CIS 5480
Recitation 1

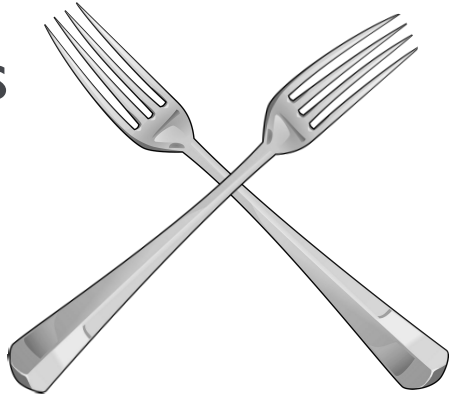
Thursday, February 6 2025

Penn Engineering
University of Pennsylvania

Agenda

- Fork
- Exec
- File Descriptors
- Pipes

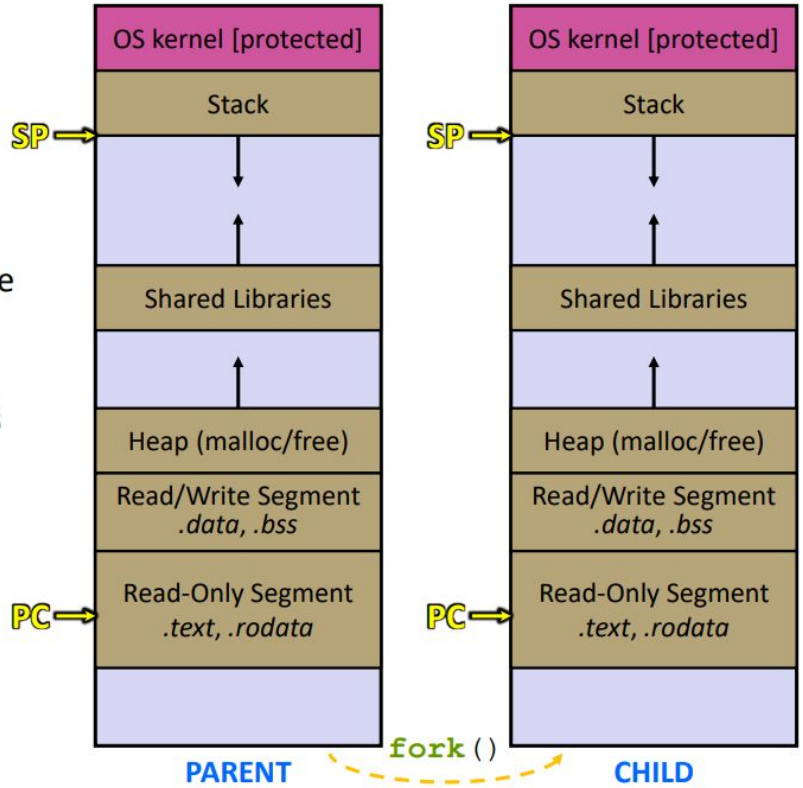
Fork



- pid_t fork(): duplicates the calling process
  - New process : child
  - Calling process: parent
- Returns
  - PID of the child process to the parent process
  - 0 in the child process
- Entire memory space is replicated in child process

Fork

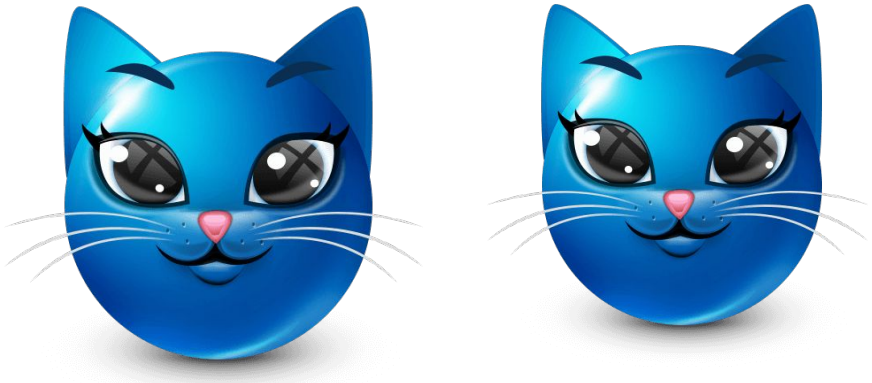
- Fork causes the OS to clone the address space
  - The *copies* of the memory segments are (nearly) identical
  - The new process has *copies* of the parent's data, stack-allocated variables, open file descriptors, etc.



