CSE 451 Section 2
XK Lab 1 Discussion

22sp - Apr 7 2022

Today’s Agenda

- Lab 1 tomorrow night
- Same lab 1 slides from last week
  - If anyone needs a refresher, happy to go through them again
- Some discussion questions
- Open Lab 1 Q/A

Where to start?

Start by reading:

- **lab/overview.md** - A description of the xk codebase. A MUST-READ!
- **lab/memory.md** - An overview of memory management in xk
- **lab/lab1.md** - Assignment write-up
- **lab/lab1design.md** - A sample design doc for the lab 1
    - You will be in charge of writing design docs for the future labs. Check out lab/designdoc.md for details.

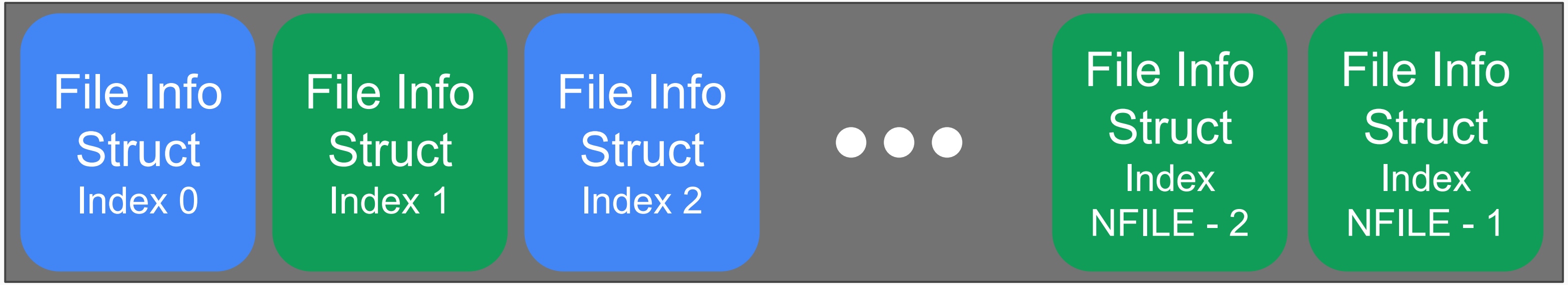
File Information

Need a way to store the following information about a file:

- A reference to the inode of the file
- Current offset
- Access permissions (readable or writable)
  - for when we add pipes and file writability later
- In memory reference count

File Info Struct

Kernel View



File Info Struct Index 0

File Info Struct Index 1

File Info Struct Index 2

• • •

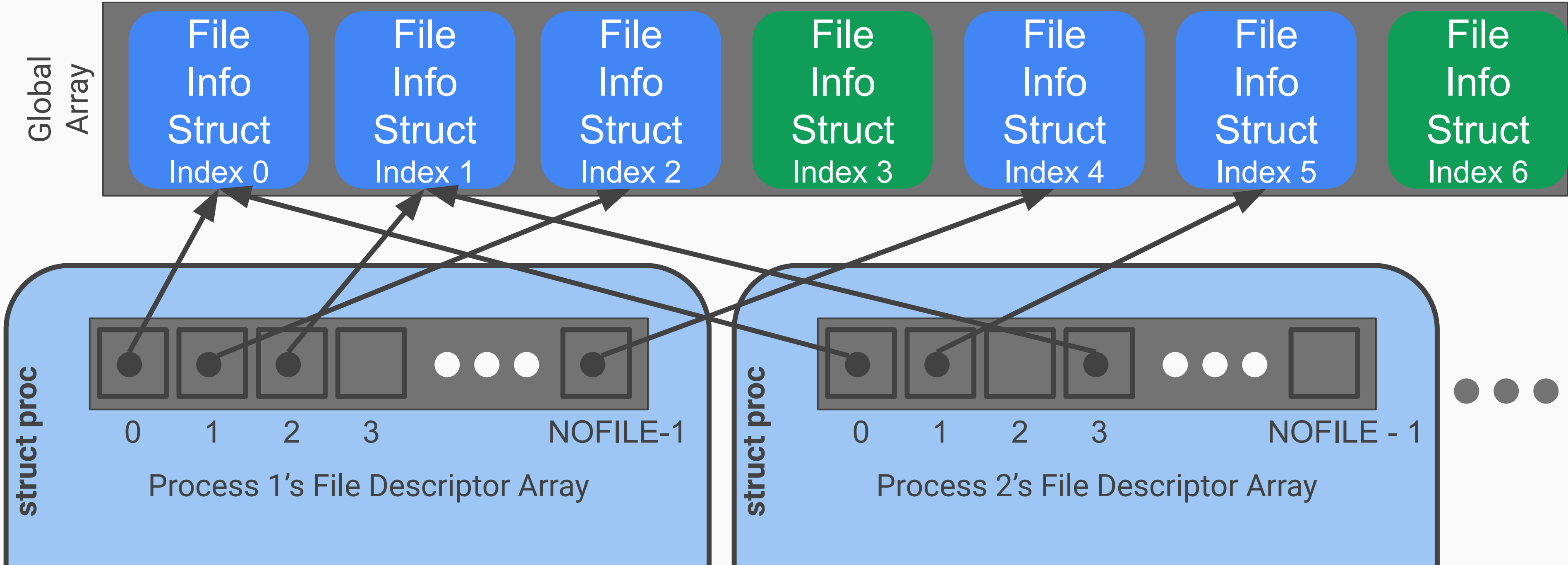
File Info Struct Index NFILE - 2

File Info Struct Index NFILE - 1

🟦 = In use   🟩 = Available

There will be a global array of all the open files on the system (bounded by NFILE) placed in static memory.

