

# Standard streams, file descriptors and I/O redirection

## Module 2 self study material

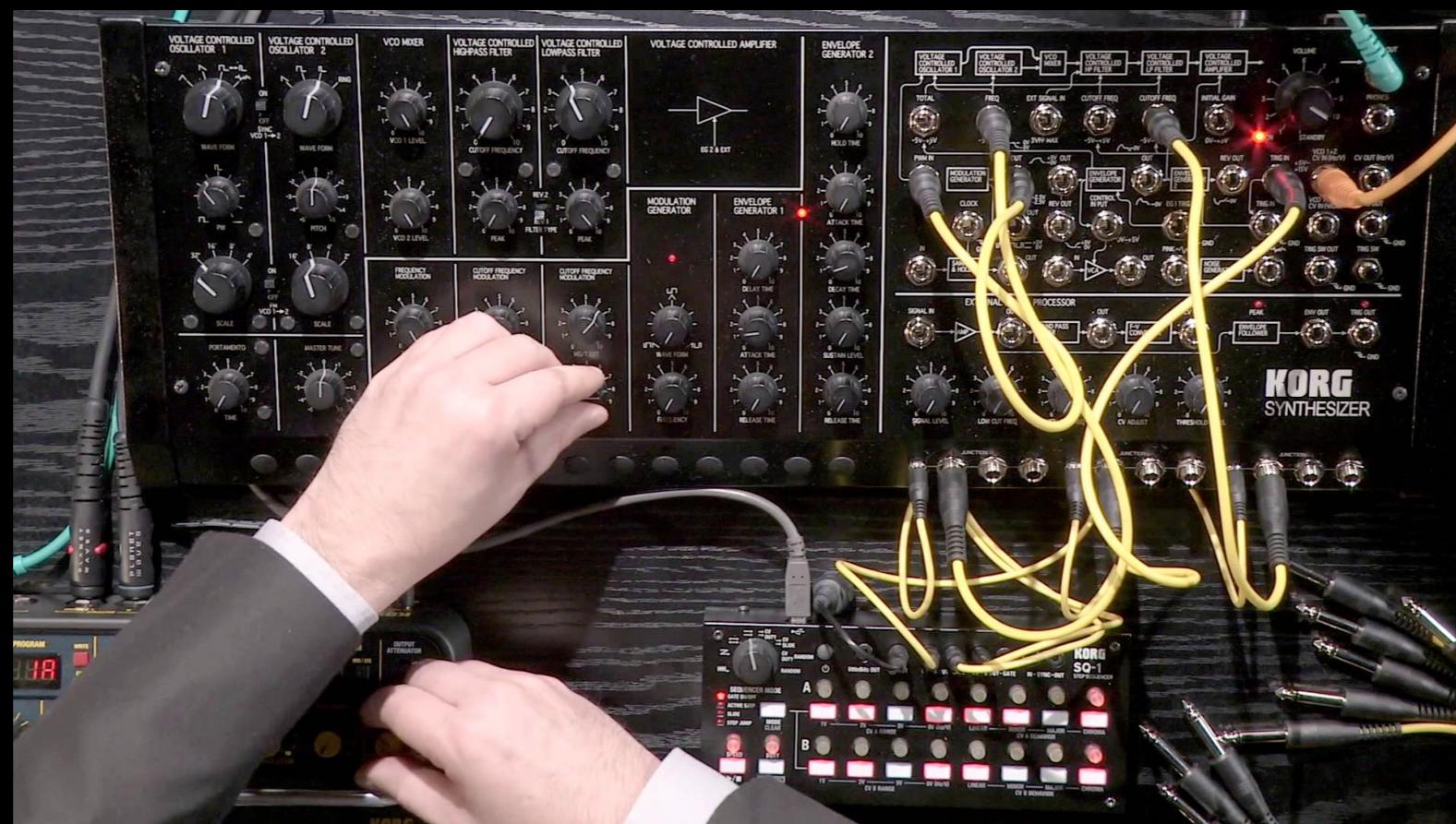


image: <http://createdigitalmusic.com/2015/02/heres-korgs-99-sq1-sequencer-can-cant>

2018-01-24

## Operating systems 2018

**1DT044 and 1DT096**

# **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	<b>Standard input</b> is the stream of data (often text) going into a program.	terminal
stdout	<b>Standard output</b> is the stream where a program writes its output data.	terminal
stderr	<b>Standard error</b> is another output stream typically used by programs to output error messages or diagnostics.	terminal

