Lab 1: More Info

File syscalls

Administrivia

- Problem Set 1 out!
  - Due **10/9/24**, 11:59pm
  - **No submissions accepted** after 2 grace day period
- Lab 1 due Wednesday, **10/9/24**
  - **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?
- System Calls
- File_* API Recap

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

```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)

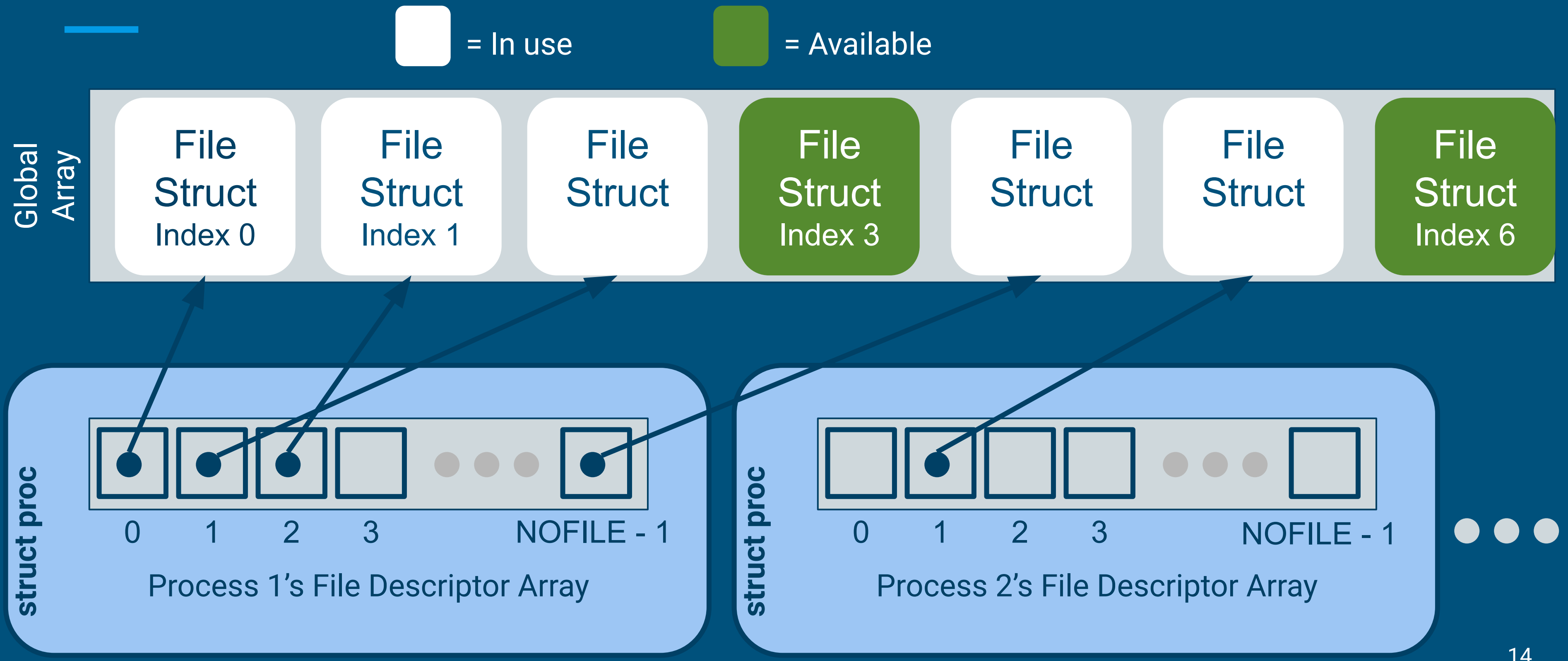
Suggested File Table Design

- 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?