Fork - Basic Example

```
5   int main() {
6       pid_t pid = fork();
7
8       if (pid == 0) {
9           printf("Meow");
10          exit();
11      } else {
12          pid_t pid = fork();
13
14          if (pid == 0) {
15              printf("Meow");
16          }
17      }
18      printf("Meow )
19      return;
20  }
21
```

Q: How many times is "Meow" printed?



Fork - Tricky Example

Q: What happens?

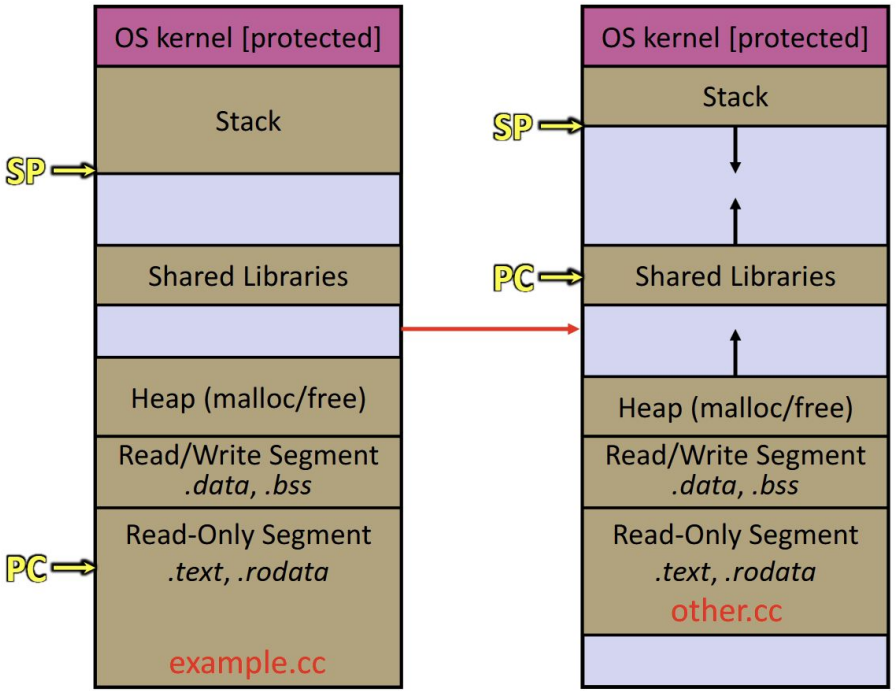
```c
int global_num = 1;

void function() {
    global_num++;
    printf("global_num = %d\n",  global_num);
}

int main() {
    printf("global_num = %d\n", global_num);

    pid_t id1 = fork();

    if (id1 == 0) {
        function();
        pid_t id2 = fork();
        if (id2 == 0) {
            function();
        } else {
            global_num += 3;
            printf("global_num = %d\n", global_num);
        }
        return EXIT_SUCCESS;
    }

    global_num *= 2;
    printf("global_num = %d\n", global_num);

    fork();

    printf("global_num = %d\n", global_num);

    return EXIT_SUCCESS;
}
```

Exec

- execve(char *pathname, char *argv[], char *envp[])
  - Pathname: executes program at this path
  - Argv: arguments
  - Envp: Environment variables
    - You don't really need to worry about these
  - You can view exec as replacing the current program image with the new program
  - Does not return

Exec

- Exec takes a process and discards or “resets” most of it



NOTE that the following DO change
- The stack
- The heap
- Globals
- Loaded code
- Registers

NOTE that the following do NOT change
- Process ID
- Open files
- The kernel

Execve

- Execution steps
  - Find the path file
  - Check that the file is actually executable, load
  - Set up the new stack (argv stored in mem)
  - Transfer control to the new program
    - CPU registers are reset
    - Instruction pointer set to start of new code
  - Fds, pid preserved; signal handlers reset

Exec Example

```
int main() {
        char *argv[] = { NULL, "-l", NULL };
        char *envp[] = { NULL };
        execve("/bin/ls", argv, envp);
        perror("execve failed");
        return 1;
}
```

- What is wrong here?

