Standard streams, file descriptors and I/O redirection

Module 2 self study material

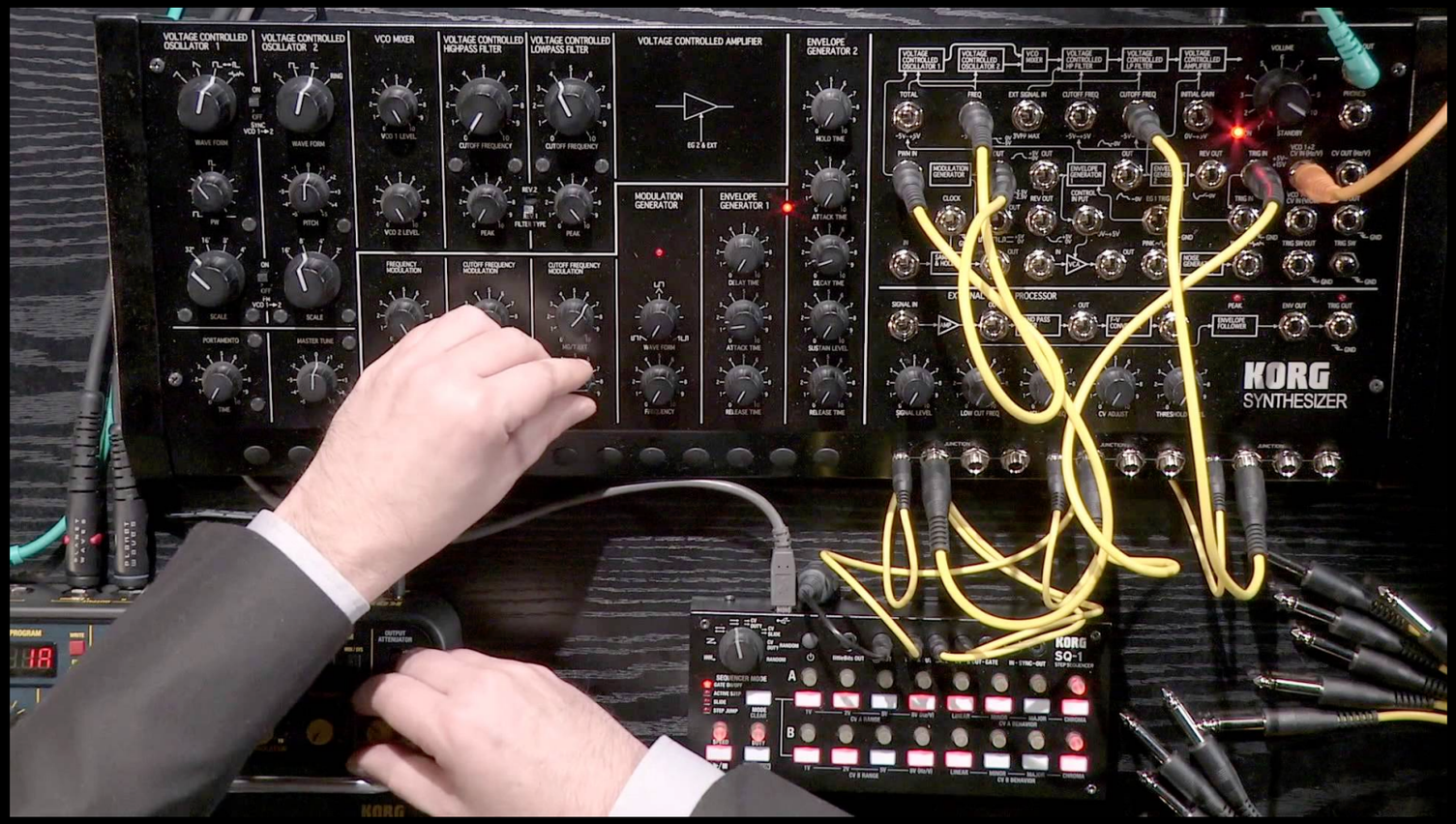


image: http://createdigitalmusic.com/2015/02/heres-korgs-99-sq1-sequencer-can-cant 2018-01-24

Operating systems 2020

1DT003, 1DT044 and 1DT096

Spring 2020 karl.marklund@it.uu.se Uppsala university

Standard I/O streams

Standard I/O streams are pre-connected input and output communication channels between a computer program and its environment when it begins execution.

