

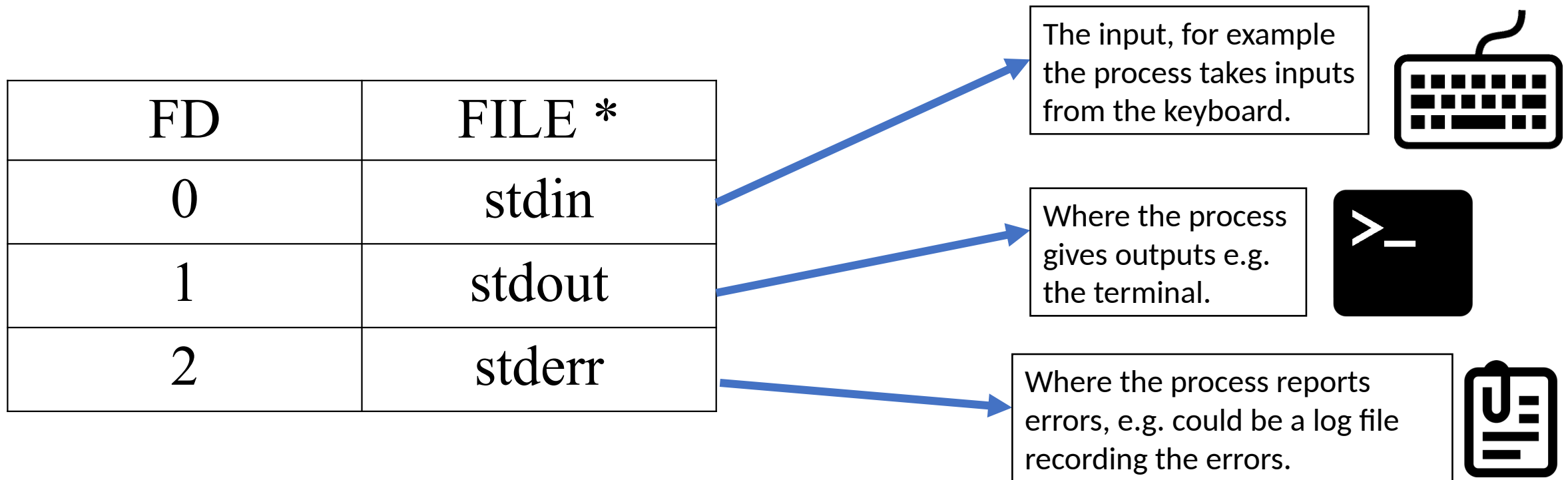
CSE 3100: Systems Programming

Part 2

Lecture 3: Redirection

Review From Last Lecture (1)

- A file descriptor is a nonnegative integer associated with a file.



Review From Last Lecture (2)

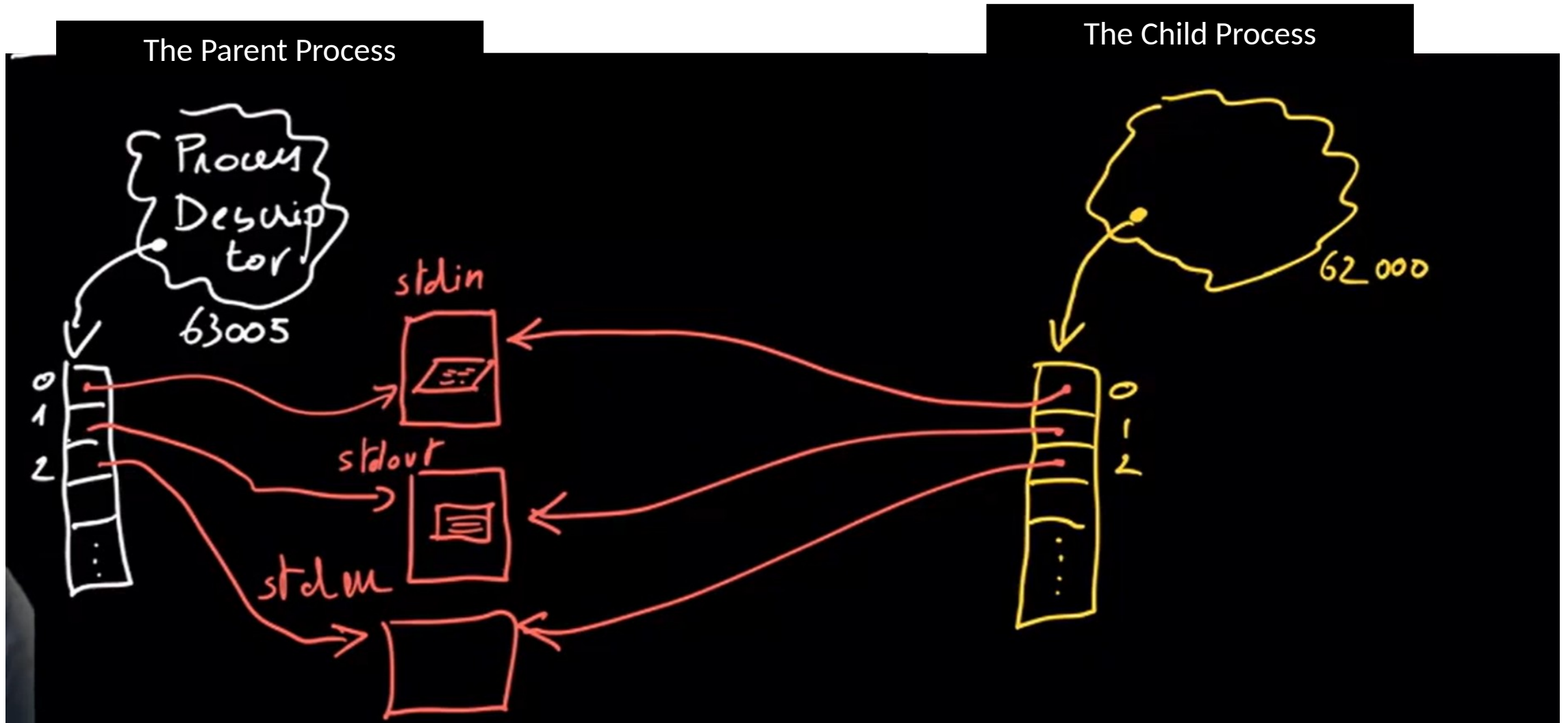


Figure Source: Professor Laurent Michel Youtube Lecture Videos

- The examples in class we have done typically have output written to the terminal.

```
1  #include <stdio.h>
2
3  int main(void){
4      printf("Hello world!\n");
5      return 0;
6  }
```



```
kaleel@CentralCompute:~$ ./HelloWorld
Hello world!
```

Question: What happens if we want processes to give us outputs in different places?

Follow up Question: Why would we want output/input from different places?



- So that we can run different operations at the same time (in parallel).
- For example, we want our word application open to write our English essay.
- But everyone knows it is impossible to do work without a good Spotify playlist. So we also want to listen to music in the background.
- To do this these programs need different file descriptors for their input and output.

How can we redirect things?

- Let's start with a simple example using the built in Unix function "sort".

```
$ sort < file.txt > sorted.txt
```

- Simple syntax

- **<** filename : Take input from the file *filename*
- **>** filename : Send output to the file *filename*
- **2>** filename : Send errors to the file *filename*

Sorting Example (without redirection)

My Unix Directory

📁 .cache	1/8/2023 10:40 PM	File folder	
📁 .vscode	1/8/2023 10:41 PM	File folder	
📁 .vscode-server	1/8/2023 10:39 PM	File folder	
📄 .bash_history	1/13/2023 10:52 PM	BASH_HISTORY File	1 KB
📄 .bash_logout	1/8/2023 10:38 PM	Bash Logout Sourc...	1 KB
📄 .bashrc	1/8/2023 10:38 PM	Bash RC Source File	4 KB
📄 .motd_shown	1/14/2023 8:00 PM	MOTD_SHOWN File	0 KB
📄 .profile	1/8/2023 10:38 PM	Profile Source File	1 KB
📄 .sudo_as_admin_successful	1/8/2023 10:45 PM	SUDO_AS_ADMIN...	0 KB
📄 .wget-hsts	1/8/2023 10:39 PM	WGET-HSTS File	1 KB
📄 adder	1/13/2023 9:47 PM	File	16 KB
📄 adder	1/11/2023 8:58 PM	C Source File	1 KB
📄 test	1/13/2023 10:46 PM	File	16 KB
📄 test	1/13/2023 10:46 PM	C Source File	3 KB
📄 UnsortedText	1/14/2023 10:37 PM	Text Document	1 KB

UnsortedText.txt

good
bad
hello
apple
morning

Sorting Example (without redirection)

First: We can just view the contents of the textfile in the command line using the “cat” command

```
kaleel@CentralCompute:~$ cat UnsortedText.txt
good
bad
hello
apple
morning
```

Second: We can sort the contents of UnsortedText.txt using the “sort” command

```
kaleel@CentralCompute:~$ sort UnsortedText.txt
apple
bad
good
hello
morning
```

But our output is still in the command line. What if we want to redirect the output to another file?

