System-Level I/O:
File Descriptor State Sharing

COMP 321

Dave Johnson

1

Consider the Following Example

We compile the source file "abc.c" to make the program "abc"

```
int main()
{
    write(1, "ABCDEFGHIJKLMNOPQRSTUVWXYZ\n", 27);
    return 0;
}
```

And we compile the source file "123.c" to make the program "123"

```
int main()
{
    write(1, "0123456789\n", 11);
    return 0;
}
```


2

Consider the Following Example

Now suppose we put the following 2 commands in a shell script file "doit.sh"

./abc
./123

What if we now run the command?

$ sh doit.sh

What we see on the screen is

ABCDEFGHIJKLMNOPQRSTUVWXYZ\n0123456789\n

Which looks like this on the screen

ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789

Now Consider the Following Further Example

What if we now run the command?

$ sh doit.sh **> OUT**

What we <u>expect</u> to see in the file OUT is

ABCDEFGHIJKLMNOPQRSTUVWXYZ\n0123456789\n

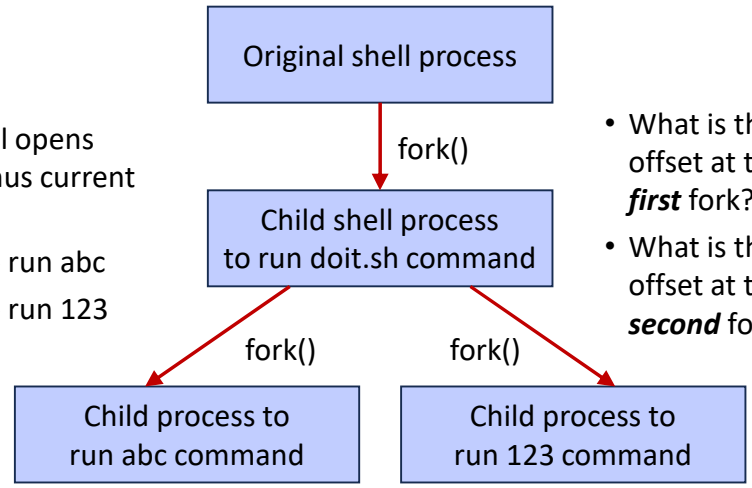
But with a "simplistic" implementation, what we'll get in OUT is

0123456789\nLMNOPQRSTUVWXYZ\n

Let's look at why . . .

Analyzing This Example

- The child shell opens "OUT", and thus current fd offset = 0
- Does fork() to run abc
- Does fork() to run 123



- What is the child shell fd offset at the time of this ***first*** fork?
- What is the child shell fd offset at the time of this ***second*** fork?

Reminder: The Current File Offset in an fd

Each open file descriptor has an associated offset (i.e., position) within the file

- The current file offset is initialized to 0 (i.e., the beginning of the file's data) when the file descriptor is opened (e.g., from open or creat)
- A ***single*** current fd offset is used jointly for both ***reading and writing*** this fd
  - Any ***read*** from this fd advances the fd's offset by the number of bytes actually ***read***
  - Any ***write*** to this fd advances ***this same*** offset by the number of bytes actually ***written***
- Example: repeated reading from the file sequentially transfers each next part of the file, until reaching the end of the file (as limited by the size of the file)

Kernel File Descriptor Data Structures

In the kernel for each process (e.g., in the process's PCB)

- An array, the ***file descriptor table*** for this process, indexed by the fd number (which are small integers)
- Each entry is a pointer to the ***open file table*** entry for that open file
- Or is NULL if that fd is not open now in this process

In the kernel, shared by all processes, the system-wide ***open file table***

- Remembers current offset position and flags (i.e., O_RDONLY, O_TRUNC, etc.)
- And a pointer to the vnode table entry for the file (i.e., object) that is open

In the kernel, shared by all processes, the system-wide ***vnode table***

- Remembers a copy of control state information (i.e., metadata) for that file

Independent File Opens vs. Shared fd State

Only one entry in the vnode table for each file (or other type of object)

- No matter how many times that file is open, and
- No matter how it was opened and by which process

