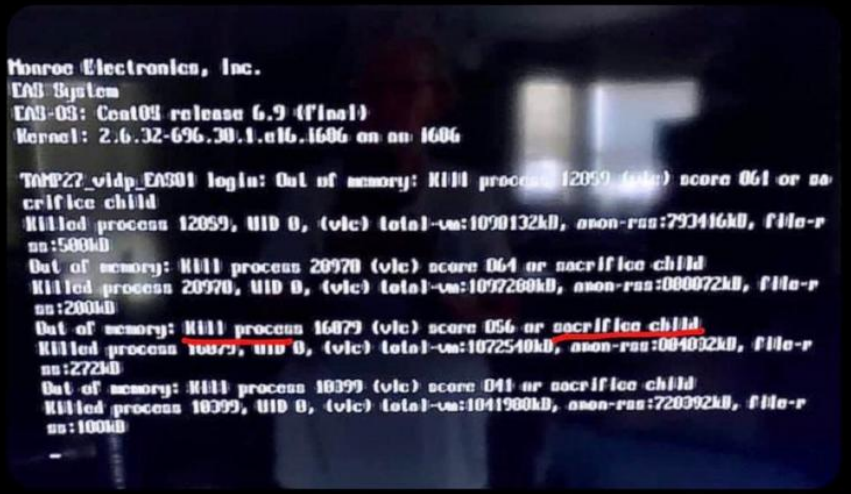


Samantha Morlote
@therealsmorlote

❗ ❗ ❗ THE BELOW MESSAGE WAS DISPLAYED ON SOMEONE'S TV. CODE WAS DISPLAYED, MIXED WITH THE WORDS "KILL PROCESS OR SACRIFICE CHILD" ❗ ❗ ❗
❗ ❗ ARE YOU ALL SEEING THIS ❗ ❗



2:58 PM · 8/11/21 · Twitter for iPhone

TODO

- Assignment 5 assigned due November 20

Overview

- Redirecting process output

Exec system call

- Exec overwrites current program with new program
  - Replaces code with new code
  - Resets all other memory
  - Resets all registers
- Exec *never returns*

How do we wait for our children to die?

```
pid_t waitpid(pid_t pid, int *_Nullable wstatus, int options);
```

- `pid`: pid of process to wait for
  - If `-1`, wait for *any* process
- `wstatus`: an “output pointer” where waitpid will write the dead child’s exit status
  - Can be `NULL` if we don’t care
- `options`: we will discuss this more later
  - 0 for no options

Reaping

- When we `wait()` on a process we *reap* it



```
while (true) {
    Print command prompt.
    Read command line from user.
    Parse command line.
    If command is built-in, execute it.
    Else, fork process
        in child:
            Execute requested command with exec
                                (never returns)
        in parent:
            Wait for child to complete with waitpid
}
```

What else can shells do?

- Shells can redirect input and output
- Lab 4:
  - `$ ./hex2raw -i 1-hiss.txt > 1-hiss.bytes`
  - `$ ./bytebomb < 1-hiss.bytes`
- May have also seen somewhere:
  - `$ something >> somethingelse.txt`

