CSE 451 Section 2
OSV Lab 1 Design

20wi

Please pick up section handout as you come in :)

File Information

```
struct file {
    int f_ref; // ref count for this file
    int oflag; // open flag
    struct inode *f_inode; // File inode
    offset_t f_pos; // Current file offset
    struct sleeplock f_lock; // Lock protecting fil
    struct file_operations *f_ops; // File operatio
};
```

```
struct inode {
    inum_t i_inum; // Inode number
    struct super_block *sb; // Superblock
    unsigned int i_ref; // Reference counter. Note tha
    unsigned int i_nlink; // Number of links
    ftype_t i_ftype; // File type
    fmode_t i_mode; // File permission
    size_t i_size; // File length in bytes
    void *i_fs_info; // Filesystem specific inode info
    state_t i_state; // State of in-memory inode
    struct sleeplock i_lock; // Lock protecting inode
    struct inode_operations *i_ops; // Inode operation
    struct file_operations *i_fops; // File operations
    struct memstore *store; // memstore to read pages
    Node node; // List of dirty inodes or inodes with
};
```

- struct inode: information for the file on disk (e.g. type, location)
- struct file: current state of the open file (e.g. mode, offset)

Additional Information

- Inode struct to file on disk: 1 to 1
  - Kernel uses a radix tree to keep track of inodes opened
  - On fs_open_file(), it checks the radix tree to fetch the inode and stores a pointer to it in the file struct
- File struct to file on disk: many to 1
  - Read from different offsets at the same time

Reference counting review

- Can we free the file structure when we close the file?

Reference counting review

- Can we free the file structure when we close the file?
  - It depends: other file descriptors might be still using it!
  - We need to keep track of how many references points to the file

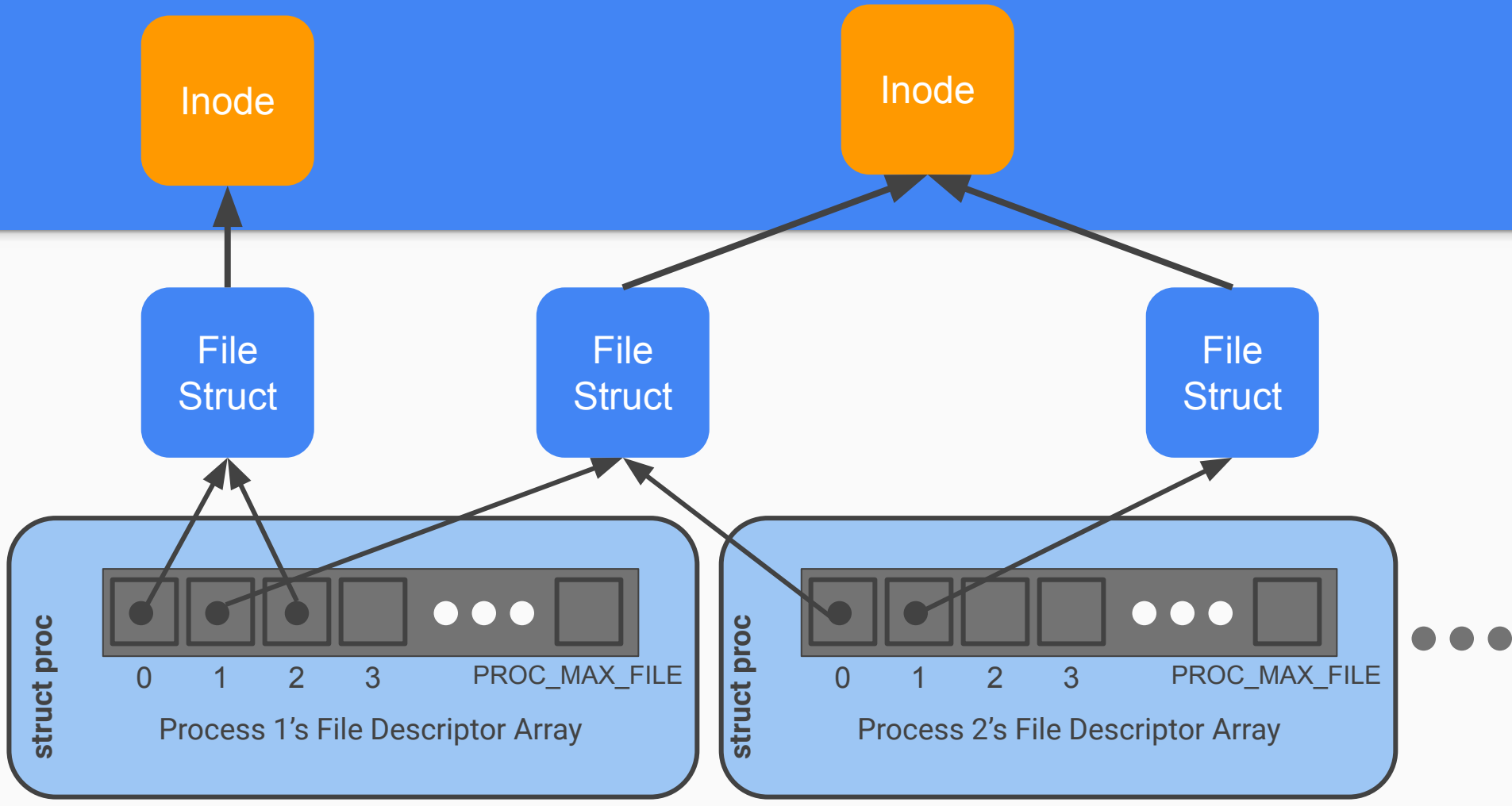
Reference counting review

```
struct file {
    int f_ref; // ref count for this file
```