Originally I/O happened via a physically connected system console (input via keyboard, output via monitor), but standard streams abstract this.

A DEC VT100 terminal. Photograph Creative Commons 2.0 by Jason Scott

Posix

POSIX (Portable Operating System Interface) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.

Standard I/O streams

POSIX defines the following three standard I/O streams.

Standard stream	Purpose	Default endpoint
stdin	**Standard input** is the stream of data data (often text) going into a program.	**terminal**
stdout	**Standard output** is the stream where a program writes its output data.	**terminal**
stderr	**Standard error** is another output stream typically used by programs to output error messages or diagnostics.	**terminal**

File descriptors

File descriptors

In Unix and related computer operating systems, a file descriptor is an **abstract indicator** used to **access** a **file** or other **input/output resources**, such as a **pipe** or network connection.

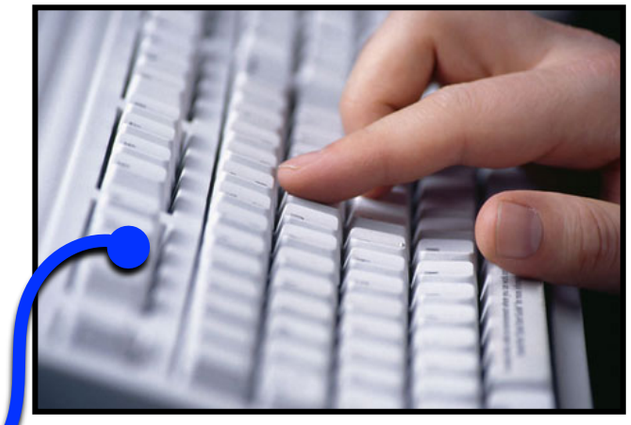
- File descriptors form part of the POSIX application programming interface.
- A file descriptor is a non-negative integer, represented in C programming language as the type int.

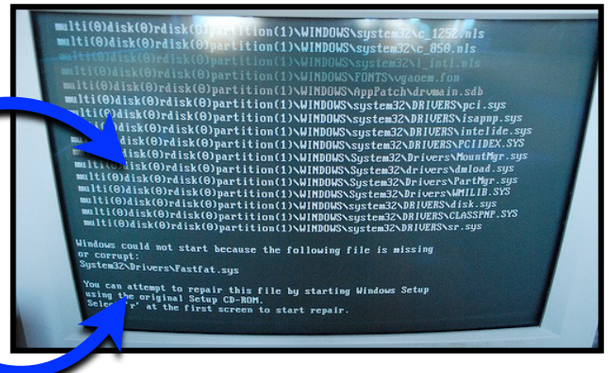
Source: https://en.wikipedia.org/wiki/File_descriptor 2016-01-25

Standard I/O streams and file descriptors

POSIX defines file descriptor values for the three standard I/O streams.

File descriptor value	Standard stream	Mode	Endpoint
0	stdin	read	terminal
1	stdout	write	terminal
2	stderr	write	terminal





File descriptor table

The kernel keeps a table with information about a process's open file descriptors.

User Space

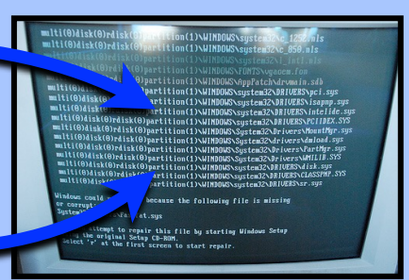
Kernel Space

A Process

By default, a process got three open descriptors:

- ★ stdin
- ★ stdout
- ★ stderr

File descriptor table			
Index	**Mode**	**Standard stream**	**Endpoint**
0	read	stdin	terminal
1	write	stdout	terminal
2	write	stderr	terminal



open()

Open is a system call that is used to open a new file and obtain its file descriptor.

open()

Open is a **system call** that is used to **open** a new **file** and **obtain** its **file descriptor.**

```
int open(const char *path, int oflags);
```

`int path`	The **relative** or **absolute path** to the file that is to be opened.
`int oflags`	A bitwise 'or' separated list of values that determine the method in which the file is to be opened (whether it should be read only, read/write, whether it should be cleared when opened, etc).
`return value`	The **file descriptor** for the opened file. The file descriptor returned is always the smallest integer greater than zero that is still available. If a negative value is returned, then there was an error opening the file.

