Lab 1 Info

File syscalls

Administrivia

Lab 1 due next Wednesday

Lab 2 Design Doc due Monday, 23 January

Agenda

- Review
- Where/how to initialize global variables?
- What is reference count for?
- Syscall/file calls
- Kernel/user memory
- trapframe/kernel stack

Where to start?

https://gitlab.cs.washington.edu/xk-public/23wi/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

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).

File Descriptors - Kernel View

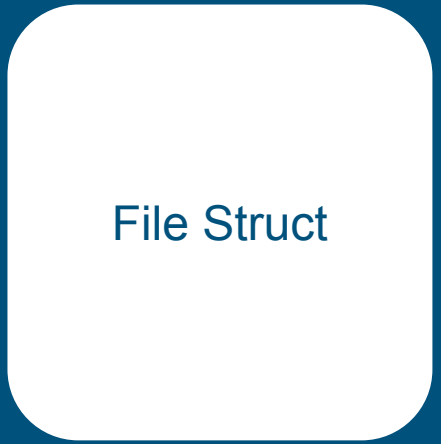
- Kernel needs to give out file descriptors upon open
  - must be give out the smallest available fd
  - fds are unique per process (fd 4 in process A can refer to a different file than fd 4 in process B)
  - need to support NOFILE number of open files for each process
    - each process should know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



File Tables

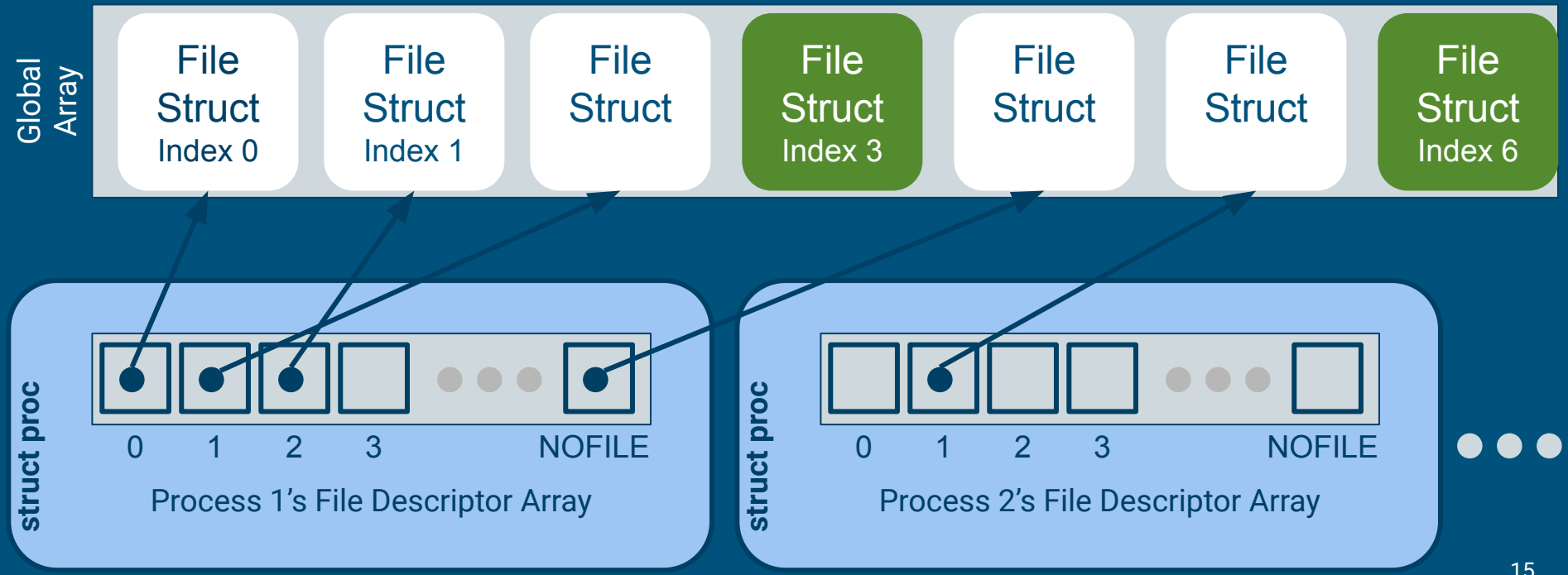
Allocation of File Structs

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)