Sorting Example With Redirection

We can sort the contents of UnsortedText.txt and redirect the output to a NEW file location.

```
kaleel@CentralCompute:~$ sort UnsortedText.txt > sort.txt
```

How can we check the output of sort now?

```
kaleel@CentralCompute:~$ cat sort.txt  
apple  
bad  
good  
hello  
morning
```

Shell Redirections

Available when executing commands in your shell (e.g. `bash`)

- **Implemented with the close/open/dup technique**

`$ command < infile > outfile`

`< infile` : Take input from file *infile*
`> outfile` : Send output to file *outfile*

- Other variants

`>> outfile` : Append output to file *outfile*
`2> outfile` : Send errors to file *outfile*
`&> outfile` : Send both output and errors to file *outfile*

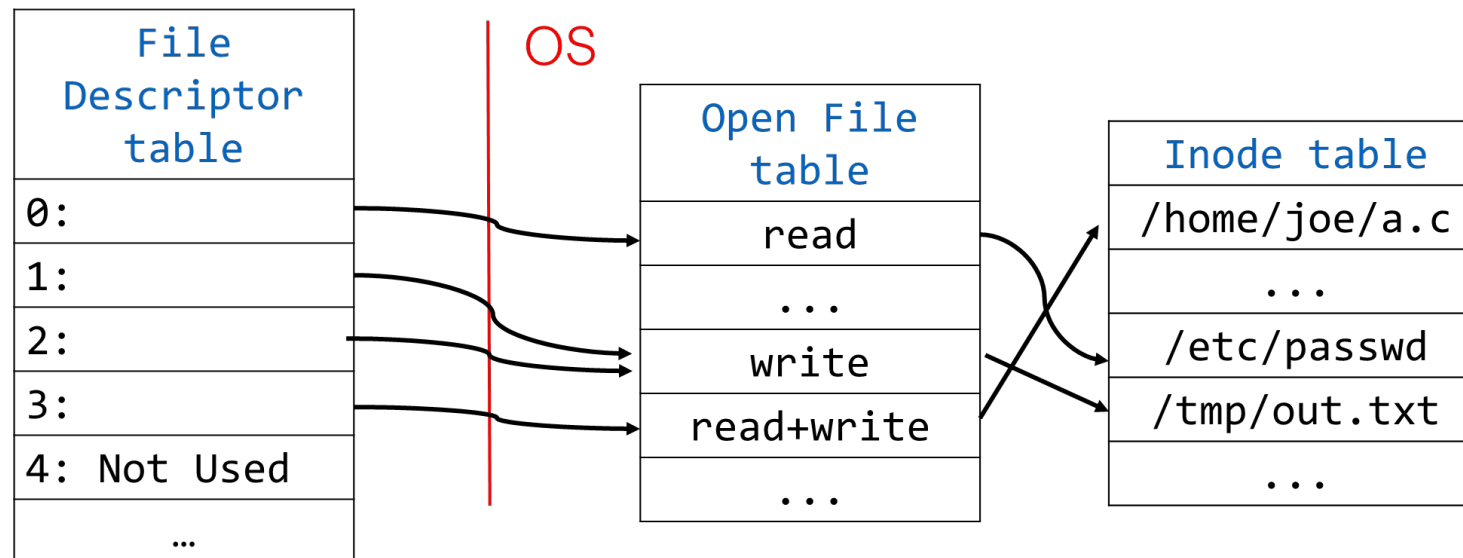
Read the manual for more variants like `2>>`, `2>&1`, etc.

Brief Recap

- Function `open()` returns a file descriptor, a non-negative integer
- The file descriptor is used later on in functions like `read()` and `close()`
- Every opened file has a file descriptor
 - `stdin`: 0, `stdout`: 1, `stderr`: 2
- Files opened in a process remain open after `fork()` and `exec`

File Descriptor Table

- Each process has a **file descriptor table**
 - Holds indices of entries into the **Open File Table** managed by OS
- The system-wide **Open File Table**
 - Records the *mode* of the opened files (e.g., reading, writing, appending)
 - Holds index into **the Inode Table** that has the actual file name and location on disk



Duplicating File Descriptors

- We do not change file descriptor table directly
- Used open() and close() and two new functions

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

- dup() copies oldfd to the **first available entry** (in FD table)
- dup2() copies oldfd to newfd
 - Closes newfd first if it is in use

There is dup3(), but it is not in POSIX. We should not use it in this course.

How would we use dup in practice?

Let's try writing to a text file using dup:

```
1  #include<stdio.h>
2  #include <unistd.h>
3  #include <fcntl.h>
4
5  int main()
6  {
7      // open() returns a file descriptor fd to the file "dup.txt"
8      int fd = open("dup.txt", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
9      printf("FD: %d\n", fd);
10     if(fd < 0)
11     {
12         printf("Cannot open the file\n");
13         return -1;
14     }
```



- First just open a text file which we call “dup.txt” for writing to.
- When opening the file we get a file descriptor fd.
- Lines 10 through 14 do some basic error checking to make sure that we have opened the file successfully.

How would we use dup in practice?

Let's try writing to a text file using dup:

```
1  #include<stdio.h>
2  #include <unistd.h>
3  #include <fcntl.h>
4
5  int main()
6  {
7      // open() returns a file descriptor fd to the file "dup.txt"
8      int fd = open("dup.txt", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
9
10
11
12
13
14
15      // dup() will create the copy of fd as the copy_fd
16      // then both can be used interchangeably.
17      int copy_fd = dup(fd);
```



- In line 17 we call dup to copy the file descriptor.
- Now fd and copy_fd can be used interchangeably.
- For this example we'll use both to write to the text file "dup.txt".

How would we use dup in practice?

Let's try writing to a text file using dup:

```
1  #include<stdio.h>
2  #include <unistd.h>
3  #include <fcntl.h>
4
5  int main()
6  {
7      // open() returns a file descriptor fd to the file "dup.txt"
8      int fd = open("dup.txt", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
9
10
11
12
13
14
15     // dup() will create the copy of fd as the copy_fd
16     // then both can be used interchangeably.
17     int copy_fd = dup(fd);
18
19     // write() will write the given string into the file
20     // referred by the file descriptors
21     //last parameter corresponds to the number of characters to be written
22     write(copy_fd,"This will be output to the file named dup.txt\n", 46);
23
```




- First we can use `copy_fd` and write to "dup.txt".

How would we use dup in practice?

Let's try writing to a text file using dup:

```
1  #include<stdio.h>
2  #include <unistd.h>
3  #include <fcntl.h>
4
5  int main()
6  {
7      // open() returns a file descriptor fd to the file "dup.txt"
8      int fd = open("dup.txt", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
9
10
11
12
13
14
15     // dup() will create the copy of fd as the copy_fd
16     // then both can be used interchangeably.
17     int copy_fd = dup(fd);
18
19     // write() will write the given string into the file
20     // referred by the file descriptors
21     //last parameter corresponds to the number of characters to be written
22     write(copy_fd,"This will be output to the file named dup.txt\n", 46);
23
24     //can also write using a copy of the file descriptor
25     write(fd,"This will also be output to the file named dup.txt\n", 51);
26
```

- 
- We can also use the original file descriptor fd to write to the text file “dup.txt”.

Now what is in dup.txt?

My Unix Directory

📁 .cache	1/8/2023 10:40 PM	File folder	
📁 .vscode	1/8/2023 10:41 PM	File folder	
📁 .vscode-server	1/8/2023 10:39 PM	File folder	
📄 .bash_history	1/16/2023 3:28 PM	BASH_HISTORY File	2 KB
📄 .bash_logout	1/8/2023 10:38 PM	Bash Logout Sourc...	1 KB
📄 .bashrc	1/8/2023 10:38 PM	Bash RC Source File	4 KB
📄 .lesshtQ	1/14/2023 10:39 PM	LESSHTSQ File	1 KB
📄 .motd_shown	1/16/2023 12:13 PM	MOTD_SHOWN File	0 KB
📄 .profile	1/8/2023 10:38 PM	Profile Source File	1 KB
📄 .sudo_as_admin_successful	1/8/2023 10:45 PM	SUDO_AS_ADMIN...	0 KB
📄 .wget-hsts	1/16/2023 12:13 PM	WGET-HSTS File	1 KB
📄 adder	1/13/2023 9:47 PM	File	16 KB
📄 adder	1/16/2023 12:14 PM	C Source File	1 KB
📄 dup	1/16/2023 4:23 PM	Text Document	1 KB
📄 DupDemo	1/16/2023 4:23 PM	File	16 KB
📄 DupDemo	1/16/2023 4:23 PM	C Source File	1 KB

dup.txt

This will be output to the file named dup.txt
This will also be output to the file named dup.txt

Using dup (Full Code)

First open the file and get the file descriptor.



```
1  #include<stdio.h>
2  #include <unistd.h>
3  #include <fcntl.h>
4
5  int main()
6  {
7      // open() returns a file descriptor fd to the file "dup.txt"
8      int fd = open("dup.txt", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
9      printf("FD: %d\n", fd);
10     if(fd < 0)
11     {
12         printf("Cannot open the file\n");
13         return -1;
14     }
15     // dup() will create the copy of fd as the copy_fd
16     // then both can be used interchangeably.
17     int copy_fd = dup(fd);
18
19     // write() will write the given string into the file
20     // referred by the file descriptors
21     //last parameter corresponds to the number of characters to be written
22     write(copy_fd,"This will be output to the file named dup.txt\n", 46);
23
24     //can also write using a copy of the file descriptor
25     write(fd,"This will also be output to the file named dup.txt\n", 51);
26
27     close(copy_fd);
28     close(fd);
29     return 0;
30 }
```

Second make a copy of the file descriptor using dup and write to the file.