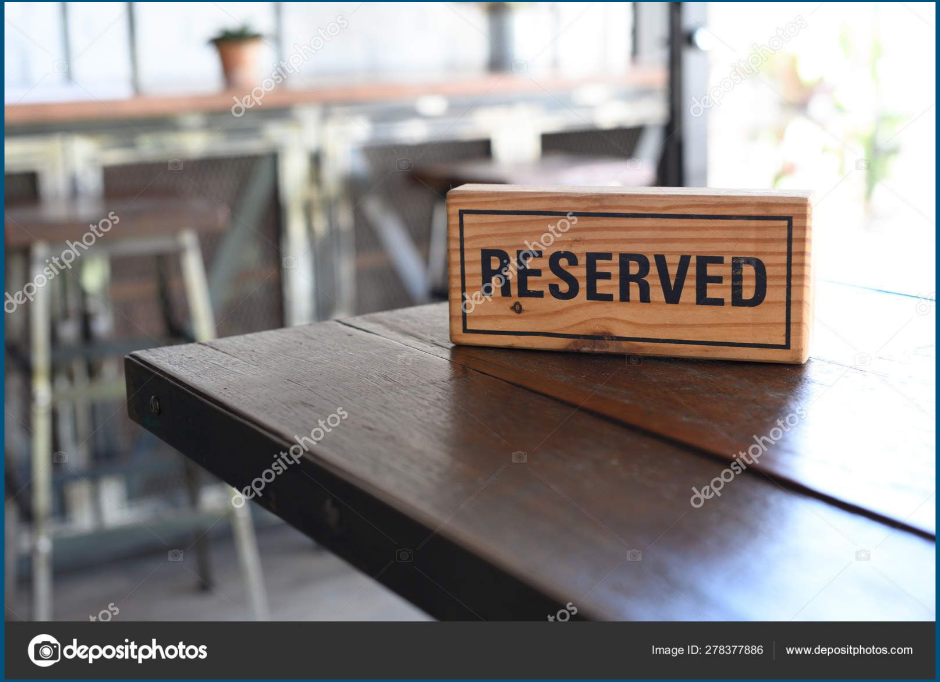
For the global table, you can statically allocate a global array of file structs! (**need to support at least NFILE entries**)

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:

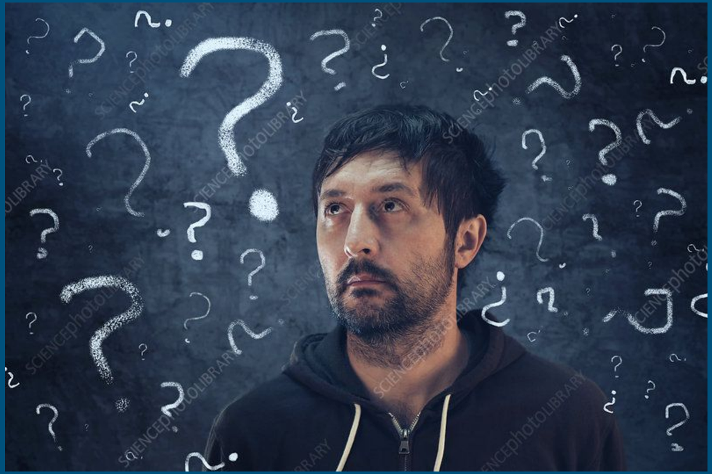
- 333’s heap allocator. It marks chunks of memory as used using bitflags.
- The global file table, each struct file_info needs to be marked as used/unused.
    - (hint: although it’s totally okay to add a “used” field, using an existing field in struct file_info may also work for this purpose)