# File descriptors

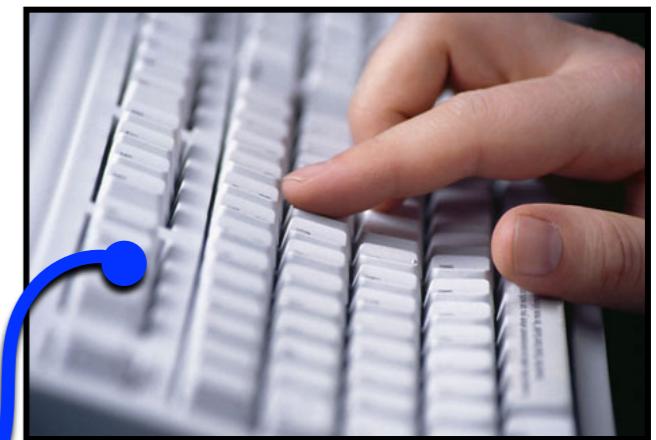
In Unix and related computer operating systems, a file descriptor is an **abstract indicator** used to **access** a file or other **input/output resources**, such as a **pipe** or network connection.

- ★ 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.

# 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.

## A Process

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

By default, a process got three open descriptors:

- ★ stdin
- ★ stdout
- ★ stderr



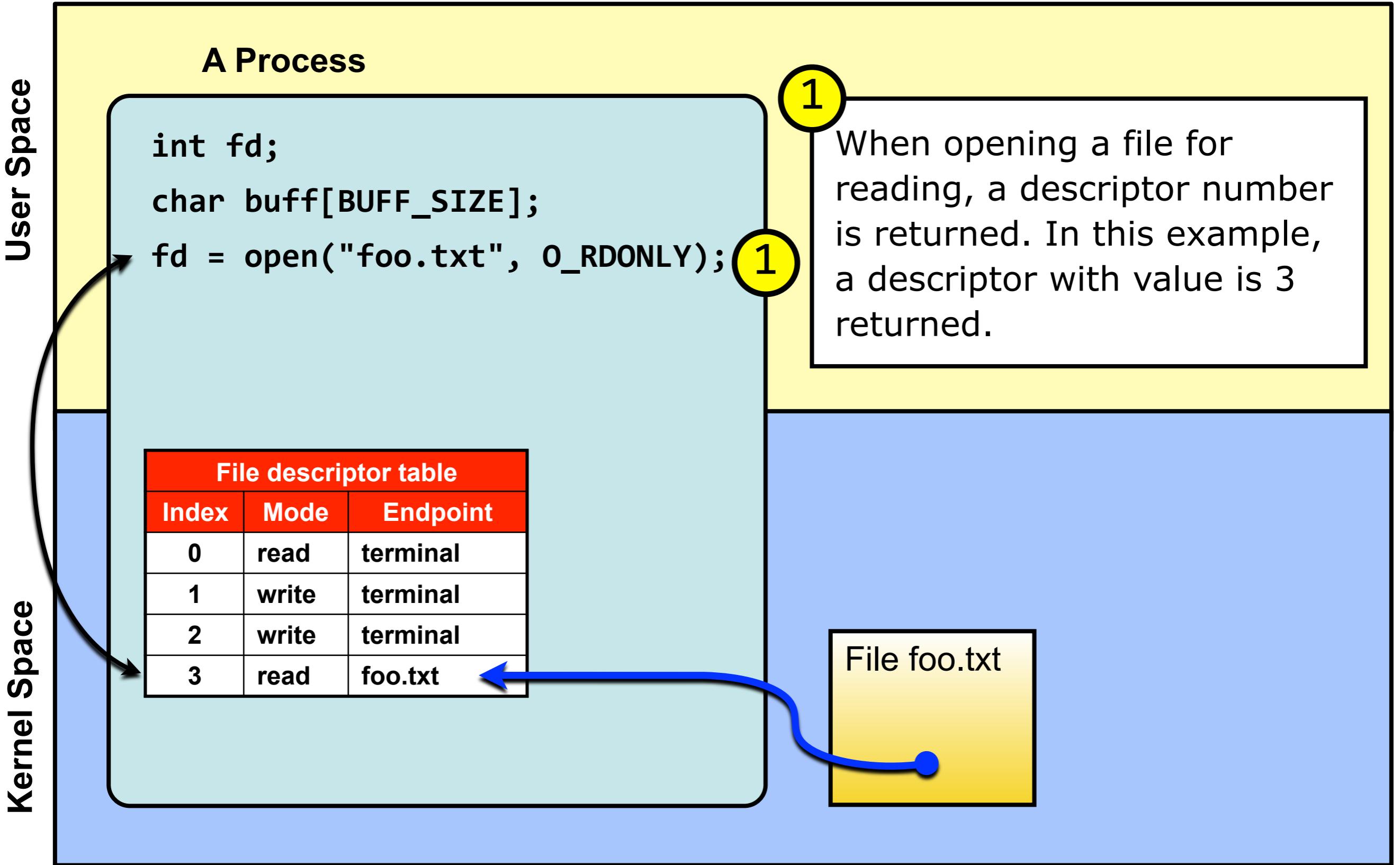
# open()