- fs_open: set reference count to 1
- fs_reopen_file: increase reference count
- Fs_close_file: decrease reference count and free the memory only if there’s no reference to it

Why we use file descriptors instead of file path? Why do we want processes to have their own sets of file descriptors?

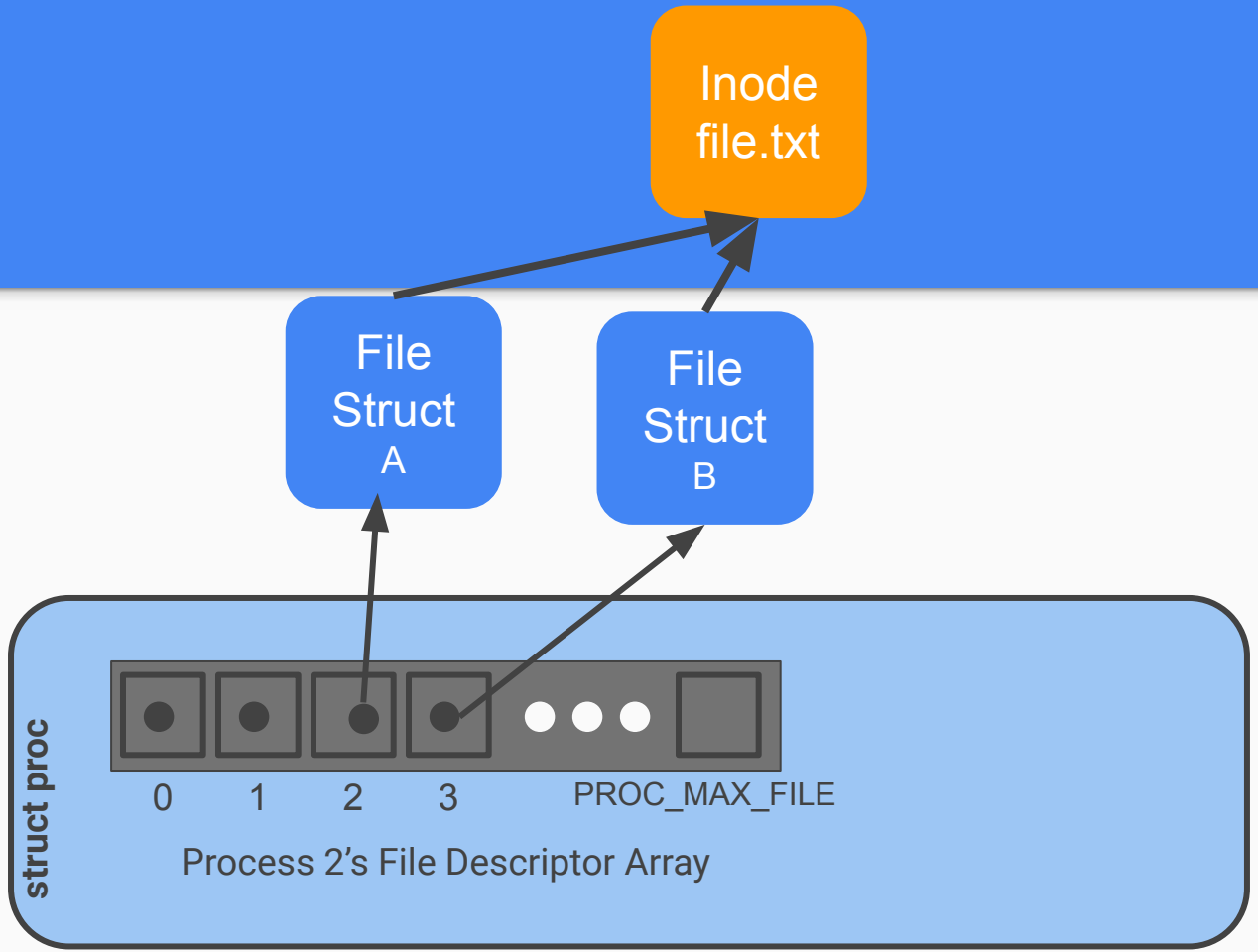
Section handout: question 1


Inode
Inode
File Struct
File Struct
File Struct
struct proc
0
1
2
3
PROC_MAX_FILE
Process 1's File Descriptor Array
struct proc
0
1
2
3
PROC_MAX_FILE
Process 2's File Descriptor Array

Draw out the memory layout after the following c code:

```
int fd1 = open("file.txt", O_RDONLY);