Open a file for reading using the open() system call

User Space

A Process

```
int fd;
char buff[BUFF_SIZE];
fd = open("foo.txt", O_RDONLY);
```

1

When opening a file for reading, a descriptor number is returned. In this example, a descriptor with value is 3 returned.

Kernel Space

File descriptor table		
Index	Mode	Endpoint
0	read	terminal
1	write	terminal
2	write	terminal
3	read	foo.txt

File foo.txt

read()

Read is a system call used to read data from a file into a buffer.

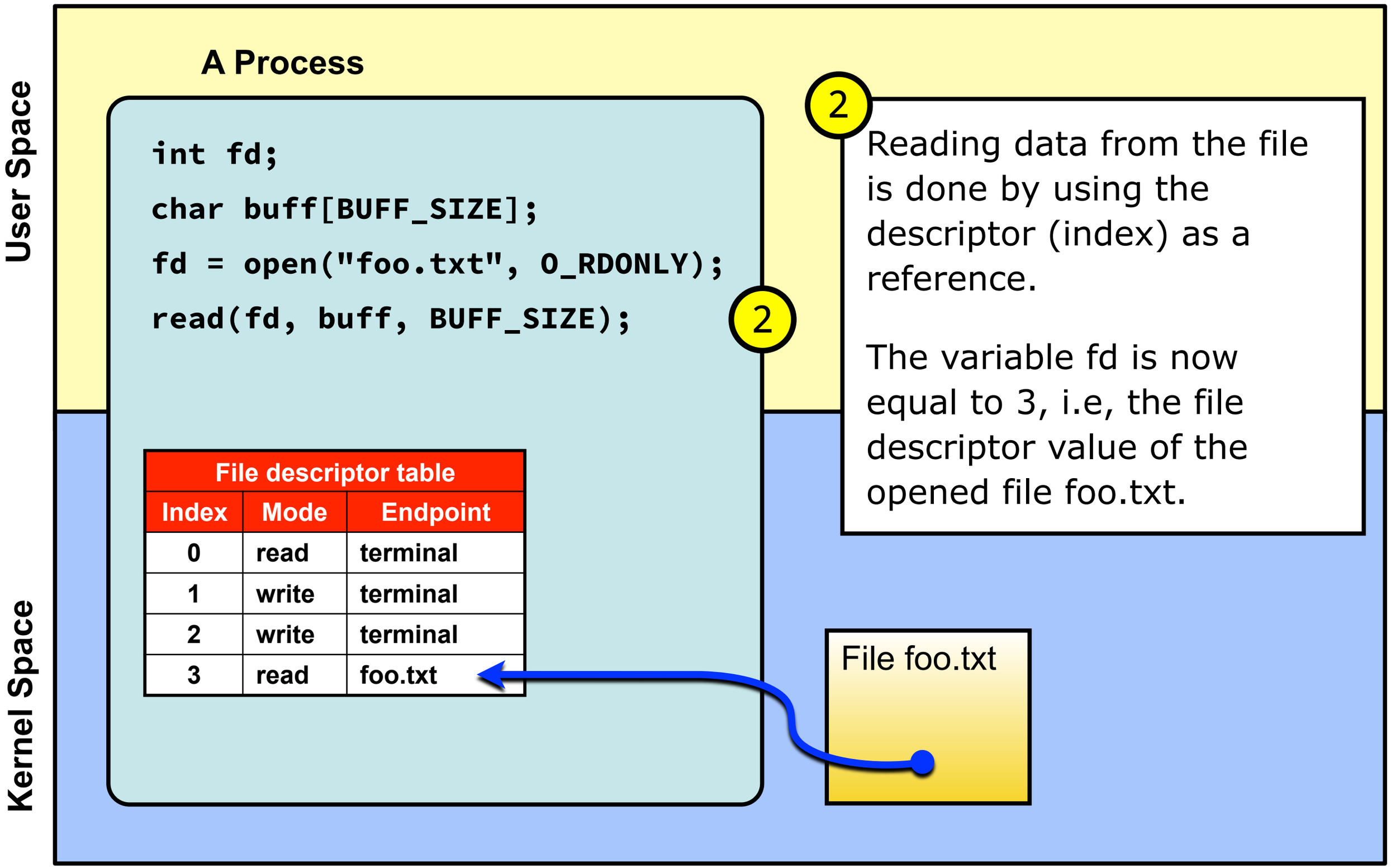
read()

Read is a system call used to read data from a file into a buffer.

```
size_t read(int fildes, void *buf, size_t nbytes);
```

`int fildes`	The **file descriptor** of where to read the input. You can either use a file descriptor obtained from the open system call, or you can use 0, 1, or 2, to refer to standard input, standard output, or standard error, respectively.
`void *buf`	A character array (**buffer**) where the read content will be stored.
`int nbytes`	The **number of bytes to read** before truncating the data. If the data to be read is smaller than nbytes, all data is saved in the buffer.
`return value`	On success, the number of bytes read is returned (**zero** indicates **end of file**). If value is **negative**, then the system call returned an **error**.