Third we can write to the file using the original file descriptor and close.



Example of using dup2()

```
1 //dup2 demo
2 //redirecting stand output to a file
3 #include<stdio.h>
4 #include <unistd.h>
5 #include <fcntl.h>
6
7 int main()
8 {
9
10     printf("Before dup2.\n");
11     printf("Can you see this on the screen?\n");
12
13     //Make sure when the file is created, the user has the read and write access
14     int fd = open("dup2-output.txt", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
15
```

Very similar code as before,
opening a file and getting
the file descriptor fd.

Example of using dup2()

```
1 //dup2 demo
2 //redirecting stand output to a file
3 #include<stdio.h>
4 #include <unistd.h>
5 #include <fcntl.h>
6
7 int main()
8 {
9
10     printf("Before dup2.\n");
11     printf("Can you see this on the screen?\n");
12
13     //Make sure when the file is created, the user has the read and write access
14     int fd = open("dup2-output.txt", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
15
```

```
22     printf("Right before dup2.\n");
23     dup2(fd, 1);
```

The dup2() system call performs the same task as dup(), but instead of using the lowest-numbered unused file descriptor, it uses the file descriptor number specified in newfd. In other words, the file descriptor newfd is adjusted so that it now refers to the same open file

<https://man7.org/linux/man-pages/man2/dup.2.html>

Call dup2(oldfd, newfd) which copies oldfd to newfd i.e. our standard output is redirected to be written to a textfile.

Example of using dup2()

```
1 //dup2 demo
2 //redirecting stand output to a file
3 #include<stdio.h>
4 #include <unistd.h>
5 #include <fcntl.h>
6
7 int main()
8 {
9
10     printf("Before dup2.\n");
11     printf("Can you see this on the screen?\n");
12
13     //Make sure when the file is created, the user has the read and write access
14     int fd = open("dup2-output.txt", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
15
16
17
18
19
20
21
22
23     dup2(fd, 1);
24     close(fd);
25     printf("After dup2.\n");
26     printf("Can you see this on the screen? I guess not.\n");
27     return 0;
28 }
```

After setting our standard output, call printf a few times and see what happens.

Example of using dup2() full code

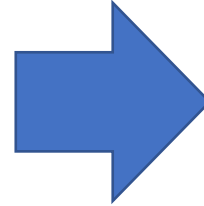
```
1 //dup2 demo
2 //redirecting stand output to a file
3 #include<stdio.h>
4 #include <unistd.h>
5 #include <fcntl.h>
6
7 int main()
8 {
9
10     printf("Before dup2.\n");
11     printf("Can you see this on the screen?\n");
12
13     //Make sure when the file is created, the user has the read and write access
14     int fd = open("dup2-output.txt", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
15
16     if(fd < 0)
17     {
18         printf("Error opening the file\n");
19         return -1;
20     }
21
22     printf("Right before dup2.\n");
23     dup2(fd, 1);
24     close(fd);
25     printf("After dup2.\n");
26     printf("Can you see this on the screen? I guess not.\n");
27     return 0;
28 }
```

Open a textfile for reading, get file descriptor fd.

Use dup2 to redirect our output. Do a few print statements.

Example of using dup2() full code

```
1 //dup2 demo
2 //redirecting stand output to a file
3 #include<stdio.h>
4 #include <unistd.h>
5 #include <fcntl.h>
6
7 int main()
8 {
9     printf("Before dup2.\n");
10    printf("Can you see this on the screen?\n");
11
12    //Make sure when the file is created, the user has the read and write access
13    int fd = open("dup2-output.txt", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
14
15    if(fd < 0)
16    {
17        printf("Error opening the file\n");
18        return -1;
19    }
20
21    printf("Right before dup2.\n");
22    dup2(fd, 1);
23    close(fd);
24    printf("After dup2.\n");
25    printf("Can you see this on the screen? I guess not.\n");
26    return 0;
27 }
28 }
```



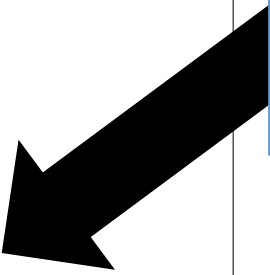
```
kaleel@CentralCompute:~$ ./test
Before dup2.
Can you see this on the screen?
Right before dup2.
```



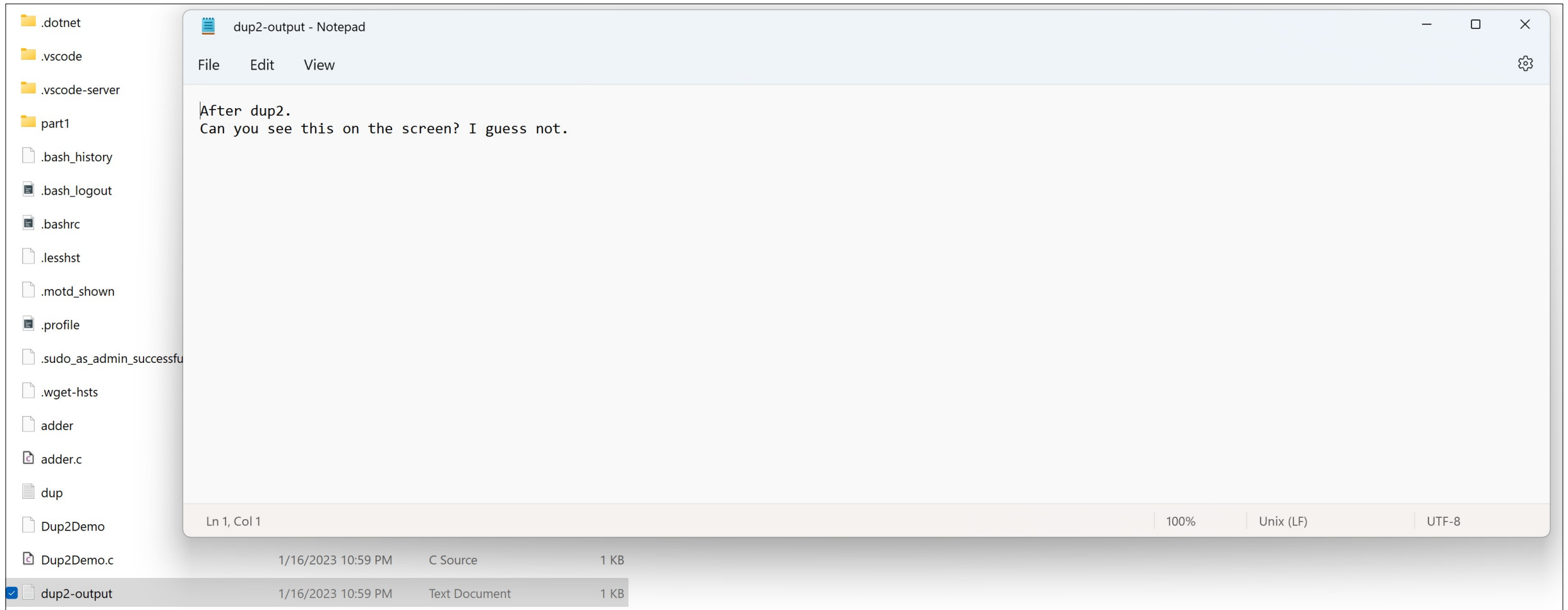
Example of using dup2() full code

```
1 //dup2 demo
2 //redirecting stand output to a file
3 #include<stdio.h>
4 #include <unistd.h>
5 #include <fcntl.h>
6
7 int main()
8 {
9
10     printf("Before dup2.\n");
11     printf("Can you see this on the screen?\n");
12
13     //Make sure when the file is created, the user has the read and write access
14     int fd = open("dup2-output.txt", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
15
16     if(fd < 0)
17     {
18         printf("Error opening the file\n");
19         return -1;
20     }
21
22     printf("Right before dup2.\n");
23     dup2(fd, 1);
24     close(fd);
25     printf("After dup2.\n");
26     printf("Can you see this on the screen? I guess not.\n");
27     return 0;
28 }
```

These are inside the
dup2-output.txt file!