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

Allocation/Deallocation: Transitioning

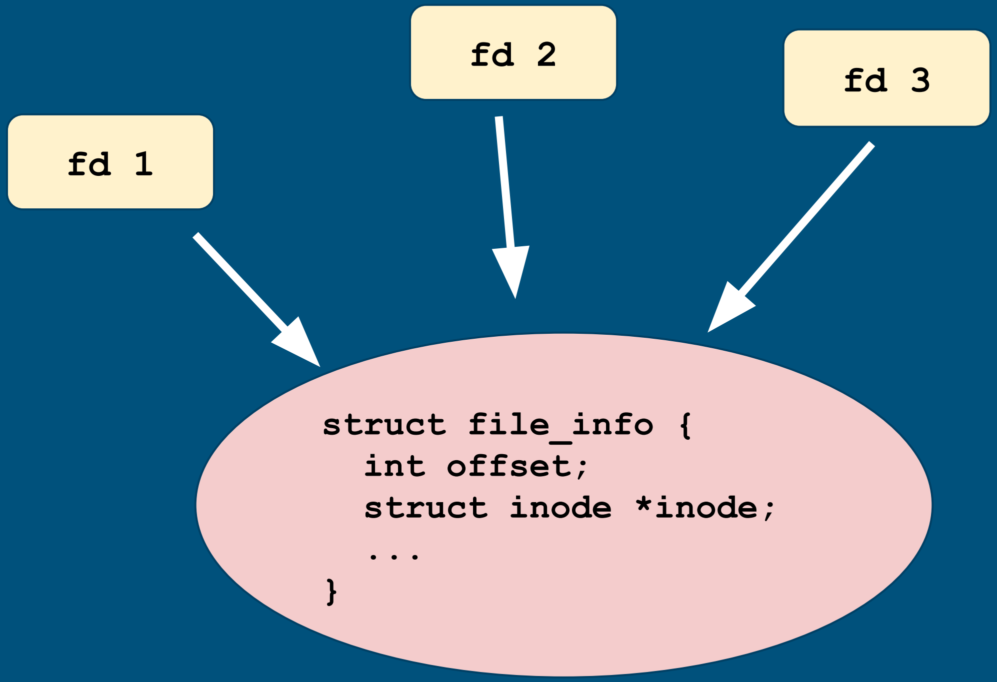
But how do we actually know when we can allocate a resource? (i.e.: how do we know it's free?)

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?

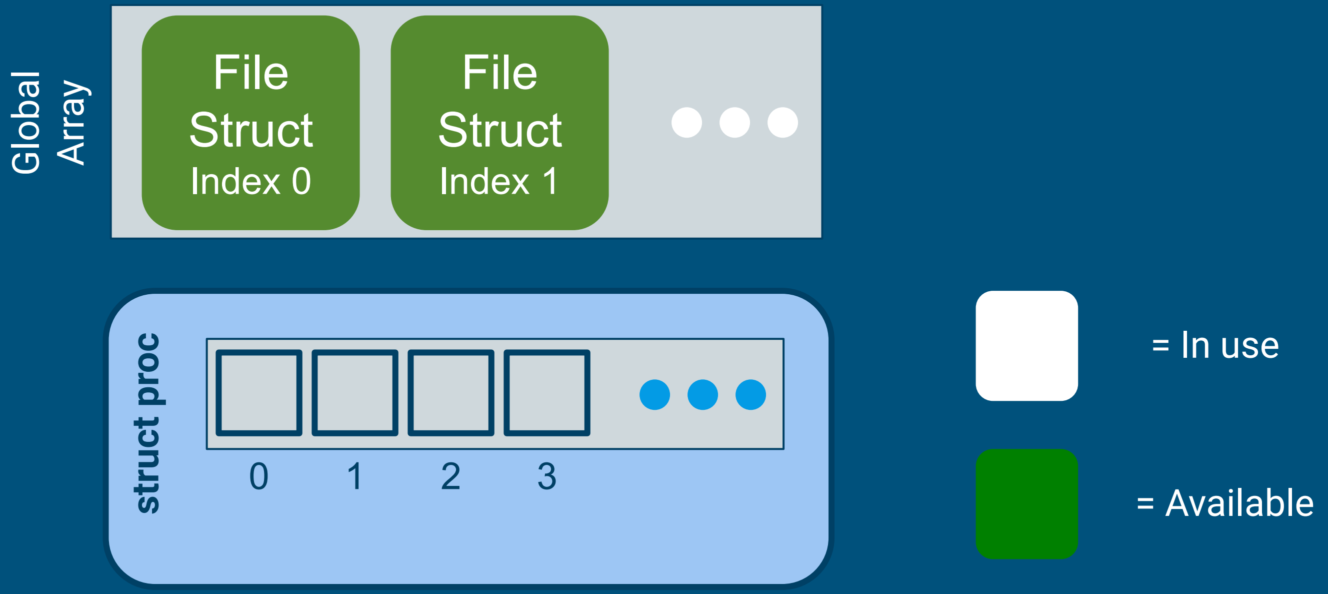
Reference Counting Notes

- 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

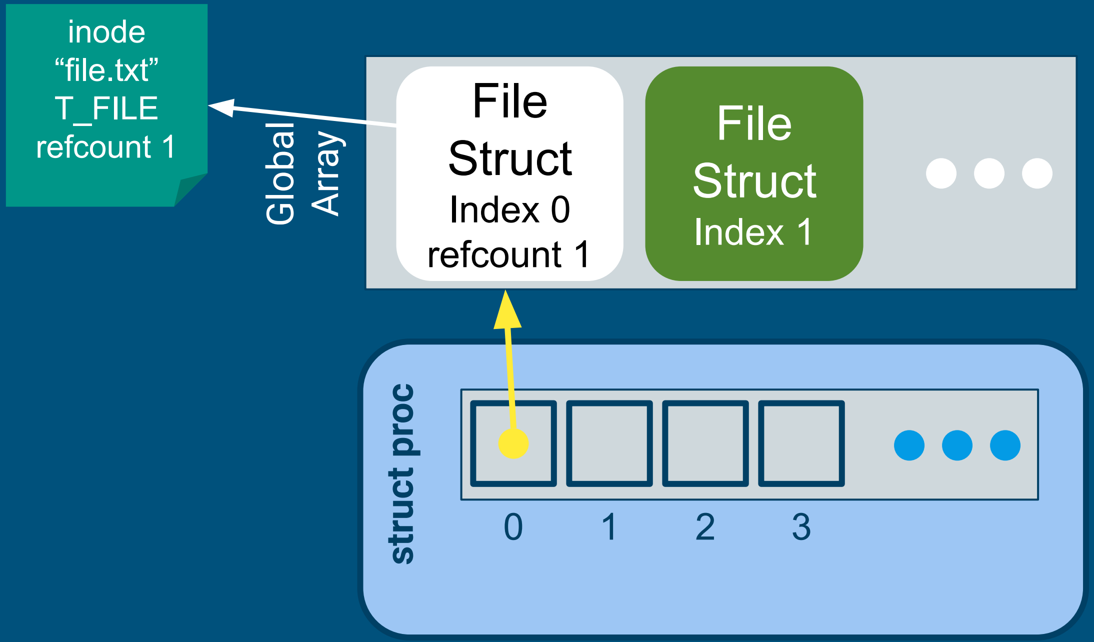
The Simple Rule: Just count the number of *direct references*.

