



Lab 1: More Info

File syscalls



Administrivia

- Problem Set 1 due tomorrow! (**4/5/24**)
- Problem Set 2 out tomorrow!
 - 24-hour grace period on Problem Sets
- Lab 1 due Monday, **4/8/24**
 - **No submissions accepted** after 2 grace day period for **lab questions**
 - **Submissions accepted** with grade penalty after 2 grace day period for **code only**

Agenda

- Common Lab 1 Questions
 - Where/how to initialize global variables?
 - What are file tables?
 - What do “allocation” and “deallocation” mean?
 - What are reference counts for?
 - When should a new file info struct be allocated?
 - Kernel/user memory
 - Syscall/trapframe/kernel stack
- File syscalls Deep Dive

Global Variables



Global Variables: Why?



“I heard on StackOverflow global variables are bad, why are we learning about them?”
~ You, a well-intentioned student

Global Variables: Motivation

- Global variables are another tool in the toolkit.
- Very convenient for sharing across functions and modules
- `xk` already makes extensive use of global variables
- You will probably want to use global variables in your designs



Global Variables: Challenges

- However, often there's confusion on how global variables are initialized.
- ⇒ Let's remedy that



Global Variable Initialization in C

```
// All variables below are allocated inside the data segment
// when the program is loaded into memory
```

```
int num1;           // initialized to 0
int num2 = 4;       // initialized to 4
```

```
static int num3;
```

```
int arr1[10];
static int arr2[10] = {1, 2, 3};
```

```
typedef struct Point {
    int x;
    int y;
} Point;
Point p = {1, 2};
```

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

What values will the variables without comments have?

Global Variable Initialization in C

```
// All variables below are allocated inside the data segment
// when the program is loaded into memory

int num1;           // initialized to 0
int num2 = 4;      // initialized to 4

// `static` means internal linkage, variable only visible
// within this translation unit (i.e.: this file).
static int num3;   // initialized to 0

int arr1[10];      // Each entry is initialized to 0
static int arr2[10] = {1, 2, 3}; // {1, 2, 3, 0, 0...}

typedef struct Point {
    int x;
    int y;
} Point;

Point p = {1, 2}; // Initialized to x = 1, y = 2.
```

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

What values will the variables without comments have?

Refocusing on the labs

So you're now an expert on C globals, but what does this have to do with the labs again?

A: Your global file table will be a global variable!

File Tables

File Tables: Motivation

You create a handy `struct file_info` for tracking your file information.

...Where will these `struct file_infos` actually exist?

- stack?
- heap?
- data segment? (static/global data)