File modes

- <: open for reading
- >: open for writing
- >>: open for appending

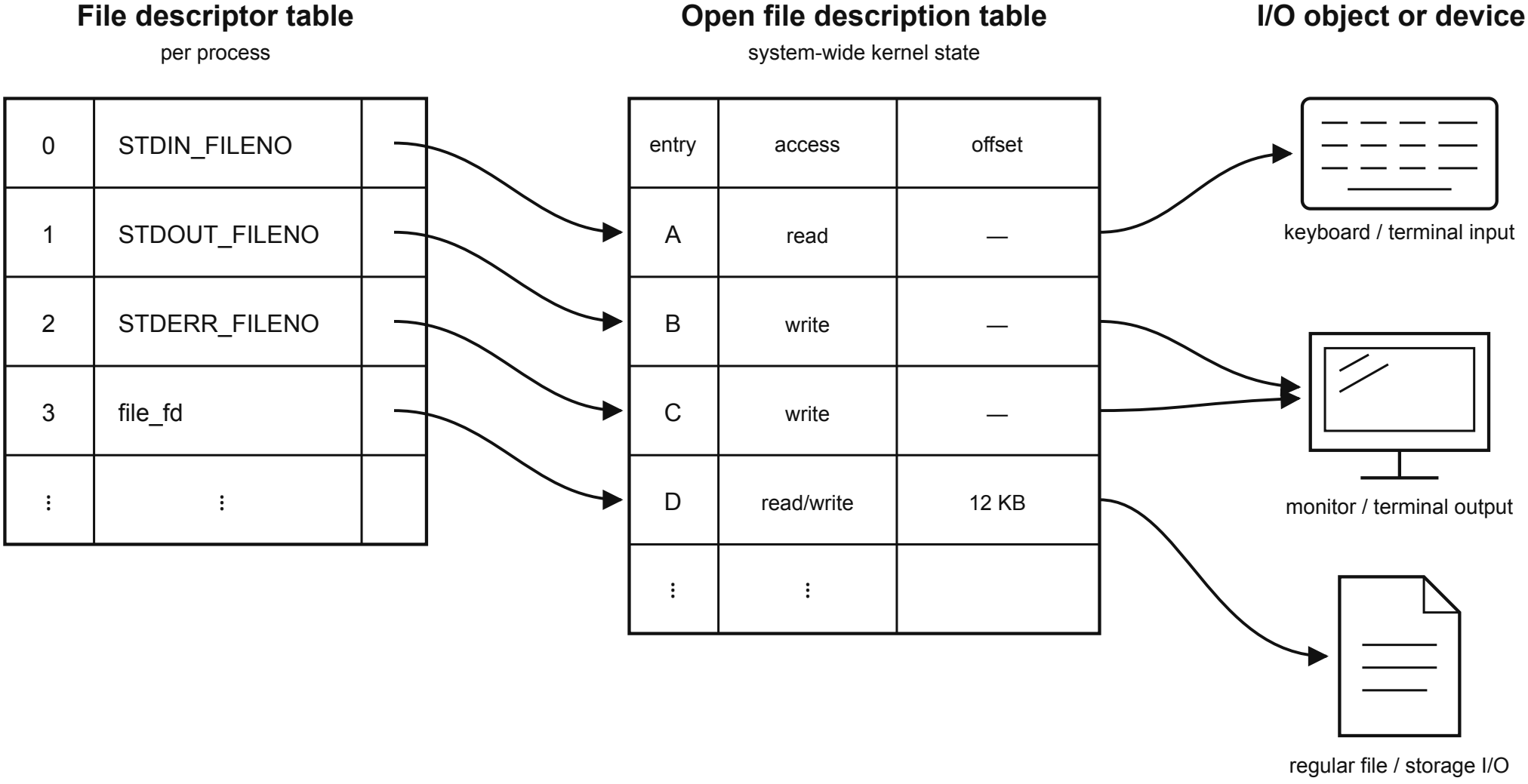
File modes

Mode	Description	Caveats
`"r"` (read)	Opens the file in read-only mode, starting at the beginning	Fails if the file does not exist
`"w"` (write)	Opens the file in write-only mode, starting at the beginning	Creates a new file if it does not exist or resets the file to be empty if it does exist
`"a"` (append)	Opens the file in write-only mode, starting at the end	Creates a new file if it does not exist

You can add a `'+'` to any mode to open the file in read-and-write mode (`"r+"`, `"w+"`, `"a+"`). However, the caveats will still apply.

File handle vs. file descriptor

- File handles are opaque wrappers around file descriptors
- System calls work directly with file descriptors
- File descriptors are lower level: they are ints
- A *file descriptor* is an index into the processes's file descriptor table


File descriptor table
per process
Open file description table
system-wide kernel state
I/O object or device
0
STDIN_FILENO
1
STDOUT_FILENO
2
STDERR_FILENO
3
file_fd
entry
access
offset
A
read
—
B
write
—
C
write
—
D
read/write
12 KB
keyboard / terminal input
monitor / terminal output
regular file / storage I/O