Now let's step through some examples.

Multiple Open Calls on Same File

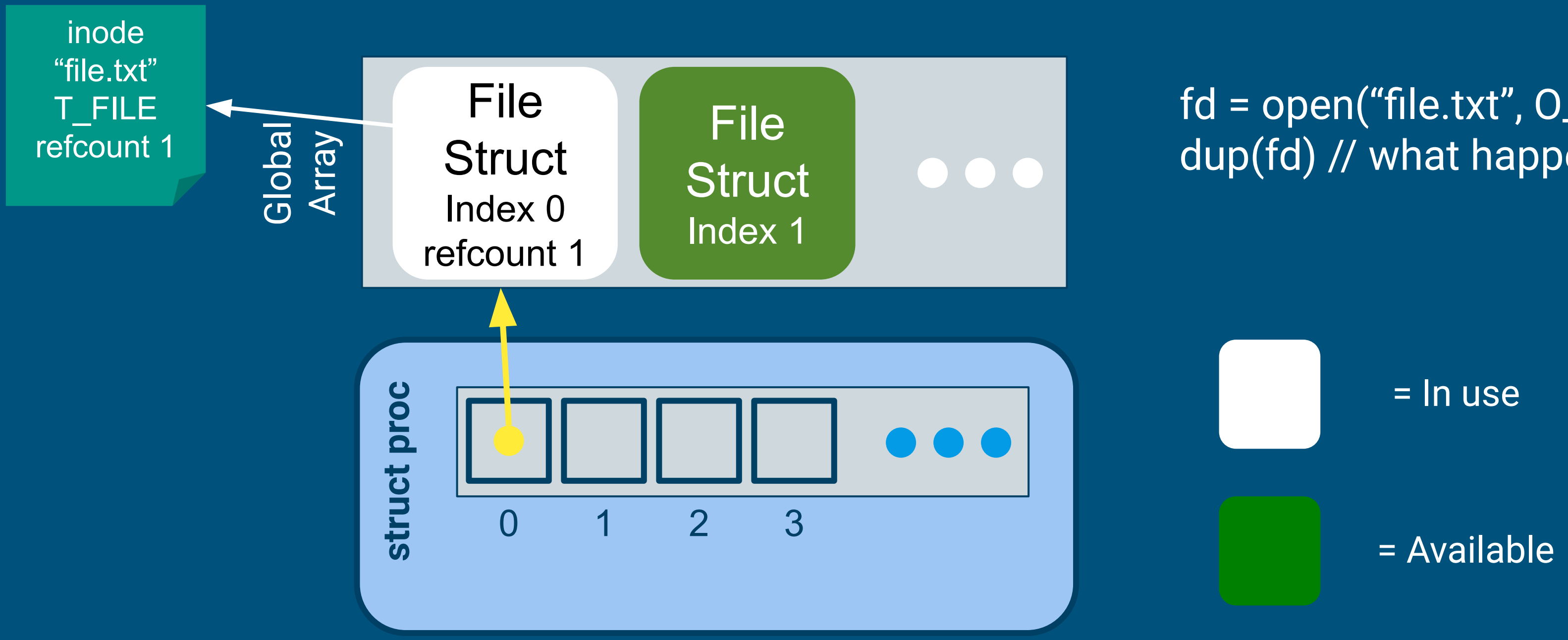


Multiple Open Calls on Same File



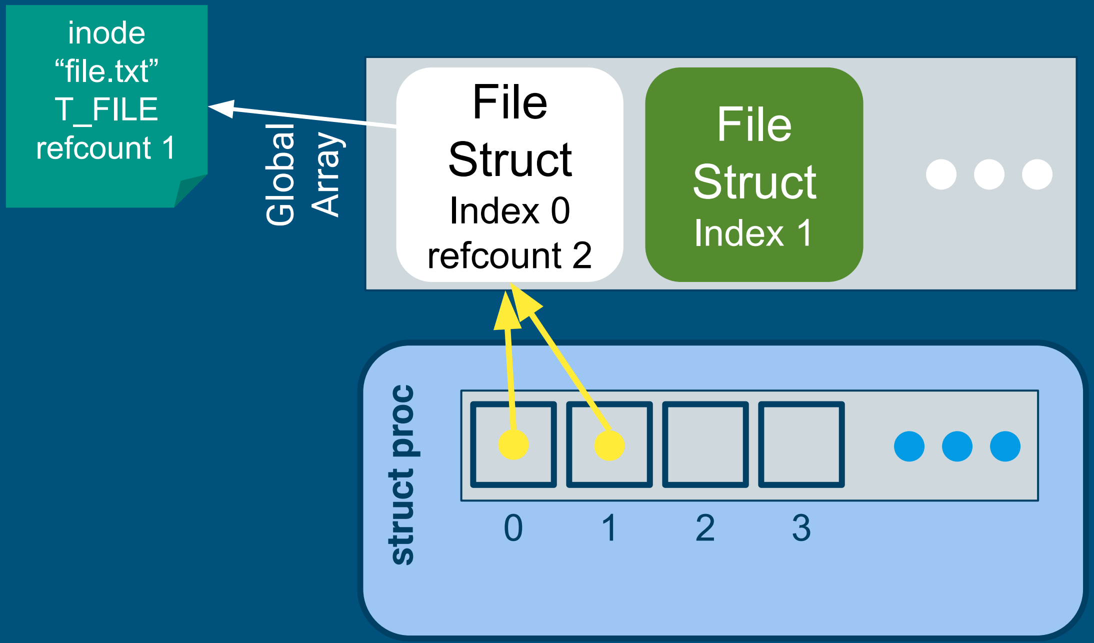
fd = open("file.txt", O_RDONLY)

Multiple Open Calls on Same File



fd = open(“file.txt”, O_RDONLY)
dup(fd) // what happens?

Multiple Open Calls on Same File

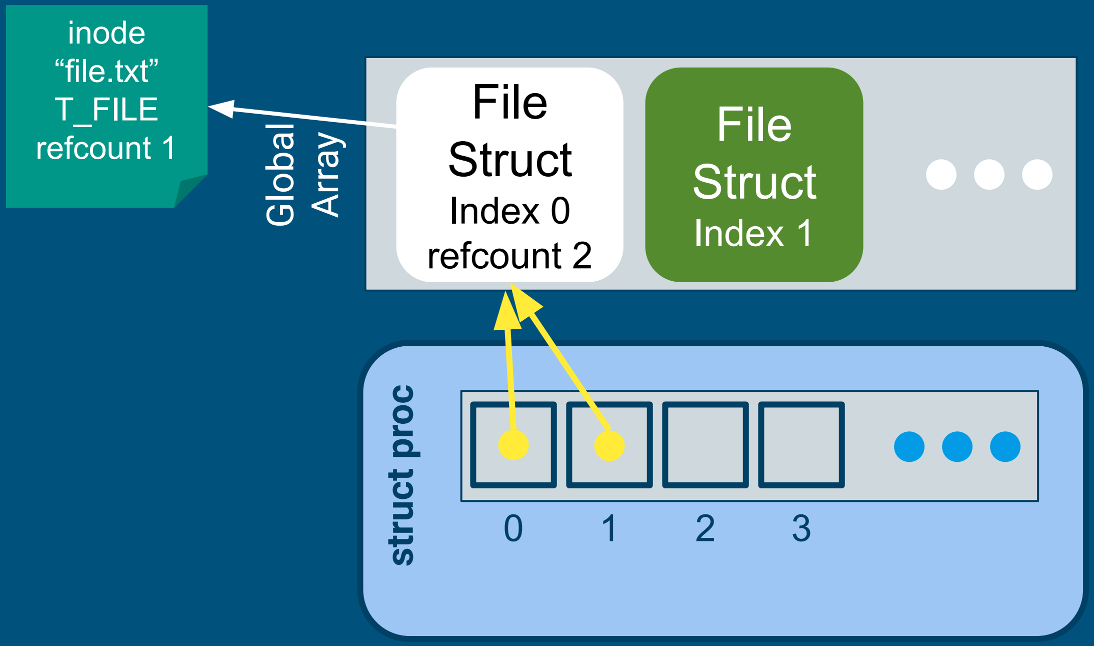


fd = open(“file.txt”, O_RDONLY)
dup(fd)

= In use

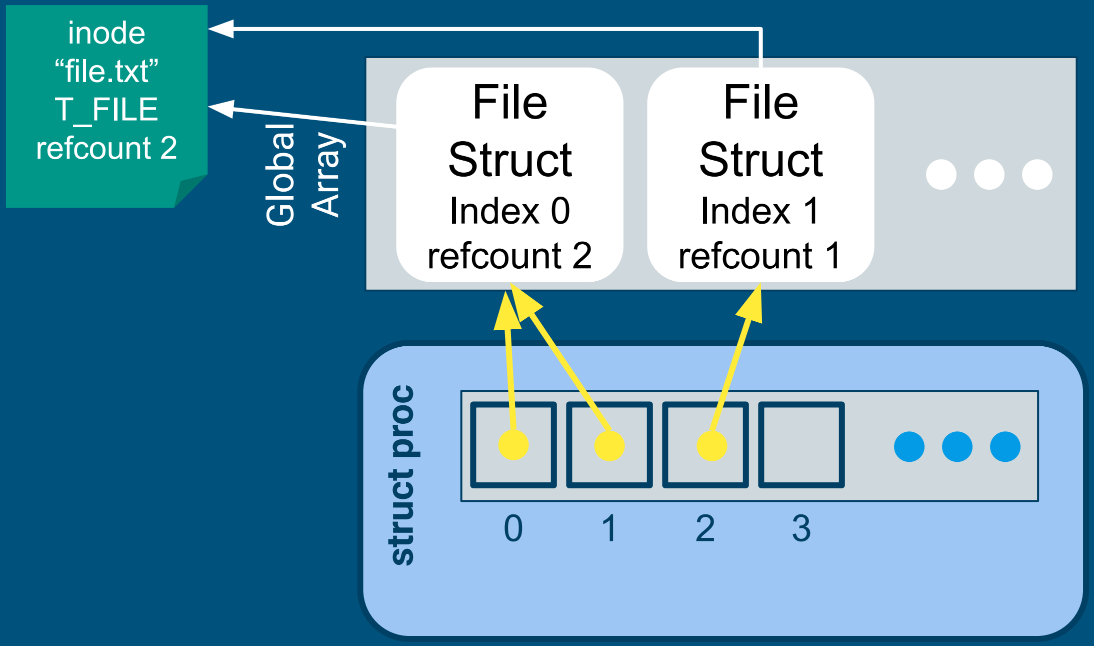
= Available

