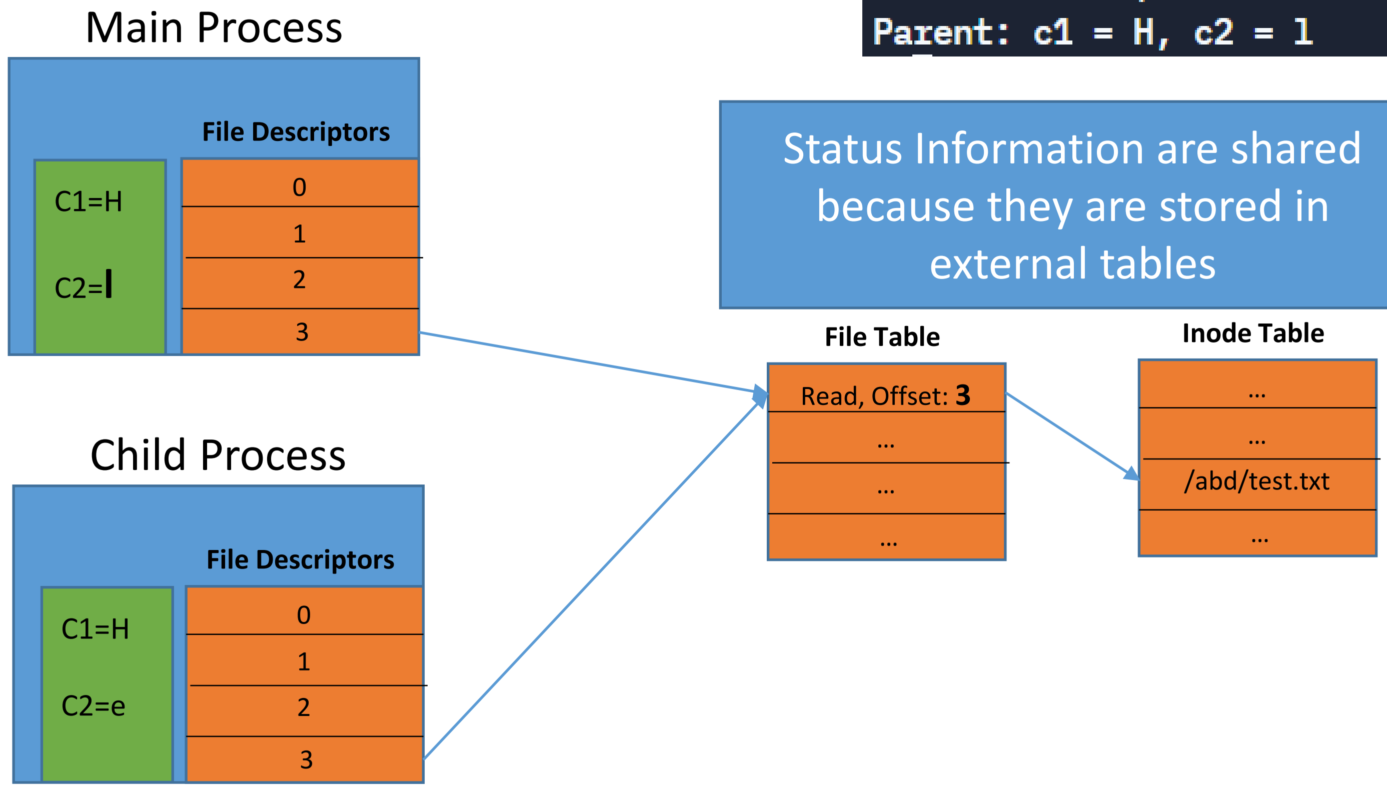
```
#include <stdio.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main()
{
  int fd1;
  char c1, c2;
  fd1 = open("test.txt", O_RDONLY, 0);
  read(fd1, &c1, 1);
  int pid = fork();
  if (pid>0){ /* Parent */
    wait(NULL);
    read(fd1, &c2, 1);
    printf("Parent: c1 = %c, c2 = %c\n", c1, c2);
  }
  else if(pid == 0){ /* Child */
    read(fd1, &c2, 1);
    printf("Child: c1 = %c, c2 = %c\n", c1, c2);
  }
  return 0;
}
```

test.txt

```
Hello
```

```
Child: c1 = H, c2 = e
Parent: c1 = H, c2 = l
```

Status Information are shared because they are stored in external tables



Unix file I/O - Review

Performed mostly using 6 commands

- open
- close
- read
- write
- lseek
- dup, dup2

The *open* system call

open	
Purpose	open or create a file for reading or writing

Include	#include<fcntl.h>
	also you need
	#include<sys/types.h>
	#include<sys/stat.h>

Useage	int open(const char *path, int flags, [mode_t mode]); (The third argument is optional.)
Arguments	• path: the (relative) path to the file • flags: see previous lecture • mode: file permissions, used when creating a new file
Returns	-1 on error file descriptor on success
Errors	Too numerous to list all: see manual if you want • ENOTDIR: A component of the path prefix is not a directory. • EACCES: Permissions do not permit reading or writing • EISDIR: The named file is a directory • EMFILE: The process has already reached its limit for open file descriptors.

The *close* system call

close	
Purpose	delete a file descriptor
Include	#include<unistd.h>
Useage	int close(int d);
Arguments	d: a file descriptor
Returns	-1 on error 0 on success (the file descriptor deleted)
Errors	• EBADF: d is not an active descriptor. • EINTR: An interrupt was received.

The *read* system call

read	
Purpose	read input from file
Include	#include<unistd.h>
Useage	ssize_t read(int d, void *buf, size_t nbytes);
Arguments	• d: a file descriptor • buf: buffer for storing bytes read • nbytes: maximum number of bytes to read
Returns	-1 on error number of bytes read and placed in buf or 0 if end of file
Errors	Too numerous to list all: see manual if you want • EBADF: d is not an active descriptor. • EFAULT: buf points outside the allocated address space. • EIO: An I/O error occurred while reading from the file system.

The *write* system call

write	
Purpose	write output to file
Include	#include<unistd.h>
Useage	ssize_t write(int d, void *buf, size_t nbytes);
Arguments	• d: a file descriptor • buf: buffer for storing bytes to be written • nbytes: maximum number of bytes to read
Returns	-1 on error number of bytes written
Errors	Too numerous to list all: see manual if you want • EBADF: d is not an active descriptor. • EFAULT: Data to be written to the file points outside the allocated address space. • EIO: An I/O error occurred while reading from the file system.

The *lseek* system call

lseek	
Purpose	reposition read/write file offset
Include	#include<unistd.h>
Useage	off_t lseek(int d, off_t offset, int base);
Arguments	• d: a file descriptor • offset: the number of bytes to be offset • base: the position from which the bytes will be offset: o SEEK_SET: offset bytes from beginning of the file. o SEEK_CUR: offset bytes from current value of offset. o SEEK_END: offset bytes from end of the file.
Returns	-1 on error The resulting offset location as measured in bytes from the beginning of the file.
Errors	• EBADF: d is not an active descriptor.. • EINVAL: base not a proper value. • ESPIPE: base associated with a non-regular file (pipe, socket or FIFO.)

Duplicating the file descriptor

dup dup2	
Purpose	duplicate an existing file descriptor
Include	#include<unistd.h>
Useage	int dup(int oldd); int dup2(int oldd, int newd);
Arguments	• oldd: an existing file descriptor • newd: the value of the new descriptor newd
Returns	-1 on error the value of newd
Errors	• EBADF: oldd or newd is not a valid active descriptor • EMFILE: Too many descriptors are active.