Process View



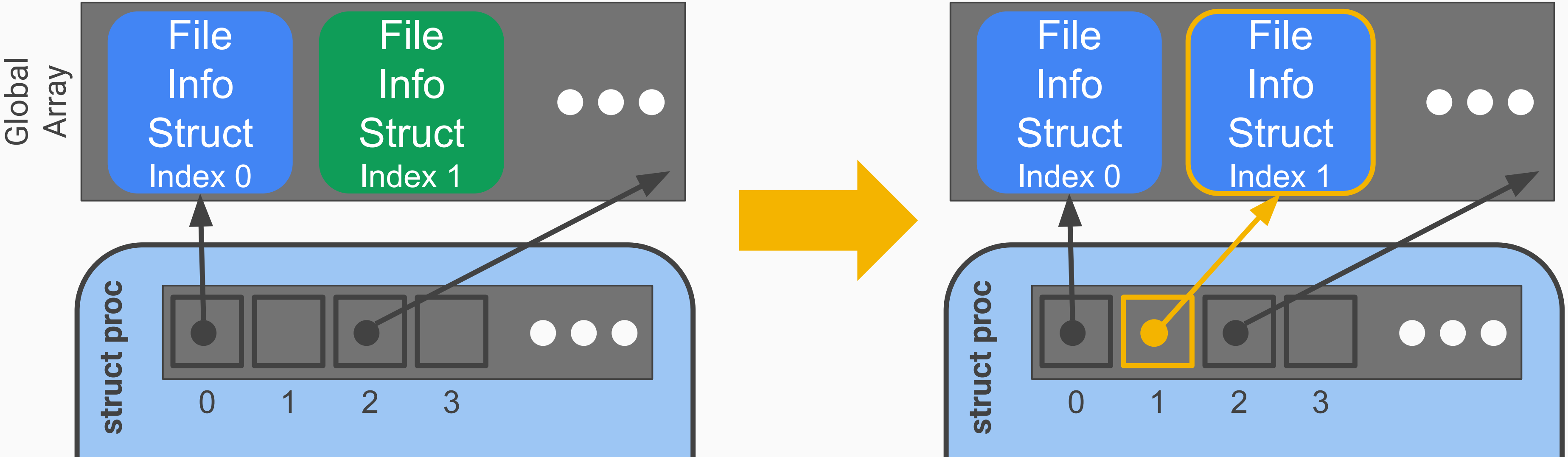
File System Functions

filewrite and *fileread*

- Writing or reading of a "file"
  - Note that file is in quotes. Many things on Unix-like systems are treated as a file. A “file” can be a real file on disk, or a console, or a pipe (lab 2)!
- Check out the functions *readi* and *writei* defined in kernel/fs.c