Multiple Open Calls on Same File



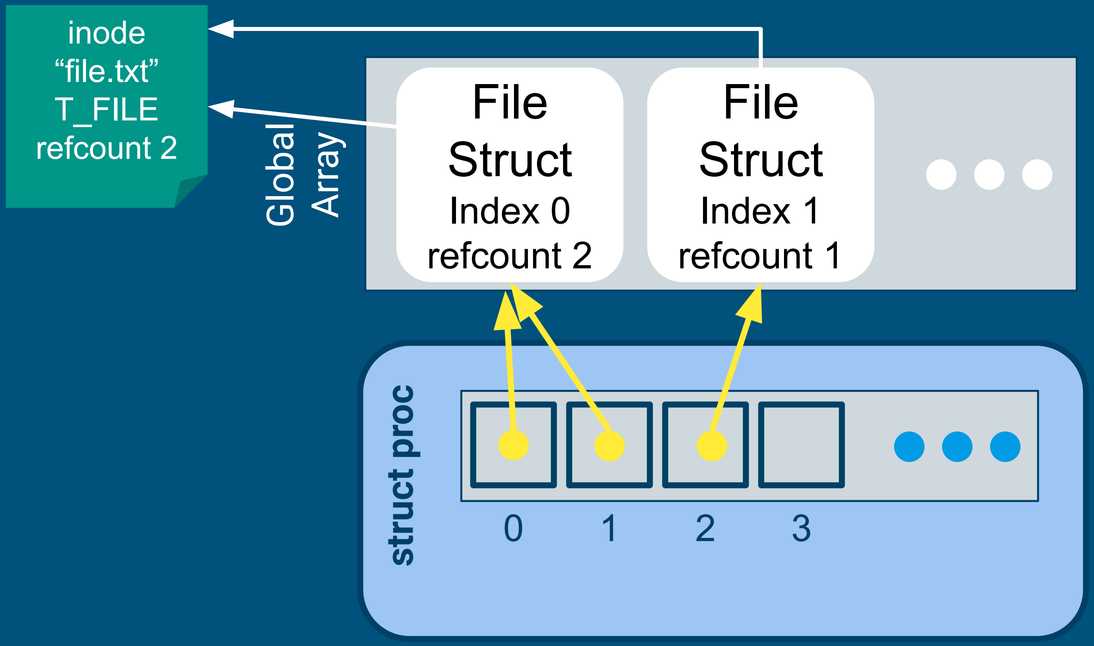
fd = open(“file.txt”, O_RDONLY)
dup(fd)
fd3 = open(“file.txt”, O_RDWR)
// what happens?

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

System Calls

In lab 1, we need to add support for the following syscalls:

- 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

Argument Parsing & Validation

First we need to parse and validate the arguments passed in via syscall

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

What does this look like in xk?

