



Lab 1

File syscalls



Agenda

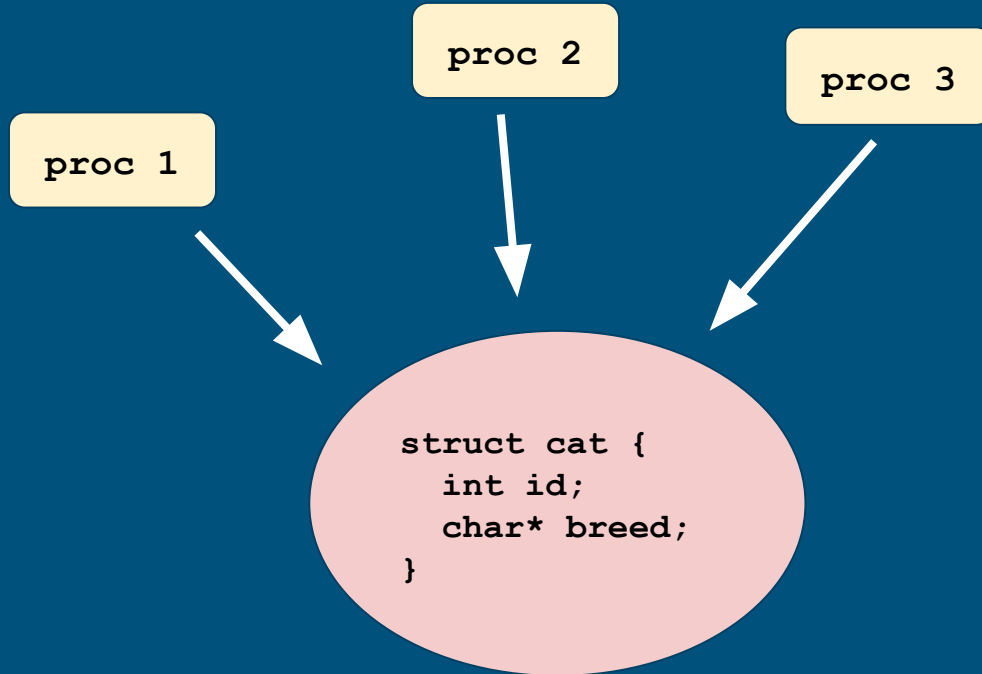
- Common Questions
 - Where/how to initialize global variables?
 - What is reference count for?
 - When should a new file info struct be allocated?
 - Kernel/user memory
 - Syscall/trapframe/kernel stack
- File syscalls Deep Dive

Global Variables in C

```
1 // All variables below are allocated inside the data segment
2 // when the program is loaded into memory
3
4 int num1;           // initialized to 0
5 int num2 = 4;      // initialized to 4
6 static int num3;   // initialized to 0
7
8 int arr1[10];      // each entry is initialized to 0
9 static int arr2[10] = {1, 2, 3}; // {1, 2, 3, 0, 0, 0, 0, ...}
10
11 void foo() {
12 | // ....
13 }
```

Global variables are automatically initialized to 0 at the time of declaration!

Reference Counting



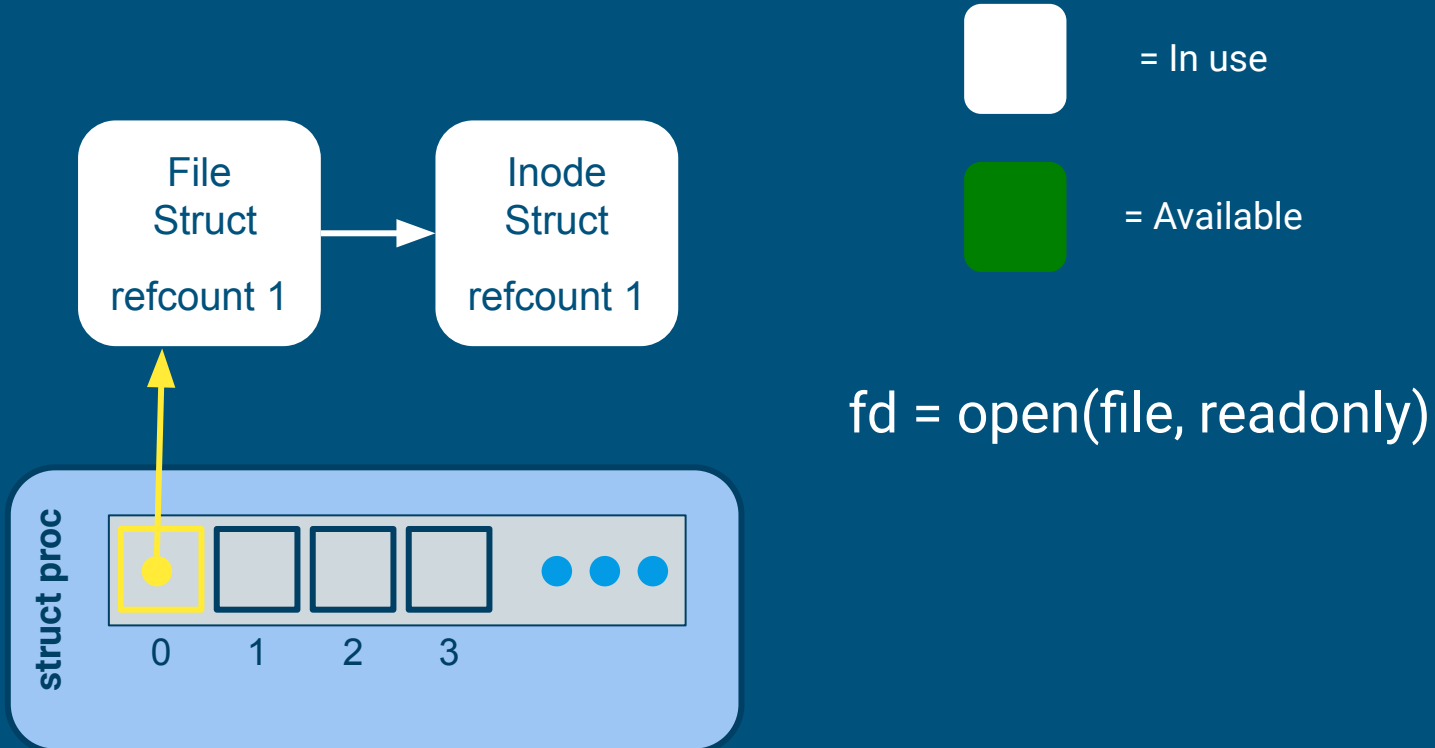
3 processes store a reference (ptr) to the struct cat