Exec Fix

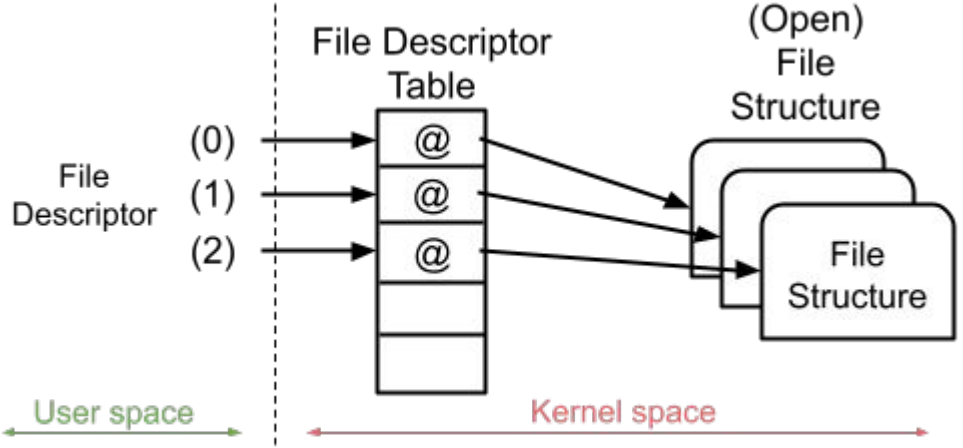
```
int main() {
        char *argv[] = { "ls", "-l", NULL }; // was missing argv[0]
        char *envp[] = { NULL };
        execve("/bin/ls", argv, envp);
        perror("execve failed");
        return 1;
}
```

- What is wrong here?

File Descriptors

File descriptors are process-unique identifiers to file-like objects:

- A regular txt file
- Terminal inputs/outputs
- Pipes



File Descriptors

Terminal inputs/outputs

- Standard input: stdin (default fd = 0 in Unix)
- Standard output: stdout (default fd = 1 in Unix)
- Standard error: stderr (default fd = 2 in Unix)

File Descriptors

In C, file descriptors are a type of int

The process of dealing with a file is generally:

- Open the file (generate the file descriptor) using open(2)
- Interact with the file using read(2) and write(2)
- Close the file (unassign the file descriptor) when the process is done with it using close(2)

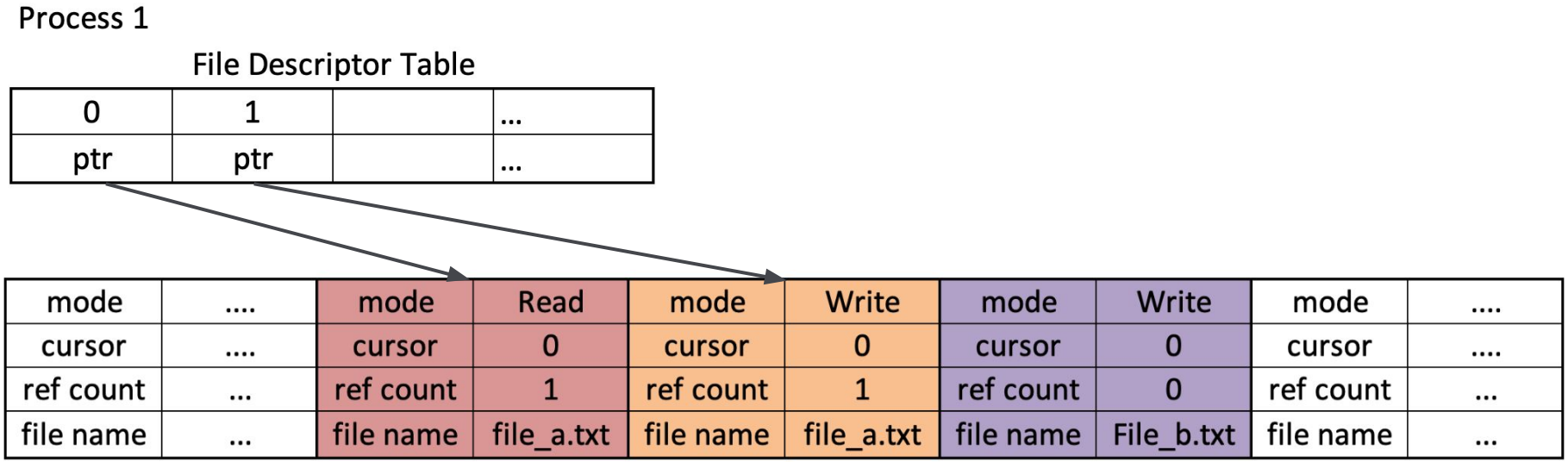
File Descriptors

- Open File Table stores the information about all the files that are open while the OS is running.