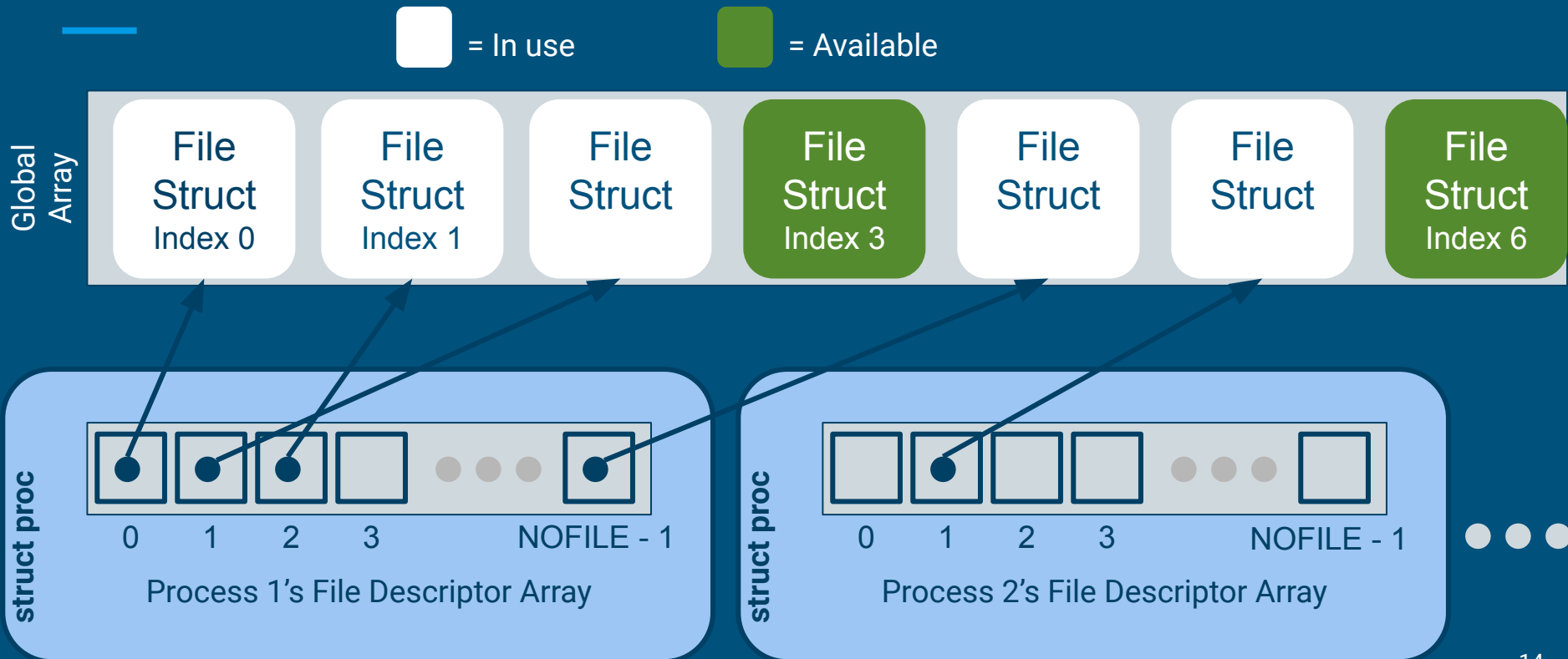
Suggested File Table Design

- The Lab 1 Spec hints at the intended file table design:
 - One “global file table”: a global array of `struct file_info`'s
 - A “process file table” per process: An array of pointers to entries in the global file table

The next slide shows what that would look like.

Global File Table Diagram

fd = *index* into local File Descriptor Array



File Tables: Why the indirection?



“Why have two layers of tables? Why not just have the per-process file tables store `struct file_infos` directly?”

~You, an astute student.

File Tables: Indirection Motivation

- Having `struct proc` directly store table of `struct file_infos` causes problems
 - How would dup work?
 - Requires an indirection mapping fds to open files
- Once we introduce multiprocessing, multiple processes can reference the same logical file
 - We'll use this to implement inter-process communication with pipe
 - It's how shells are often implemented
- So open files need to be available globally

File Tables: Where do they go?

So where/how is memory allocated for these tables?

For the global table, you can statically allocate a global array of file structs! (**need to support at least NFILE entries**)

For the per-process tables: you can include the table as a field of `struct proc` (**need to support at least NOFILE entries**)

Global File Table Notes

- Process file table entries point to elements (`struct file_info`) of global open file table.
- The “file descriptor” (fd) is the index into the process file table.

Defining “Allocate” and “Deallocate”



Motivation

- Earlier in the file table section we say some file table slots are “used” or “unused”
 - Clearly we need to know so that we don’t trample other files’ metadata
 - ... but how do we know if a file is in use?

Defining “allocation” and “deallocation”

“Allocation” means *marking a resource as used*. Examples:

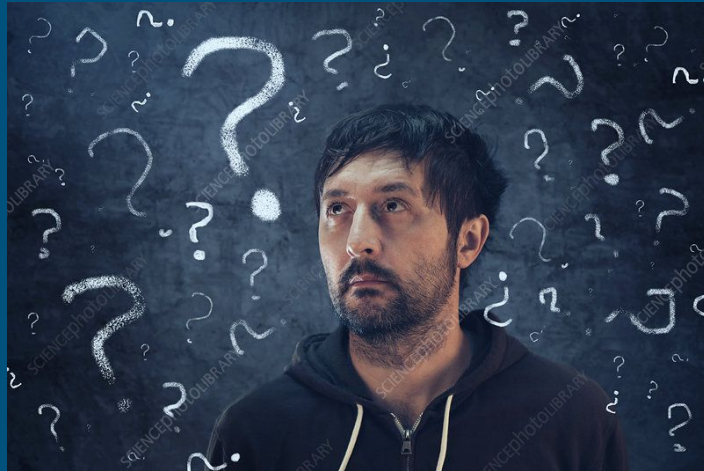
- 333’s heap allocator. It marks chunks of memory as used using bitflags.
- The global file table, each `struct file_info` needs to be marked as used/unused.
 - (hint: although it’s totally okay to add a “used” field, using an existing field in `struct file_info` may also work for this purpose)

“Deallocation” just means marking a resource as unused (inverse of however its done for allocation).

Allocation/Deallocation: Transitioning

But how do we actually know when we can allocate a resource? (i.e.: how do we know it's free?)