Looking inside dup2-output.txt

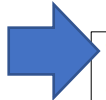


Pictorial: What is going on in the dup2 example?

```
7  int main()
8  {
9
10     printf("Before dup2.\n");
11     printf("Can you see this on the screen?\n");
12
13     //Make sure when the file is created, the user has the read and write access
14     int fd = open("dup2-output.txt", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
15
16     if(fd < 0)
17     {
18         printf("Error opening the file\n");
19         return -1;
20     }
21
22     printf("Right before dup2.\n");
23     dup2(fd, 1);
24     close(fd);
25     printf("After dup2.\n");
26     printf("Can you see this on the screen? I guess not.\n");
27     return 0;
28 }
```


FD	FILE *
0	stdin
1	stdout
2	stderr

Line 7: start the program, start initial file descriptor table




```
7  int main()
8  {
9
10     printf("Before dup2.\n");
11     printf("Can you see this on the screen?\n");
12
13     //Make sure when the file is created, the user has the read and write access
14     int fd = open("dup2-output.txt", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
15
16     if(fd < 0)
17     {
18         printf("Error opening the file\n");
19         return -1;
20     }
21
22     printf("Right before dup2.\n");
23     dup2(fd, 1);
24     close(fd);
25     printf("After dup2.\n");
26     printf("Can you see this on the screen? I guess not.\n");
27     return 0;
28 }
```


FD	FILE *
0	stdin
1	stdout
2	stderr



keyboard




Terminal



Your Screen

Line 10 and 11: Print some stuff



```
7  int main()
8  {
9
10     printf("Before dup2.\n");
11     printf("Can you see this on the screen?\n");
12
13     //Make sure when the file is created, the user has the read and write access
14     int fd = open("dup2-output.txt", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
15
16     if(fd < 0)
17     {
18         printf("Error opening the file\n");
19         return -1;
20     }
21
22     printf("Right before dup2.\n");
23     dup2(fd, 1);
24     close(fd);
25     printf("After dup2.\n");
26     printf("Can you see this on the screen? I guess not.\n");
27     return 0;
28 }
```

FD	FILE *
0	stdin
1	stdout
2	stderr

keyboard


Terminal

Your Screen

We call printf() in lines 10 and 11. Where will it get printed?

We check the file descriptor table and see that stdout (entry 1) is pointing to the terminal
So we go print to the terminal!

Line 14: Open a new file



```
7  int main()
8  {
9
10     printf("Before dup2.\n");
11     printf("Can you see this on the screen?\n");
12
13     //Make sure when the file is created, the user has the read and write access
14     int fd = open("dup2-output.txt", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
15
16     if(fd < 0)
17     {
18         printf("Error opening the file\n");
19         return -1;
20     }
21
22     printf("Right before dup2.\n");
23     dup2(fd, 1);
24     close(fd);
25     printf("After dup2.\n");
26     printf("Can you see this on the screen? I guess not.\n");
27     return 0;
28 }
```

FD	FILE *
0	stdin
1	stdout
2	stderr
3	

keyboard

Terminal


Your Screen

Dup2-output.txt

In line 14 we open a new file.
Time to create a new entry in the file descriptor table!

Line 22: Call printf() one more time

```
7  int main()
8  {
9
10     printf("Before dup2.\n");
11     printf("Can you see this on the screen?\n");
12
13     //Make sure when the file is created, the user has the read and write access
14     int fd = open("dup2-output.txt", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
15
16     if(fd < 0)
17     {
18         printf("Error opening the file\n");
19         return -1;
20     }
21
22     printf("Right before dup2.\n");
23     dup2(fd, 1);
24     close(fd);
25     printf("After dup2.\n");
26     printf("Can you see this on the screen? I guess not.\n");
27     return 0;
28 }
```



FD	FILE *
0	stdin
1	stdout
2	stderr
3	

keyboard

Terminal

Your Screen

Dup2-output.txt

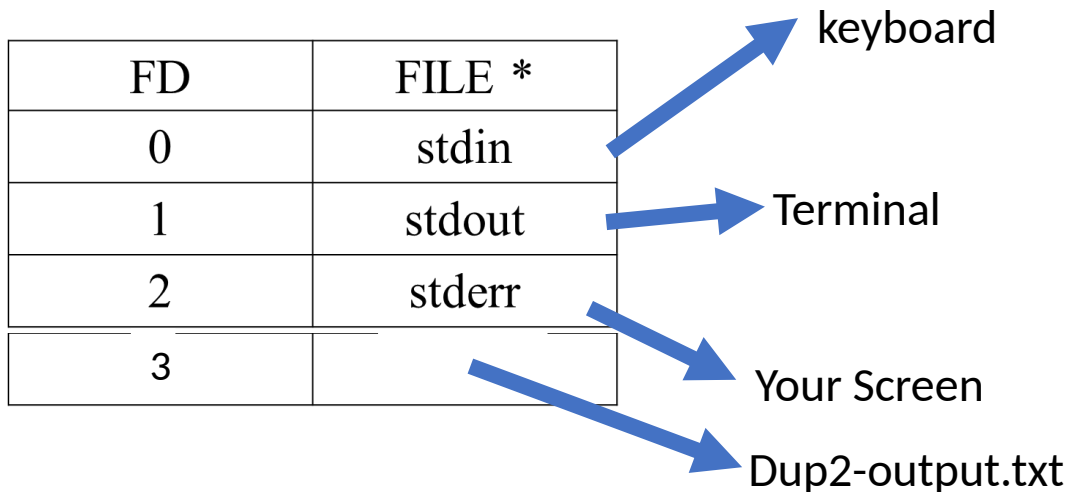
Call printf() one more time.

Because stdout is still pointing to terminal, we'll get printing on the terminal.

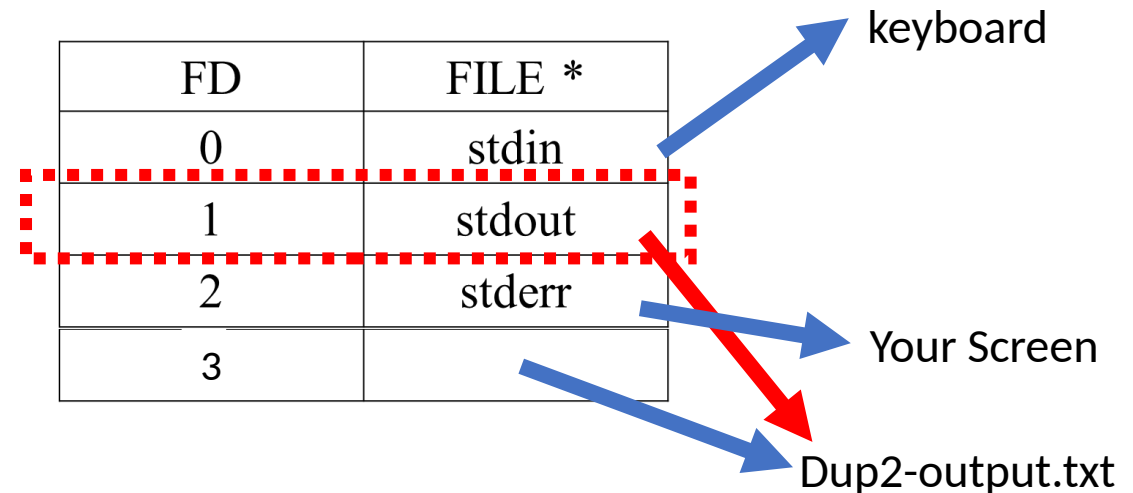
Line 23: Call dup2(fd,1)

```
23      dup2(fd, 1);
```

BEFORE LINE 23:



AFTER LINE 23:



Line 24: Close fd (entry 3)