When is it safe to deallocate the struct cat?

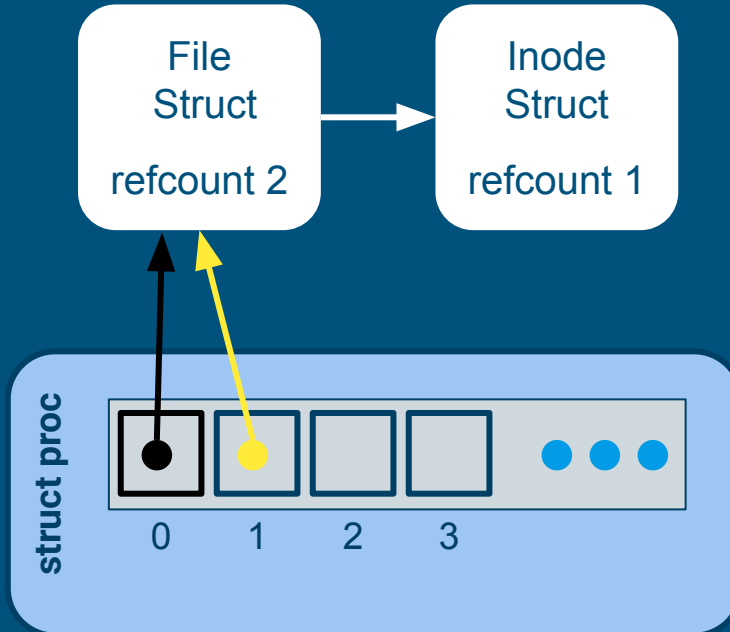
Reference Counting

- Purpose of referencing counting
 - keeps track of how many references are there for the object
 - so we can know when it's safe to deallocate things!
- Reference count is specific to each struct
 - file's reference count might be different from inode's
 - everytime you store the pointer of a file struct somewhere, the refcount goes up
 - open, dup
 - everytime you remove a reference of a file struct, refcount should go down
 - close