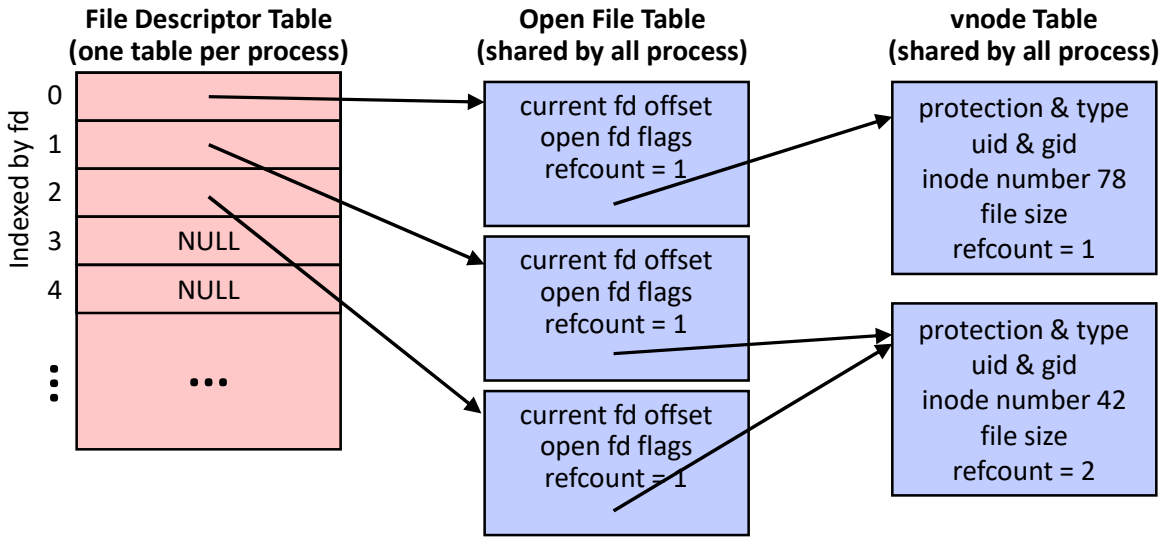
Each independent open (or creat, etc.) results in a new open file table entry

- And thus an independent offset (i.e., position) in that open file
- And a new set of remembered flags (e.g., O_RDONLY, O_TRUNC, etc.)
- All point to the same entry in the vnode table

Creating a new fd from some existing fd shares existing open file table entry

- Thus shares the open file offset and open file flags

Kernel File Descriptor Data Structures



Aside: What Is an inode and What Is a vnode?

Unix/Linux supports many different types of file systems

Each has an <u>on-disk</u> data structure for each file known as an <u>inode</u>

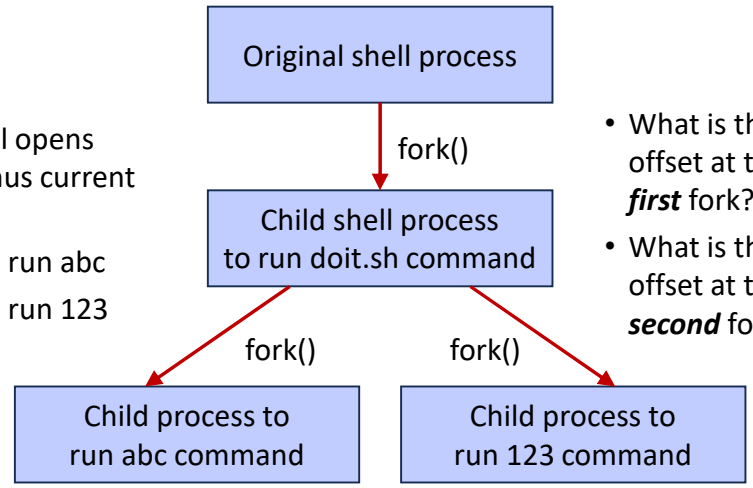
- Short for "index node" since it provides (among other information) an "index" of which "blocks" of disk space store each part of the file's data
- We'll look more at inodes later when we look at file systems

A vnode is an <u>in-memory</u> data structure for a file, with two purposes

- Remembers in memory a copy of the inode information from disk for that file
- Provides a "wrapper" layer over the inode information in memory
  - Most of the kernel treats all vnodes the same (a "virtual inode")
  - And the differences between the inode format for one type of file system and another are handled all together through this "wrapper"

Again, Analyzing This Example

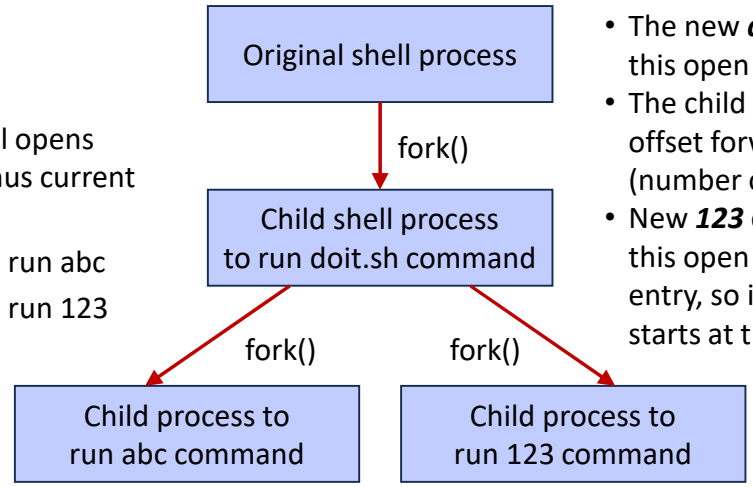
- The child shell opens "OUT", and thus current fd offset = 0
- Does fork() to run abc
- Does fork() to run 123