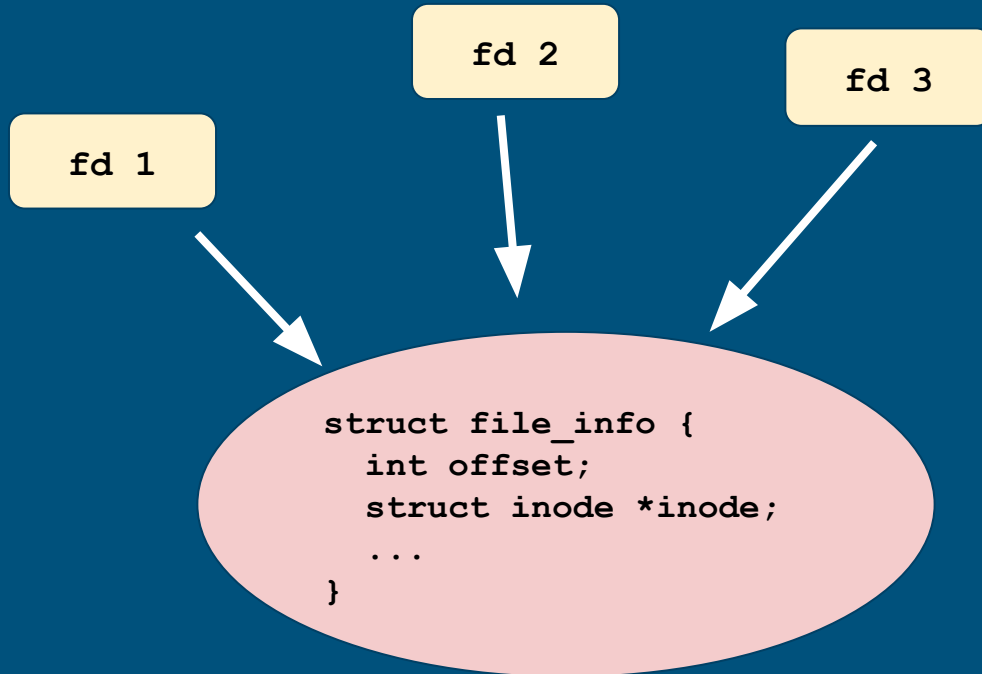
How can we know when we can deallocate it?



Reference Counting



Reference Counting Diagram



3 fds reference the
struct file_info

When is it safe to
deallocate the struct
file_info?

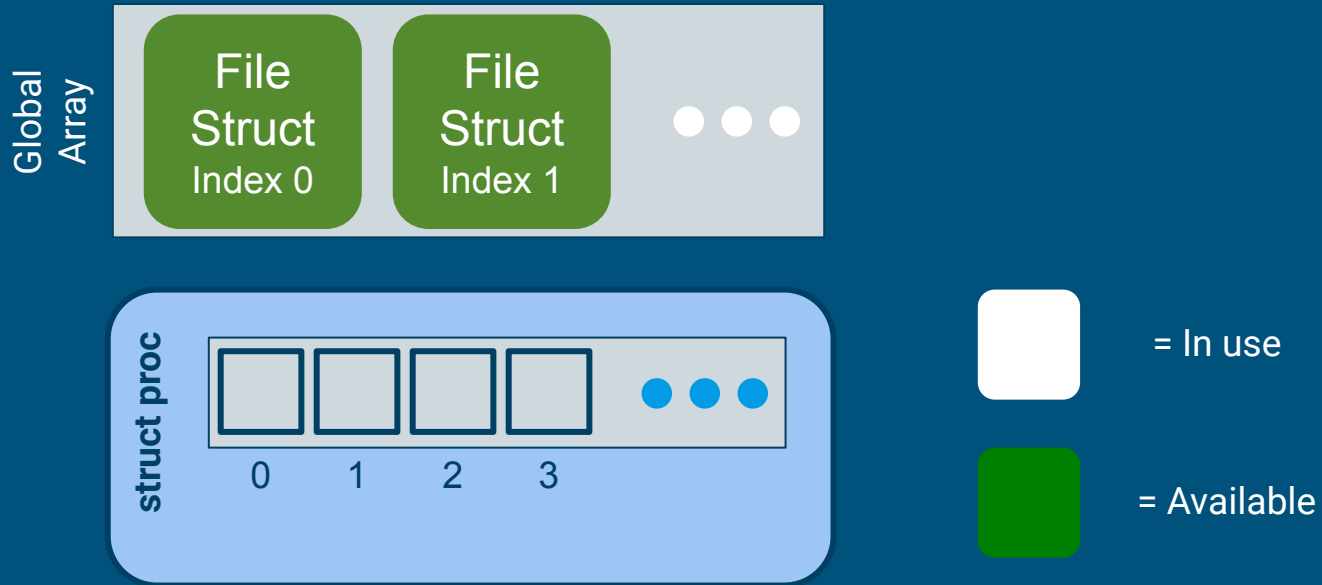
Reference Counting Notes

- Reference count is specific to each struct
 - Note that a file's ref count is different from an inode's refcount
- Everytime you store the pointer of a file struct somewhere, refcount goes up
 - open, dup
- everytime you remove a reference to a file struct, refcount goes down
 - close