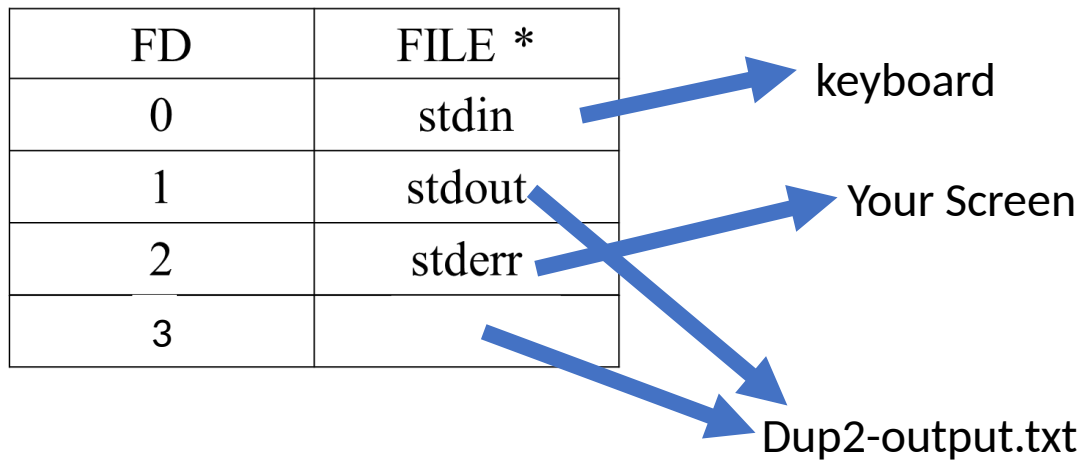
24

```
close(fd);
```

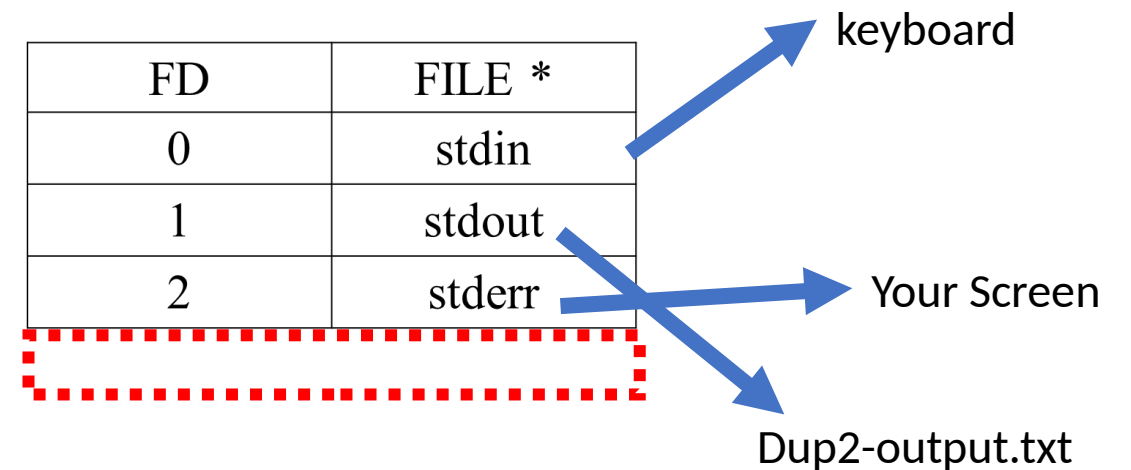
Why can we close fd if we still want to write to it?

Answer: Because now entry 1 in the file descriptor table points to the same place.


BEFORE LINE 24:



AFTER LINE 24:



Line 25 and 26: Call printf() again



```
7  int main()
8  {
9
10     printf("Before dup2.\n");
11     printf("Can you see this on the screen?\n");
12
13     //Make sure when the file is created, the user has the read and write access
14     int fd = open("dup2-output.txt", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
15
16     if(fd < 0)
17     {
18         printf("Error opening the file\n");
19         return -1;
20     }
21
22     printf("Right before dup2.\n");
23     dup2(fd, 1);
24     close(fd);
25     printf("After dup2.\n");
26     printf("Can you see this on the screen? I guess not.\n");
27     return 0;
28 }
```

FD	FILE *
0	stdin
1	stdout
2	stderr

keyboard

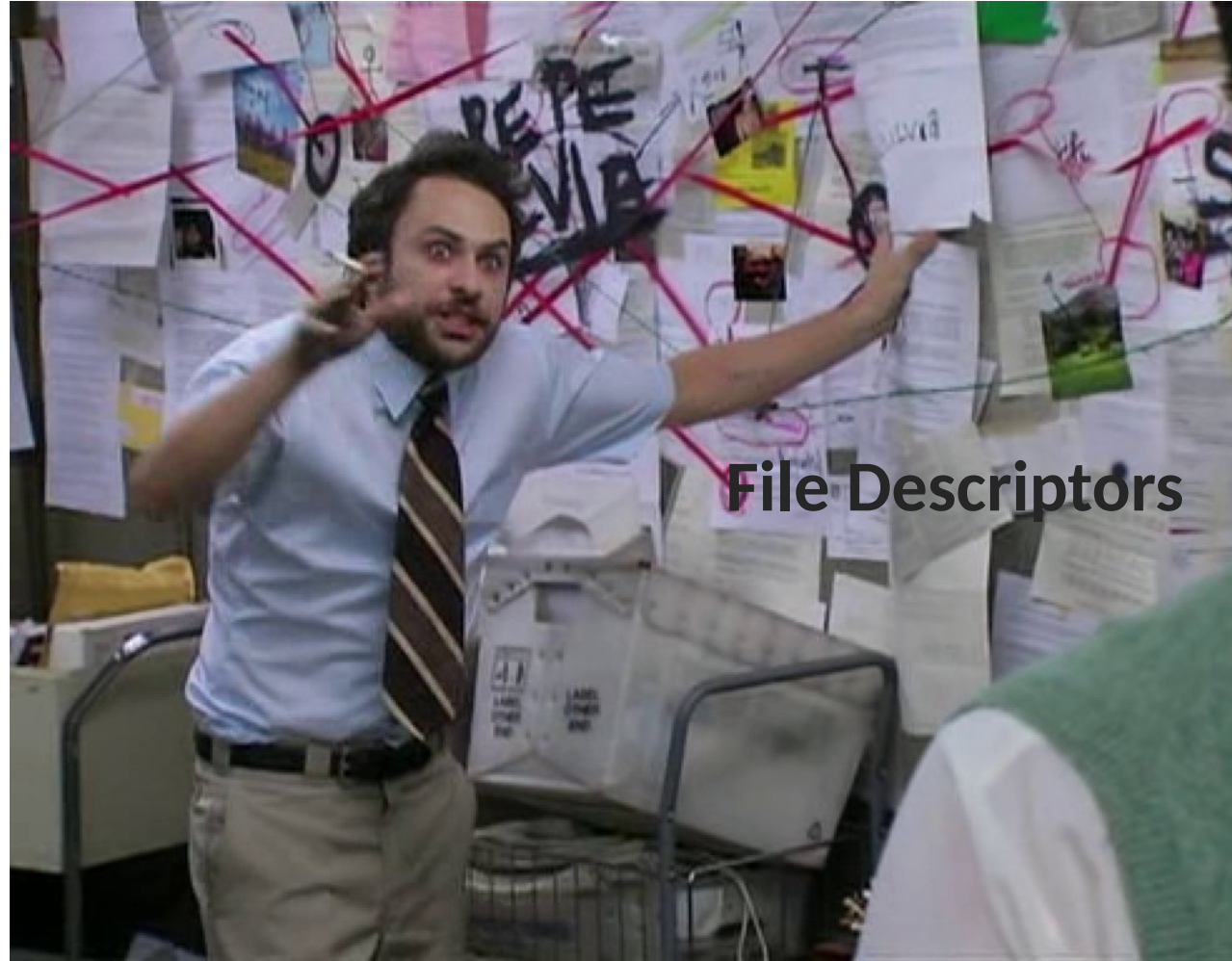
Dup2-output.txt

Your Screen

Call printf(). Where will we print?

Because stdout is still pointing to a text file, we print in the text file.

Why do we care about all this?



Understanding the Big Picture

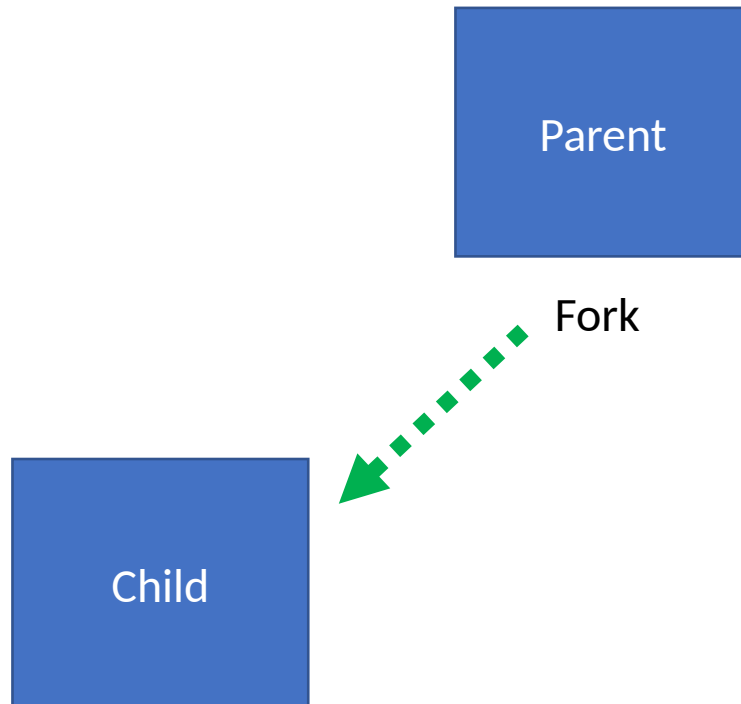
We start with one process (program).



Parent

Understanding the Big Picture

1. We start with one process (program)
2. But we want a process to do multiple things so we'll call fork



Understanding the Big Picture

1. We start with one process (program)
2. But we want a process to do multiple things so we'll call fork
3. However when we fork, what happens to the file descriptor table?