Global File Table

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



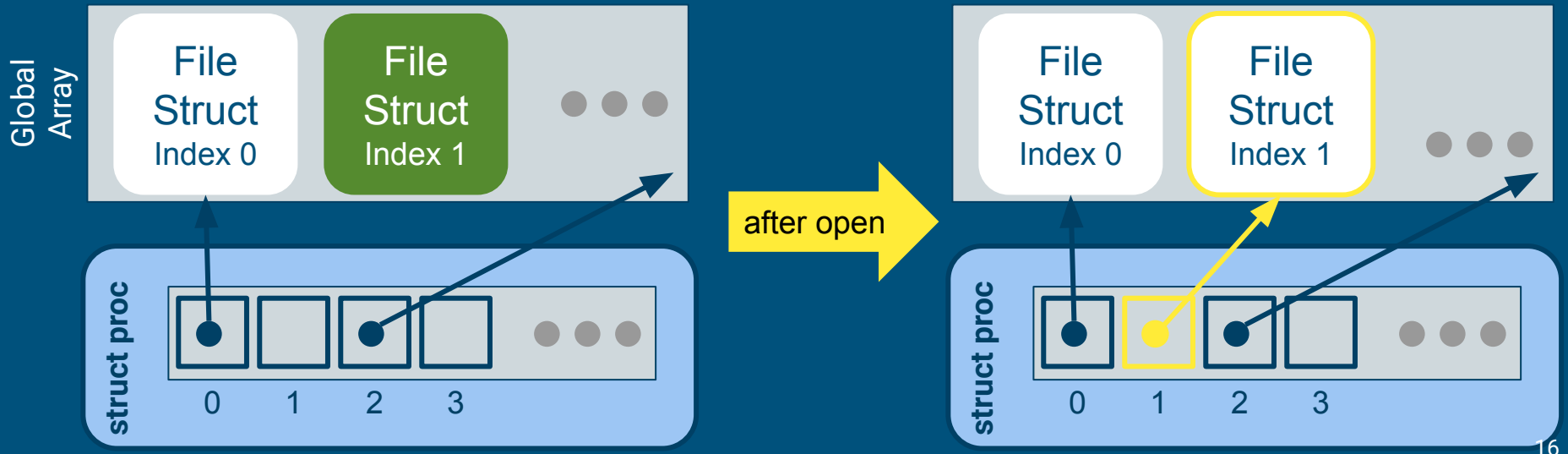
fileopen

= In use

= Available

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

Hint: take a look at namei()