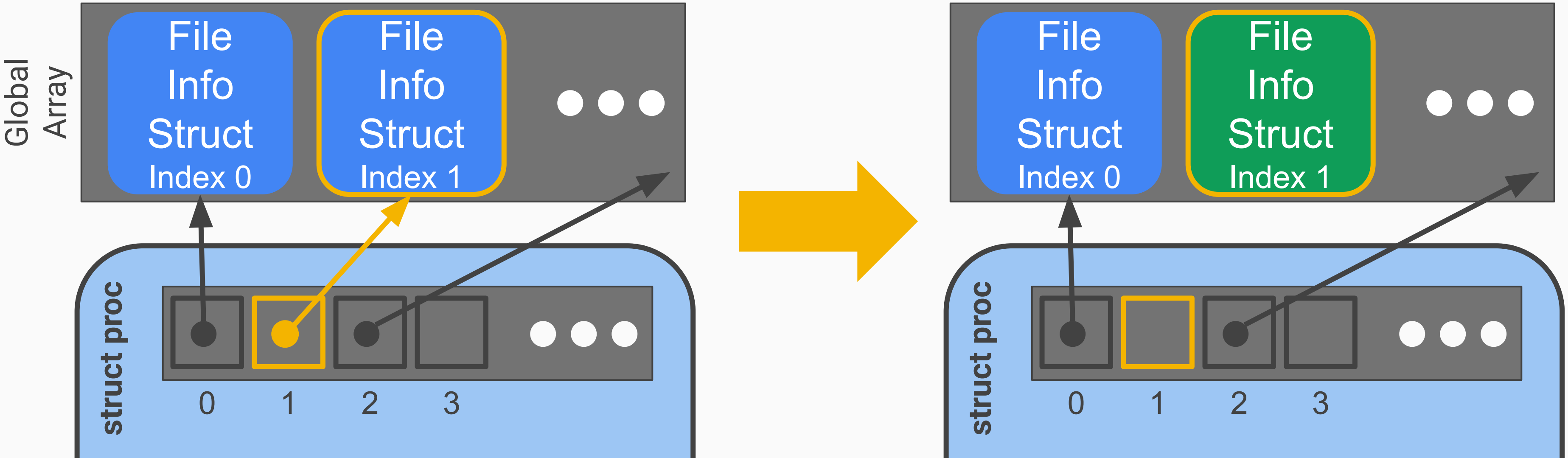
fileopen

Finds an open file in the global file table to give to the process



fileclose

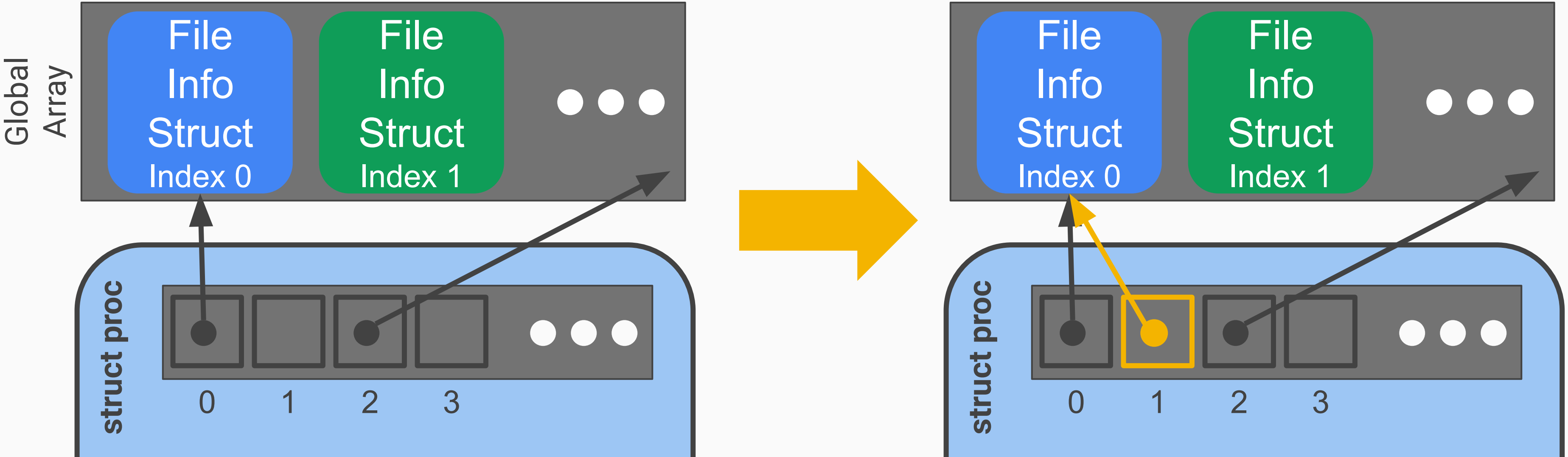
Release the file from this process, will have to clean up if this is the last reference



filedup

Duplicates the file descriptor in the process' file descriptor table

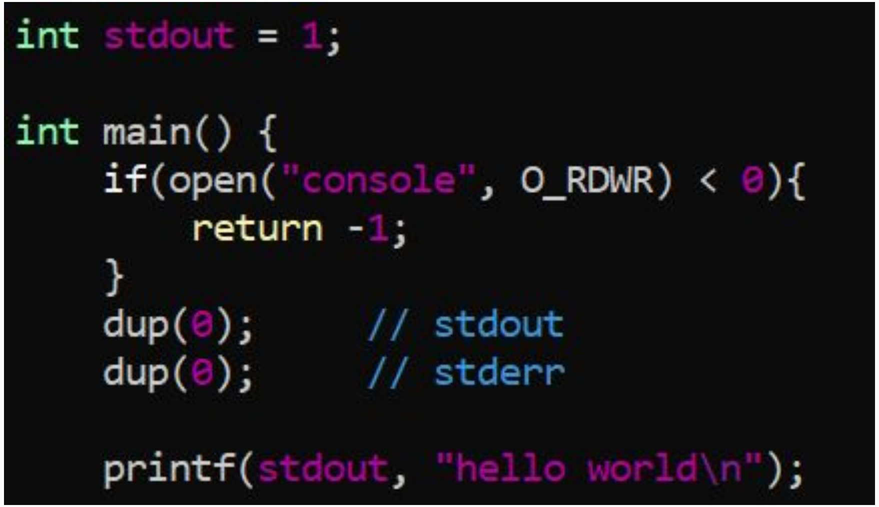
Why do we need this?



filestat

- Return statistics to the user about a file
- Check out the function stati in kernel/fs.c