```
// Fetch the nth word-sized system call argument as a pointer
// to a block of memory of size bytes.  Check that the pointer
// lies within the process address space.
int argptr(int n, char **pp, int size) {
  int64_t i;
  struct vregion *r;
  struct vspace *v;

  if (argint64(n, &i) < 0)
    return -1;
  if (size < 0)
    return -1;

  v = &myproc()->vspace;
  for (r = v->regions; r < &v->regions[NREGIONS]; r++) {
    if (vregioncontains(r, i, size)) {
      *pp = (char*)i;
      return 0;
    }
  }
  return -1;
}
```

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

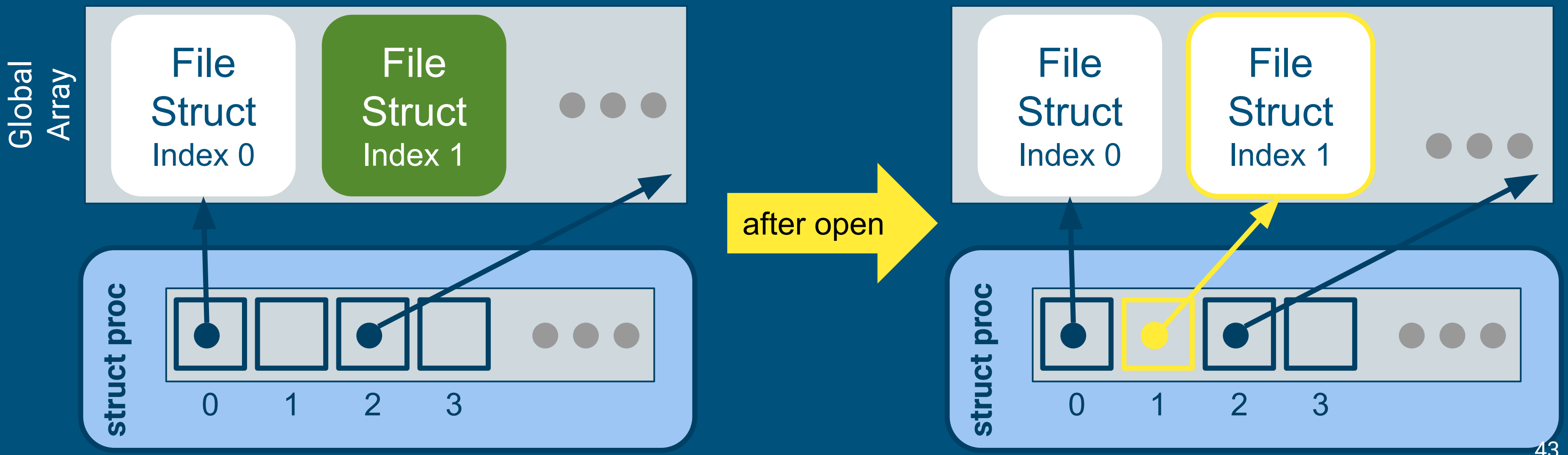
fileopen

= In use

= Available

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

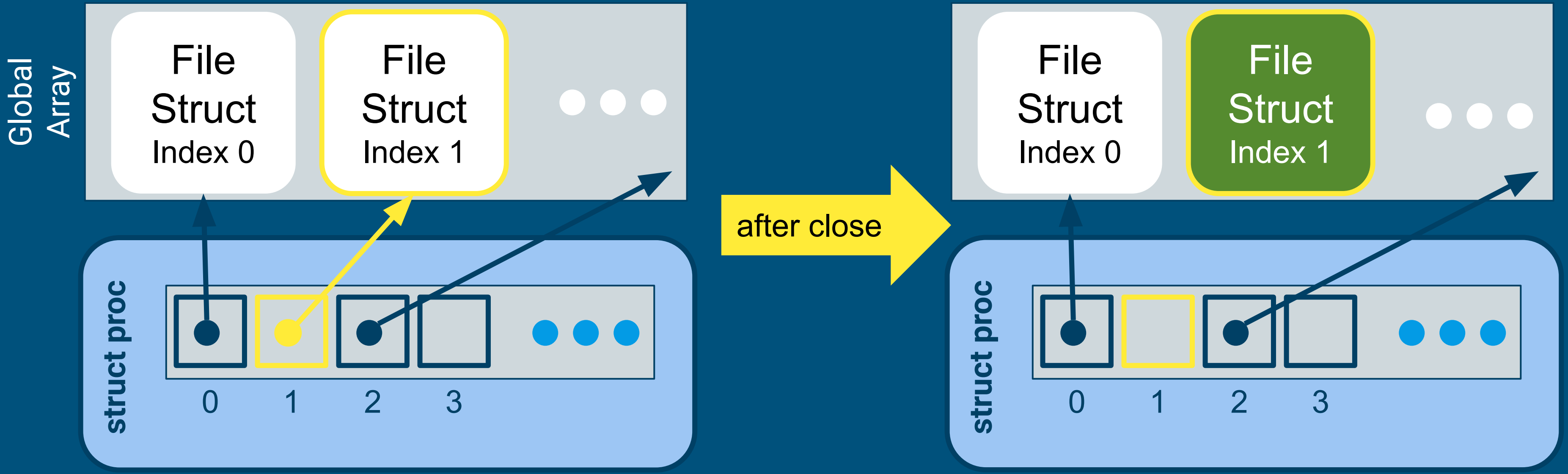
Hint: to obtain the inode for the desired file, take a look at iopen()



fileclose

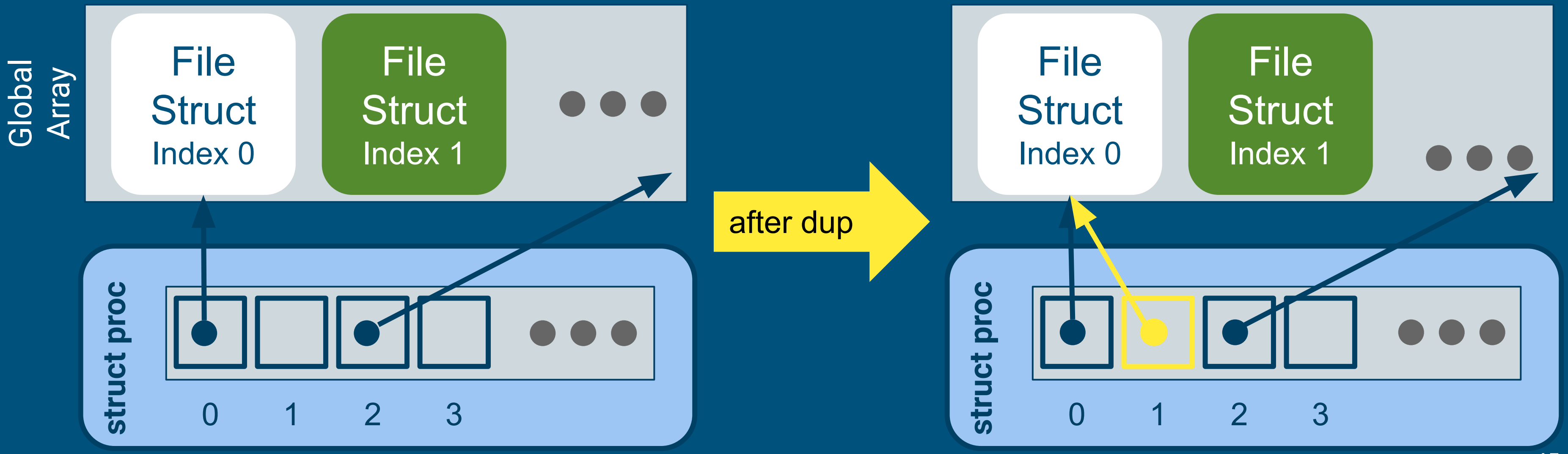
Release the file from this process, will have to clean up if this is the last reference

- 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_stati* 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?

Lecture Questions

Question topics?

- Mode transfer mechanism
- Process abstraction
- Program becoming a process
- Time sharing in a CPU (scheduling)
- Process life cycle
- Fork
- Exec
- Copy on Write Fork
- Signals
- Pipes

Memory

Relevant for Lab 2

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 virtual addresses to physical addresses

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

- AKA “interrupt stack”
- Each process has its own kernel stack
- In the kernel section of memory
- In xk, the kernel allocates one page which acts as the kernel stack during process creation
- From kernel.proc.c:allocproc:

