Get to know xk
And Lab 1

new slides; please free to give feedback to improve these slides - 22au

What is xk?

- xk stands for “e**x**perimental **k**ernel”
- Configured to run on qemu (hw emulator)
- A simpler version of the early linux kernel
- 64 bit port of xv6

Which file is in which directory?

- inc
  - contains all the headers (.h) files
  - Most of the structs are/will be defined in the header files

- kernel
  - Kernel source code for all the different components.
  - Big chunk of the lab is based on this folder

Which file is in which directory? - CONTD

- user
  - All the “user” files, i.e everything that is not part of the kernel
  - Lab tests, shell, source code for binaries like ls, wc, ln etc.

- Lab
  - Lab related docs, specs and design docs

Different components of the xk kernel (*roughly*)

- Syscalls
- File System
  - file.c deals with open files management and managing the file info struct (lab1)
  - fs.c deals with writing and reading blocks from disk and other helper functions (lab4)
- Processes
  - fork/exec/wait implementation
  - proc.c and exec.c (lab 2)
- Memory management
  - writing the page fault handler (for stack, heap, and else) , trap.c (lab3)

Lab 1

File syscalls

Where to start?

https://gitlab.cs.washington.edu/xk-public/22au/blob/main/lab/lab1.md

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 (which will be a bit more comprehensive than the one provided for lab 1). Check out lab/designdoc.md for details.

Summary of Lab 1

- File info
    - struct storing info for each open file
- File descriptor
    - per-process file identifier (one for each open file) to use in syscalls
- File syscalls
    - Uses both file descriptor and file info to implement file related system calls

File API (UNIX, xk)

file-descriptor = open(filename)

Returns a per-process handle to be used in subsequent calls (implemented as a C int)

Shell pre-assigns stdin, stdout as file descriptors (0, 1)

read/write(file-descriptor, buffer, numBytes)

Read or write numBytes into/out of buffer, changes position in file

file-descriptor = dup(file-descriptor)

Make a new file descriptor, copy of the previous one (used in shell)

close(file-descriptor)

We're done with using this file descriptor

More on the UNIX File API

File descriptors are used for all I/O, eg, network sockets, pipes for interprocess communication

Applications use read/write regardless of which thing it is reading/writing to

File descriptors are per-process but can be passed between processes

Important for how fork/exec and the shell works

Examples: ls | wc   ls > tmpfile   wc < tmpfile

Kernel *should not* trust file descriptor (might not be previously opened, etc.)

App should not be able to crash kernel

File Syscalls

You will need to implement a number of file related system calls.

Implementing syscalls consists of two steps:

- parsing and validating syscall arguments
  - see implemented syscalls for reference (sysfile.c)
  - argptr, argstr, argint, what do these functions do?
- perform the requested file operations
  - need to write your own file operations using the provide inode layer

File Descriptors - Kernel View

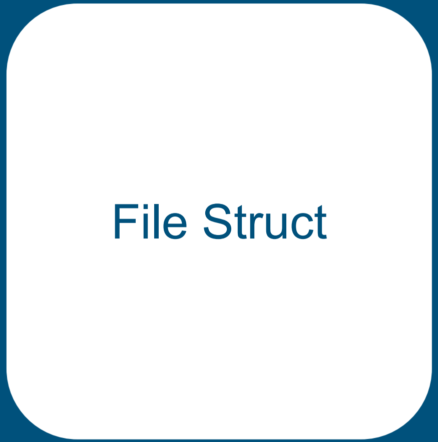
- Kernel needs to give out file descriptors upon open
    - must be give out the smallest available fd
    - fds are unique per process (fd 4 in process A can refer to a different file than fd 4 in process B)
    - need to support NOFILE number of open files for each process
        - each process should 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

File Information

The current xk file system only implements a primitive inode layer, so you need to create a file abstraction yourself. We need to track the following information for each open file:

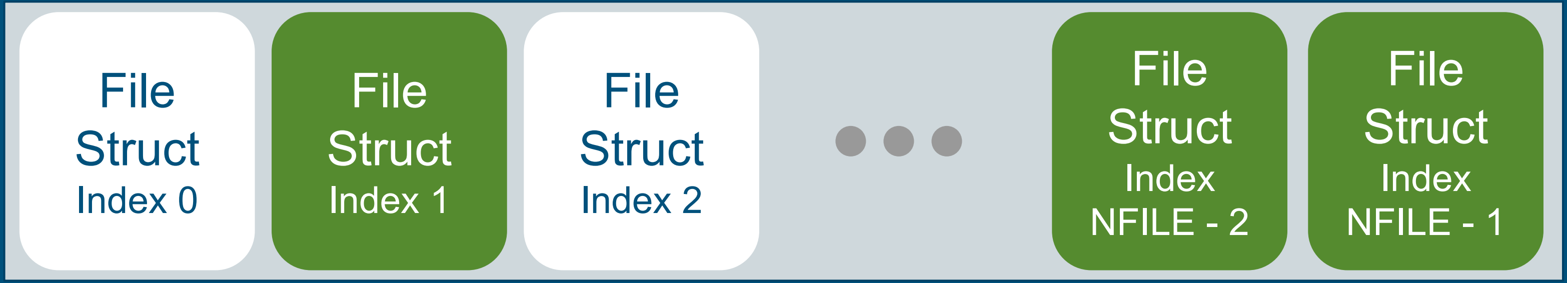
- In memory reference count
- A pointer to the inode of the file
- Current offset
- Access permissions (readable or writable)