int fd2 = open("file.txt", O_RDWR);
```

Section handout: question 2


Inode
file.txt
File
Struct
A
File
Struct
B
struct proc
0
1
2
3
PROC_MAX_FILE
Process 2's File Descriptor Array

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!)
  - Verify permission
  - Call associated file functions to handle the request

Argument Parsing & Validation

Currently process to thread is 1:1, don't need to copy syscall arguments

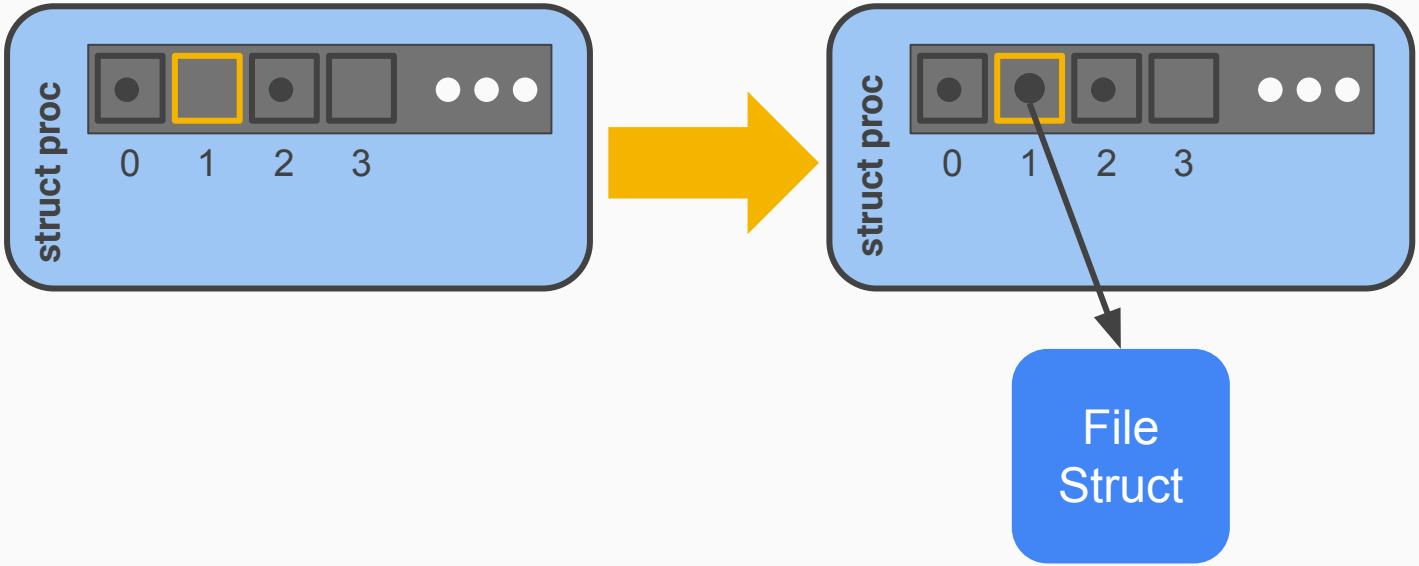
- **bool fetch_arg(void *arg, int n, sysarg_t *ret)** get nth argument
- **bool validate_str(char *s)**: validate string
- **bool validate_bufptr(void* buf, size_t size)**: validate buffer