- What is the child shell fd offset at the time of this ***first*** fork?
- What is the child shell fd offset at the time of this ***second*** fork?

How it Really Works

- The child shell opens "OUT", and thus current fd offset = 0
- Does fork() to run abc
- Does fork() to run 123



- The new ***abc*** child ***shares*** this open file table entry
- The child write() moves the offset forward by 27 (number of bytes written)
- New ***123*** child also ***shares*** this open file table entry, so its new write() starts at that position

Instead, With a "Simplistic" Implementation

If the current offset was <u>not</u> shared across the fork()

- The ***abc*** child would start writing at offset 0, copied from its parent like the rest of its address space
- When the ***abc*** child exits, any changes to the offset would be thrown away with the rest of that process's address space
- The ***123*** child would thus start also at offset 0, copied from its parent like the rest of its address space

So we'd (incorrectly) end up with the following in the file OUT

0123456789\nLMNOPQRSTUVWXYZ\n

123's write() incorrectly starts at offset 0

Again, With the Correct Implementation

Because the current offset <u>is</u> shared across the fork()

- The ***abc*** child starts writing at offset 0, ***since that is the current shared offset*** (the shell hasn't written to that fd after opening it)
- When the ***abc*** child exits, its address space is thrown away, but ***the shared offset for that file descriptor remains in the shared open file table entry***
- The ***123*** child thus starts at the same offset that the ***abc*** child was at when it completed, since ***that offset is still in the open file table entry***

So we'd <u>correctly</u> end up with the following in the file OUT

ABCDEFGHIJKLMNOPQRSTUVWXYZ\n0123456789\n

123's write() correctly starts at the shared offset position

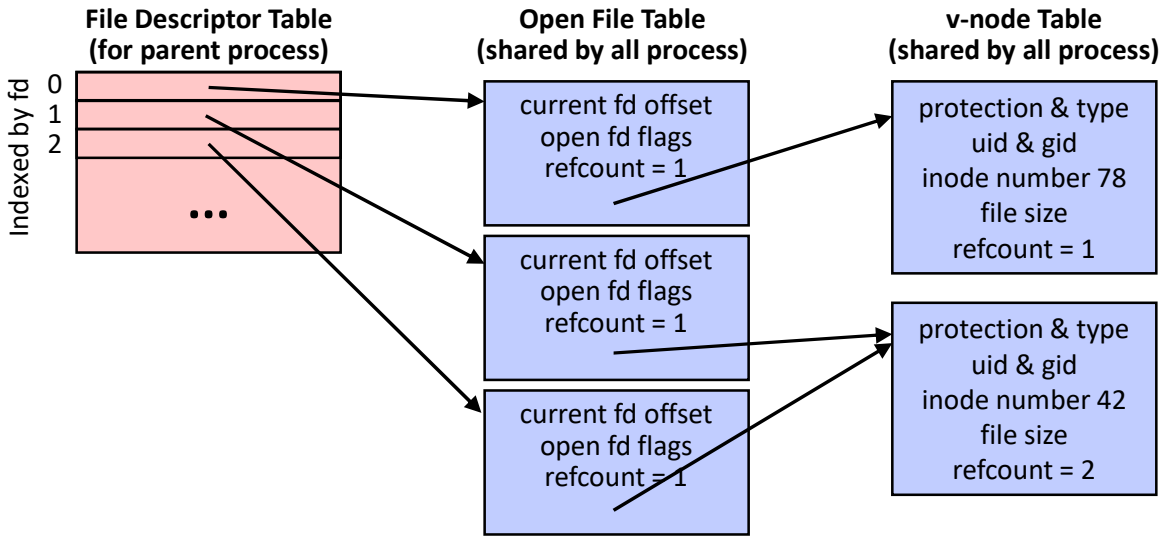
Summary: Handling File Descriptors During a fork()

The kernel copies the parent process's entire address space to create the child's