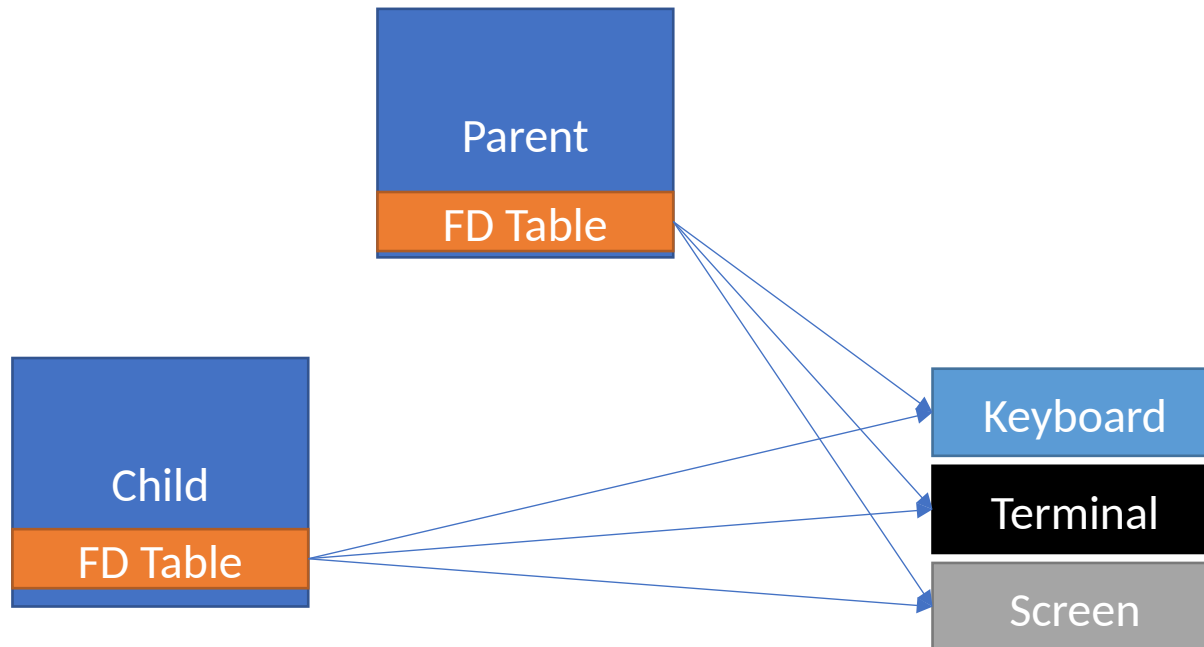
Parent

Child

Understanding the Big Picture

1. We start with one process (program)
2. But we want a process to do multiple things, so we'll call fork
3. However when we fork, what happens to the file descriptor table?

Both parent and child FD tables are pointing to the same places!

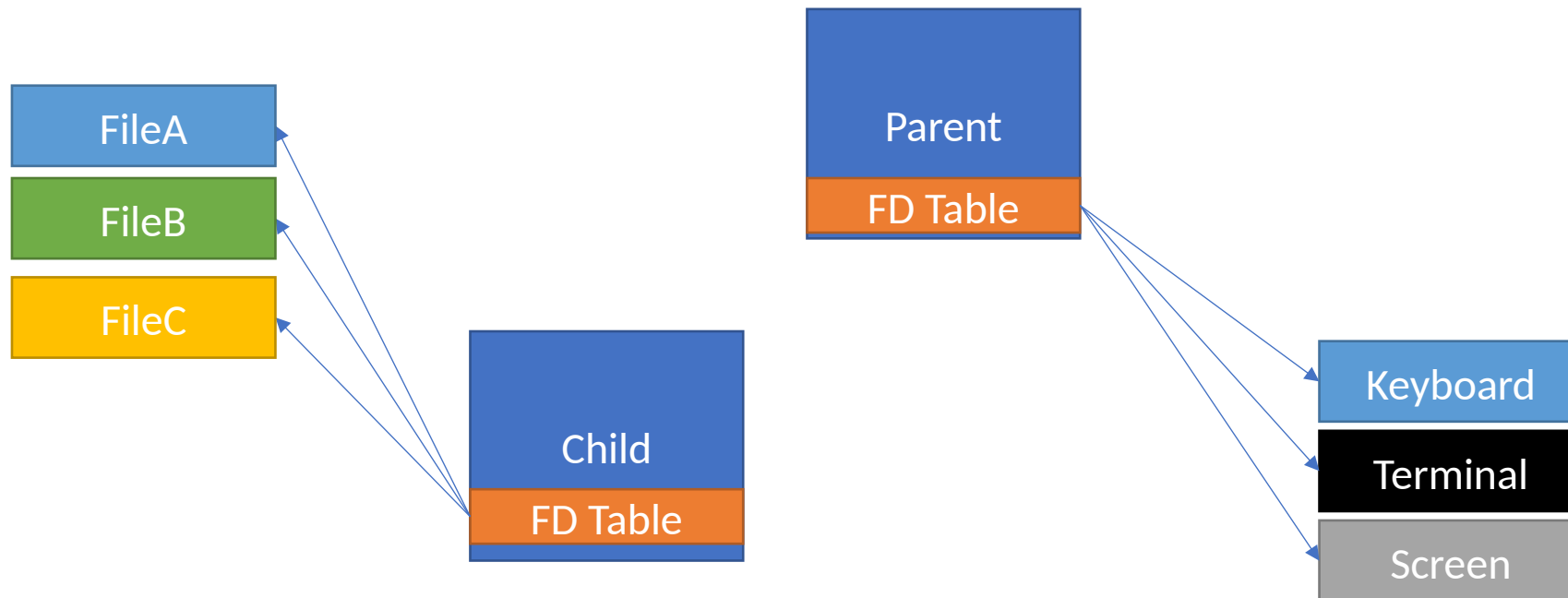


Understanding the Big Picture

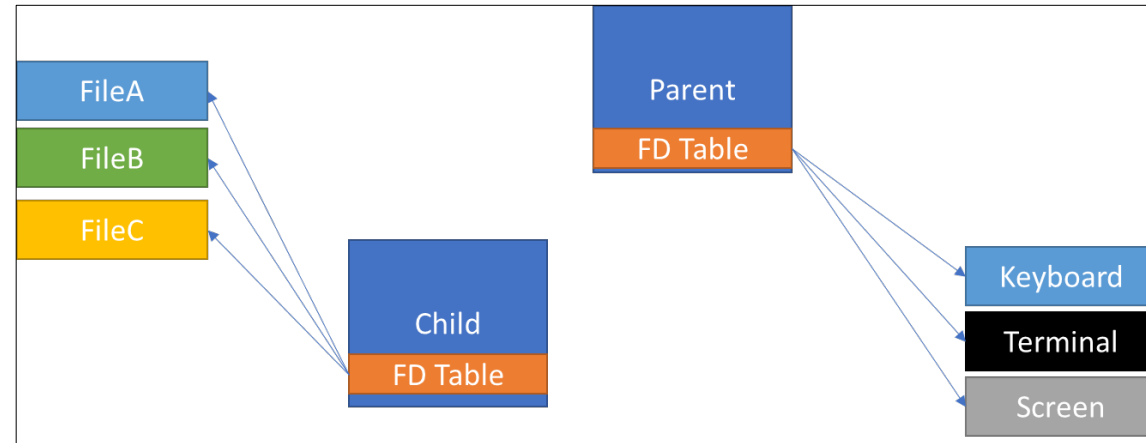
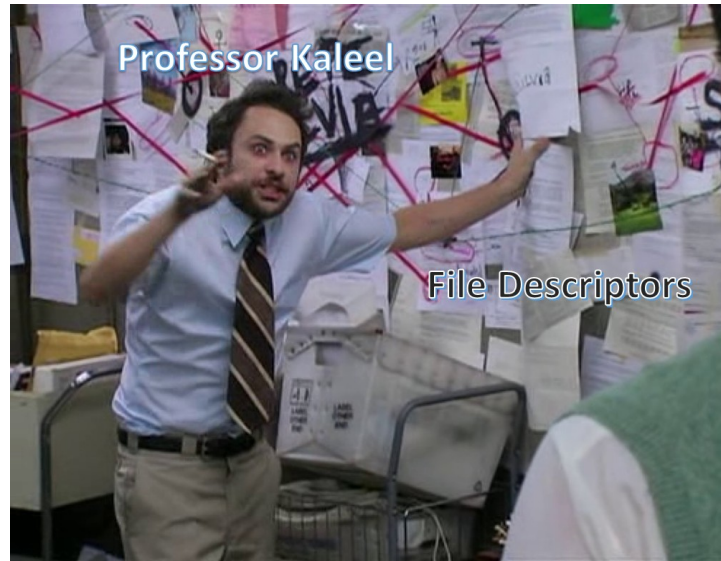
3. However when we fork, what happens to the file descriptor table?

Both parent and child FD tables are pointing to the same places!