Our working model for a process

- Registers
- Memory
- PID
- File descriptor table 🆕

File handle vs. file descriptor

- Return file handles:
  - In Python, `open()`
  - In C, `fopen()`
- Return file descriptors:
  - In Python, `os.open()`
  - In C, `open()`
- The ones that return file handles wrap the ones that return file descriptors

files.py

```
with open('readme.txt', 'r') as f:
    print(f.fileno())
```

```
$ python3 files.py
3
```

files.py

```
with open('writeme.txt', 'w') as f:
    print(f.fileno())
```

```
$ python3 files.py
3
```

files.py

```
with open('appendtome.txt', 'a') as f:
    print(f.fileno())
```

```
$ python3 files.py
3
```

files.c

```c
#include <stdio.h>

int main(void) {
    FILE *file = fopen("readme.txt", "r");
    if (file != NULL) {
        printf("%d\n", fileno(file));
        fclose(file);
    }
    return 0;
}
```

```
$ ./foo
3
```

files.c

```c
#include <stdio.h>

int main(void) {
    FILE *file = fopen("writeme.txt", "w");
    if (file != NULL) {
        printf("%d\n", fileno(file));
        fclose(file);
    }
    return 0;
}
```

```
$ ./foo
3
```

files.c

```c
#include <stdio.h>

int main(void) {
    FILE *file = fopen("appendtome.txt", "a");
    if (file != NULL) {
        printf("%d\n", fileno(file));
        fclose(file);
    }
    return 0;
}
```

```
$ ./foo
3
```

open() example

- Read:
  - `int fd = open(filename, O_RDONLY);`
- Write:
  - `int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0666);`
- Append:
  - `int fd = open(filename, O_WRONLY | O_CREAT | O_APPEND, 0666);`