Reference Counting



Reference Counting



= In use



= Available

`dup(fd)`

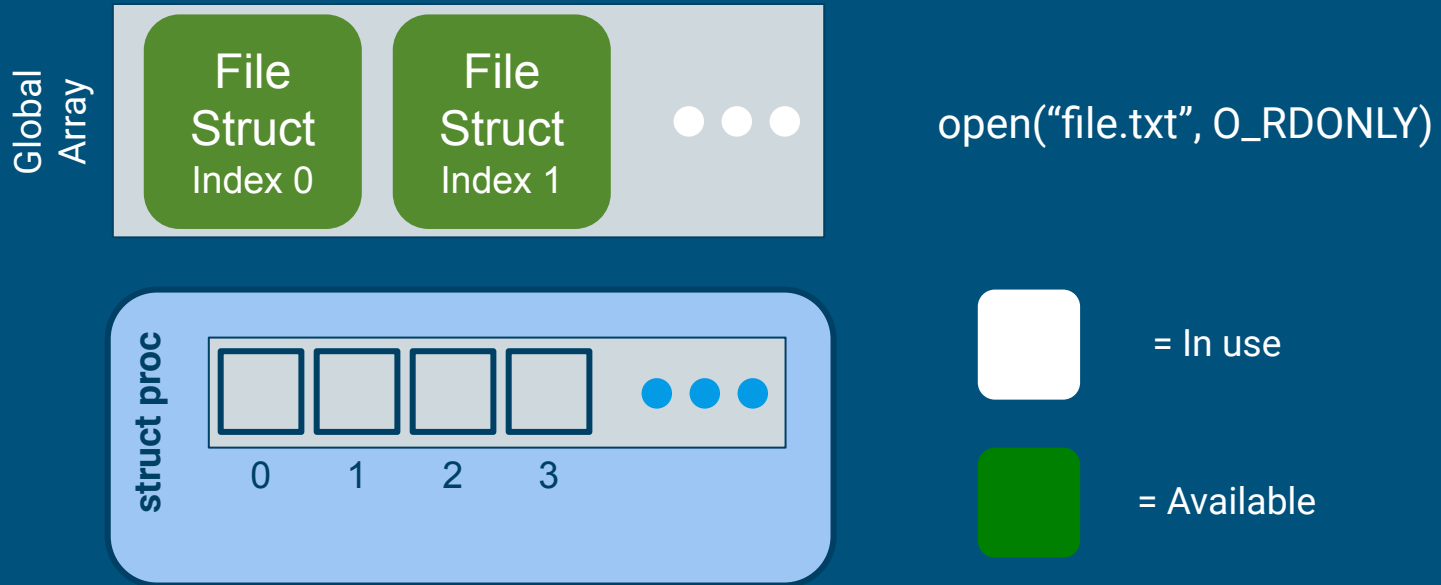
Multiple Open Calls on Same File

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

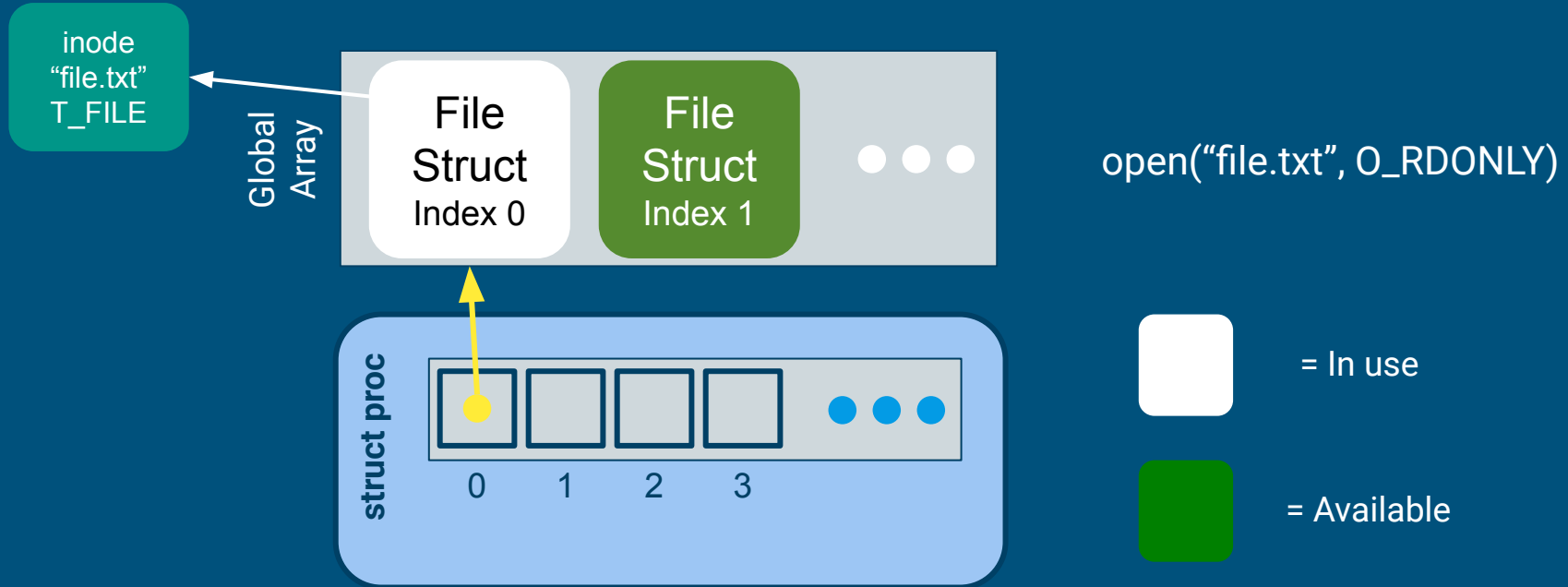
```
int fd1 = open("file.txt", O_RDONLY);
```

```
int fd2 = open("file.txt", O_RDONLY);
```