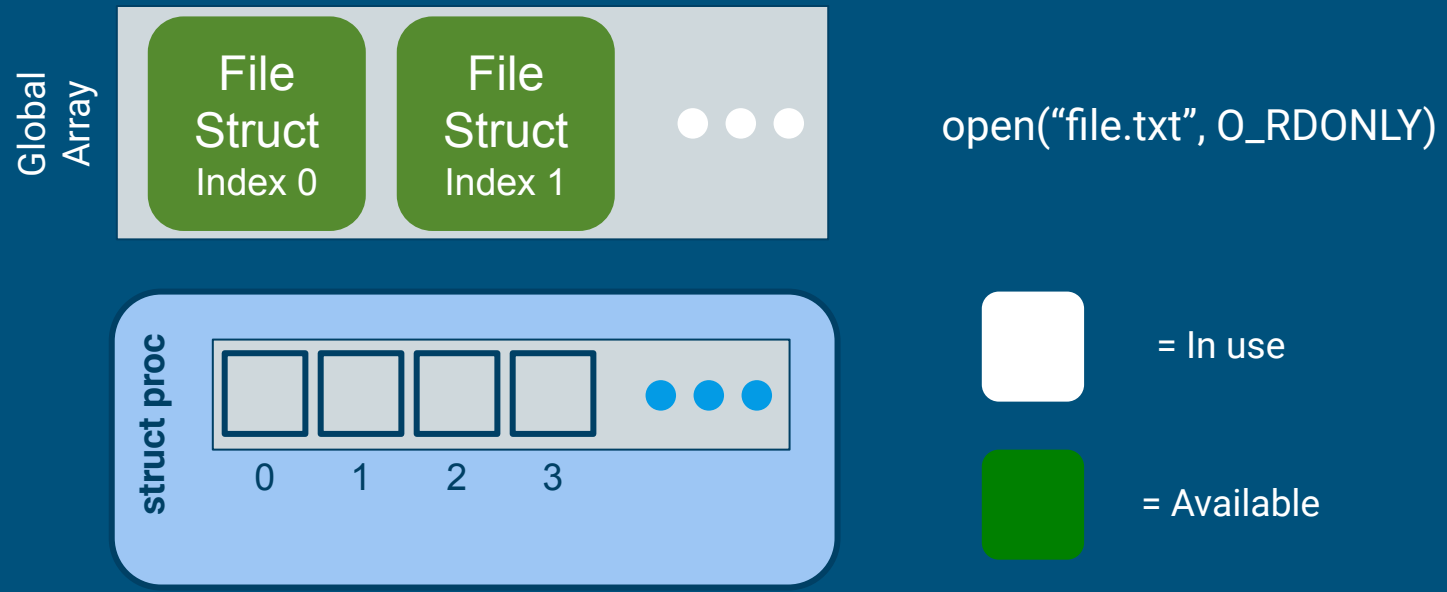


Multiple Open Calls on Same File

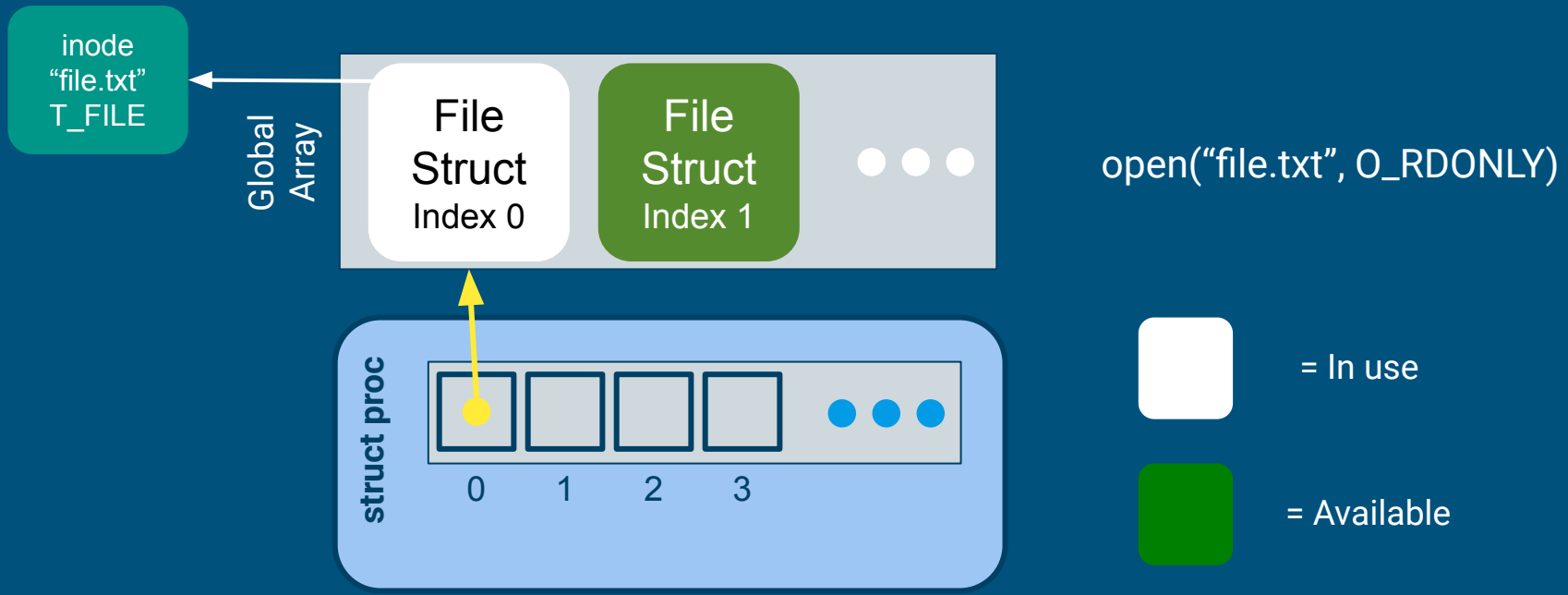
- Draw out the process and global open file table layout after the following:

```
int fd1 = open(“file.txt”, O_RDONLY);
int fd2 = open(“file.txt”, O_RDONLY);
```

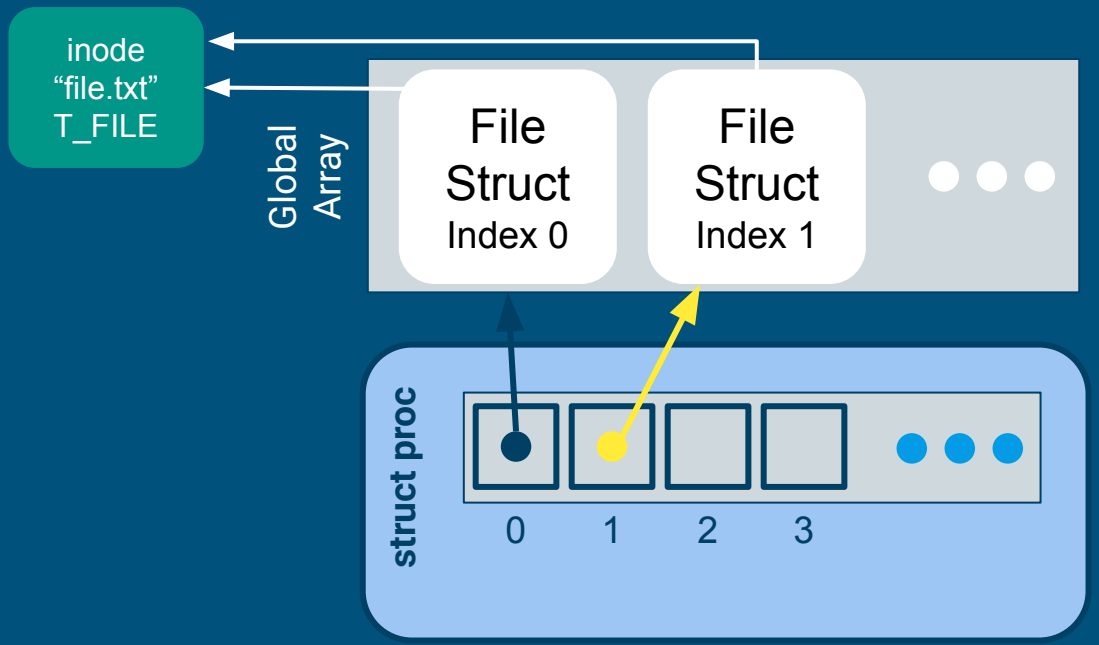
Multiple Open Calls on Same File



Multiple Open Calls on Same File



Multiple Open Calls on Same File



open(“file.txt”, O_RDONLY)