It’s a good practice to implement and use helper functions:

- **int alloc_fd()**: allocate a file descriptor
- **bool validate_fd(int fd)**: validate if a fd is valid

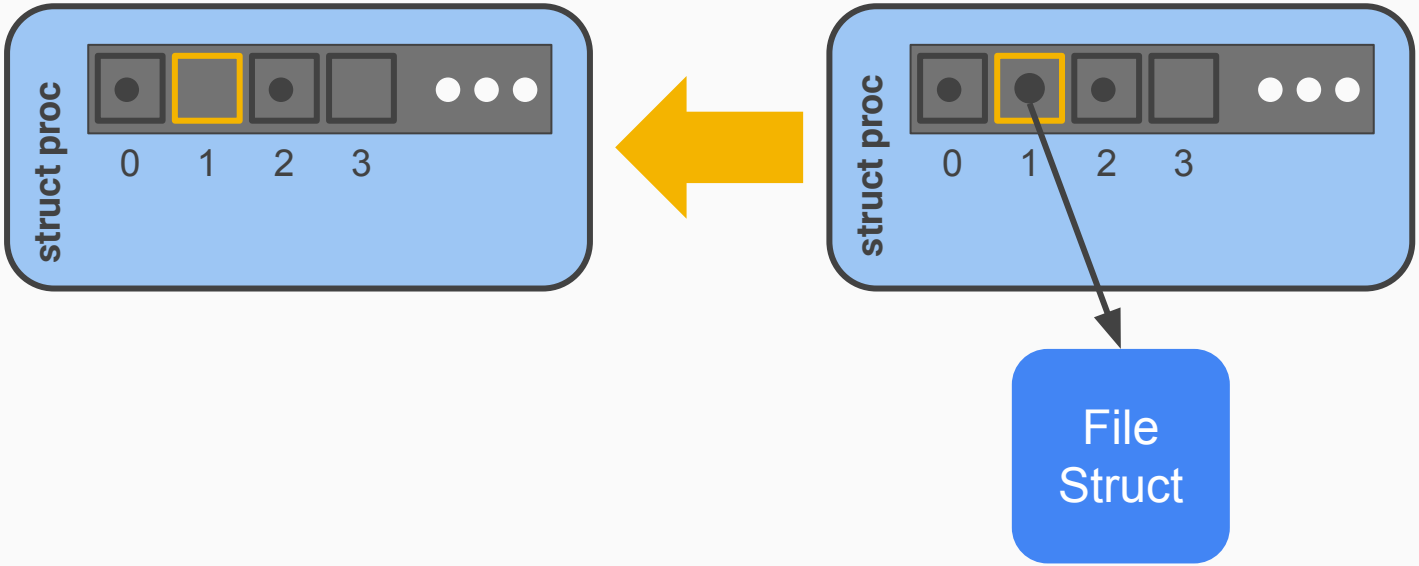
sys_open

Open file and find an open spot in the file descriptor table



sys_close

Clear a spot in the file descriptor table



sys_read and sys_write

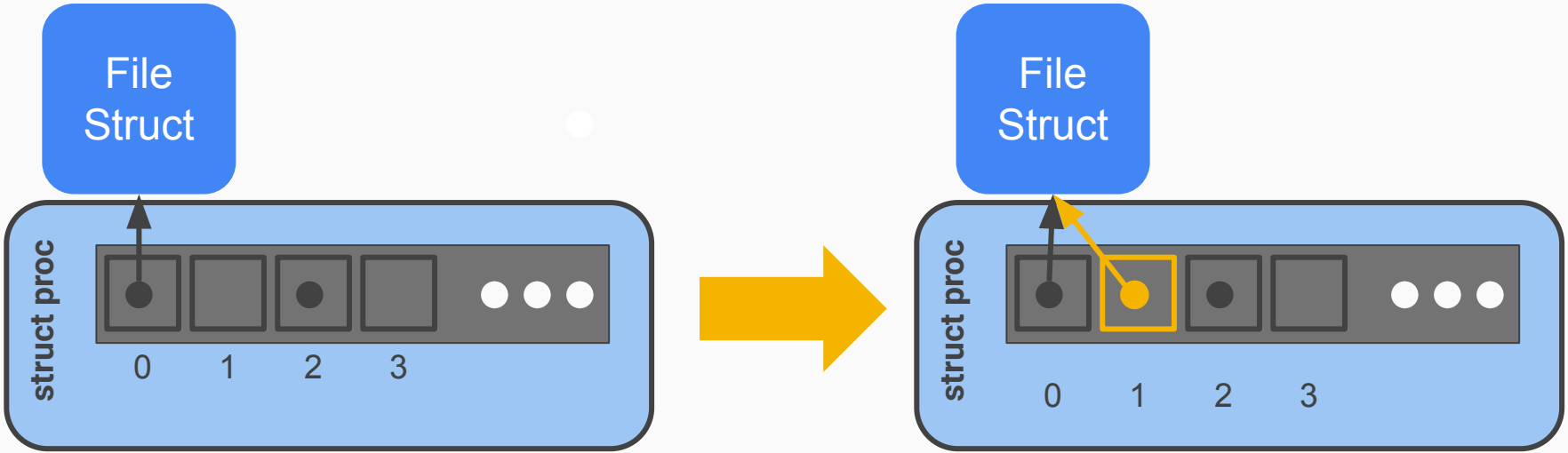
- Writing or reading of a "file", based on whether the file is an inode or a pipe.
  - Note that file is in quotes. A file descriptor can represent many different things. You could be reading from a file, or you could be reading from console or a pipe!
- Don’t need to worry about the pipe part for this lab, just the inode files.
  - We will learn pipes in lab 2

sys_stat

- Return useful statistics information from inode
- Can't stat on files that are not on disk
  - if file doesn't have an inode, it is not a real file and we don’t have statistics for it
  - Example: pipes in lab 2

sys_dup

Duplicates the file descriptor in the process' file descriptor table



Exercise: dup2

Section handout: question 3

Console Input/Output

- Console input and output are declared as special files
  - Look at kernel/console.c to see how it's done
- How are they related to stdin/stdout?
  - Automatically initialized in proc_init as file descriptor 0 and 1
  - When the child process is forked from parent, the file descriptors are copied

Where is X?

From the top level of the repo, run:

grep -R “X” .

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.

For vscode: press Ctrl+T to search for declaration/definition

Multiprocessing :)

- Take a look at proc_spawn in proc.c, what does it do?