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);
```

<b>int path</b>	The <b>relative or absolute path</b> to the file that is to be opened.
<b>int oflags</b>	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).
<b>return value</b>	The <b>file descriptor</b> 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



# 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);`

<code>int <b>fildes</b></code>	The <b>file descriptor</b> 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.
<code>void *<b>buf</b></code>	A character array ( <b>buffer</b> ) where the read content will be stored.
<code>int <b>nbytes</b></code>	The <b>number of bytes to read</b> before truncating the data. If the data to be read is smaller than nbytes, all data is saved in the buffer.
<code>return value</code>	On success, the number of bytes read is returned ( <b>zero</b> indicates <b>end of file</b> ). If value is <b>negative</b> , then the system call returned an <b>error</b> .

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

User Space

## A Process

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

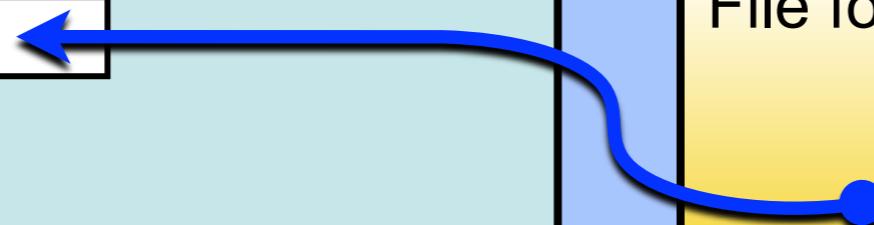
2

Reading data from the file is done by using the descriptor (index) as a reference.

Kernel Space

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

File foo.txt



# close()

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

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

<b>int fildes</b>	The <b>file descriptor</b> to be closed.
<b>return value</b>	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);
```

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

3

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

3

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.

```
int dup(int fildes);
```

<b>int <b>fildes</b></b>	The file descriptor that you are attempting to create an alias for.
<b>return value</b>	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 occurred.

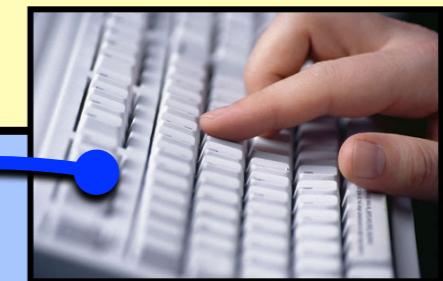
# Copy a file descriptor using the `dup()` system call