So we need to call `dup()` or `dup2()` in the child to get different functionality



Understanding the Big Picture



1. We start with one process (program).
2. But we want a process to do multiple things, so we'll call fork.
3. However when we fork, both file descriptor tables point to the same things.
4. So we need to call `dup()` or `dup2()` in the child to get different functionality

The Big Picture Example

```
//In this example, we demo how to redirect a child process's standard  
output  
//Moreover, the child process does exec to run a command line  
//wc filename  
//to count the # of lines and word count of the file  
//The standard output is redirected to output.txt
```

```
16  int main(int argc, char *argv[])
17  {
18
19      if(argc!=2)
20      {
21          printf("Usage: %s filename\n", argv[0]);
22          return -1;
23      }
```

Start by taking the name of a textfile as input in the main. Do some basic error checking.

```
16  int main(int argc, char *argv[])
17  {
```

```
24  pid_t pid;
25  pid = fork();
26  if(pid == 0)
27  {
```

} Make a call to fork and check to make sure we are in the child (pid==0)

```
30  }
```

```
16  int main(int argc, char *argv[])
17  {
18
26      if(pid == 0)
27      {
28
29          //Make sure when the file is created, the user has the read and write access
30          int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
31          if(fd < 0)
32          {
33              printf("Cannot open the file\n");
34              return -1;
35          }
```

In the child process open a new textfile for writing to and make sure the file can be opened.

```
16 int main(int argc, char *argv[])
17 {
18
26     if(pid == 0)
27     {
28
29         //Make sure when the file is created, the user has the read and write access
30         int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
```

```
36     dup2(fd, 1);
37     close(fd);
```

Use dup2() to set the output of the child to go to the output textfile.

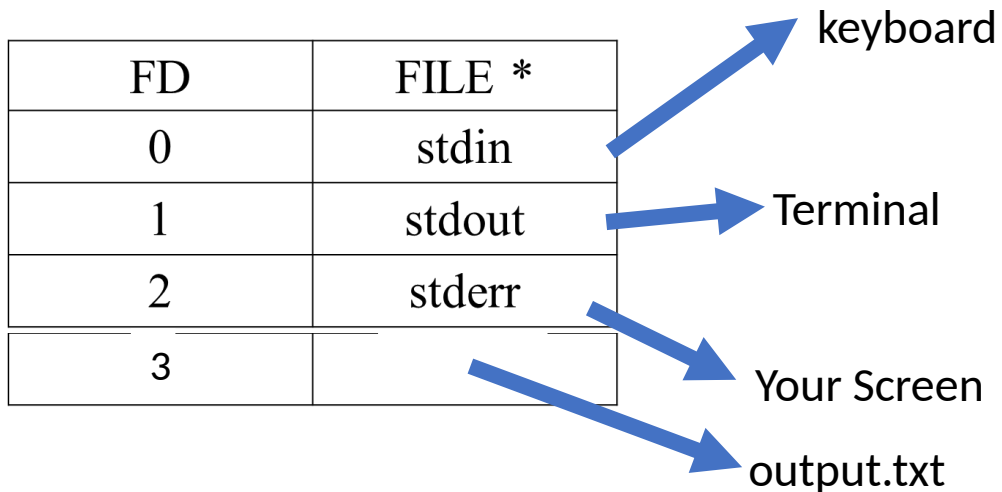
Close the original file descriptor fd as we don't need it.

Side Note: What does the FD table look like before and after line 36?

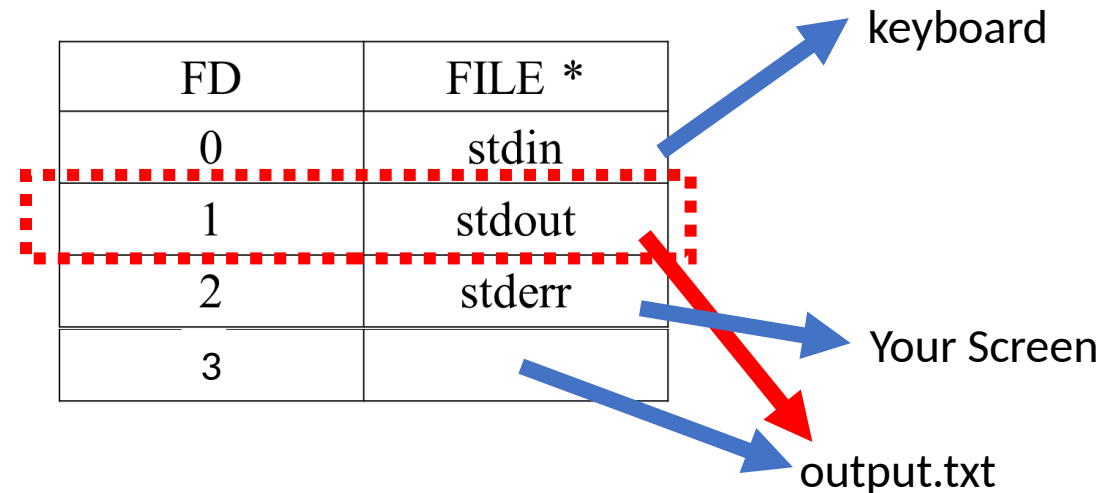
36

```
dup2(fd, 1);
```

BEFORE LINE 36:



AFTER LINE 36:



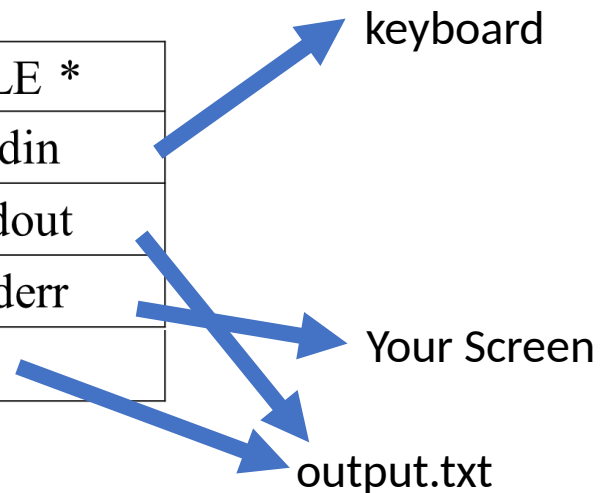
Side Note: What does the FD table like before and after line 37?

37 ✓

```
close(fd);
```

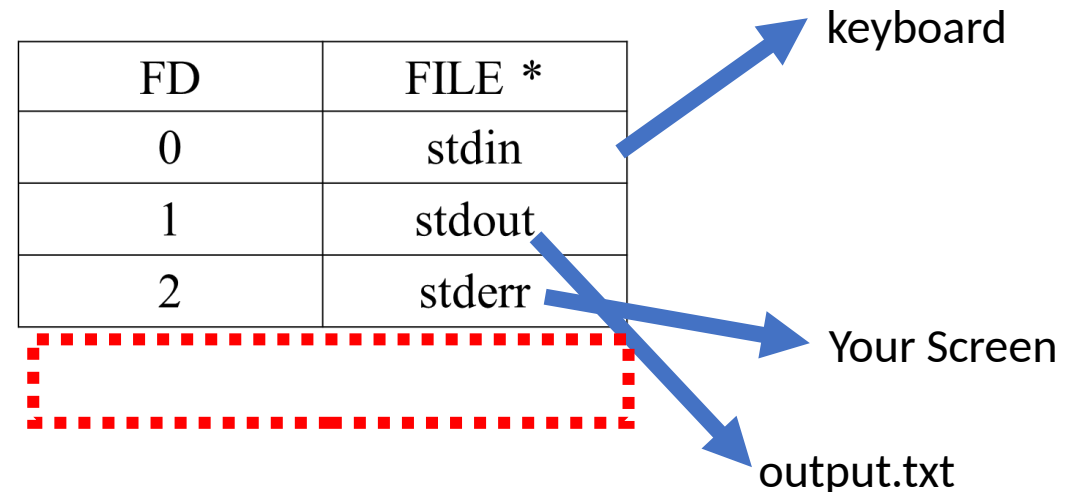
BEFORE LINE 37:

FD	FILE *
0	stdin
1	stdout
2	stderr
3	



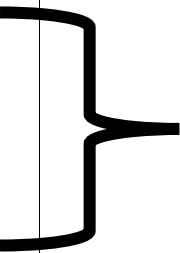
AFTER LINE 37:

FD	FILE *
0	stdin
1	stdout
2	stderr




```
16  int main(int argc, char *argv[])
17  {
18
26      if(pid == 0)
27      {
28
29          //Make sure when the file is created, the user has the read and write access
30          int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
```

```
36      dup2(fd, 1);
37      close(fd);
38      char * argv_list[] = {"wc", argv[1], NULL};
39
40      // the execv() only return if error occurred.
41      // The return value is -1
42      execvp("wc", argv_list);
```



We use `execvp` to call “wc” to count the number of lines in our text file and write the output to `output.txt`.

```
16 int main(int argc, char *argv[])
```

```
17 {
```

```
26     if(pid == 0)
```

```
27     {
```

```
45     }
```

```
46     else{
```

```
47         printf("I'm the parent printing to the terminal...\n");
```

If we are the parent and not the child, just print something for fun.