Lab 1 Test Program Code Fragment

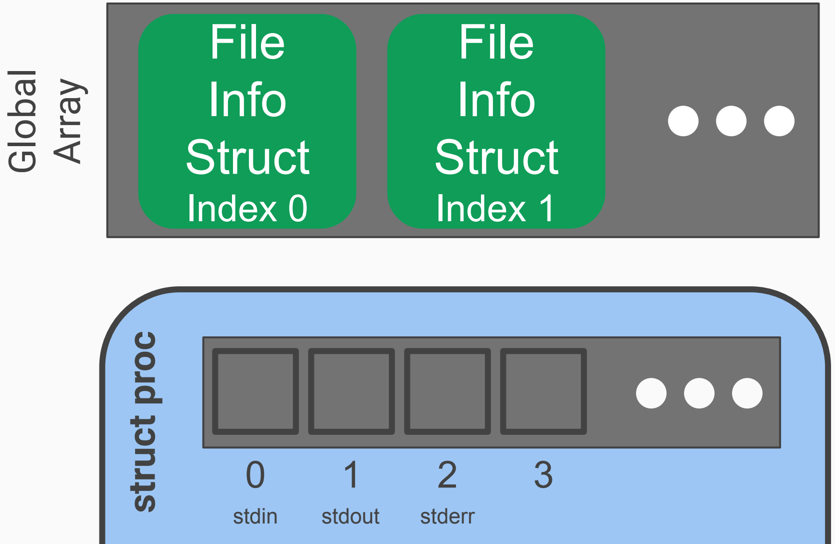


```
int stdout = 1;

int main() {
    if(open("console", O_RDWR) < 0){
        return -1;
    }
    dup(0);     // stdout
    dup(0);     // stderr

    printf(stdout, "hello world\n");
```

- What’s going on here?
- We mention the file system is read only…
  - Why can we write to stdout?

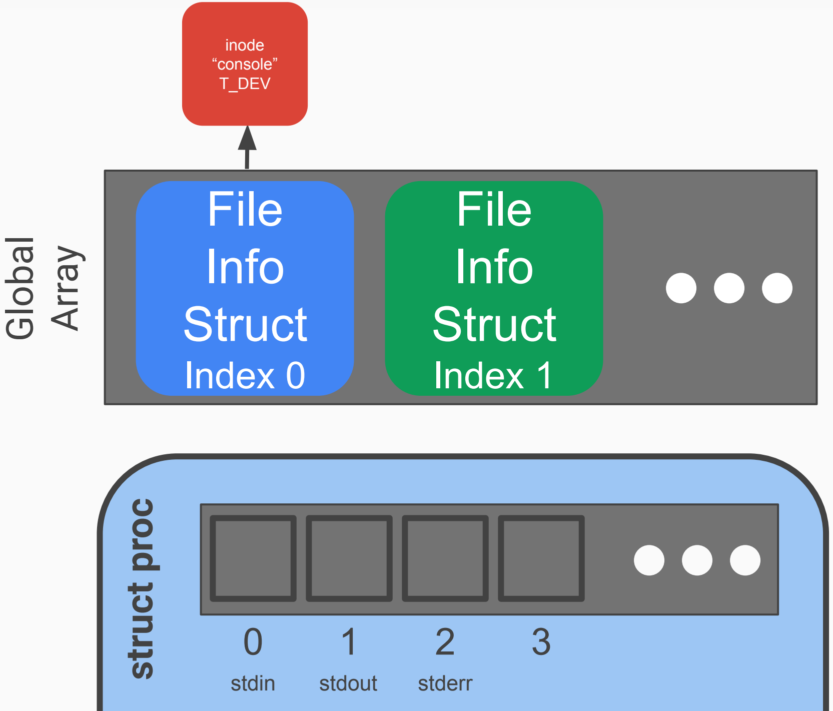
File Table View

open(“console”, O_RDWR)



File Table View

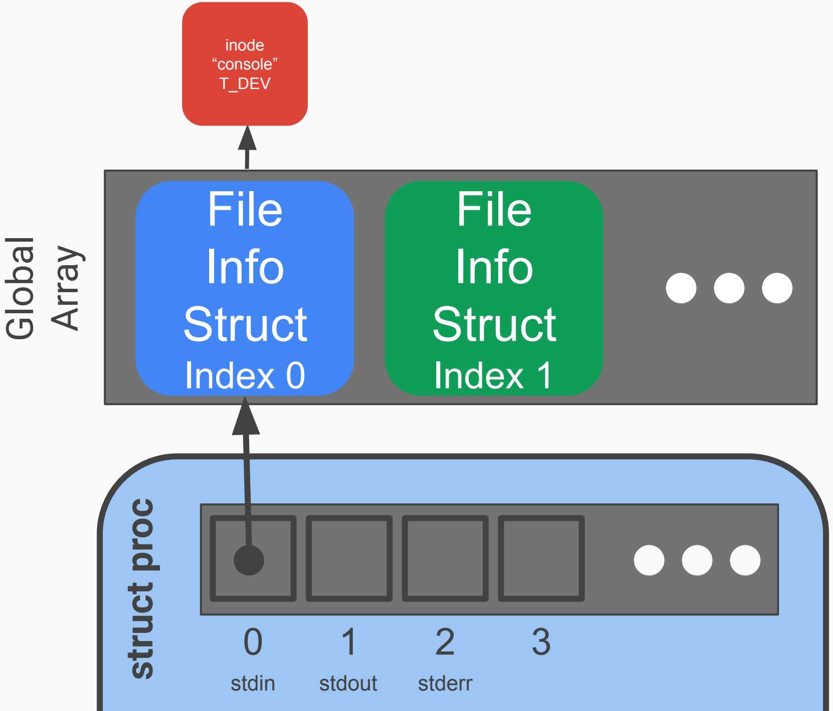
open(“console”, O_RDWR)



- Resolve inode for “console”
- Find next unused slot in global array, allocate for inode