Other file system calls

```
ssize_t read(int fd, void buf[.count], size_t count);

ssize_t write(int fd, const void buf[.count], size_t count);

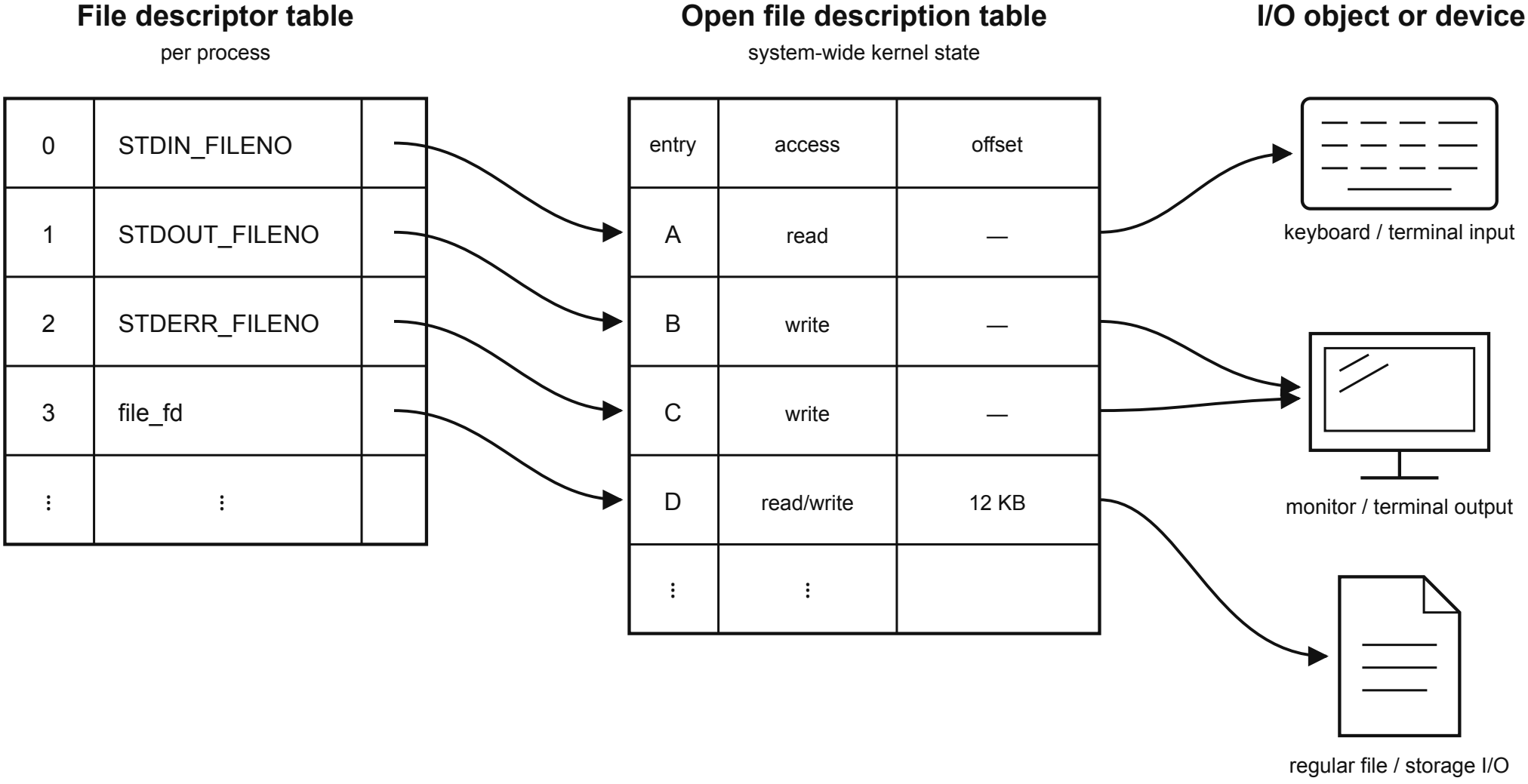
int close(int fd);

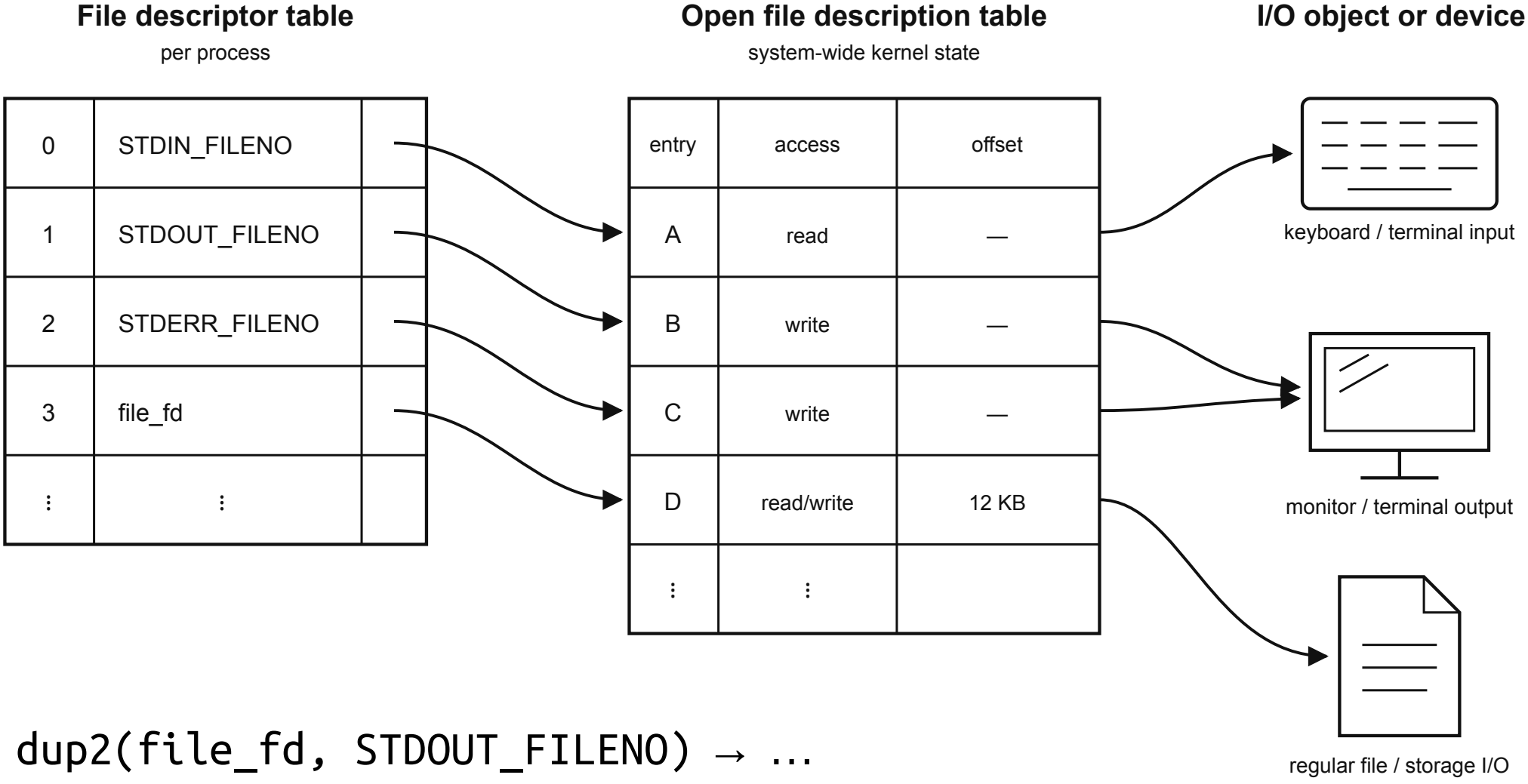
int dup2(int oldfd, int newfd);
```