mode	....	mode	Read	mode	Write	mode	Write	mode	....
cursor	....	cursor	0	cursor	0	cursor	0	cursor	....
ref count	...	ref count	1	ref count	1	ref count	0	ref count	...
file name	...	file name	file_a.txt	file name	file_a.txt	file name	File_b.txt	file name	...

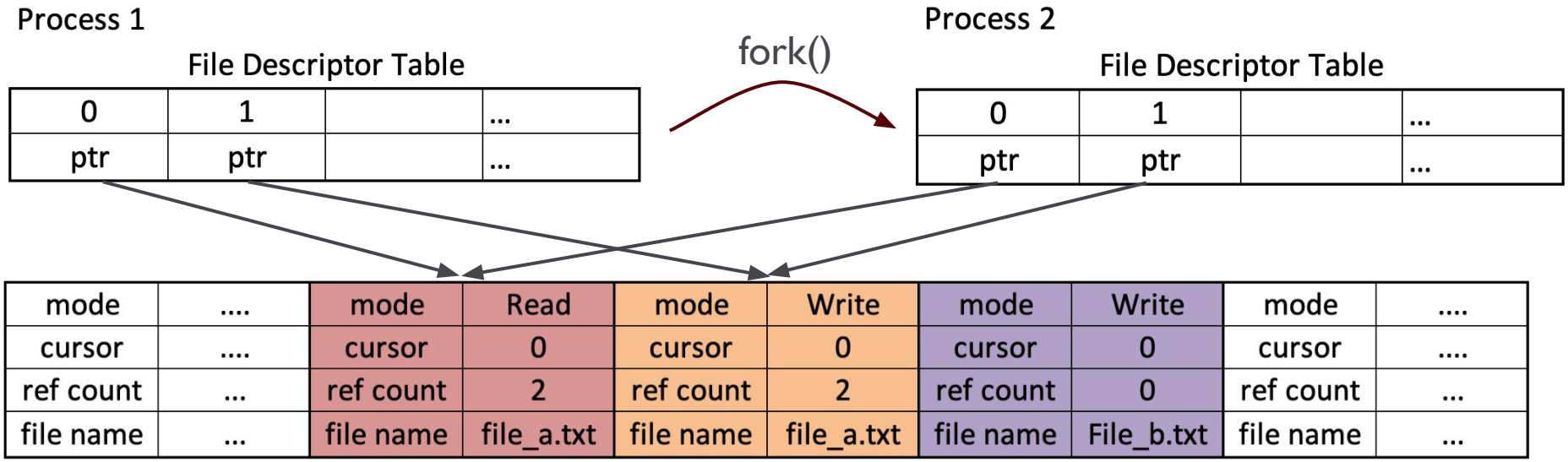
File Descriptors

- As we open a file, we add to the reference count, for each file descriptor pointing to that file



