Lecture 5: File Descriptors

Fall 2019
Jason Tang

Slides based upon Operating System Concept slides,
http://codex.cs.yale.edu/avi/os-book/OS9/slide-dir/index.html

Topics

- File Concept
- Access Methods
- Pipes
- Piping Data

File Concepts

- OS abstraction
- Various data (numeric, character, binary, executables) in secondary storage
- Accessed by user process via a handle (also known as a file descriptor)

File Operations

- File is an **abstract data type** that supports at least the following operations:
  - create
  - write - at pointer location
  - read - at pointer location
  - reposition read/write pointer location within file (a seek)
  - delete
  - truncate

Opening Files

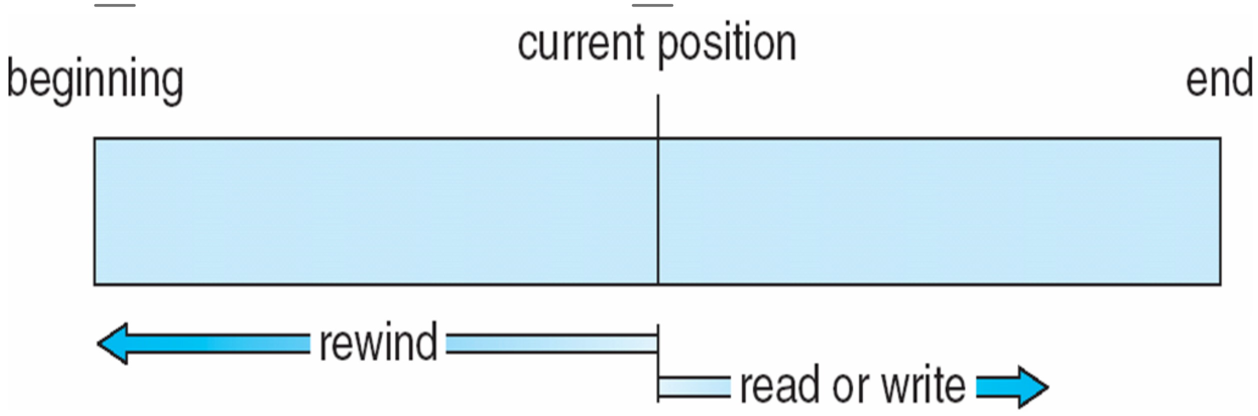
- OS needs following data to manage open files:
    - Open-file table: tracks open files on per-process and system-wide basis
    - File pointer: pointer to last read/write location, per-process
    - File-open count: counts number of times a file has been opened, so as to remove entry from system open-file table when last process closes file
    - Disk location of file: cached so that OS does not re-read from disk
    - Access rights: per-process access mode information (read-only, write-only, etc.)

File Structure

- Most modern OSes treat files as a sequence of 8-bit bytes
    - Underlying processes are responsible for interpreting the files' contents
- Some older OSes required file to have a record structure
    - Consisting of lines, of fixed length or of variable length
    - Example: WinFS is a cancelled project from Microsoft where data are stored in a relational database-like system

Access Methods

- Sequential access: information processed in order, like from a tape
  - Operations: `read_next(), write_next(), reset()`



- Direct access: file is fixed length logical records of size **n** bytes (where **n** is often 1)
  - Operations: read(**n**), write(**n**), reposition(**n**) for read_next() and write_next(), rewind(**n**)

Simulating Sequential Access

Sequential Access	Equivalent Implementation using Direct Access
reset()	off_t cp = 0; reset(cp);
read_next()	read(cp); cp++;
write_next()	write(cp); cp++;

Unix File Descriptors

- **Unsigned** integer, unique on a **per-process basis**, used to access files (and other resources)

- When a process opens a file (for reading and/or writing), OS will assign a file descriptor (FD) to represent that resource

  - Process uses that FD to access the file's data (and also its metadata)

  - OS maintains an internal array that maps a process's FDs to actual files (the process's open-file table)

