CSE 451 Section 2
XK Lab 1 Design

Wi18

Where to start?

Start by reading:

- **lab/overview.md** - A description of the xk codebase. A MUST-READ!
- **lab/lab1.md** - Assignment write-up
- **lab/memory.md** - An overview of memory management in xk
- **lab1design.md** - A design doc for the lab 1 code
    - 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:

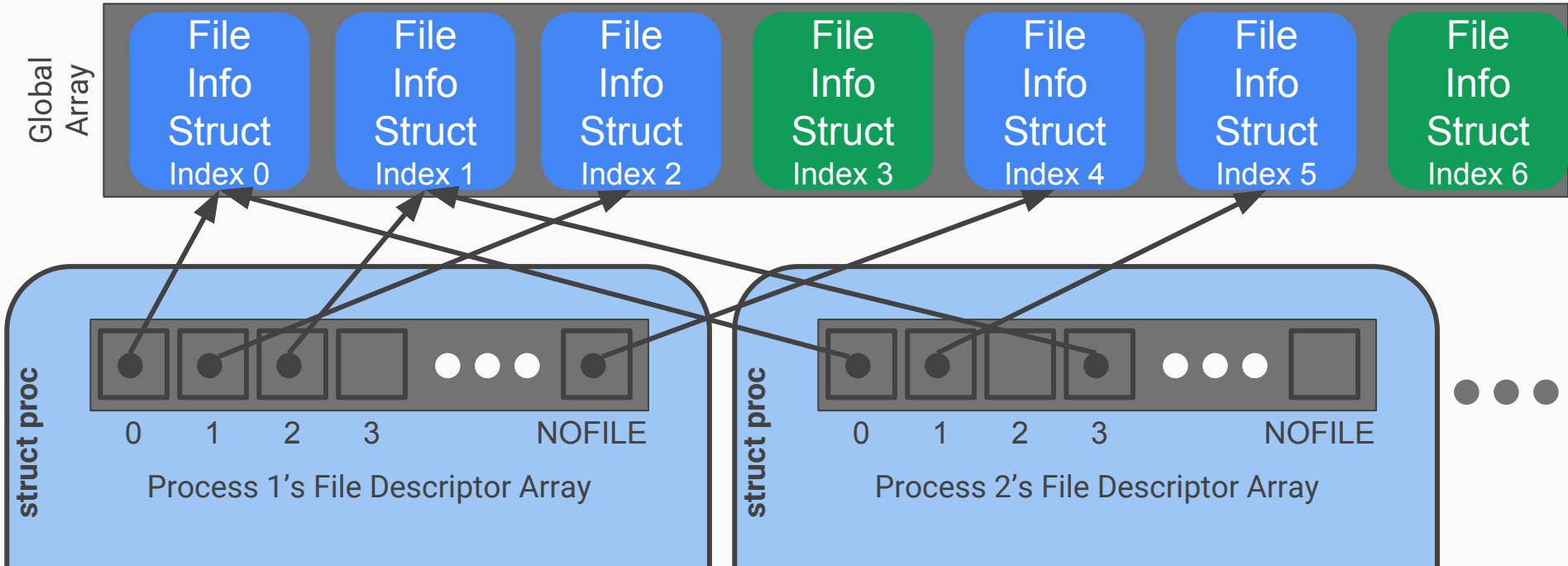
- In memory reference count
- Whether the "file" is an on disk inode, or a pipe (later assignment)
- A reference to the inode of the file
- Current offset
- Access permissions (readable or writable) [for when we add pipes and file writeability later]

File Info Struct

Kernel View

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

Process View



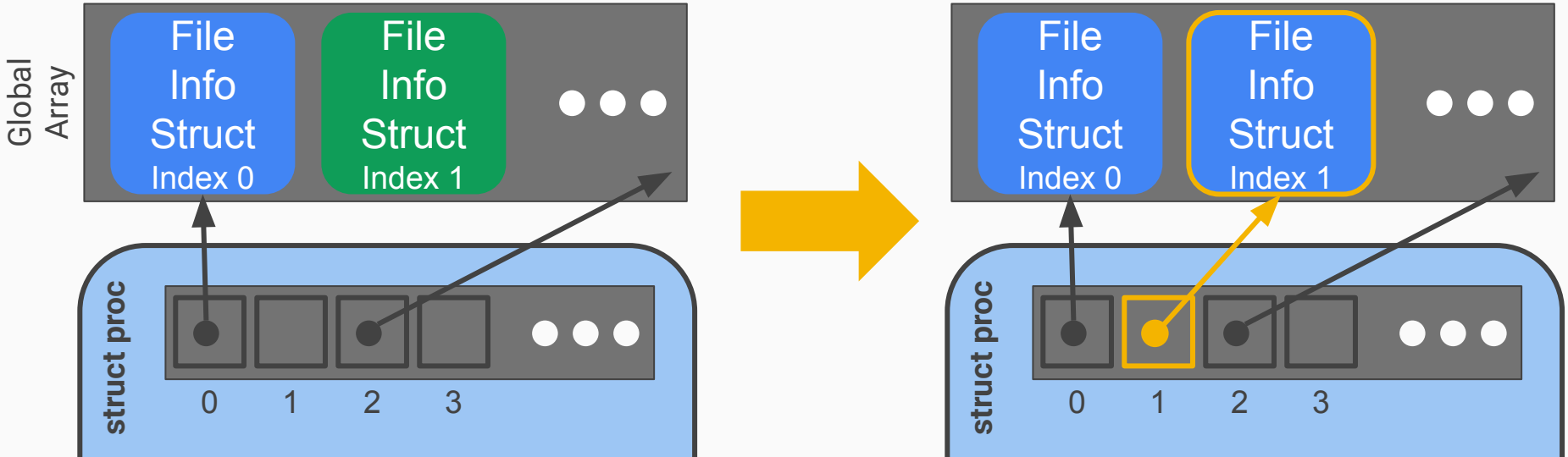
Functions

filewrite and *fileread*

- 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 standard in or a pipe!
- Don’t need to worry about the pipe part for this lab, just the inode files.
- Check out the functions *readi* and *writei* defined in kernel/fs.c