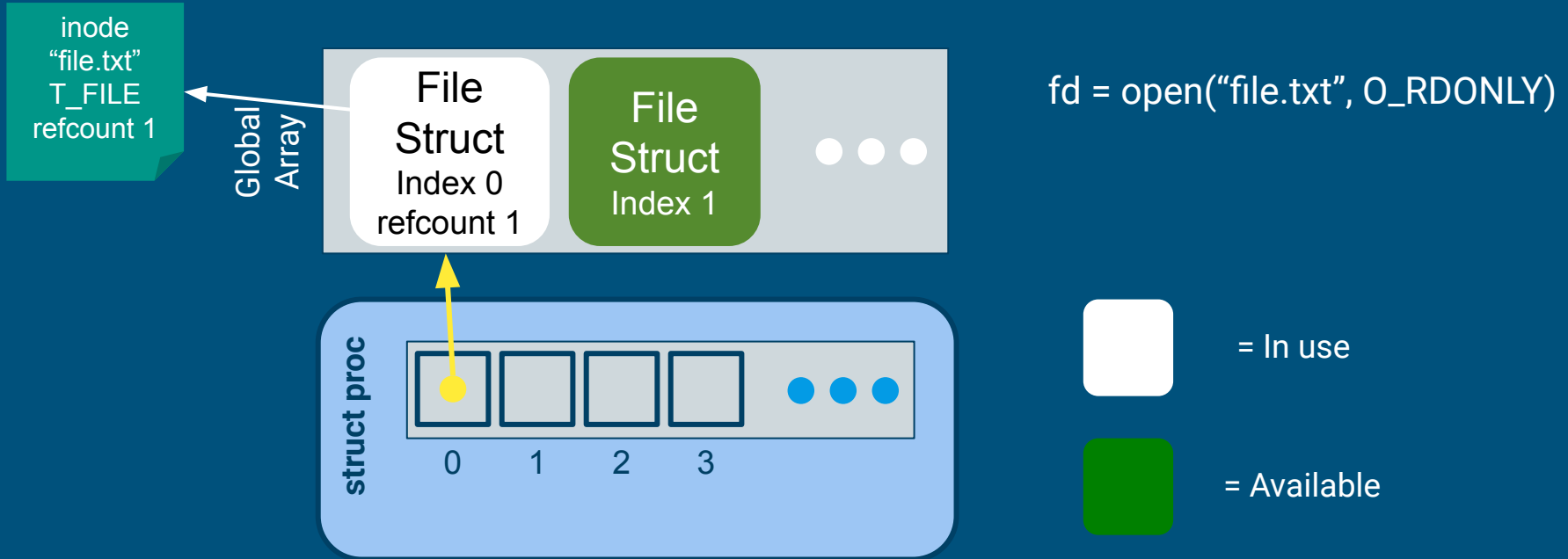
The Simple Rule:
Just count the
number of *direct*
references.

Now let's step through some examples.