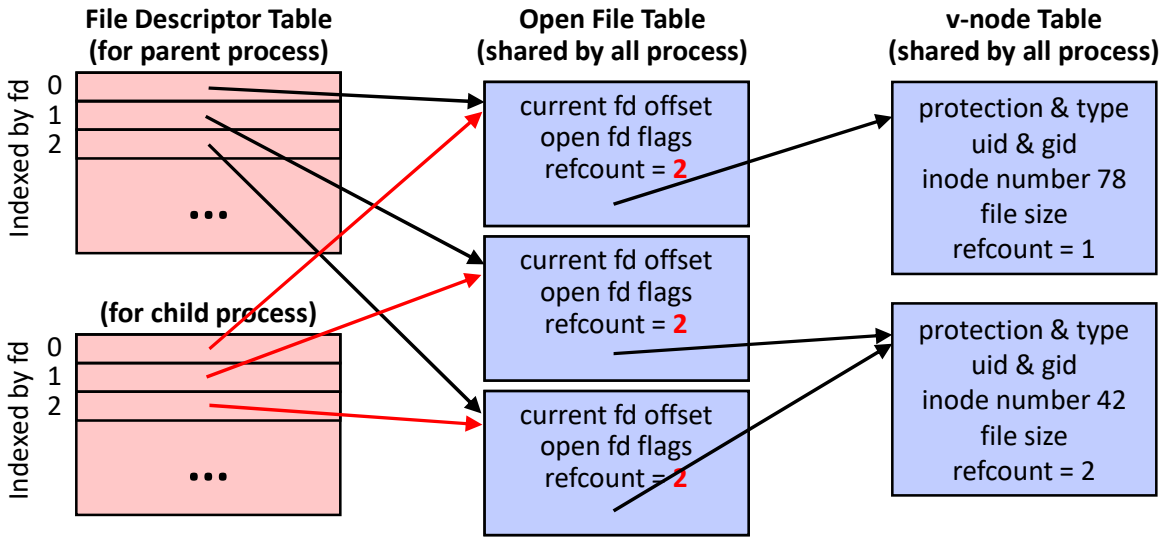
In addition, the kernel effectively "copies" the parent PCB file descriptor table

- Each entry in this array is either NULL or is a pointer to the corresponding open file table entry
- Copy each entry (i.e., each pointer) to corresponding entry in child's PCB
- And, if not equal to NULL, increase the reference count on the corresponding open file table entry
  - This open file table entry is shared between the parent and the child

Kernel File Data Structures Before the Fork



Kernel File Data Structures After the Fork



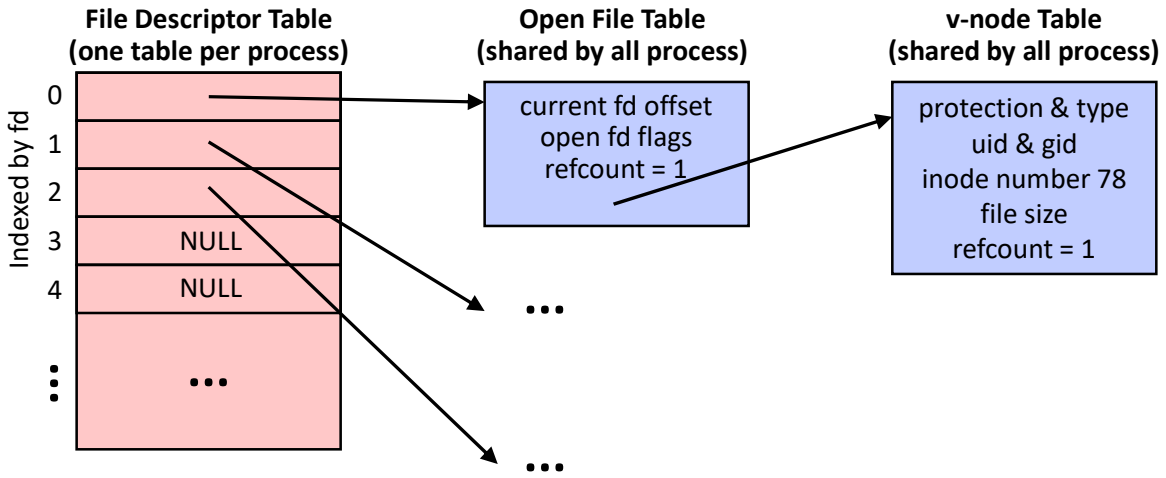
Duplicating a File Descriptor within a Process