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)



⬜ = In use 🟩 = Available

Inode Layer

namei() = opens an inode in memory

readi() / concurrentreadi() = read data using this inode

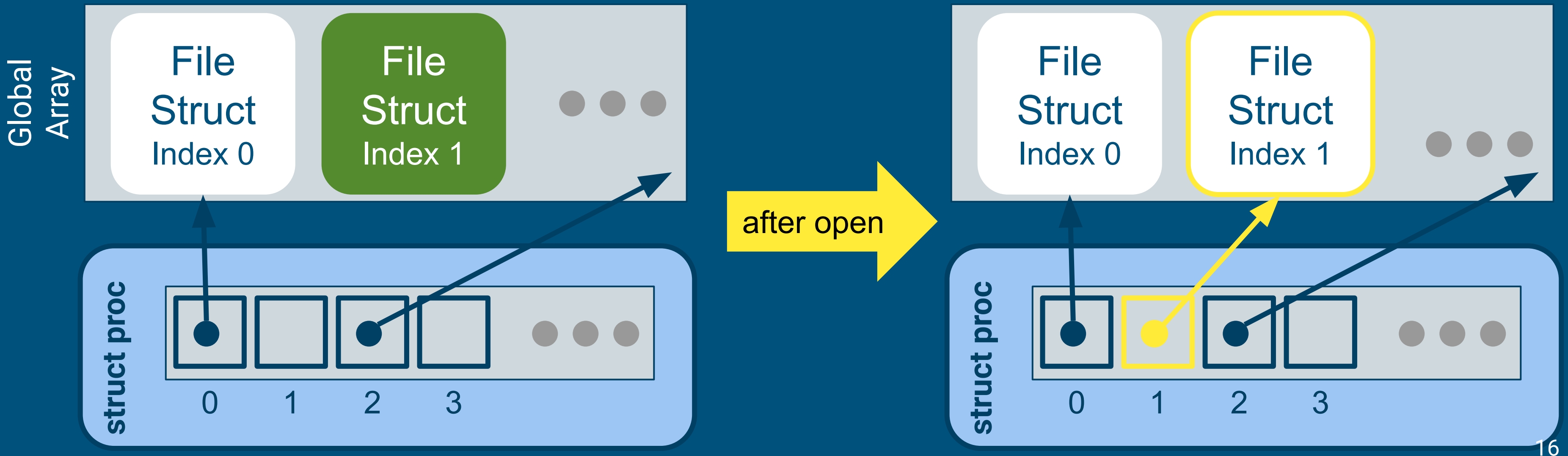
writei() / concurrentwritei() = write data using this inode

File layer provides “policy” for accessing files, inode layer provides “mechanism” for reading/writing