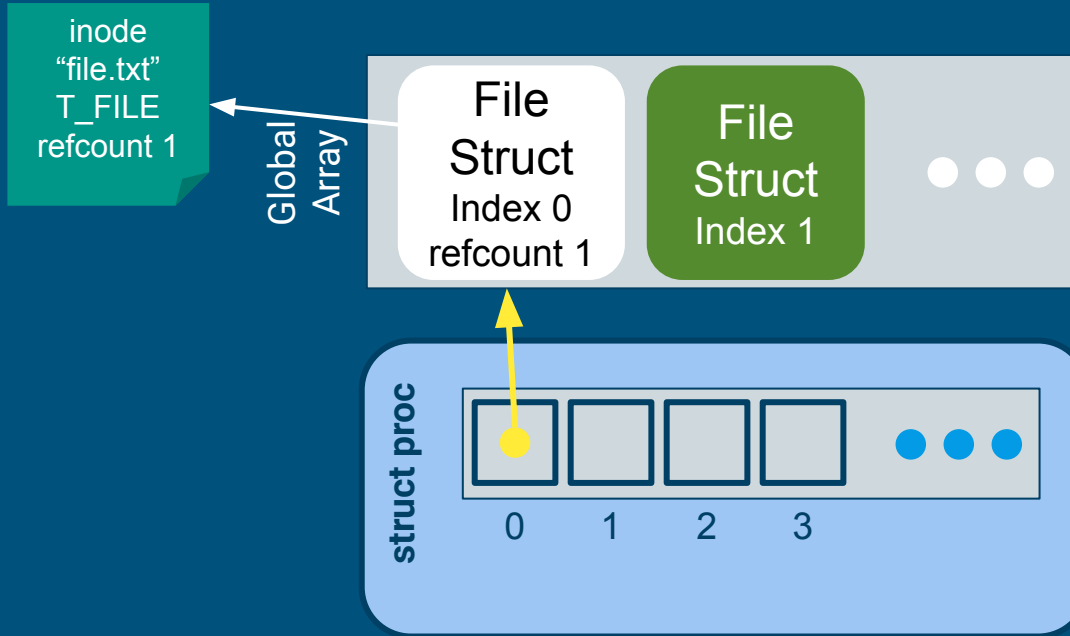
Multiple Open Calls on Same File



Multiple Open Calls on Same File



Multiple Open Calls on Same File

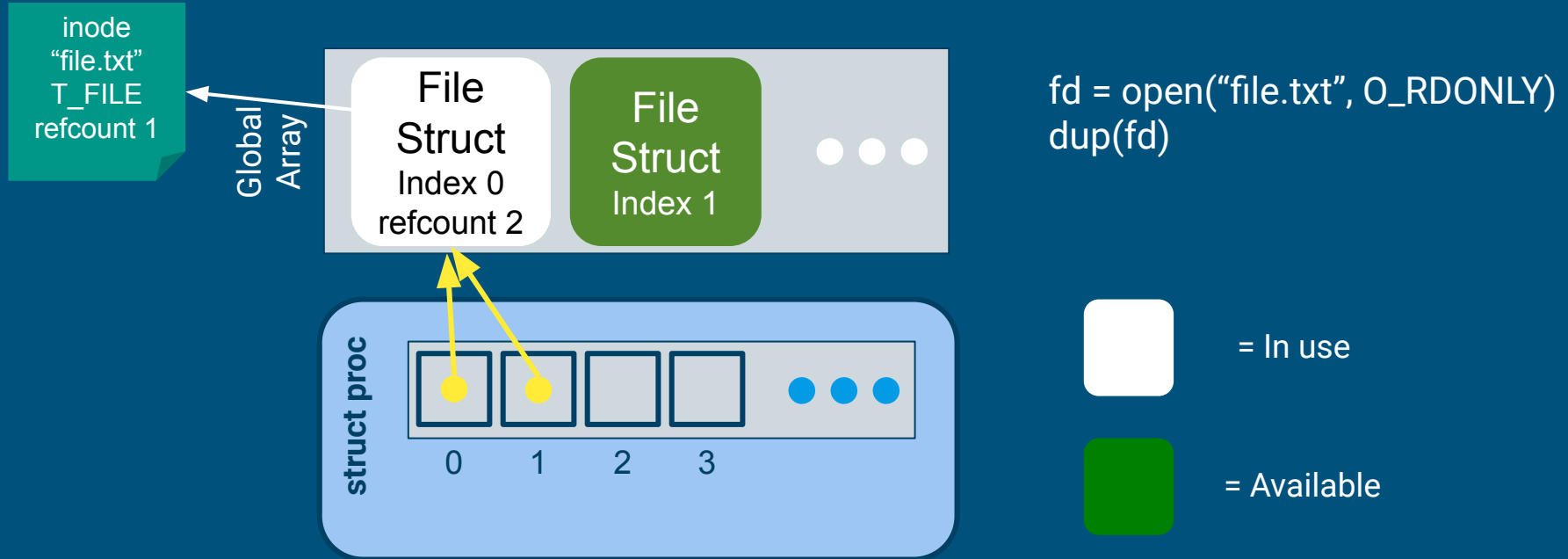


```
fd = open("file.txt", O_RDONLY)
dup(fd) // what happens?
```