File Table View

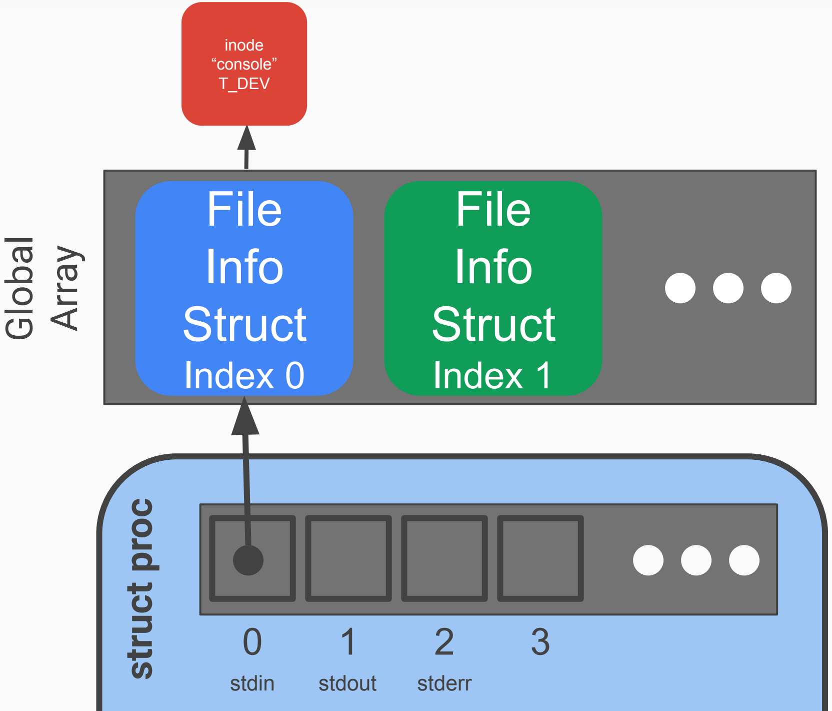
open(“console”, O_RDWR)



- Find next open slot in local FD array
- Return FD to user

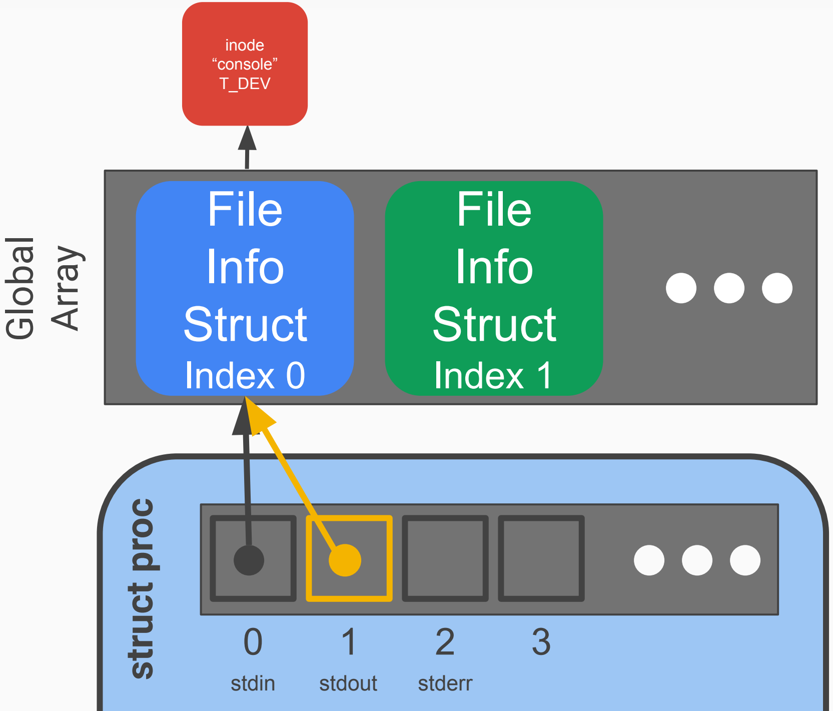
File Table View

```
open("console", O_RDWR)
dup(0)
```



File Table View

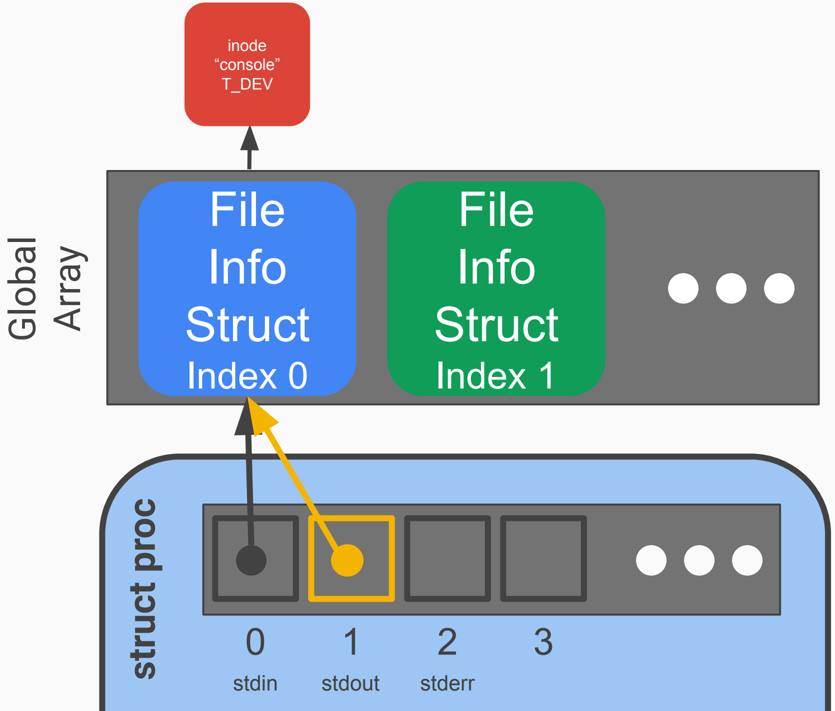
open(“console”, O_RDWR)
dup(0)