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

Assigns a new (additional) descriptor number to existing open file instance

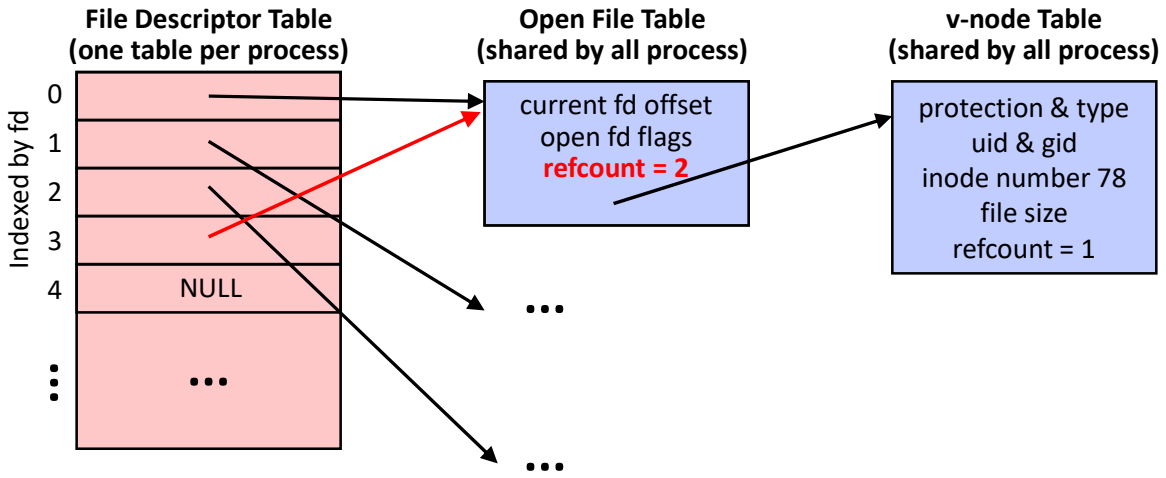
- dup() returns the ***lowest numbered*** file descriptor number that is not currently open in this process to something – just like open() does
- dup2() instead uses the specified newfd file descriptor number
  - if newfd is already open, it is automatically closed first
- On return, both old and new file descriptors refer to the same ***shared*** open file table entry

The Kernel Data Structures – Doing dup(0)



19

The Kernel Data Structures – Doing dup(0)



20

Example: Using dup() or dup2() in the Shell

Redirecting standard output (e.g., command > file) the unsafe way

```
close(STDOUT_FILENO);
open(file, O_WRONLY);
```

- Doing open() will pick the lowest unused descriptor number, which here should be STDOUT_FILENO
- But for a time, you have no open standard output file!

Doing it the correct, safe way

```
newfd = open(file, O_WRONLY);
dup2(newfd, STDOUT_FILENO);
close(newfd);
```

- dup2() closes old STDOUT_FILENO; then we close unneeded newfd

Summary: Kernel File Descriptor Data Structures

File descriptor table for each process (e.g., in the process's PCB)

- An array, indexed by the fd number (which are small integers)
- Each entry is a pointer to the ***open file table*** entry for that open file
- Or is NULL if that fd is not open now in this process

Open file table, shared by all processes

- A new one ***only*** for each ***independent*** open (or creat, etc.)
- Remembers current offset position and flags (i.e., O_RDONLY, O_TRUNC, etc.)
- And a pointer to the vnode table entry for the file (i.e., object) that is open

vnode table, shared by all processes

- Remembers a copy of control state information (i.e., metadata) for that file

Summary: Creating a New fd Based on Existing fd

The new fd shares the open file table entry with the original fd

- Example: fork() creating each fd in the child based on existing fd in the parent
- Example: dup() or dup2() creating a new fd in this process based on the specified existing fd also in this process
- ***The new fd shares the existing open file table entry with the existing fd***
- The existing fd already has some position (i.e., offset) within the file
  - Might be at ***any*** position, depending on what I/O has already been done on that existing fd
- The existing fd has existing flags (e.g., O_RDONLY, O_TRUNC, etc.)
  - And creating the new fd has no way to specify then any new/different flags
- ***The new fd, based on this existing fd, thus shares all of this***

Summary: A New Independent Open

The new fd must also create a new open file table entry

- Example: a new call to open (or creat, etc.)
- The new open may use any (different) flags (e.g., O_RDWR, O_TRUNC, etc.)
- Kernel thus ***must*** create a new ***open file table*** to remember those new flags
- The position in the open file is also thus not shared
  - ***Can't*** be shared since the open file table entry thus ***can't*** be shared
  - But also makes sense not to share the position
    - this was a new independent open
    - and any other existing (independent) open fds are thus unrelated and wouldn't expect unrelated sharing of the open file position