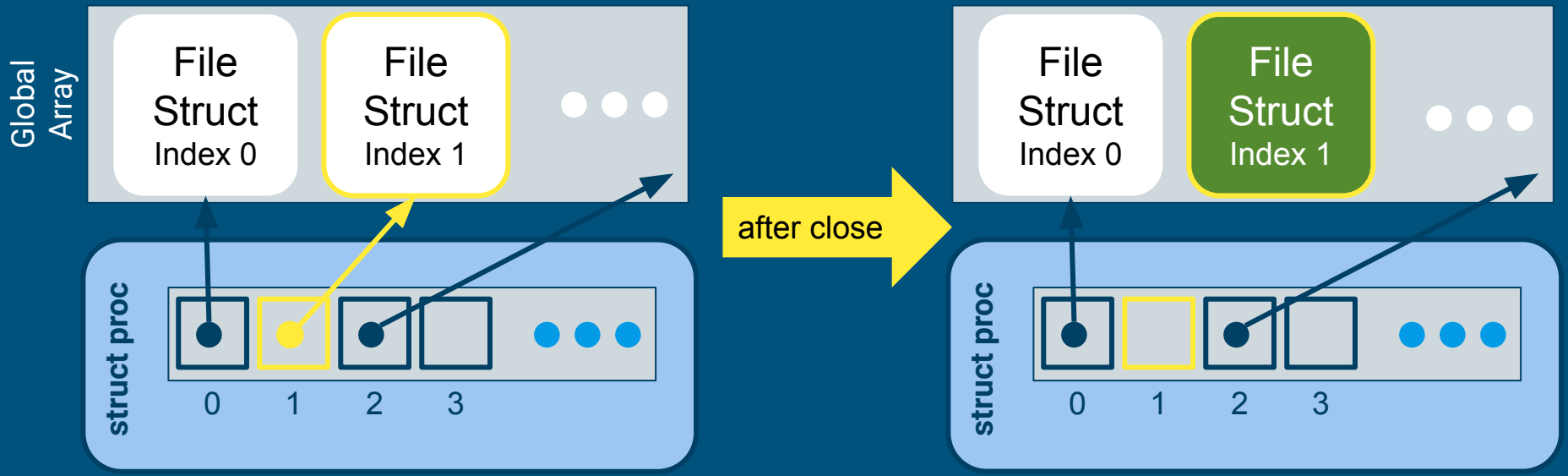
open(“file.txt”, O_RDONLY)

- Each open call allocates a new file_info struct
- Name lookup returns same inode

fileclose

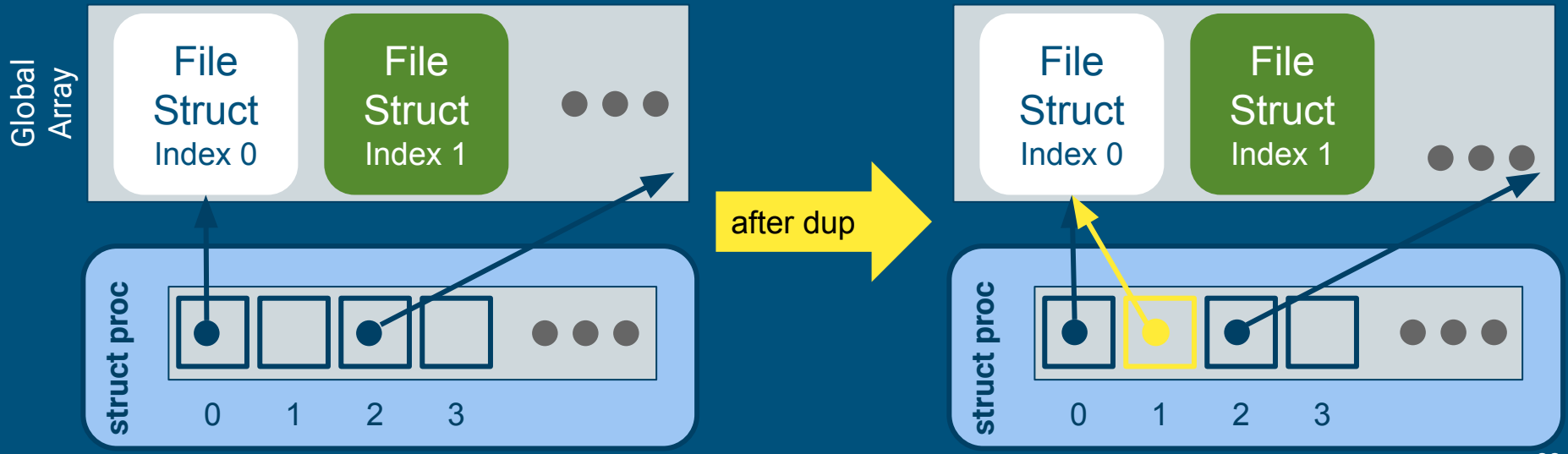
Release the file from this process, will have to clean up if this is the last reference