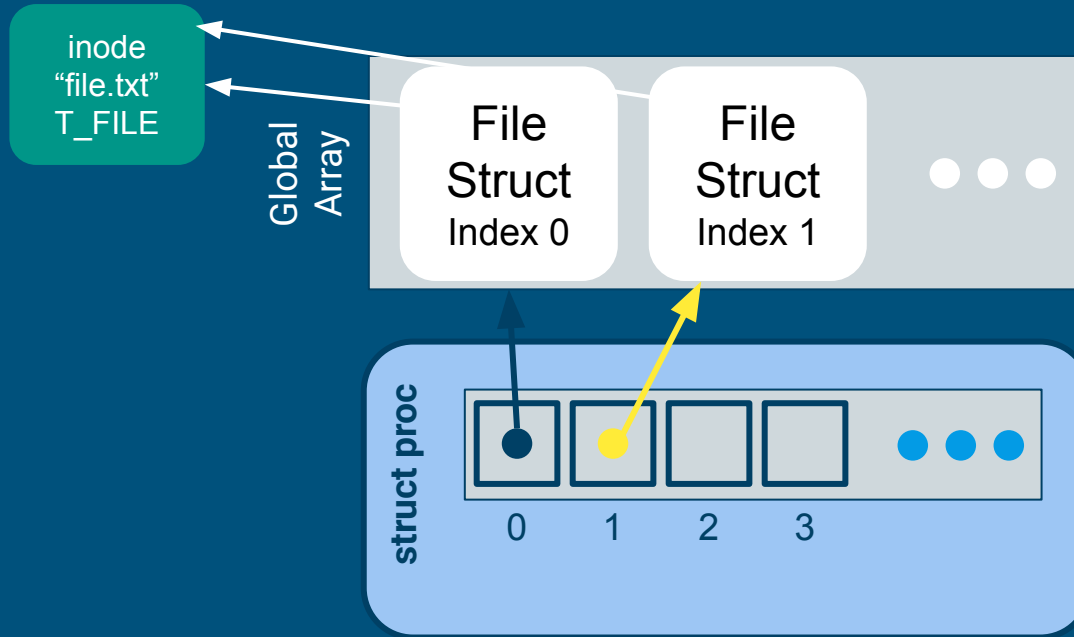

Multiple Open Calls on Same File



Multiple Open Calls on Same File



Multiple Open Calls on Same File



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

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

Memory: Kernel and User mode

- Read lab/memory.md (useful for lab 3, but also to understand some parts of lab 2)
- Each process has its own page tables that translate a virtual address to a physical address

Virtual memory
for a process:

The kernel is
mapped to the
top for every
process:

Why? Are there
any risks?

```
+-----+ <- 0xFFFFFFFFFFFFFFFF (18 exabytes)
|
| Kernel |
|
+-----+ <- KERNBASE = 0xFFFFFFFF80000000
|
| Unused |
|
+-----+ <- 2GB (vspace.regions[VR_USTACK].va_base)
|
| Stack  |
|
+-----+ <- vspace.regions[VR_USTACK].va_base - vspace.regions[VR_USTACK].size
|
| Unused |
|
+-----+ <- vspace.regions[VR_HEAP].va_base + vspace.regions[VR_HEAP].size
|
| Heap   |
+-----+ <- vspace.regions[VR_HEAP].va_base
|
| Text   |
|
+-----+ <- vspace.regions[VR_CODE].va_base
```

Kernel stack

- Can also be referred to as interrupt stack
- Each process has its own kernel stack
- However, this is in the kernel section of the memory
- In xk, the kernel allocates one page which acts as the kernel stack during process creation
- From kernel.proc.c:allocproc:

```
// Allocate kernel stack.  
if ((p->kstack = kalloc()) == 0) {  
    p->state = UNUSED;  
    return 0;  