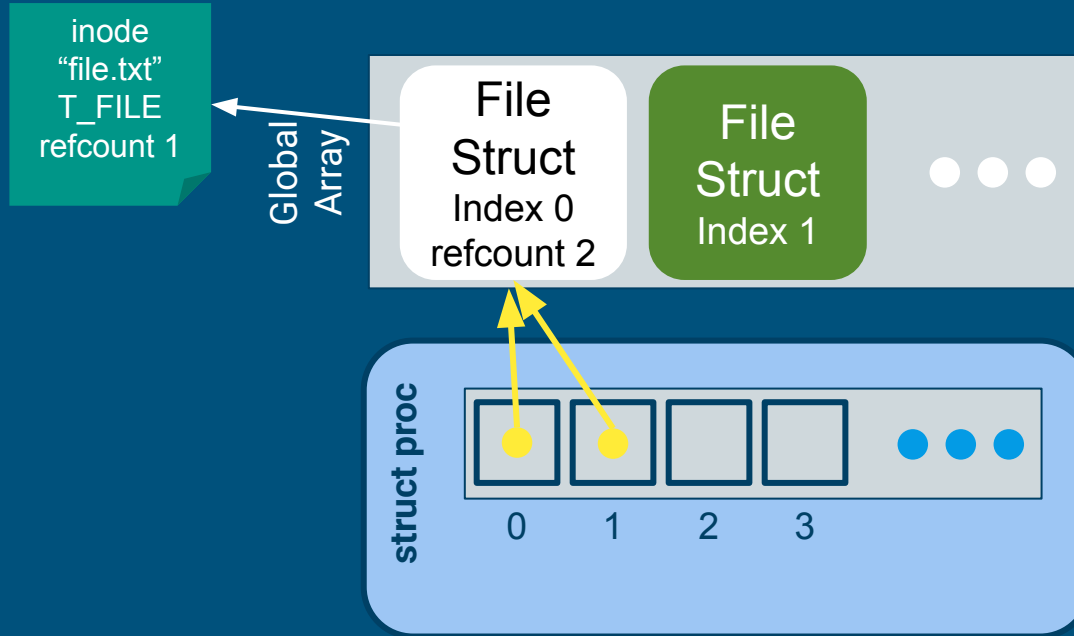
 = In use

 = Available

Multiple Open Calls on Same File



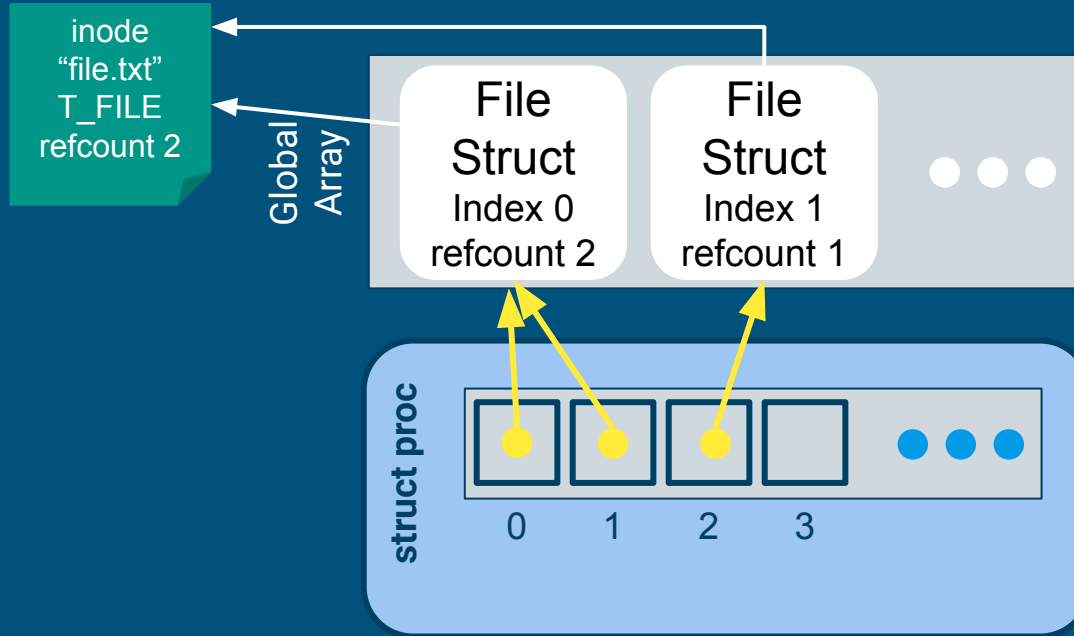
Multiple Open Calls on Same File



```
fd = open("file.txt", O_RDONLY)
dup(fd)
fd3 = open("file.txt", O_RDWR)
// what happens?
```



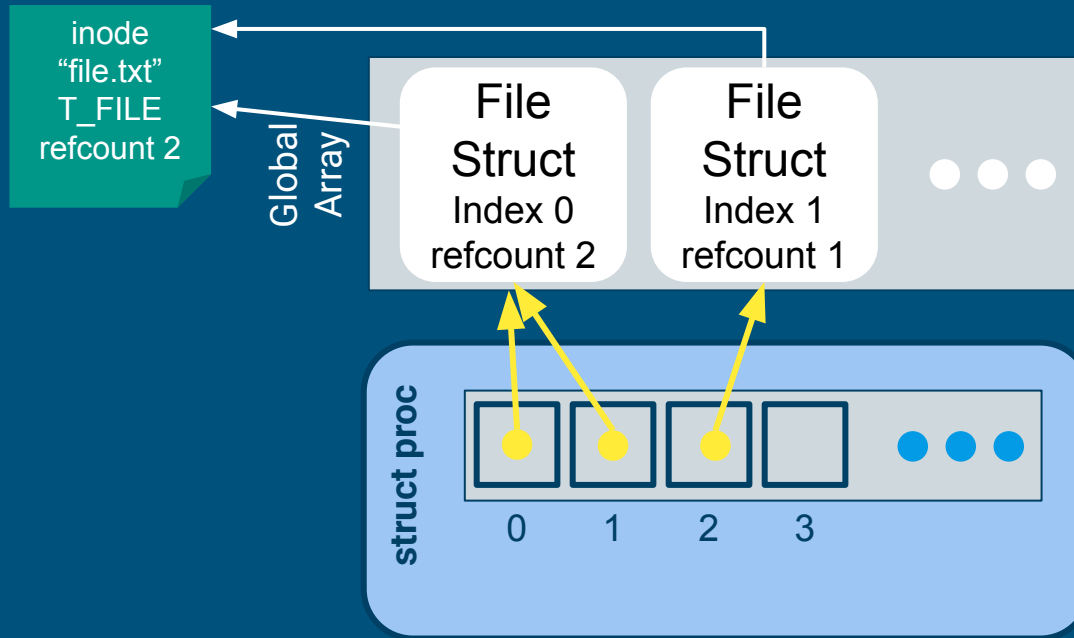
Multiple Open Calls on Same File



```
fd = open("file.txt", O_RDONLY)
dup(fd)
fd3 = open("file.txt", O_RDWR)
```



Multiple Open Calls on Same File



```
fd = open("file.txt", O_RDONLY)
dup(fd)
fd3 = open("file.txt", O_RDWR)
```

- Each open call allocates a new file_info struct
- Name lookup returns same inode
- **Don't worry** about managing inode refcount for this lab!