- make sure to irelease() before deallocating the file struct



filedup

Duplicates the file descriptor in the process' file descriptor table

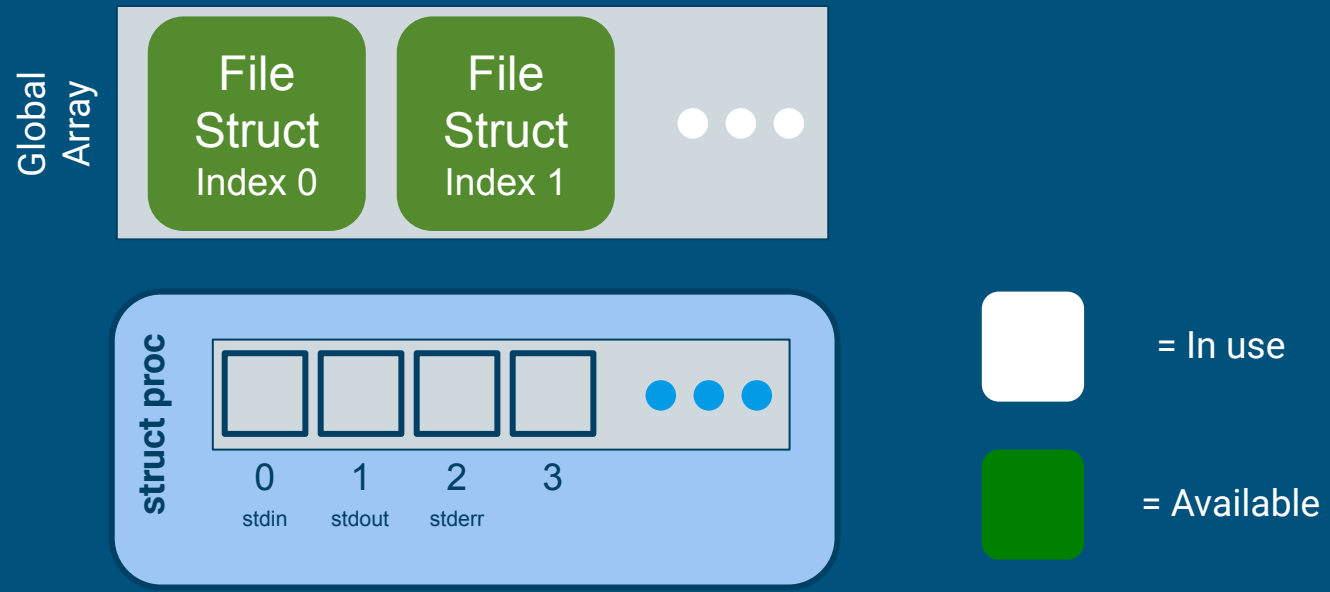


Lab 1 Test Program Code Fragment

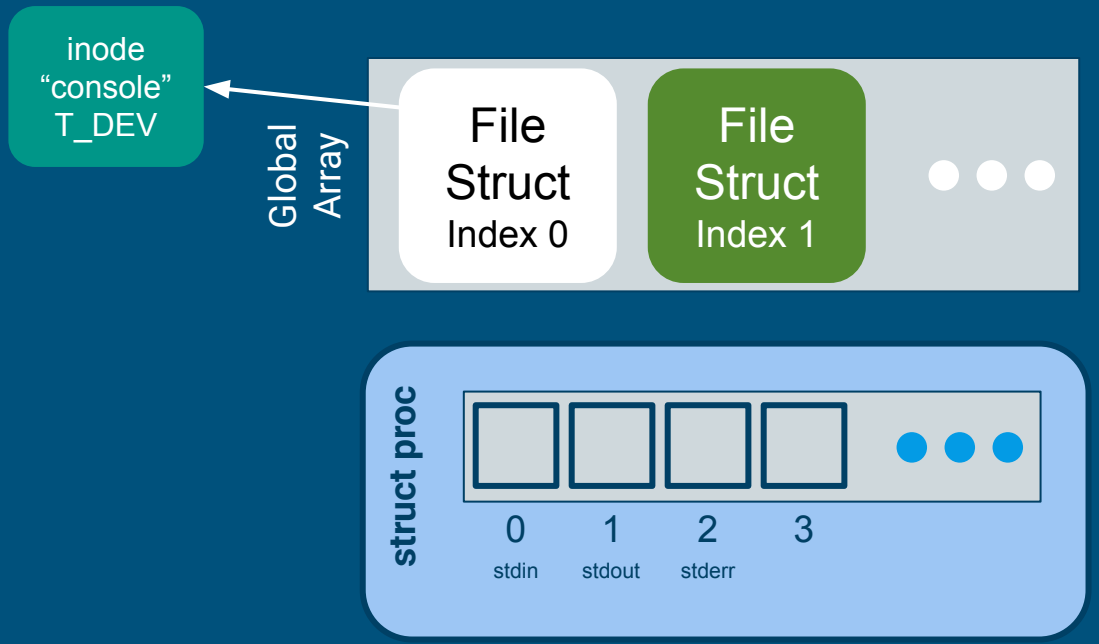
```
int stdout = 1;

int main() {
    if(open("console", O_RDWR) < 0){
        return -1;
    }
    dup(0);     // stdout
    dup(0);     // stderr

    printf(stdout, "hello world\n");
```