- Find next open slot in local FD array
- Duplicate reference from user’s given FD
- Return new FD to user

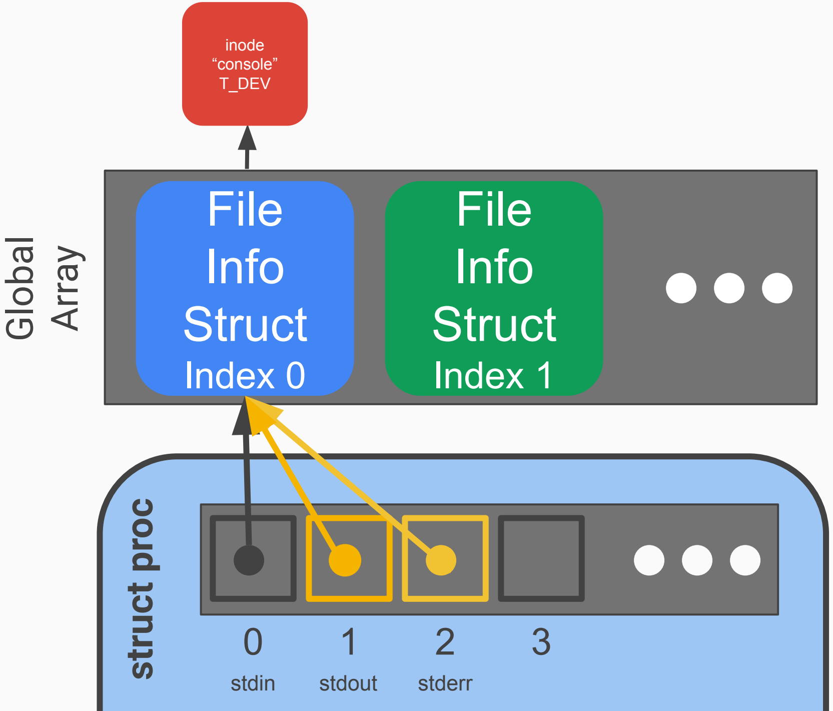
File Table View

```
open("console", O_RDWR)
dup(0)
dup(0)
```



File Table View

open(“console”, O_RDWR)
dup(0)
dup(0)



Console Input/Output

- The console device is just a special file called “console”!
- Code to handle device files is already handled for you
  - Its information is already provided for you when you open the device file.
  - Where? Look at kernel/fs.c, inc/file.h and how the T_DEV file type is used.

- I thought stdin/stdout/stderr were always available?
  - Recall that fork() copies the file descriptor table and there’s always a root process. The root process is actually what opens the console device file, and every process inherits from root, which is why stdin/stdout/stderr are available on non-root processes.

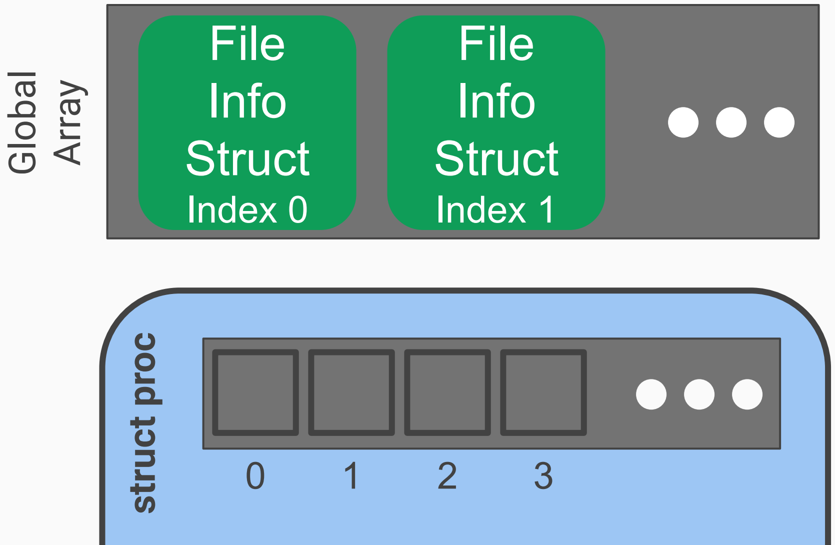
Multiple Open Calls on Same File

- Draw out the process and global open file table layout after the following:

```
int fd1 = open(“file.txt”, O_RDONLY);
int fd2 = open(“file.txt”, O_RDWR);  // assume we allow writes to files
```