- If a process is forked, what steps in proc_spawn should remain the same, and what should change?

Where will a process resume execution after coming back to user mode? Where is this information stored?

How to set return value on syscall returns?

To differentiate the new processes from the old process, we call the new process a child of the parent old process. The return value of `fork` is different between the child and the parent. The parent will return the child process id and the child will return 0.

How can we alter the return value of the fork function to simulate the situation above?

Lab2 additional info: List in osv

- List declaration and initialization

```
struct addrspace {
    List regions;
```

```
err_t
as_init(struct addrspace* as)
{
    kassert(as);
    spinlock_init(&as->as_lock, False);
    list_init(&as->regions);
```

List in osv

- Before you can add an element to a list, you first have to allocate a node inside the element.

```
struct memregion {
    struct addrspace *as;
    Node as_node;           // used to connect all memregions within an addrspace
```

List in osv

- Adding to the list

```
// Link into address space's region list
list_append_ordered(&as->regions, &r->as_node, memregion_comparator, NULL);
```

```
void
proc_attach_thread(struct proc *p, struct thread *t)
{
    kassert(t);
    if (p) {
        list_append(&p->threads, &t->thread_node);
    }
}
```

List in osv

- Retrieving from the list

```
struct memregion *region = (struct memregion*) list_entry(n, struct memregion, as_node);
```

Provide address of the node added, type of struct, and the name of the node to retrieve the element

List in osv

- Iterate through the list