}  
sp = p->kstack + KSTACKSIZE;
```

Interrupts, exceptions, syscall (review)

- Interrupts: triggered by hardware events (I/O), unrelated to the current instr
 - Ex: timer interrupt, keyboard input, disk I/O completion
- Exceptions: error caused by the current instr
 - Ex: divide by zero, segfault, pagefault
- Syscall: user requesting a service from the kernel
 - Ex: open(), close(), read()

All 3 involve a mode switch into the kernel!

Trap Frame

When an interrupt/exception/sys call occurs,

There is mode switch from User -> Kernel

However, we need to eventually move back to user space eventually

The kernel has a different `$rsp`, `$rip` and would change registers during execution

Trap frame stores all the registers into a struct so that it can be later restored when switching to user mode

Review



File Descriptors - Kernel View

- Kernel needs to give out file descriptors upon open
 - must be give out the smallest available fd
 - fds are unique per process (fd 4 in process A can refer to a different file than fd 4 in process B)
 - need to support NOFILE number of open files for each process
 - each process should know its fd to file mapping
- Kernel needs to deallocate file descriptors upon close
 - `close(1)` means that fd 1 is now available to be recycled and given out via open

Allocation of File Structs

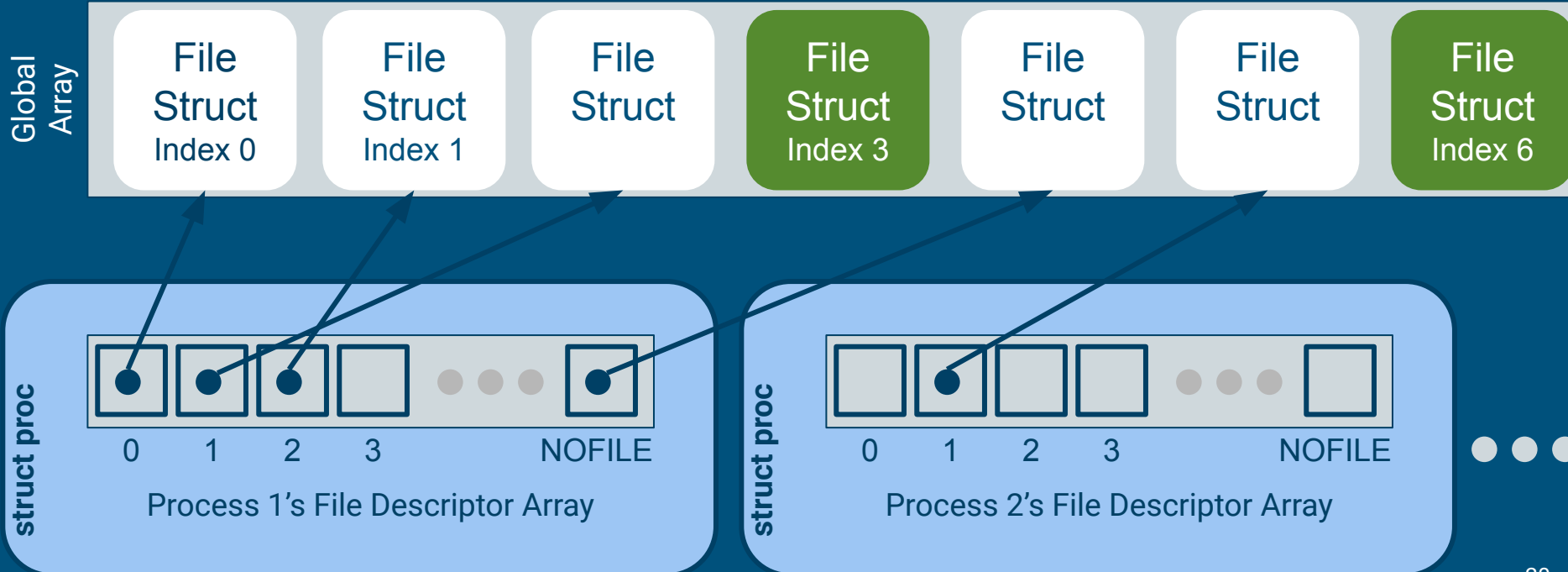
After defining the file struct, you need a way to allocate it.