Console

Console Input/Output

- The console is a special file called “console”!
 - Special file marked as a device
 - Where? Look at kernel/fs.c, inc/file.h and how the `T_DEV` file type is used.
- Code to support devices is already handled for you
 - Its information is already provided when you fetch the device file from inode layer.
- 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

Argument Parsing & Validation

What do we mean by “parsing” and “validating”?

What does this look like in xk?

```
98 // Fetch the nth word-sized system call argument as a pointer
99 // to a block of memory of size bytes. Check that the pointer
100 // lies within the process address space.
101 int argptr(int n, char **pp, int size) {
102     int64_t i;
103     struct vregion *r;
104     struct vspace *v;
105
106     if (argint64(n, &i) < 0)
107         return -1;
108     if (size < 0)
109         return -1;
110
111     v = &myproc()->vspace;
112     for (r = v->regions; r < &v->regions[NREGIONS]; r++) {
113         if (vregioncontains(r, i, size)) {
114             *pp = (char*)i;
115             return 0;
116         }
117     }
118     return -1;
119 }
```

System Calls

- `sys_open`, `sys_read`, `sys_write`, `sys_close`, `sys_dup`, `sys_fstat`
- What are the main goals of `sys` functions?
 - Argument parsing and validation (never trust the user!)
 - E.g. resolve FD -> `file_info*`
 - Call associated file functions

Parsing & Validation Helper Functions

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).

File API Recap

Let's recap the specs for the filesystem
syscall APIs!