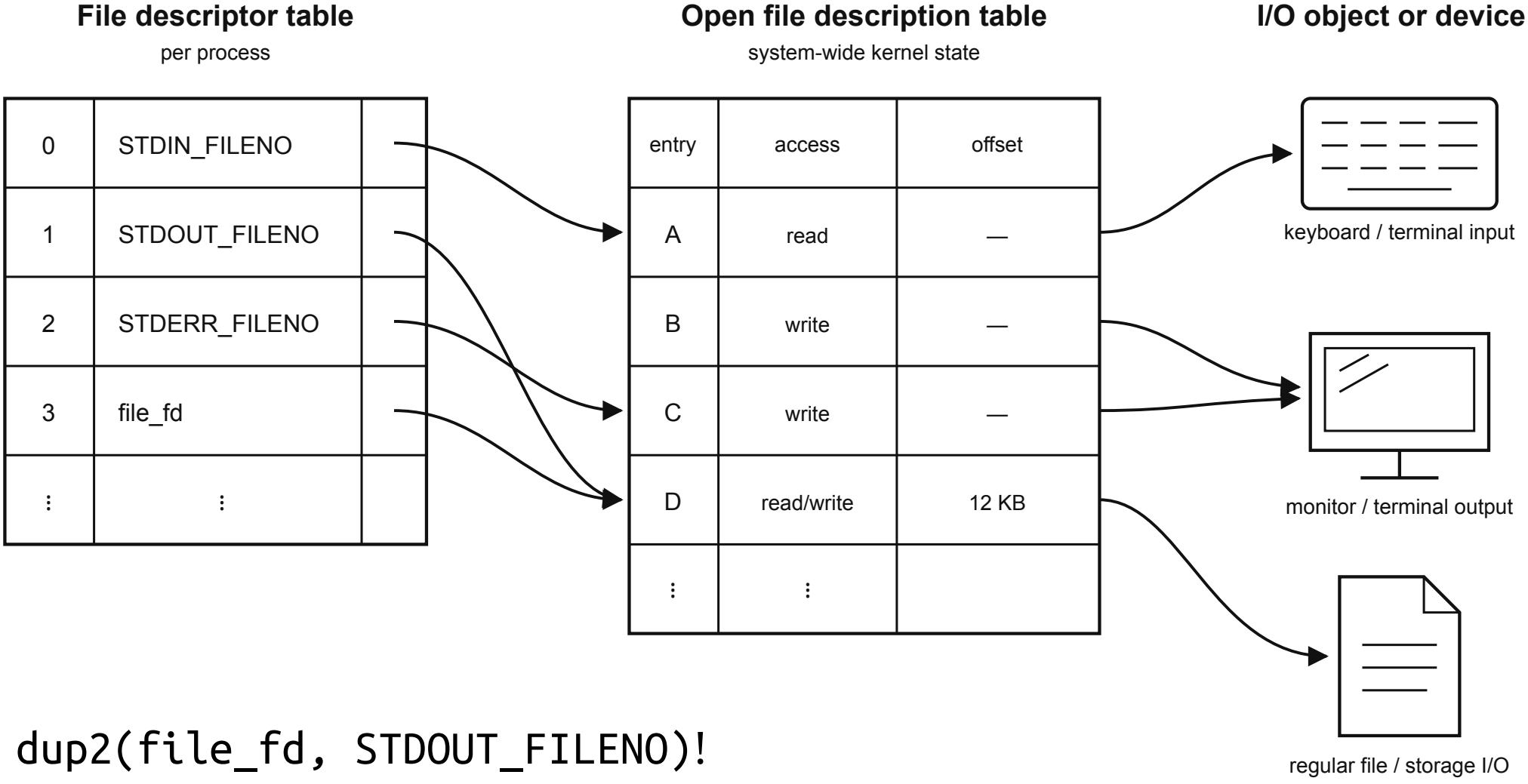
dup2

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

- Duplicates an entry in the file descriptor table
- Replaces the second argument with the first


File descriptor table
per process
Open file description table
system-wide kernel state
I/O object or device
0
STDIN_FILENO
1
STDOUT_FILENO
2
STDERR_FILENO
3
file_fd
entry
access
offset
A
read
—
B
write
—
C
write
—
D
read/write
12 KB
keyboard / terminal input
monitor / terminal output
regular file / storage I/O


File descriptor table
per process
Open file description table
system-wide kernel state
I/O object or device
0
STDIN_FILENO
1
STDOUT_FILENO
2
STDERR_FILENO
3
file_fd
entry
access
offset
A
read
—
B
write
—
C
write
—
D
read/write
12 KB
keyboard / terminal input
monitor / terminal output
regular file / storage I/O
dup2(file_fd, STDOUT_FILENO) → ...


File descriptor table
per process
Open file description table
system-wide kernel state
I/O object or device
0
STDIN_FILENO
1
STDOUT_FILENO
2
STDERR_FILENO
3
file_fd
entry
access
offset
A
read
—
B
write
—
C
write
—
D
read/write
12 KB
keyboard / terminal input
monitor / terminal output
regular file / storage I/O
dup2(file_fd, STDOUT_FILENO)!

hellodup2

- Let’s add a `dup2()` call to `hello.c`

```
while (true) {
    Print command prompt.
    Read command line from user.
    Parse command line.
    If command is built-in, execute it.
    Else, fork process
        in child:
            Execute requested command with exec
                                (never returns)
        in parent:
            Wait for child to complete with waitpid
}
```

In shells

```
if (fork() == 0) { /* is child process */
    execvp(...);
}
waitpid(...)
```

In shells

```
if (fork() == 0) { /* is child process */
    if (is_stdin_redirected) {
        int fd = open(...);
        dup2(fd, STDIN_FILENO);
        close(fd);
    }
    if (is_stdout_redirected) {
        int fd = open(...);
        dup2(fd, STDOUT_FILENO);
        close(fd);
    }
    execvp(...);
}
waitpid(...)
```

Syscall exercises

Open examples:

- `int fd = open(filename, O_RDONLY);  /* "r" */`
- `int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0666);  /* "w" */`
- `int fd = open(filename, O_WRONLY | O_CREAT | O_APPEND, 0666); /* "a" */`

Dup2 examples:

- `dup2(fd, STDIN_FILENO);`
- `dup2(fd, STDOUT_FILENO);`

So what else can shells do?

- Normally commands are run in the *foreground*
- That means that the shell *waits* for your command to finish before prompting
- They can also run things in the background!

Background processes

- `$ sleep 5`
- `[wait 5 seconds]`
- `$`

Background processes

- `$ sleep 5 &`
- `[immediately]`
- `$`

Background processes

- `$ sleep 5 &`
- `[immediately]`
- `$ sleep 5 &`
- `[immediately]`
- `...`

- Note: multiple processes can be in background!

Background processes

- Shell prints notice when command is finished
- Bash defers printing the notice until next prompt…
- But some shells print immediately!

In shells

```
if (fork() == 0) { /* is child process */
    /* dup2() as needed */
    execvp(...);
}
waitpid(...)
```

In shells

```
if (fork() == 0) { /* is child process */
    /* dup2() as needed */
    execvp(...);
}
if (is_foreground) {
    waitpid(...)
}
```

Next up

- How do processes reap background processes?

What does this have to do with zombies?



When your parents die, which daemon adopts you?

TODO

- Assignment 5 assigned due November 20