You can statically allocate an array of file structs (need to support a total of NFILE entries)



Global File Table

fd = *index* into local File Descriptor Array



File System Functions

fileopen



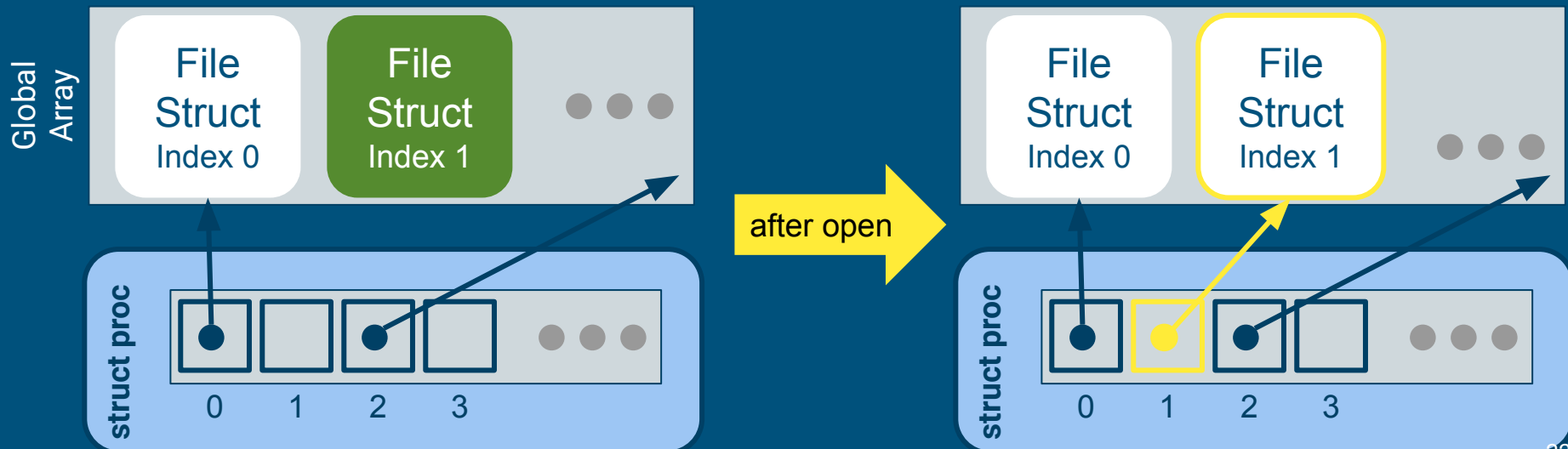
= In use



= Available

Finds an available file struct in the global file table to give to the process

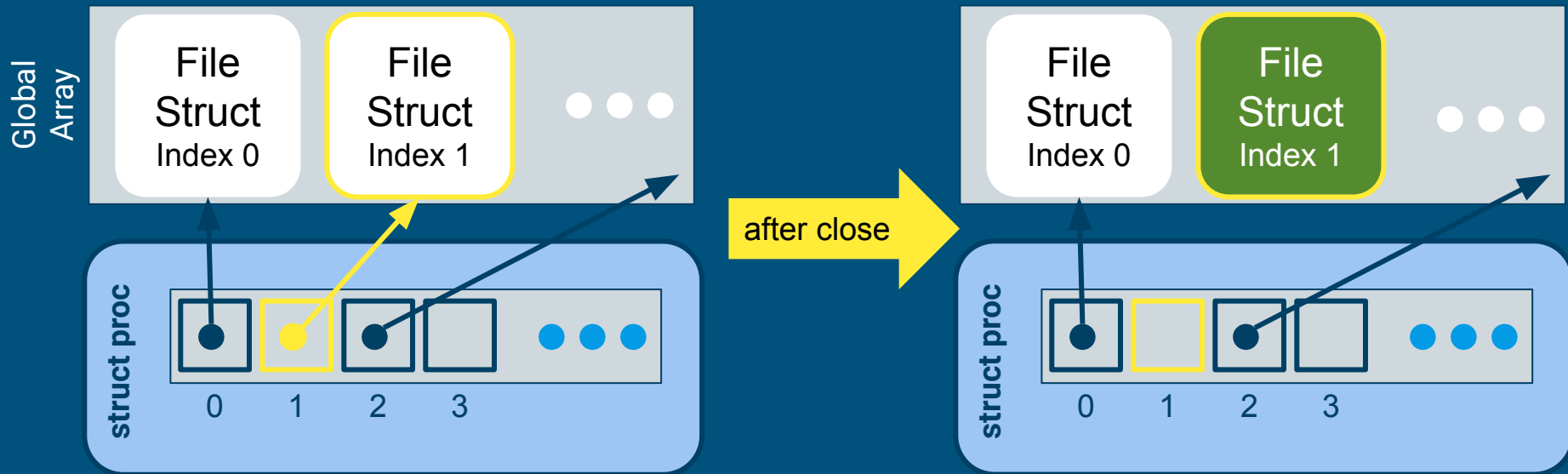
Hint: take a look at namei()



fclose

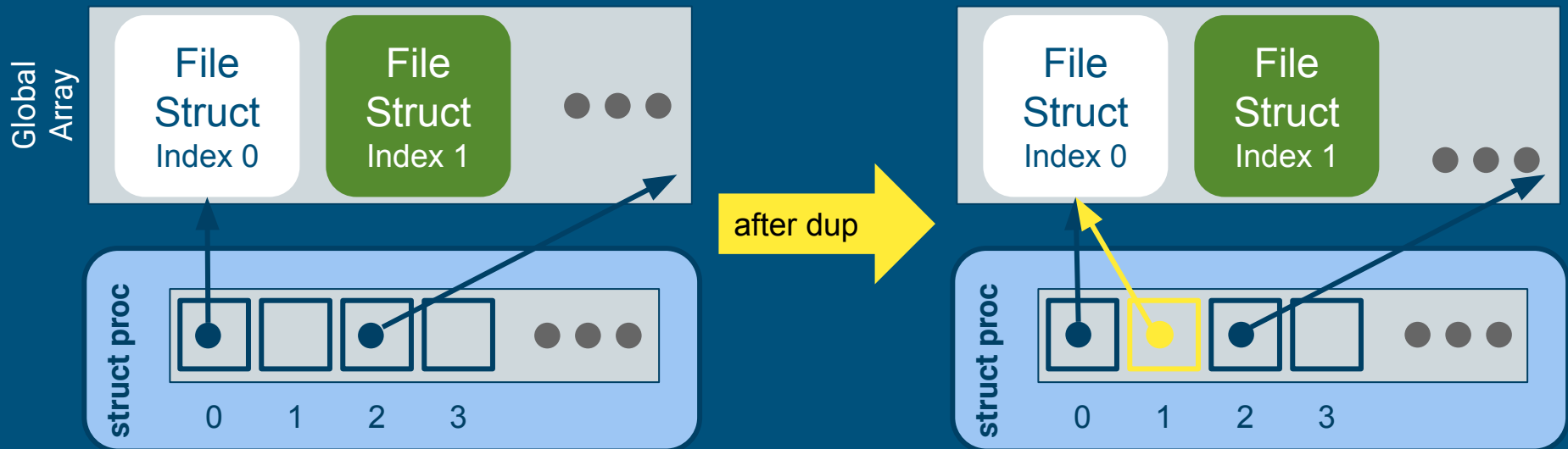
Release the file from this process, will have to clean up if this is the last reference