- Convenience function `fopen()` has a file descriptor within its `FILE` struct

Example of Opening a File

```c
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(void) {
    int fd = open("/etc/passwd", O_RDONLY);
    if (fd < 0) {
        fprintf(stderr, "Error opening\n");
    } else {
        printf("Got FD %d\n", fd);
    }
    return 0;
}
```

`open()` returns a file descriptor, or negative on error

File descriptors are just `int`s, that can be printed like other `int`s

Example of Reading a File

`read()` returns the number of bytes actually read (which could be smaller than amount requested, for a short read)

```
…
char buf[2048];
size_t amount_to_read = sizeof(buf);
ssize_t amount_read = read(fd, buf, amount_to_read);
if (amount_read < 0) {
    fprintf(stderr, "Error reading\n");
} else {
    printf("Read %zd bytes\n", amount_read);
}
…
```

`ssize_t` is not an `int`; they could have different ranges

Blocking Reads

- What if there is a request for read(**n**), but only **n**-1 records currently exist?
    - Non-blocking I/O: Return immediately with error code
    - Blocking I/O: Pause process until some other process creates record **n**
- Blocking I/O very useful when writing networking code
    - Typical pattern is for a process to block itself until one (or more) file descriptor becomes available to be read

Synchronous Write

- What if there is a request for read(**n**), but another process is simultaneously writing?
  - When OS caches write operations (default in Linux), read(**n**) returns previous [stale] data
- When opening file, can specify synchronous write flag
  - When writing to file, OS guarantees that data committed to underlying hardware before function returns
  - Similar to `write()` + `fsync()`

Default File Descriptors

- By *convention*, every Unix process begins with three opened file descriptors

FD number	Name	Usage
0	stdin	Process input, usually from console
1	stdout	Buffered output to console
2	stderr	Unbuffered output, for reporting errors

- A shell can redirect these FDs to other sources, such as to a file
    - Example: `./foo > /tmp/foo.log`

Sharing File Descriptors

- Every process has **its own set** of file descriptors
- Example: there are two processes named *foo* and *bar*
  - Process *foo*’s FD 4 does not necessarily refer to the same location as *bar*’s FD 4
  - If *foo* opens the file `/tmp/baz` and the OS assigns it FD 5, then if *bar* also opens `/tmp/baz` then the OS may happen also assign to *bar* FD 5
    - Or OS could also assign some other FD to *bar*

Duplicate File Descriptors

- OS does not release a resource until **all** file descriptors referring to it are **closed**
- When a process `fork()`s, its child process inherits all opened file descriptors
  - If *foo* has a FD 5 to `/tmp/baz`, and it calls `fork()`, then the child will also have a FD 5 that refers to the same `/tmp/baz`
  - If *foo* then calls `close()` on FD 5, the file is not really closed until the child also calls `close()` on its FD 5
- Different behaviors between Windows and Unix if one process deletes a file while another process has an opened file descriptor to it