fileopen

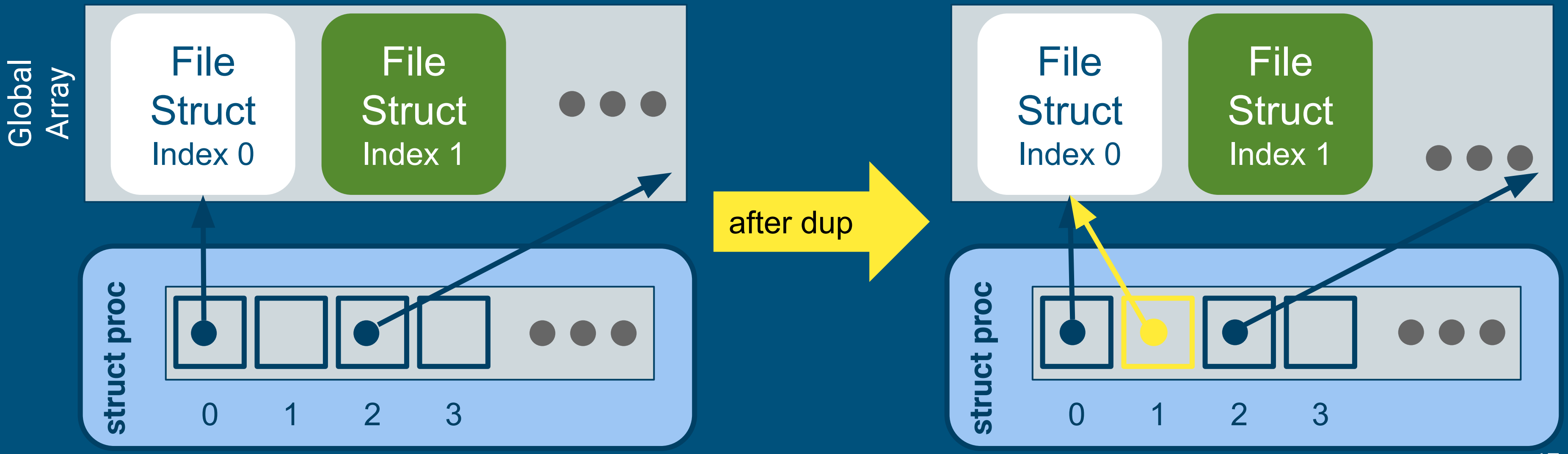
Finds an available file struct in the global file table to give to the process

Hint: take a look at namei()



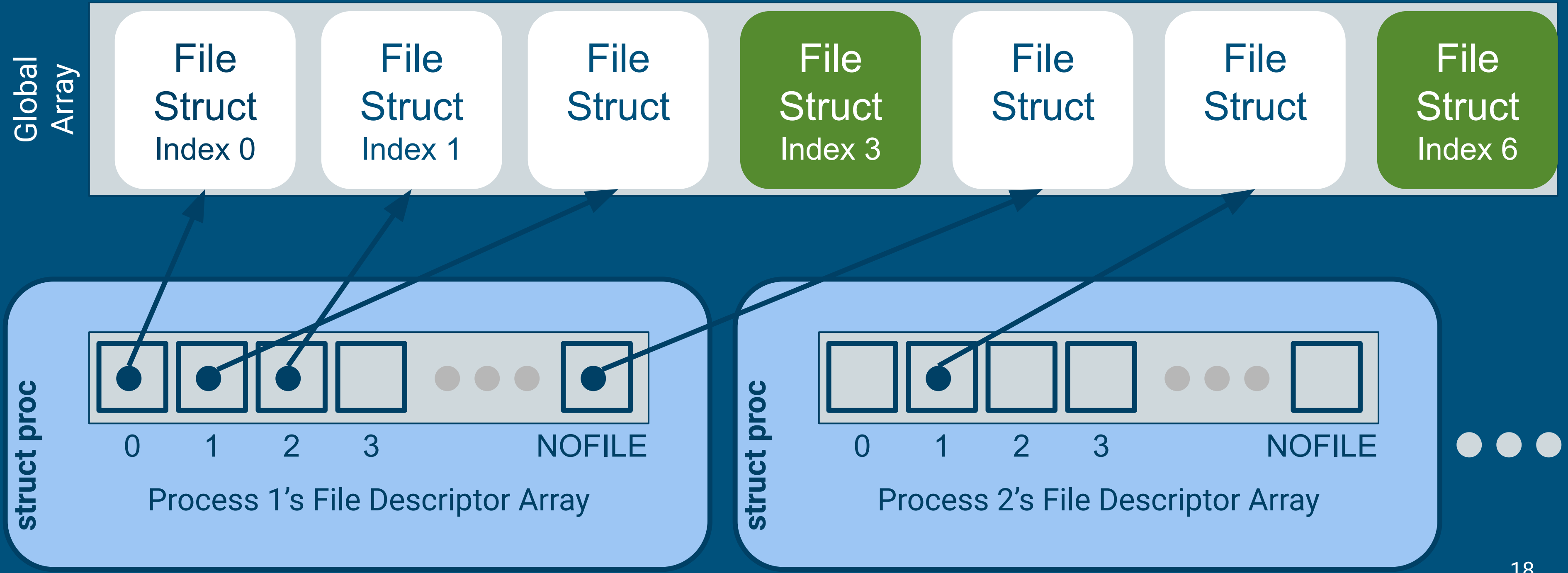
filedup

Duplicates the file descriptor in the process' file descriptor table



Global File Table

fd = ***index*** into local File Descriptor Array



Global Array
File Struct Index 0
File Struct Index 1
File Struct
File Struct Index 3
File Struct
File Struct
File Struct Index 6
struct proc
0
1
2
3
NOFILE
Process 1's File Descriptor Array
struct proc
0
1
2
3
NOFILE
Process 2's File Descriptor Array