- make sure to `irelease()` before deallocating the file struct



filedup

Duplicates the file descriptor in the process' file descriptor table



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`

filestat

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

Lab 1 Test Program Code Fragment

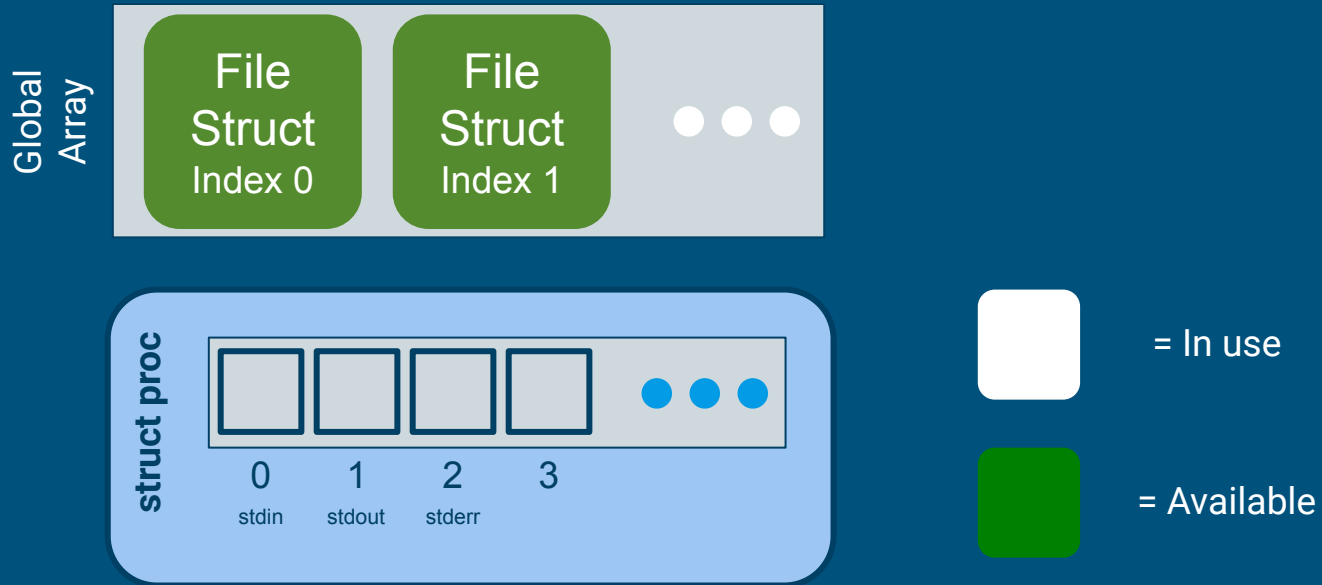
```
int stdout = 1;

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

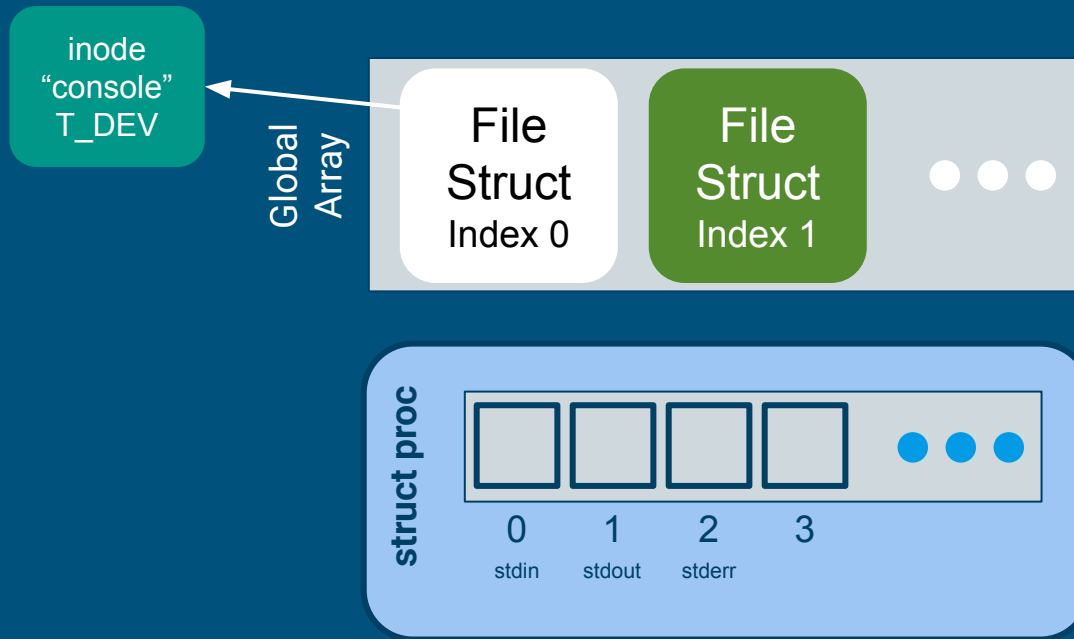
    printf(stdout, "hello world\n");
}
```