## A Process

```
int fd1, fd2;  
char buff[BUFF_SIZE];  
  
fd1 = open("foo.txt", O_RDONLY);  
  
fd2 = dup(fd1);
```

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

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



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

```
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.

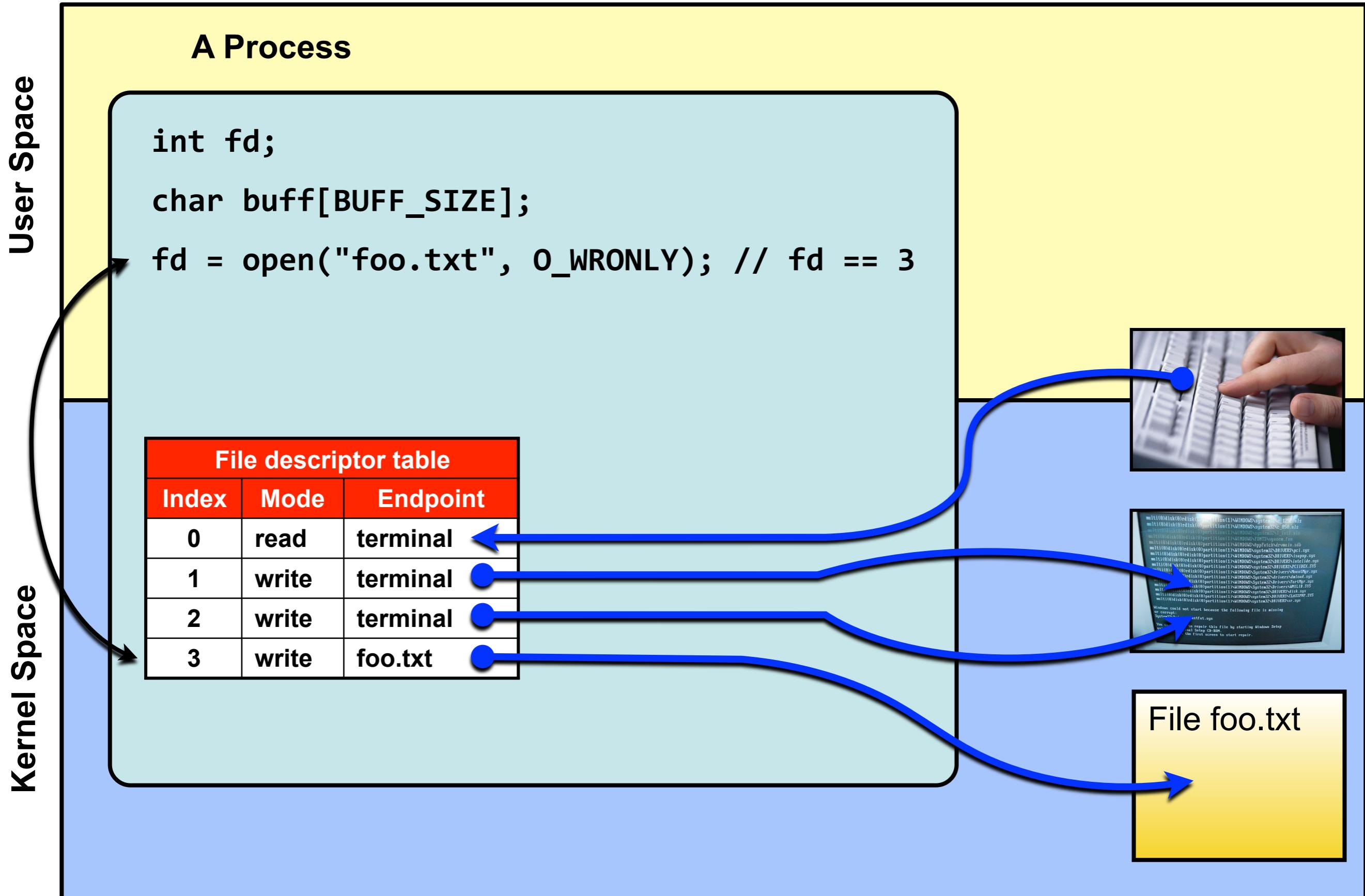
```
int dup2(int src, int dst);
```

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

# I/O redirection

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

# Redirect stdin using the `dup2()` system call



# Redirect stdin using the `dup2()` system call

## 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

```
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

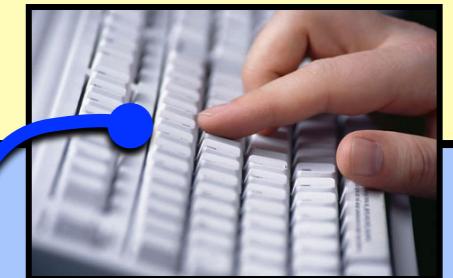
```
A Process  
  
int fd;  
  
char buff[BUFF_SIZE];  
  
fd = open("foo.txt", O_WRONLY); // fd == 3  
  
dup2(fd, 1);
```

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.

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

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**.

- ★ 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

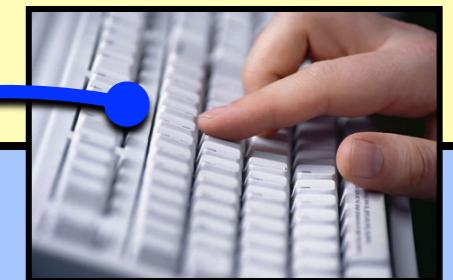
## A Process

```
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

User Space

Kernel Space



File foo.txt

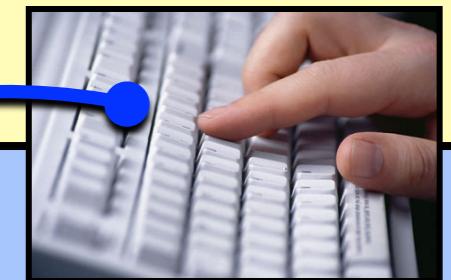
# Replacing the process image using the `execvp()` system call

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`

# General problem with `exec()` and file descriptors

User 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

Kernel Space

# General problem with `exec()` and file descriptors

## A Process

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

New file descriptor table		
Index	Mode	Endpoint
0	read	terminal
1	write	terminal
2	write	terminal

...

41	read	foo.txt
42	write	foo.txt

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

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

File foo.txt

File bar.txt