Reading data from a file using the read() system call



close()

Close is a system call that is used to close an open file descriptor.

close()

Close is a system call that is used to close an open file descriptor.

```
int close(int fildes);
```

`int fildes`	The **file descriptor** to be closed.
`return value`	Retuns a 0 upon success, and a -1 upon failure. It is important to check the return value, because some network errors are not returned until the file is closed.

Closing an open file using the close() system call

User Space

A Process

```
int fd;
char buff[BUFF_SIZE];
fd = open("foo.txt", O_RDONLY);
read(fd, buff, BUFF_SIZE);
close(fd);
```

3

After closing, the file cannot be accessed using the previously open file descriptor.

3

File descriptor table		
Index	**Mode**	**Endpoint**
0	read	terminal
1	write	terminal
2	write	terminal
~~3~~	~~read~~	~~foo.txt~~

File foo.txt

Kernel Space

dup()

The dup() system call creates a copy of a file descriptor, using the lowest-numbered unused descriptor for the new descriptor.

dup()

The dup() system call **creates a copy of a file descriptor**, using the lowest-numbered unused descriptor for the new descriptor.

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

`int fd`	The file descriptor that you are attempting to create an alias for.
`return value`	dup returns the value of the new file descriptor that it has created (which will always be the smallest available file descriptor). A negative return value means that an error occured.

Copy a file descriptor using the dup() system call

User Space

Kernel Space

A Process

```
int fd1, fd2;
char buff[BUFF_SIZE];

fd1 = open("foo.txt", O_RDONLY);

fd2 = dup(fd1);
```

Create a copy of the file descriptor **fd1** (index 3) and insert the copy at index 4 (**fd2**) in the file descriptor table.

File descriptor table		
Index	**Mode**	**Endpoint**
0	read	terminal
1	write	terminal
2	write	terminal
3	read	foo.txt
4	read	foo.txt

File foo.txt

I/O redirection

I/O redirection is a function common to most command-line interpreters, including the various Unix shells that can redirect standard streams to user-specified locations.

Redirection of stdout (1)

The **date** command (system program) writes the current date to **stdout**.

```
$ date
Tue Jan 26 23:49:05 CET 2016
$
```

Redirection of stdout (2)

```
$ date > date.txt
$
```

In most shell, the **>** operator is used to **redirect stdout** to a file.

Now, the output of calling date does not show up in the terminal (stdout). Where did it go?

```
$ ls
date.txt
$
```

Aha, now there exist a new file named date.txt

```
$ cat date.txt
Tue Jan 26 23:49:12
CET 2016
$
```

The result of the date command was redirected from stout to the file date.txt

dup2()

The dup2() system call copies one file descriptor entry to another entry in the file descriptor table.

int dup(int fd);

The dup() system call **creates a copy of a file descriptor** fd, using the lowest-numbered unused descriptor for the new descriptor.

int dup2(int src, int dst);

The `dup2()` system call performs the same task as dup(), but instead of using the lowest-numbered unused file descriptor for the copy, it uses the file descriptor number specified by dst.

If the file descriptor dst was previously open, it is silently closed before being reused.

dup2()

Copies one file descriptor entry to another entry in the file descriptor table.

```
int dup2(int src, int dst);
```

`int scr`	The source file descriptor. This remains open after the call to dup2.
`int dst`	The destination file descriptor. This file descriptor will point to the same file as scr after this call returns. If the file descriptor dst was previously open, it is silently closed before being reused.
`return value`	dup2 returns the value of the second parameter (dst) upon success. A negative return value means that an error occured

I/O redirection using dup2()

The dup2() system call is used to redirect I/O.