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

File Table View



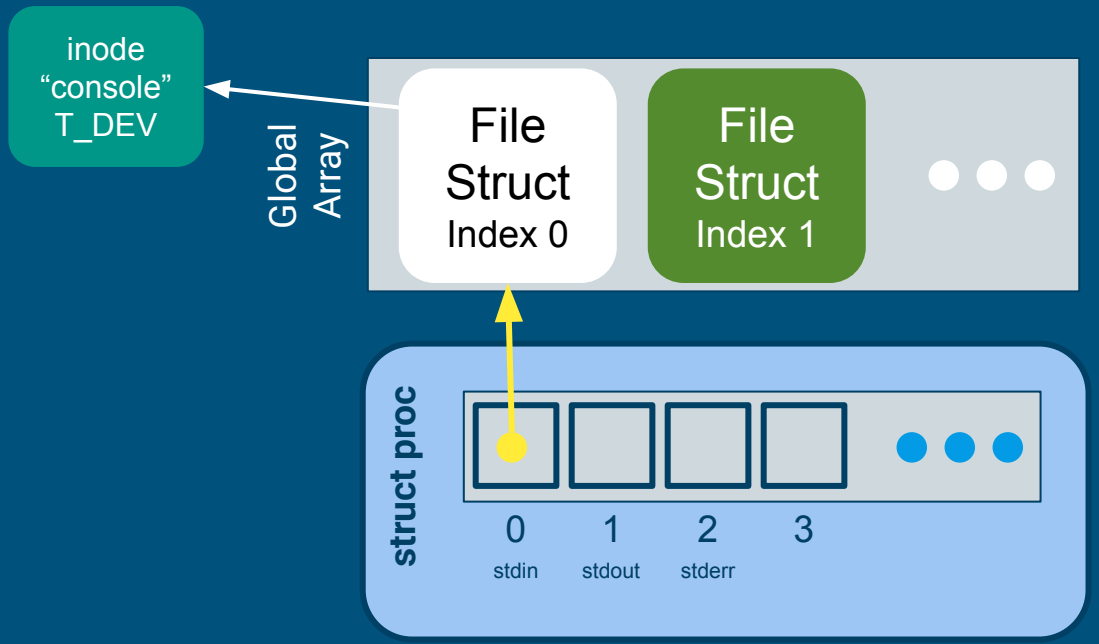
File Table View



open(“console”, O_RDWR)

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

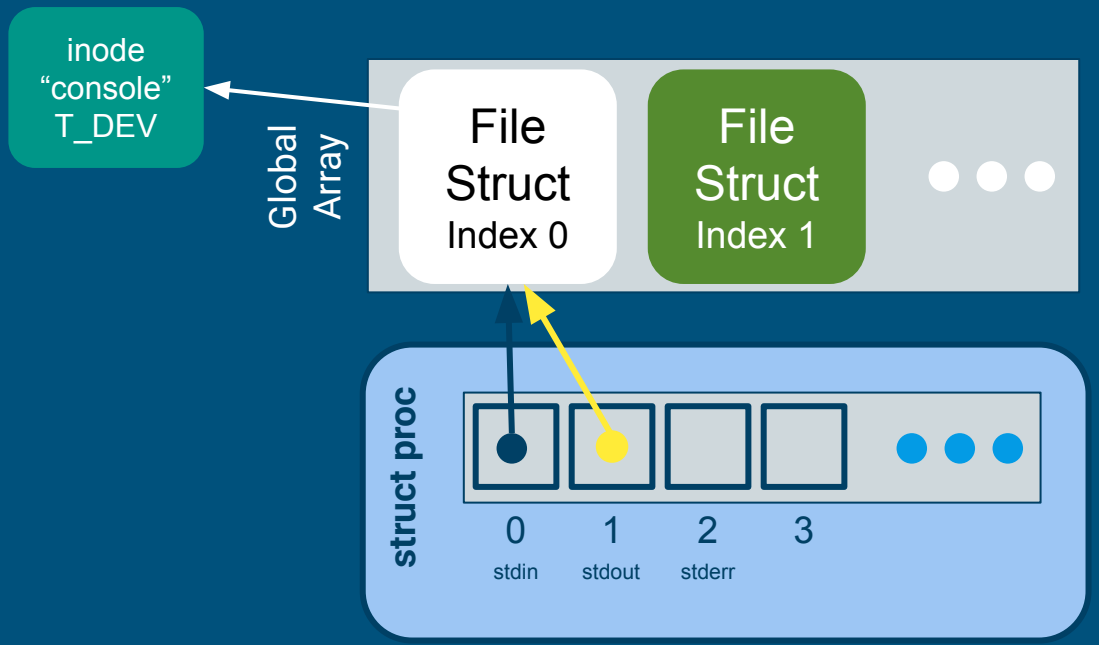
File Table View



open(“console”, O_RDWR)