Multiple Open Calls on Same File

```
open(“file.txt”, O_RDONLY)
```



Global Array

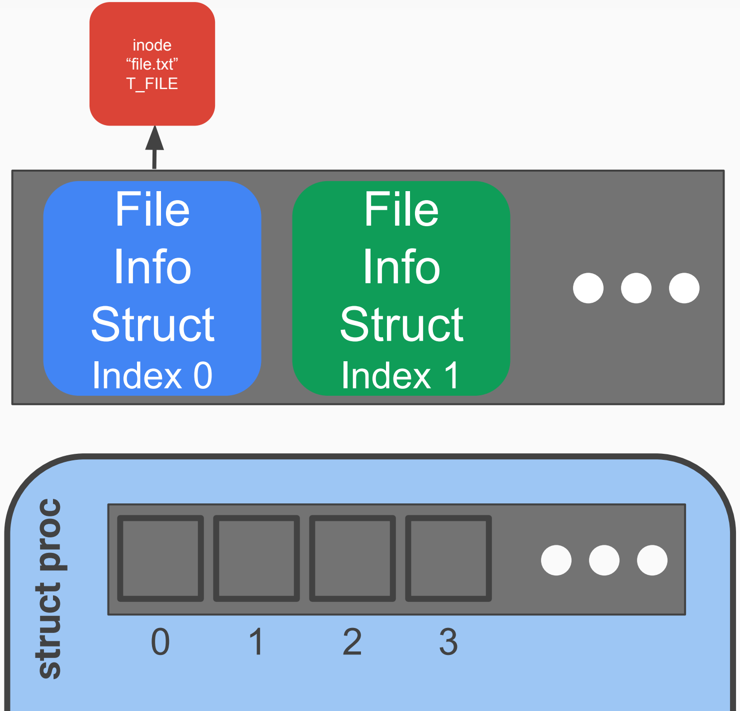
File Info Struct Index 0

File Info Struct Index 1

struct proc

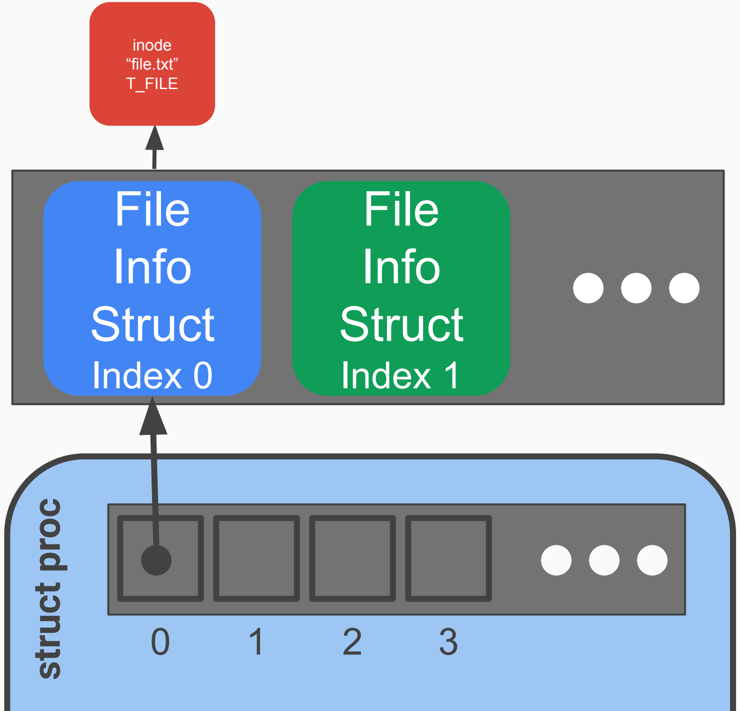
Multiple Open Calls on Same File

open(“file.txt”, O_RDONLY)



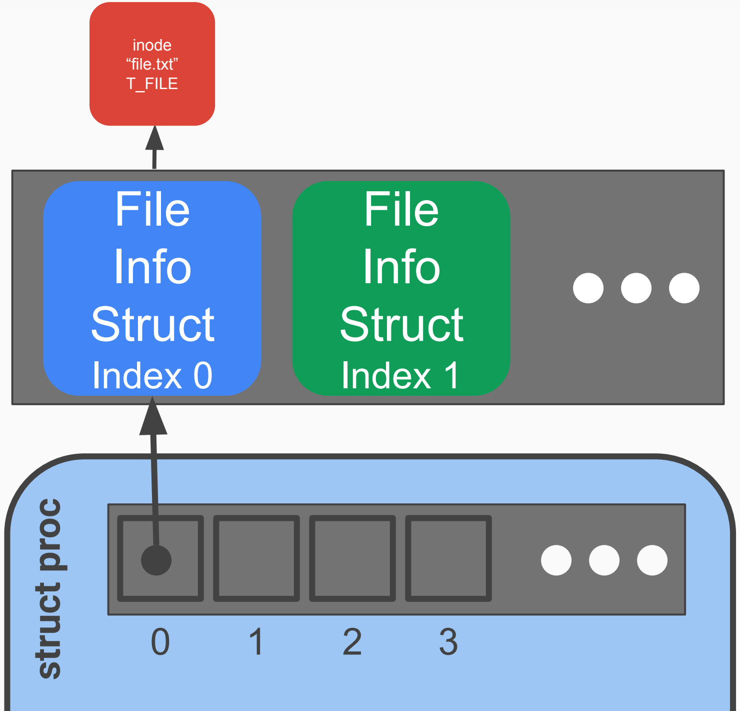
Multiple Open Calls on Same File

open(“file.txt”, O_RDONLY)