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

File Table View



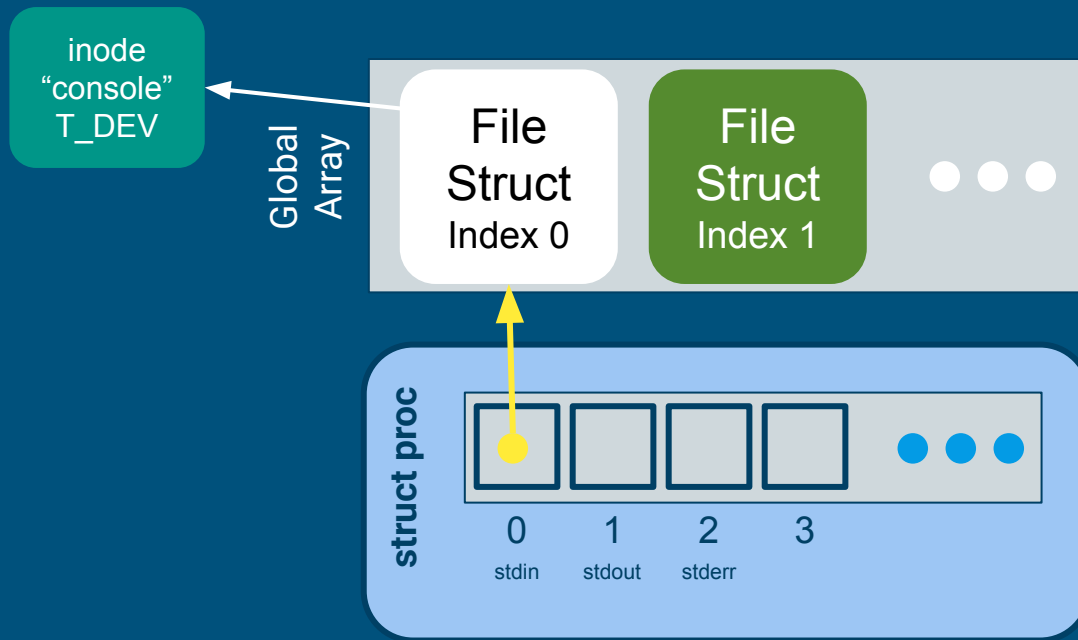
File Table View



`open("console", O_RDWR)`

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

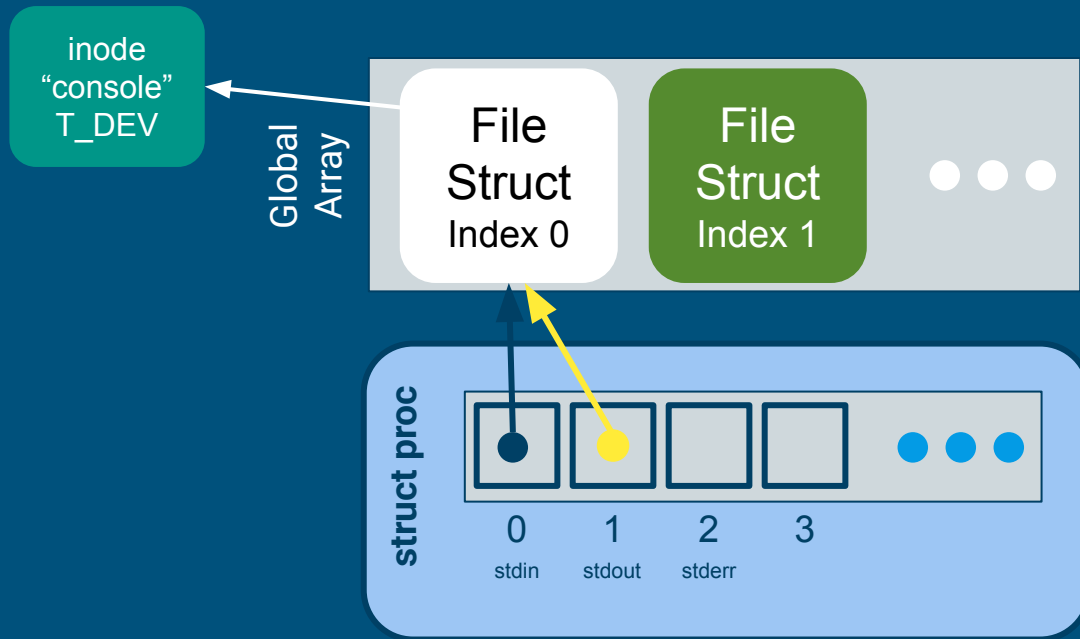
File Table View



`open("console", O_RDWR)`

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

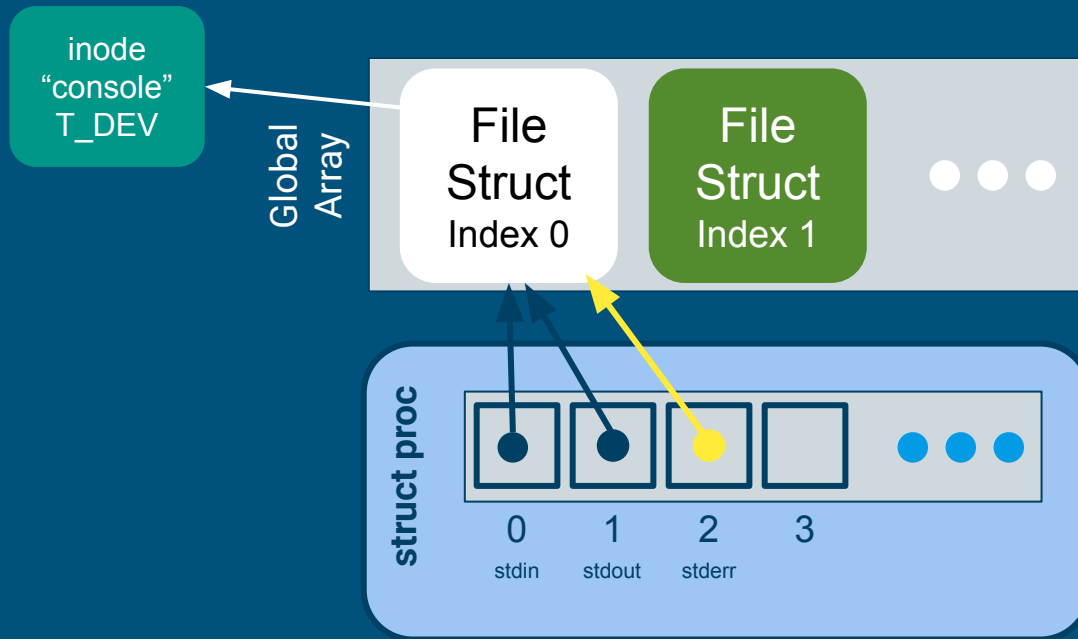
File Table View



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

- 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)
```


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 an `init` process. The `init` process is actually what opens the console device file, and every process inherits from `init`, which is why `stdin/stdout/stderr` are available on non-`init` processes.

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).

Questions?