```c
// 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

Accessing Global Variables Across Files

“Great so I can create an initialize them, but how do I access file1.c:foo from file2.c?”
~ You, a student with excellent questions

Accessing Global Variables Across Files: A Brief Aside

Common to want to access variable foo defined in file2.c in file1.c.

Google is your friend for C questions: StackOverflow Answer
To recap:

```
/// file1.c
// Note that this line also could just be in
// a header file which is included instead.
extern int foo;

// Now `foo` can be freely used in this file.
```

```
/// file2.c
// This is it. Defines the variable foo
// in the current translation unit.
int foo;

// Now `foo` can be freely used in this
file.
```

Reference Counting: Why do you care?

- Consider a process which opens then closes a file one million times in a loop.
- Do you expect the file table to be exhausted? ⇒ No! Of course not
- **This implies that file table slots must be reclaimed** (i.e.: deallocated).
- On the other hand, slots must NOT be reclaimed before they're done being used.

The solution? Reference counting.

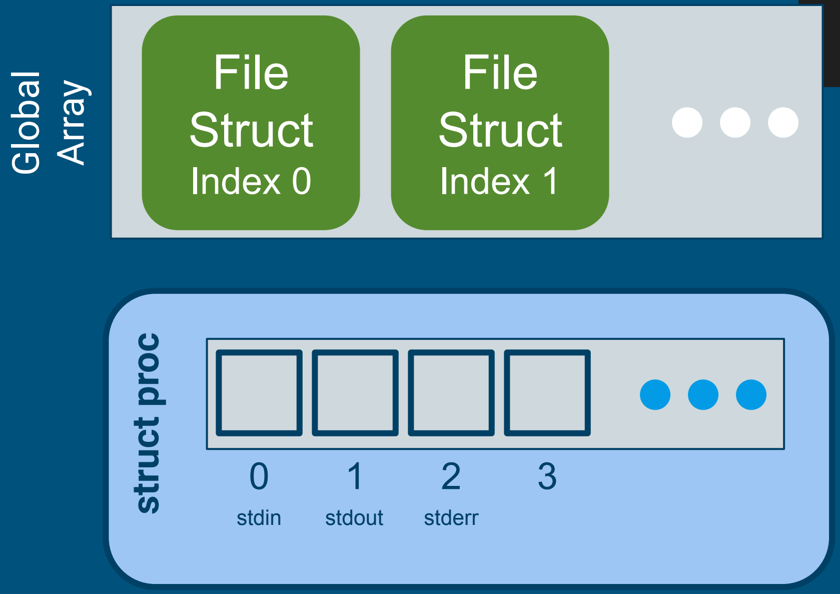
Lab 1 Test Program Code Fragment

```
int main() {
  // printf(stdout, "hello world\n");
  // while (1);
  if (open("console", O_RDWR) < 0) {
    error("fail");
  }
  dup(0); // stdout
  dup(0); // stderr
```

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

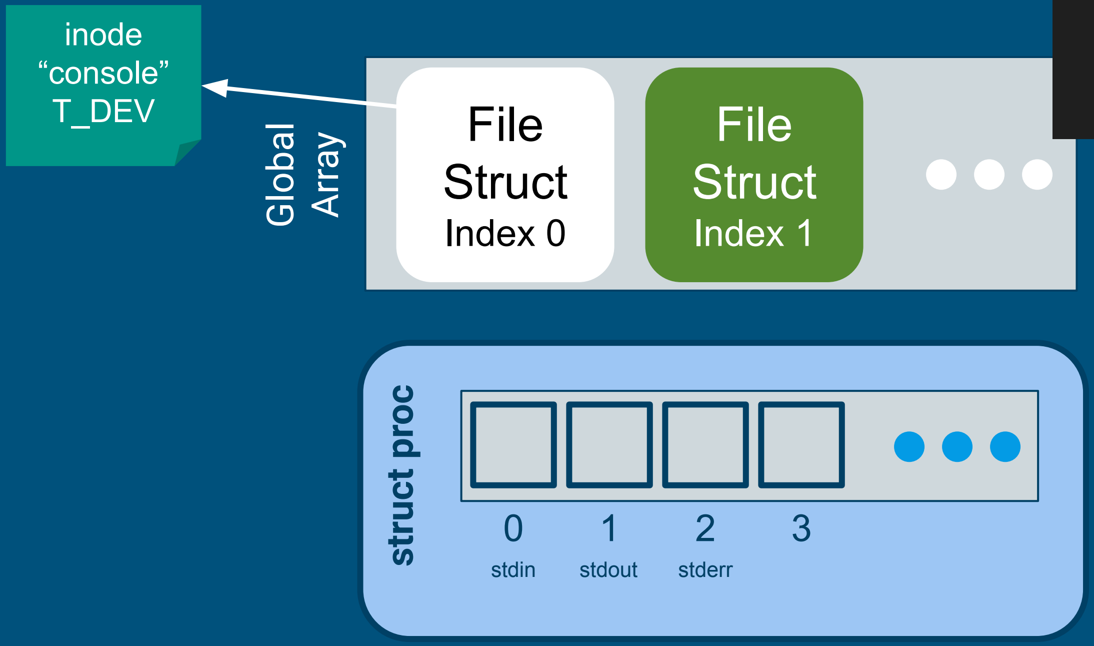
File Table View

```
int main() {
  // printf(stdout, "hello world\n");
  // while (1);
  if (open("console", O_RDWR) < 0) {
    error("fail");
  }
  dup(0); // stdout
  dup(0); // stderr
```



File Table View

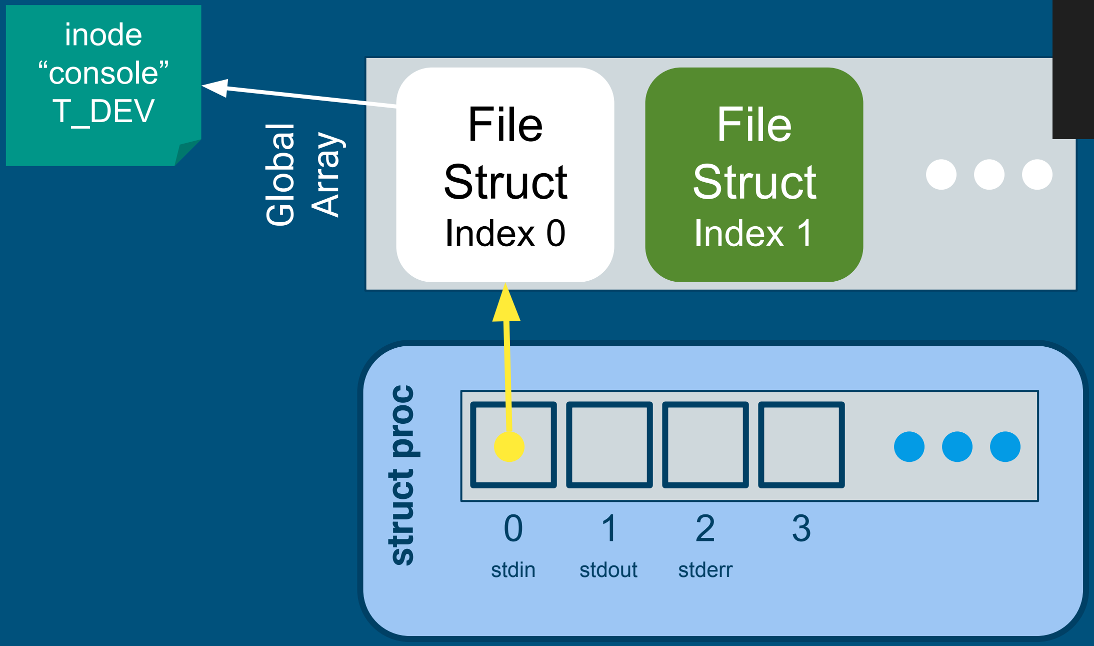
```c
int main() {
  // printf(stdout, “hello world\n”);
  // while (1);
  if (open(“console”, O_RDWR) < 0) {
    error(“fail”);
  }
  dup(0); // stdout
  dup(0); // stderr
```