Pipes

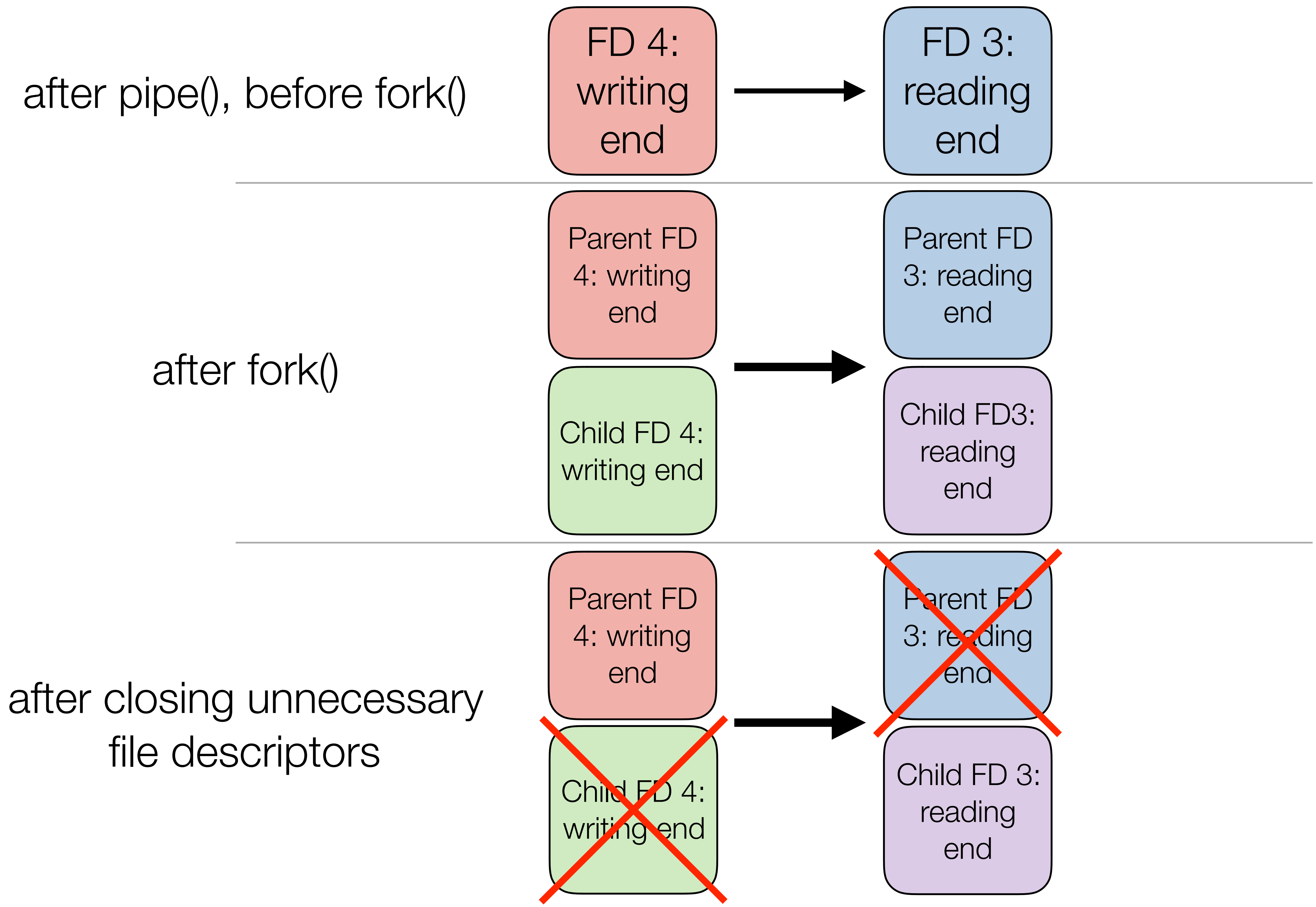
- Conduit allowing two processes to communicate, via file-like I/O
  - Unnamed pipe: parent process creates pipe, then uses it to communicate with its child processes
  - Named pipe (FIFO): any two processes (potentially unrelated) can use
- In Linux/Unix, pipes are always unidirectional
  - Unix-domain socket (named socket): similar to a pipe, but is bidirectional

Traditional Unix Unnamed Pipe

```
int fd[2];
pipe(fd);
if (fork() == 0) {
    /* child process */
    read(fd[0], …);
} else {
    /* parent process */
    write(fd[1], …);
}
```

- `pipe()` function creates an unnamed pipe, where **read-end** of pipe assigned to element 0 and **write-end** to element 1 of FD array
- Next program `fork()`s, so that one process writes and the other reads from the unnamed pipe
- See man page for pipe(2) for full code example