fileopen



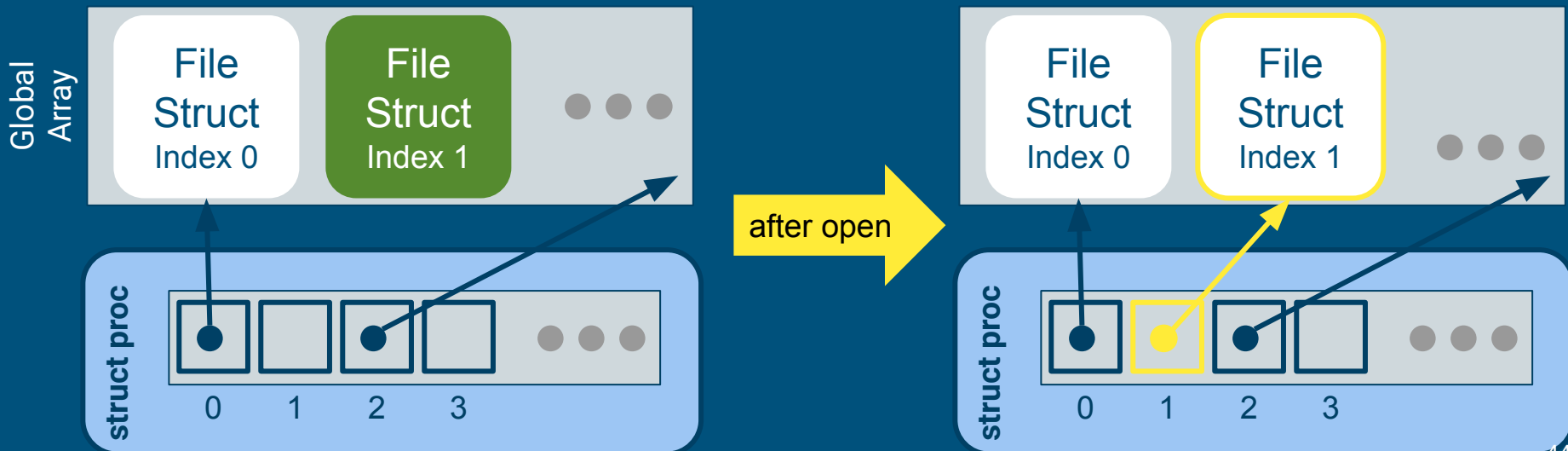
= In use



= Available

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

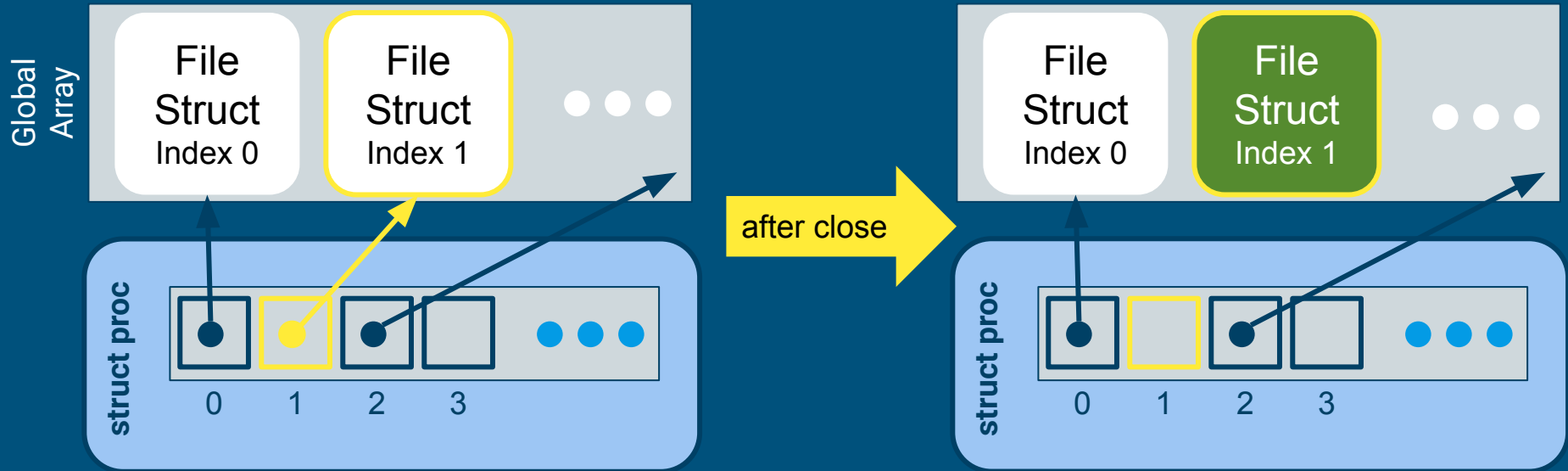
Hint: to obtain the inode for the desired file, take a look at `iopen()`



fileclose

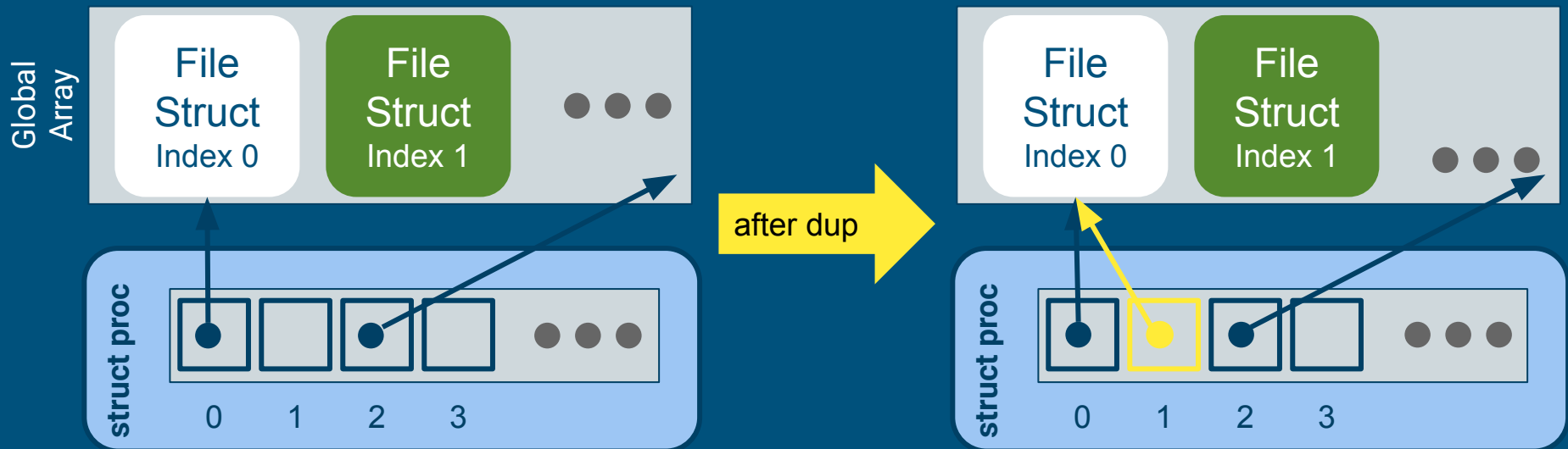
Release the file from this process, will have to clean up if this is the last reference

- make sure to irelease() the inode 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 *concurrent_readi* and *concurrent_writei* defined in `kernel/fs.c`

filestat

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

Useful for testing

- For example, you can use it to find the size of a file
- We use it extensively to test your implementation :)

Questions?