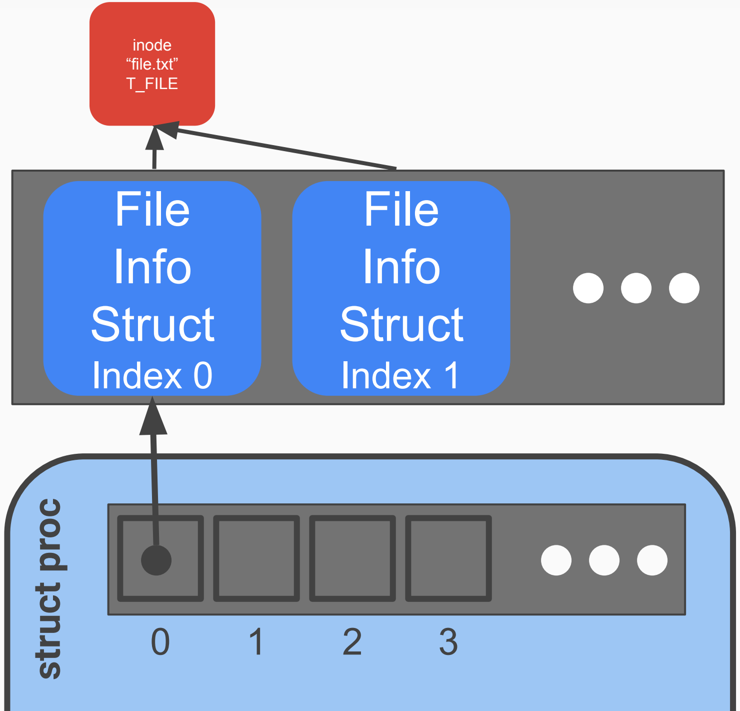
Multiple Open Calls on Same File

```
open("file.txt", O_RDONLY)
open("file.txt", O_RDWR)
```



Multiple Open Calls on Same File

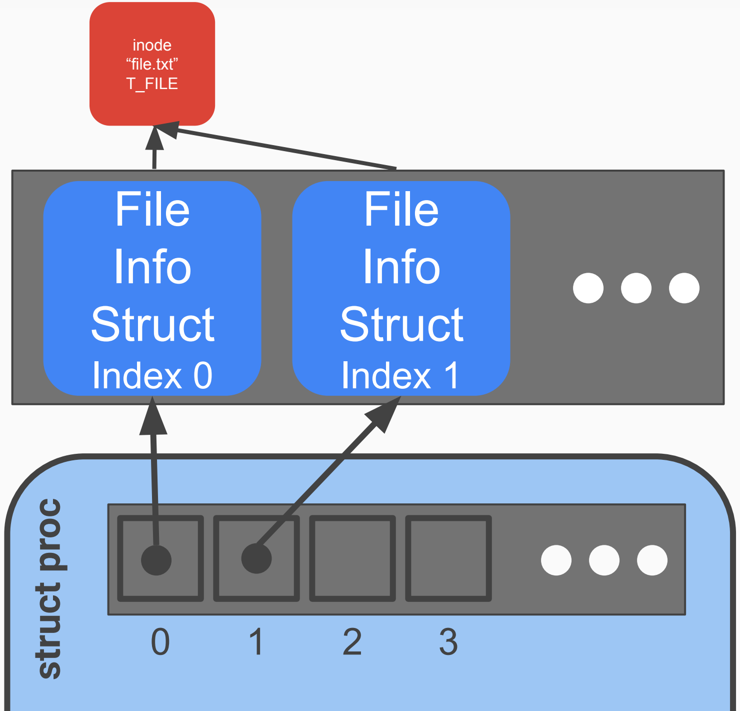
open(“file.txt”, O_RDONLY)
open(“file.txt”, O_RDWR)



- Each open call allocates a new file_info struct
- Name lookup returns same inode

Multiple Open Calls on Same File

open(“file.txt”, O_RDONLY)
open(“file.txt”, O_RDWR)



- Each open call allocates a new file_info struct
- Name lookup returns same inode

System calls

System Calls

- sys_open, sys_read, sys_write, sys_close, sys_dup, sys_fstat
- Main goals of sys functions
  - Argument parsing and validation (never trust the user!)
  - Call associated file functions

Argument Parsing & Validation

All functions have int n, which will get the n'th argument. Returns 0 on success, -1 on failure

- **int argint(int n, int *ip)**: Gets an int argument
- **int argint64_t(int n, int64_t *ip)**: Gets a int64_t argument
- **int argptr(int n, char **pp, int size)**: Gets an array of size. Needs size to check array is within the bounds of the user's address space
- **int argstr(int n, char **pp)**: Tries to read a null terminated string.

You should implement and then use:

- **int argfd(int n, int *fd)**: Will get the file descriptor, making sure it's a valid file descriptor (in the open file table for the process).

Where is X?

From the top level of the repo, run:

```
grep -nR “X” .
```

-n gives the line numbers

For better results, ctags is a useful tool on attu (**man ctags**) with support built into vim and emacs. There are shortcuts in vim/emacs for jumping to where a function/type/macro/variable is defined when using ctags.

Staging of work

1. The global file table
2. User/Process file table
3. File functions
4. System calls

Questions?