View of the entire code

```
7 #include <stdio.h>
8 #include <stdlib.h>
9 #include <string.h>
10 #include <unistd.h>
11 #include <fcntl.h>
12 #include <ctype.h>
13 #include <sys/stat.h>
14 #define MAX_LINE 1024
15
16 int main(int argc, char *argv[])
17 {
18
19     if(argc!=2)
20     {
21         printf("Usage: %s filename\n", argv[0]);
22         return -1;
23     }
24     pid_t pid;
25     pid = fork();
26     if(pid == 0)
27     {
28
29         //Make sure when the file is created, the user has the read and write access
30         int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
31         if(fd < 0)
32         {
33             printf("Cannot open the file\n");
34             return -1;
35         }
36         dup2(fd, 1);
37         close(fd);
38         char * argv_list[] = {"wc", argv[1], NULL};
39
40         // the execv() only return if error occurred.
41         // The return value is -1
42         execvp("wc",argv_list);
43         printf("Something is wrong.\n");
44         exit(-1);
45     }
46     else{
47         printf("I'm the parent printing to the terminal...\n");
48     }
49     return 0;
50 }
```

Call fork to get processes.

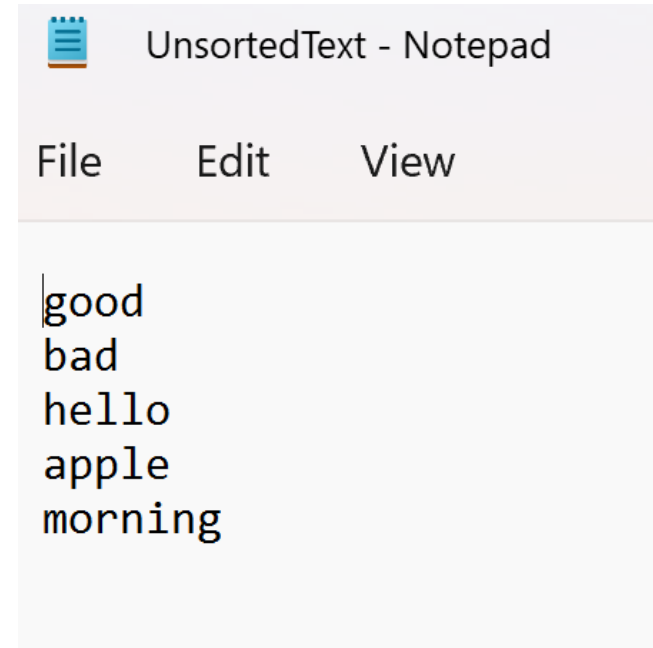
In the child open a text file, set the output of the process to the text file.
Call "wc" to count the lines of a different text file passed by argv.

In the parent print something to terminal.

The Big Picture Example : Input

Text file whose lines I want to count:

```
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <string.h>
10 #include <unistd.h>
11 #include <fcntl.h>
12 #include <ctype.h>
13 #include <sys/stat.h>
14 #define MAX_LINE 1024
15
16 int main(int argc, char *argv[])
17 {
18     if(argc!=2)
19     {
20         printf("Usage: %s filename\n", argv[0]);
21         return -1;
22     }
23     pid_t pid;
24     pid = fork();
25     if(pid == 0)
26     {
27         //Make sure when the file is created, the user has the read and write access
28         int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
29         if(fd < 0)
30         {
31             printf("Cannot open the file\n");
32             return -1;
33         }
34         dup2(fd, 1);
35         close(fd);
36         char * argv_list[] = {"wc", argv[1], NULL};
37         // the execv() only return if error occurred.
38         // The return value is -1
39         execvp("wc", argv_list);
40         printf("Something is wrong.\n");
41         exit(-1);
42     }
43     else{
44         printf("I'm the parent printing to the terminal...\n");
45     }
46     return 0;
47 }
```



UnsortedText - Notepad

File Edit View

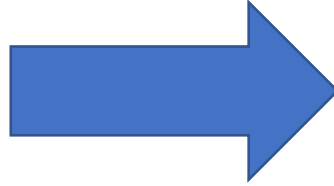
good
bad
hello
apple
morning

How I would call the program:

```
kaleel@CentralCompute:~$ ./test UnsortedText.txt
```

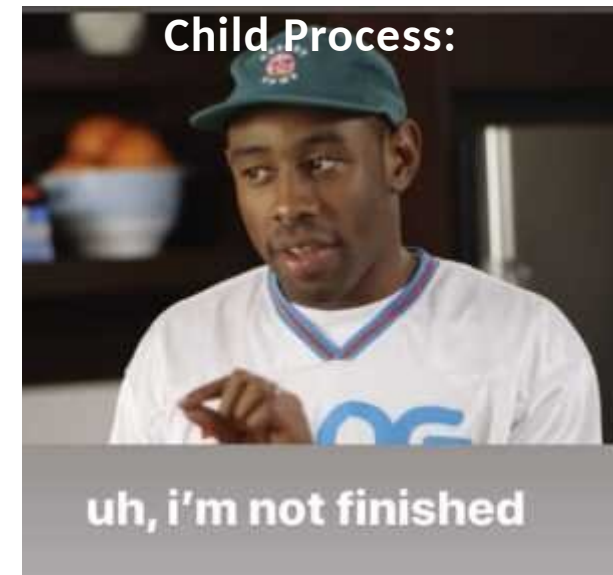
The Big Picture Example : Output

```
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <string.h>
10 #include <unistd.h>
11 #include <fcntl.h>
12 #include <ctype.h>
13 #include <sys/stat.h>
14 #define MAX_LINE 1024
15
16 int main(int argc, char *argv[])
17 {
18
19     if(argc!=2)
20     {
21         printf("Usage: %s filename\n", argv[0]);
22         return -1;
23     }
24     pid_t pid;
25     pid = fork();
26     if(pid == 0)
27     {
28
29         //Make sure when the file is created, the user has the read and write access
30         int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
31         if(fd < 0)
32         {
33             printf("Cannot open the file\n");
34             return -1;
35         }
36         dup2(fd, 1);
37         close(fd);
38         char * argv_list[] = {"wc", argv[1], NULL};
39
40         // the execv() only return if error occurred.
41         // The return value is -1
42         execvp("wc", argv_list);
43         printf("Something is wrong.\n");
44         exit(-1);
45     }
46     else{
47         printf("I'm the parent printing to the terminal...\n");
48     }
49     return 0;
50 }
```



```
kaleel@CentralCompute:~$ ./test UnsortedText.txt
I'm the parent printing to the terminal...
```

But is that all?



The Big Picture Example : Output 2

```
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <string.h>
10 #include <unistd.h>
11 #include <fcntl.h>
12 #include <ctype.h>
13 #include <sys/stat.h>
14 #define MAX_LINE 1024
15
16 int main(int argc, char *argv[])
17 {
18
19     if(argc!=2)
20     {
21         printf("Usage: %s filename\n", argv[0]);
22         return -1;
23     }
24     pid_t pid;
25     pid = fork();
26     if(pid == 0)
27     {
28
29         //Make sure when the file is created, the user has the read and write access
30         int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
31         if(fd < 0)
32         {
33             printf("Cannot open the file\n");
34             return -1;
35         }
36         dup2(fd, 1);
37         close(fd);
38         char * argv_list[] = {"wc", argv[1], NULL};
39
40         // the execvp() only return if error occurred.
41         // The return value is -1
42         execvp("wc",argv_list);
43         printf("Something is wrong.\n");
44         exit(-1);
45     }
46     else{
47         printf("I'm the parent printing to the terminal...\n");
48     }
49     return 0;
50 }
```



output - Notepad

File

Edit

View

5 5 35 UnsortedText.txt

One Last Question

If you understood the lecture, hopefully you should be able to answer this confidently:

A process would like to start a new program, and redirect stdout of the new process to a file.

Where & when should the file be opened?

Select the best option:

- A. Before fork(), in parent
- ☒ B. After fork() in parent
- ☒ C. Before exec in child (after fork())
- D. After exec in child (after fork())
- E. None of the above

Figure Sources

1. <http://marveltoynews.com/wp-content/uploads/2018/05/Avengers-Infinity-War-Hot-Toys-Doctor-Strange-Figure-with-Many-Arms.jpg>
2. <https://play-lh.googleusercontent.com/P2VMEenhplsubG2oWbvULGrs0GyyzLiDosGTg8bi8htRXg9Uf0eUtHiUjC28p1jgHzo>
3. https://upload.wikimedia.org/wikipedia/commons/thumb/f/fd/Microsoft_Office_Word_%282019%E2%80%93present%29.svg/1200px-Microsoft_Office_Word_%282019%E2%80%93present%29.svg.png
4. <https://imgflip.com/s/meme/Grandma-Finds-The-Internet.jpg>
5. https://i.kym-cdn.com/entries/icons/original/000/022/524/tumblr_o16n2kBlpX1ta3qyvo1_1280.jpg
6. <https://www.idlememe.com/wp-content/uploads/2021/10/tyler-the-creator-meme-idlememe-1-300x287.jpg>