Redirect stdin using the `dup2()` system call

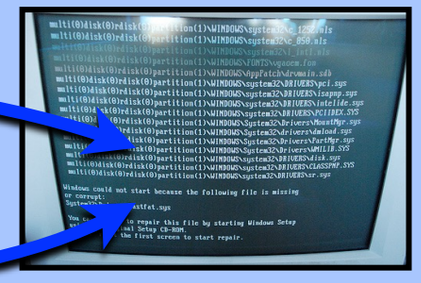
User Space

Kernel Space

A Process

```
int fd;
char buff[BUFF_SIZE];
fd = open("foo.txt", O_WRONLY); // fd == 3
```

File descriptor table		
Index	**Mode**	**Endpoint**
0	read	terminal
1	write	terminal
2	write	terminal
3	write	foo.txt



File foo.txt

Redirect stdin using the `dup2()` system call

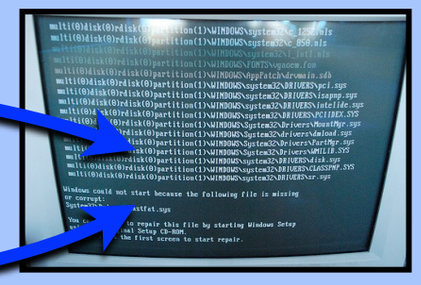
User Space

Kernel Space

A Process

```
int fd;
char buff[BUFF_SIZE];
fd = open("foo.txt", O_WRONLY); // fd == 3
dup2(fd, 1);
```

File descriptor table		
Index	**Mode**	**Endpoint**
0	read	terminal
1	write	terminal
2	write	terminal
3	write	foo.txt



File foo.txt

Redirect stdin using the `dup2()` system call

User Space

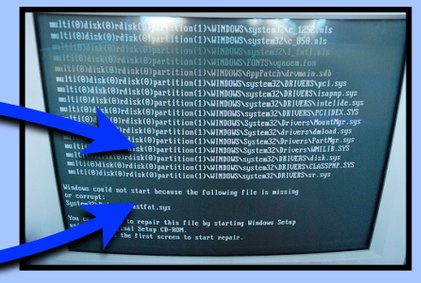
Kernel Space

A Process

```
int fd;
char buff[BUFF_SIZE];
fd = open("foo.txt", O_WRONLY); // fd == 3
dup2(fd, 1);
```

Old file descriptor table		
Index	**Mode**	**Endpoint**
0	**read**	**terminal**
1	**write**	**terminal**
2	**write**	**terminal**
3	**write**	**foo.txt**

Copy the row at index fd (3) to row at index 1.



File foo.txt

Redirect stdin using the dup2() system call

User Space

Kernel Space

A Process

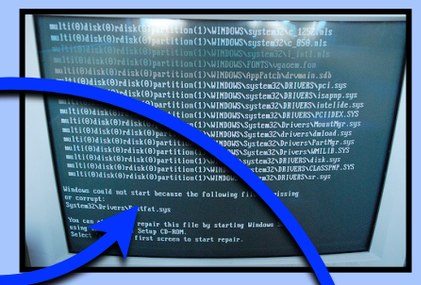
```
int fd;
char buff[BUFF_SIZE];
fd = open("foo.txt", O_WRONLY); // fd == 3
dup2(fd, 1);
```

Old file descriptor table		
Index	Mode	Endpoint
0	read	terminal
1	write	terminal
2	write	terminal
3	write	foo.txt

New file descriptor table		
Index	Mode	Endpoint
0	read	terminal
1	write	foo.txt
2	write	terminal
3	write	foo.txt

Copy the row at index fd (3) to row at index 1.

When writing to file descriptor 1, data will now be written to the file foo.txt instead of stdout.



File foo.txt

exec()

In Unix-like operating systems, the exec family of system calls runs an executable file in the context of an already existing process, replacing the previous executable.

exec()

In Unix-like operating systems, the **exec family of system calls** runs an executable file in the context of an already existing process, **replacing** the previous **executable**.

- ★ As a new process is not created, the process identifier (PID) does not change, but the machine code, data, heap, and stack of the process are replaced by those of the new program.
- ★ A file descriptor open when an exec call is made remain open in the new process image, unless was fcntled with FD_CLOEXEC. This aspect is used to specify the standard streams (stdin, stdout and stderr) of the new program.
- ★ As a result of exec, all data in the old program that were not passed to the new program, or otherwise saved, become lost.