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

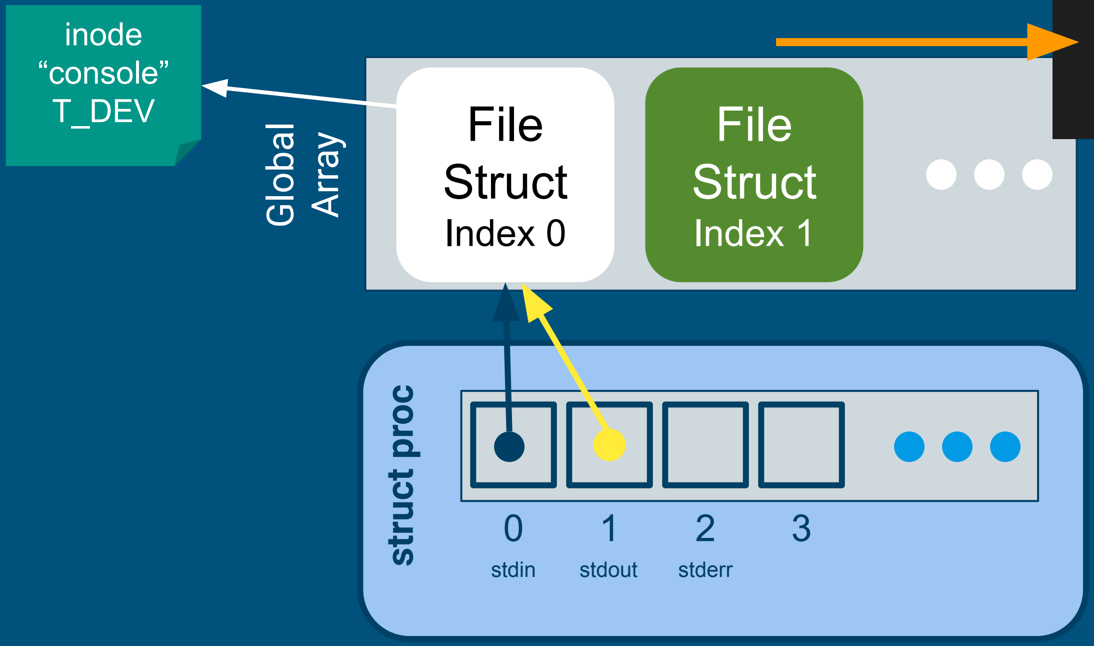
File Table View

```
int main() {
  // printf(stdout, "hello world\n");
  // while (1);
  if (open("console", O_RDWR) < 0) {
    error("fail");
  }
  dup(0); // stdout
  dup(0); // stderr
```



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

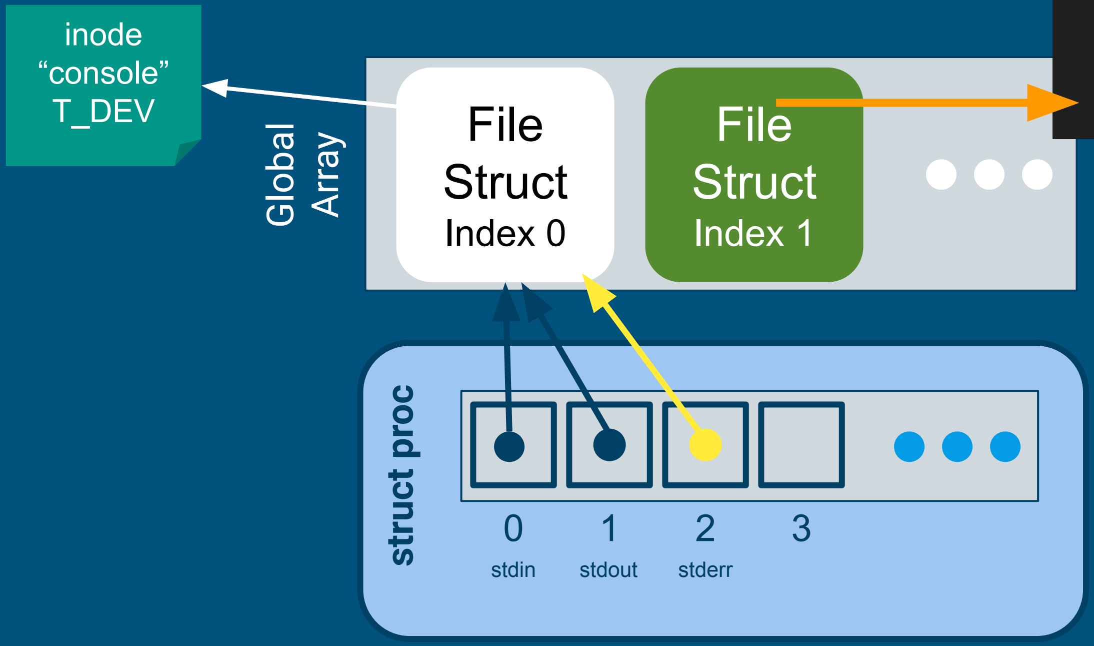
File Table View




```
int main() {
  // printf(stdout, “hello world\n”);
  // while (1);
  if (open(“console”, O_RDWR) < 0) {
    error(“fail”);
  }
  dup(0); // stdout
  dup(0); // stderr
```

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

File Table View




```
int main() {
  // printf(stdout, “hello world\n”);
  // while (1);
  if (open(“console”, O_RDWR) < 0) {
    error(“fail”);
  }
  dup(0); // stdout
  dup(0); // stderr
```