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

File Table View

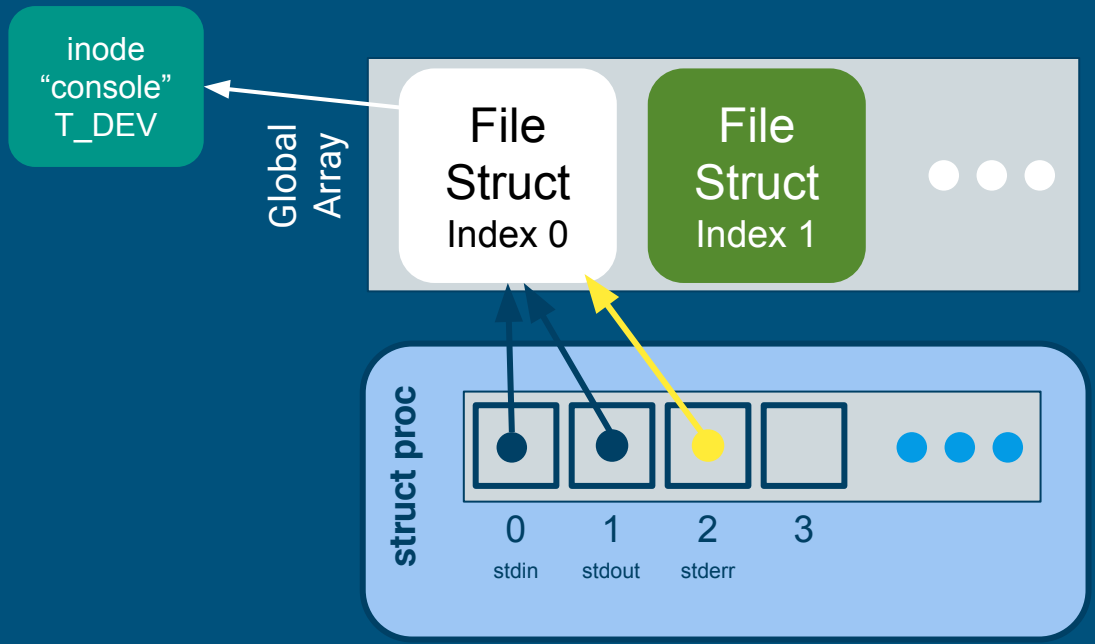


open(“console”, O_RDWR)

dup(0)

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

File Table View



open(“console”, O_RDWR)

dup(0)

dup(0)

Writing and Reading

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 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.

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 *readi* and *writei* defined in kernel/fs.c

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

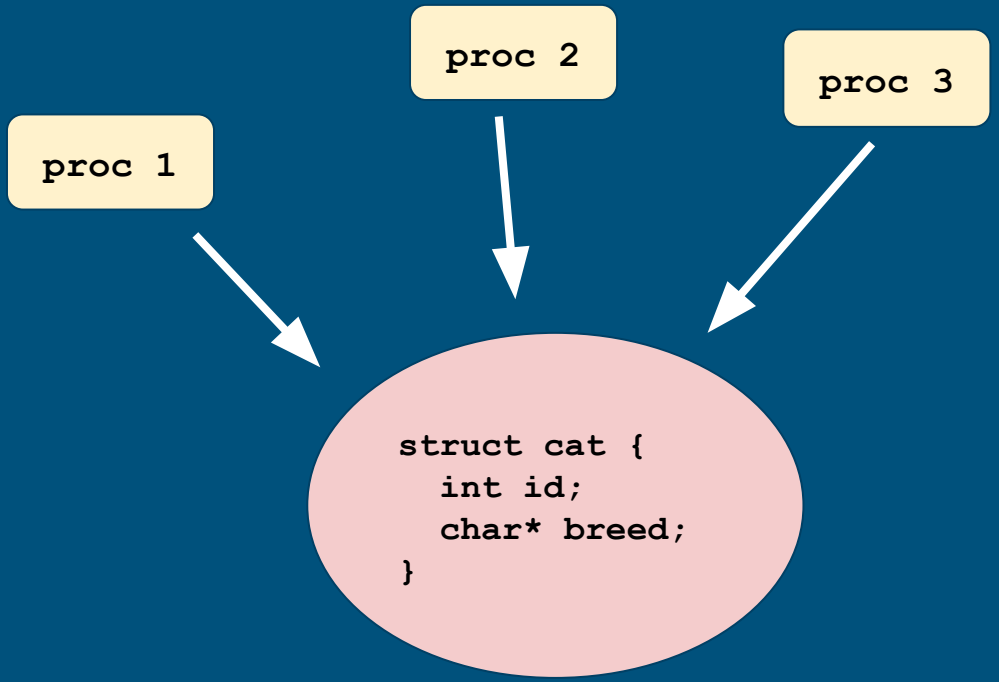
Userland	KERNEL LAND			
System Calls	File API	Inode API	Block API	IDE API
write() open()	filewrite() fileappend() filecreate()	writei() readi()	bread() bwrite() brelse()	iderw()