Replacing the process image using the execvp() system call

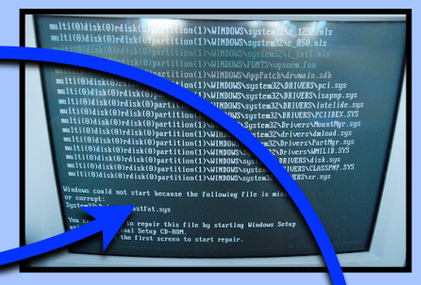
User Space

Kernel Space

A Process

```
int fd;
char buff[BUFF_SIZE];
fd = open("foo.txt", O_WRONLY); // fd == 3
dup2(fd, 1);
char* argv[] = {"ls", "-l", NULL};
execvp(argv[0], argv);
```

New file descriptor table		
Index	**Mode**	**Endpoint**
0	read	terminal
1	write	foo.txt
2	write	terminal
3	write	foo.txt



File foo.txt

Replacing the process image using the `execvp()` system call

User Space

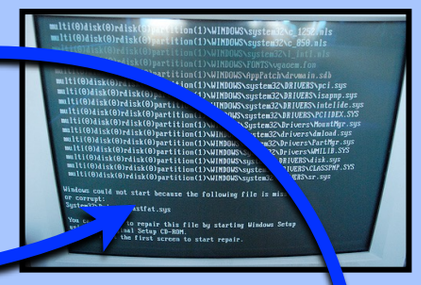
Kernel Space

A Process

Executable for the ls system program

New file descriptor table		
Index	**Mode**	**Endpoint**
0	read	terminal
1	write	terminal
2	write	terminal
3	write	foo.txt

Output written by **ls** to file descriptor 1 (stdout) will now be written to the file foo.txt.



File foo.txt

Replacing the process image using the execvp() system call

User Space

Kernel Space

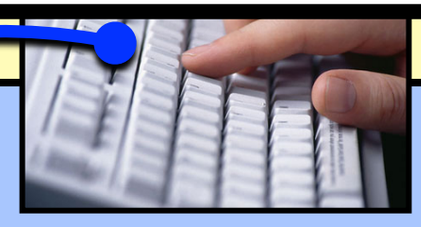
A Process

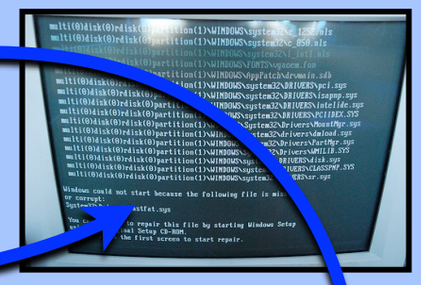
```
int fd;
char buff[BUFF_SIZE];
fd = open("foo.txt", O_WRONLY); // fd == 3
dup2(fd, 1);
char* argv[] = {"ls", "-l", NULL};
execvp(argv[0], argv);
```

New file descriptor table		
Index	**Mode**	**Endpoint**
0	read	terminal
1	write	foo.txt
2	write	terminal
3	write	foo.txt

First we use **`dup2()`** to change the file descriptor table in order to redirect the endpoint of the **stdout** file descriptor (1) to write to a file instead of writing to the terminal.

Next we used **`exec()`** to replace the process memory image to execute the **`ls`** system program.





The file descriptor table is not affected by **`exec()`**.

The **`ls`** program writes to **stdout**. By default, the stdout has the terminal as its endpoint but since stdout has been redirected to the file, the output is now written to the file.

The **`ls`** program has no way of knowing where the data written to **stdout** will actually end up.

File foo.txt

General problem with exec() and file descriptors

User Space

Kernel Space

A Process

```
// Somewhere in the program
int fd;
fd = open("bar.txt", O_WRONLY);
exec();
```

New file descriptor table		
Index	**Mode**	**Endpoint**
0	read	terminal
1	write	terminal
2	write	terminal
. . .		
41	read	foo.txt
42	write	foo.txt

File foo.txt

File bar.txt

General problem with exec() and file descriptors

User Space

A Process

- ★ After **exec()**, the executable is replaced.
- ★ Can **bar.txt** still be accessed using file descriptor **42**?

Yes, after **exec()** file descriptor **42** can still be used to access **bar.txt**.

But, the variable **fd** holding the value **42** before **exec()** does not exist anymore!

Kernel Space

New file descriptor table		
Index	**Mode**	**Endpoint**
0	**read**	**terminal**
1	**write**	**terminal**
2	**write**	**terminal**
...		
41	**read**	**foo.txt**
42	**write**	**foo.txt**