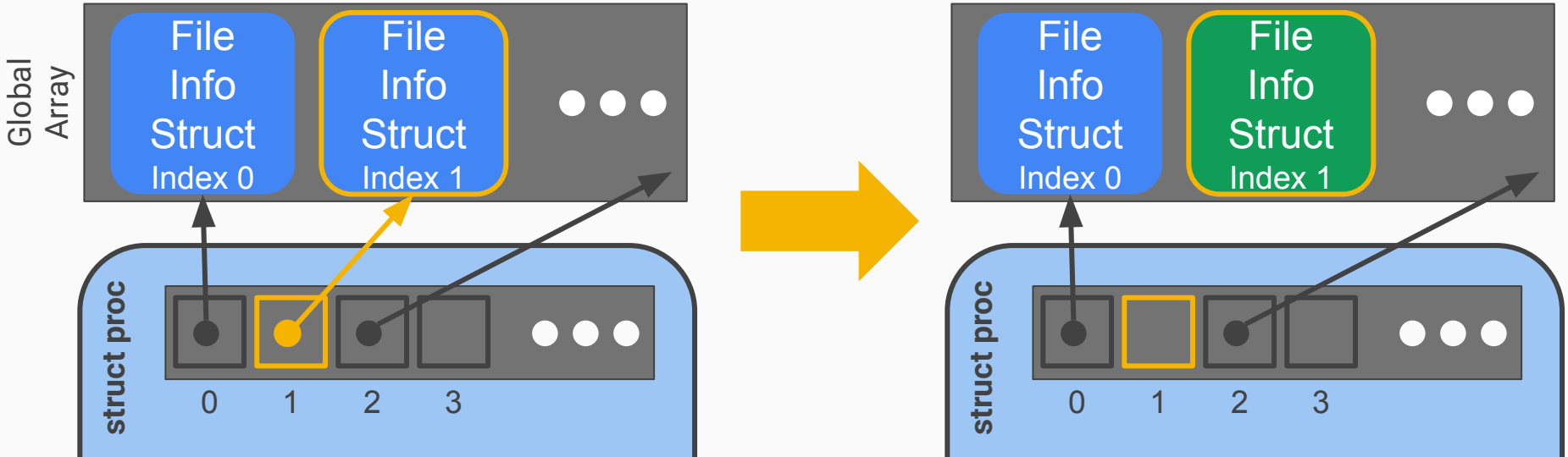
fileopen

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



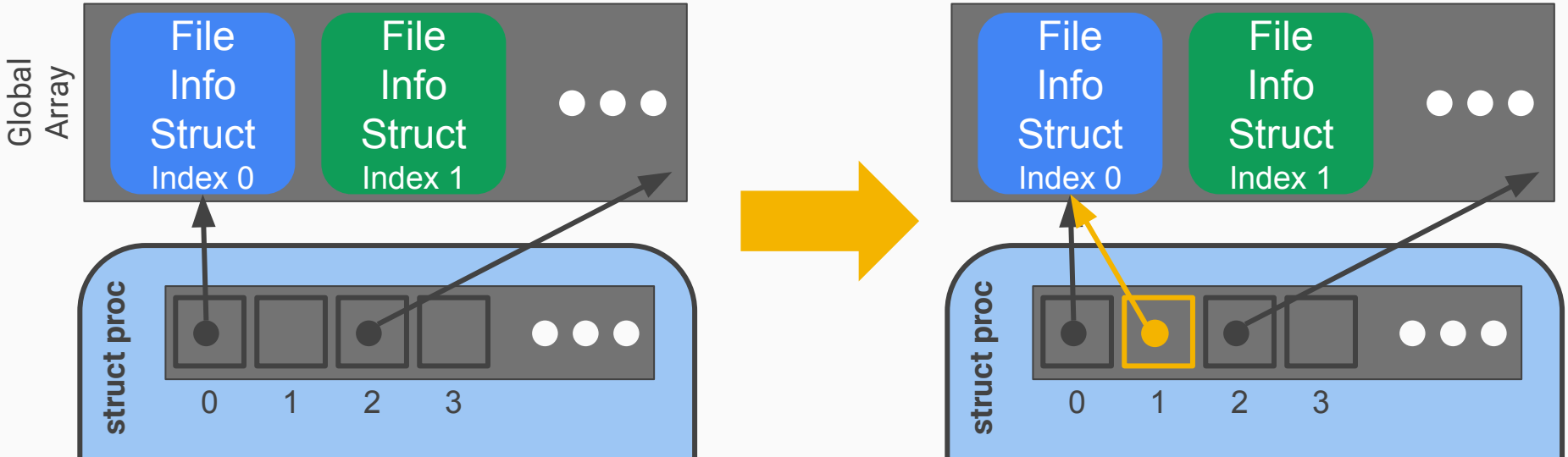
fileclose

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



filestat

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

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).

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.

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.

Staging of work

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