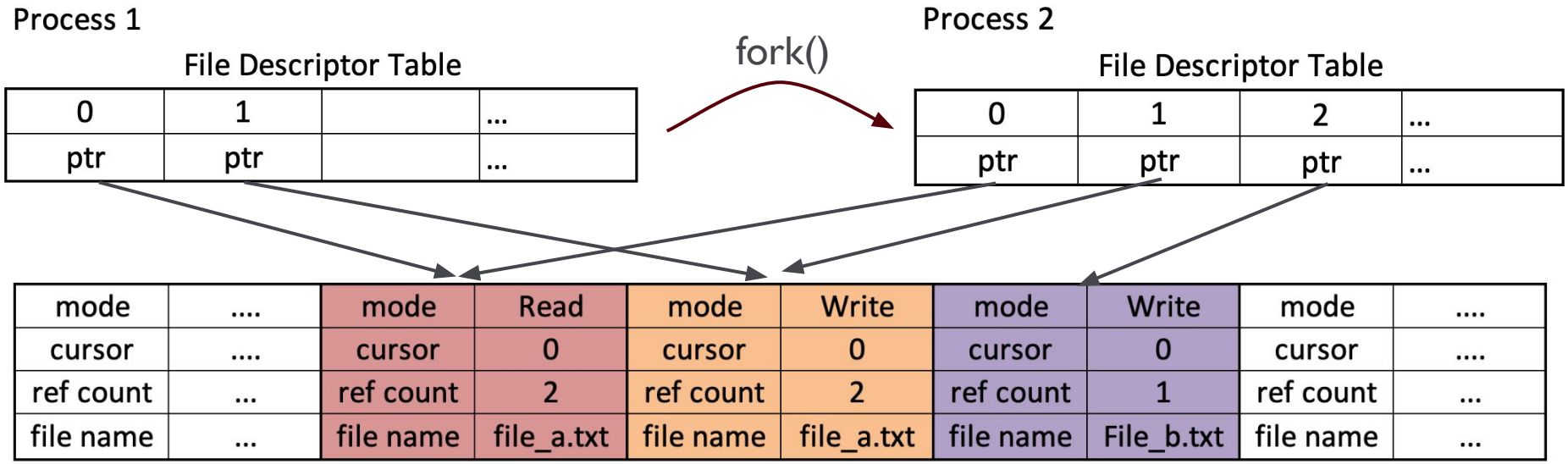
File Descriptors

- Fork duplicates the File descriptor of its parent



File Descriptors

- New file descriptors are not shared!

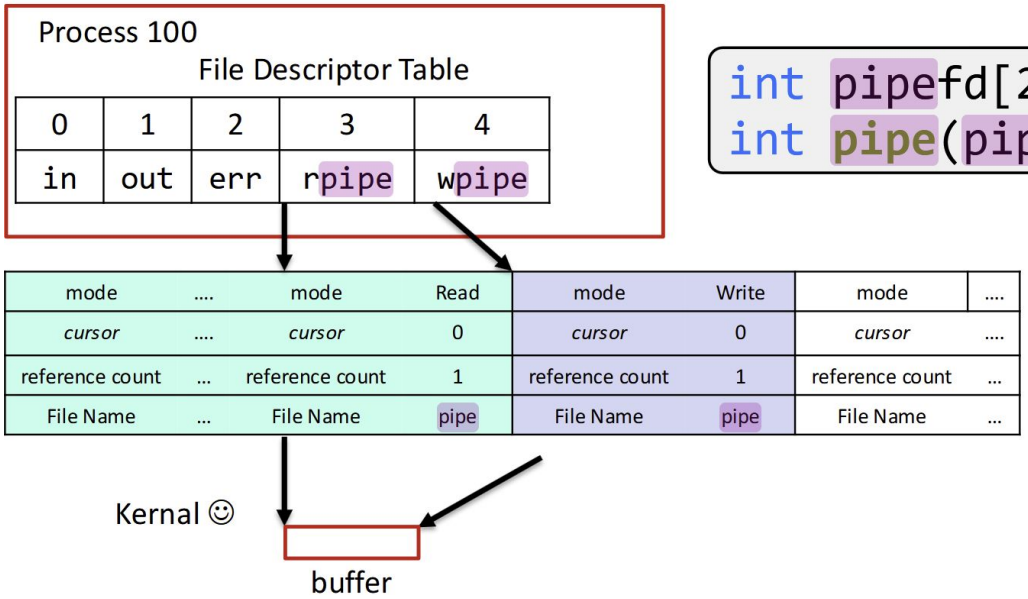


Pipes

- int pipe(int pipefd[2])
  - Creates a **unidirectional** data channel for IPC
  - Sets pipefd[0] to be an fd corresponding to the reading end of the pipe
  - pipefd[1]: fd corresponding to write end !
- Pipe "file" only exists as long as there are references to it and it is maintained by the OS

Pipes

- Creating a pipe initializes two file descriptors in the process FD Table.
- Makes two entries in the system wide file table!




```
int pipefd[2];
int pipe(pipefd);
```

note: the buffer has limited sp
If it is full, you can not write t
read (consume) what is there

Pipes

- When reading from a pipe, you read until a certain number of bytes, or until EOF is received
- EOF is read when **all** write end of the pipe are closed