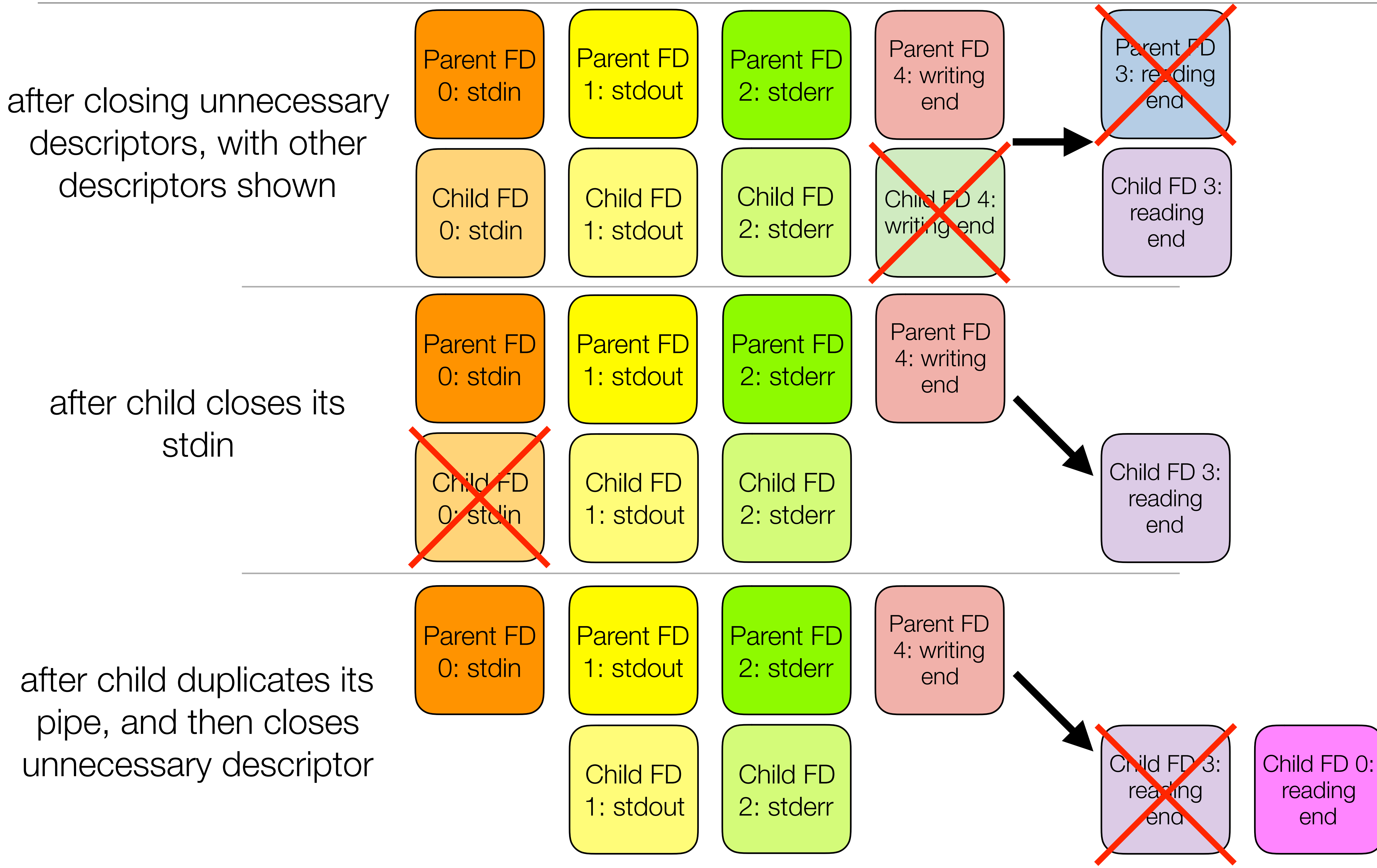
Using a Pipe to Communicate to a Child



Piping from One Process to Another

- In Unix, common to pipe data from one process to another
    - Example: `dmesg | head`
- This takes advantage of several properties of file descriptors:
    - Resource not released until **all** file descriptors are closed
    - Child processes inherit all of its parent's opened descriptors
    - Opened file descriptors are preserved across `exec()`
    - OS chooses lowest available number when assigning new descriptors

Piping Data to New Process



Piping Data Between Processes

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(void) {
    int fd = open("/tmp/baz", O_RDWR);
    char buf[16];
    if (fork() == 0) {
        /* child process */
        ssize_t a;
        a = read(fd, buf, sizeof(buf));
        printf("Read %zd bytes\n", a);
    } else {
        /* parent process */
        write(fd, buf, sizeof(buf));
    }
    return 0;
}
```

What would this display if `/tmp/baz`:

- Was empty?
- Already had 4 bytes?
- Already had 32 bytes?