filestat

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

Reference Counting

Reference Counting



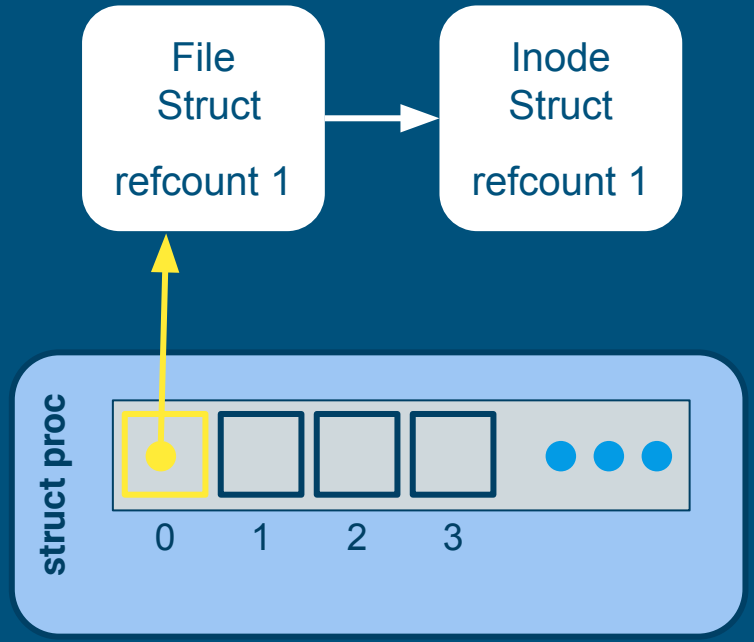
3 processes store a reference (ptr) to the struct cat

When is it safe to deallocate the struct cat?

Reference Counting

- Purpose of referencing counting
  - keeps track of how many references are there for the object
  - so we can know when it's safe to deallocate things!
- Reference count is specific to each struct
  - file's reference count might be different from inode's
  - everytime you store the pointer of a file struct somewhere, the refcount goes up
    - open, dup
  - everytime you remove a reference of a file struct, refcount should go down
    - close

Reference Counting

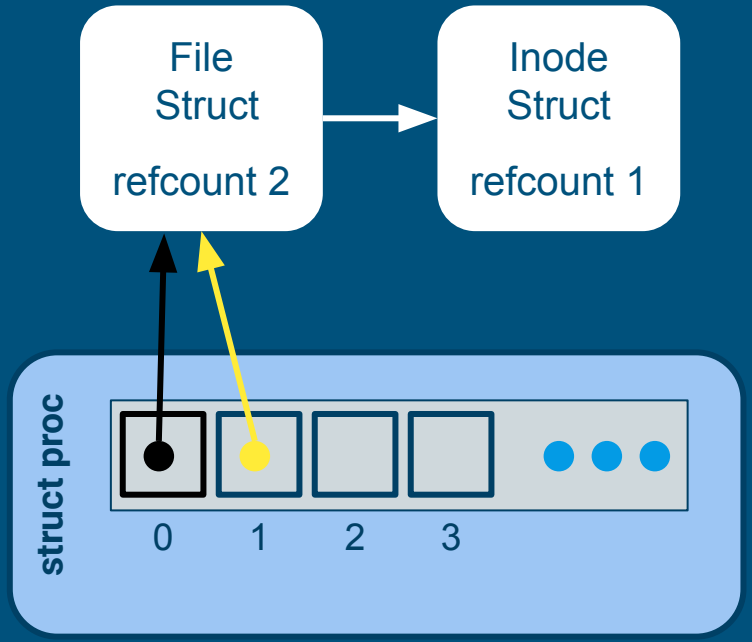


= In use

= Available

fd = open(file, readonly)

Reference Counting



= In use

= Available

dup(fd)

Memory in xk

Global Variables in C

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

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

// a static global variable is "private" - can only be used in
// the same file
static int num3;  // initialized to 0

// extern means the definition is external to this file
extern int num4;
// there needs to be a non-static definition of num4 in another file

// to make a global variable shared between files:
//  put an extern declaration in the header file
//  put the definition in the c file
//  every c file that includes the header file will have access to that variable

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

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

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 a virtual address to a physical address

Virtual memory for a process:

The kernel is mapped to the top for every process:

Why? Are there any risks?

Kernel stack

- Can also be referred to as interrupt stack
- Each process has its own kernel stack
- However, this is in the kernel section of the memory
- In xk, the kernel allocates one page which acts as the kernel stack during process creation
- From kernel.proc.c:allocproc:

```
// 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

Questions?