Systems I – CS 6013
Computer Architecture and Operating Systems
Lecture 14: The Unix Shell

MASTER OF SOFTWARE DEVELOPMENT (MSD) PROGRAM

J. DAVISON DE ST. GERMAIN

SPRING 2024

*(adapted from slides by Scott Brandt at UC Santa Cruz and other general sources, including previous MSD slides)

Lecture 14 – Topics

- The Unix Shell Assignment
  - How it fits together
  - Processes, Forking, Pipes, Exec

Miscellaneous

- Vocabulary from Industry Seminar, etc
  - Front end / back end / full stack
  - Excel / VBA
  - ***Know*** the things on your resume!

Unix Shell Assignment

- What processes are out there?
- What is "Shell" doing?

Shell

- Loop
  - Show ">", read user input
  - Type of user input?
    - String
  - What is the user input?
    - Command Line
  - What do we do with the CL?
    - Parse it… into what?
    - Tokens
    - Next step?
      - Create `commands`
      - …

sshd

emacs

/sbin/init

- PID?
  - 1

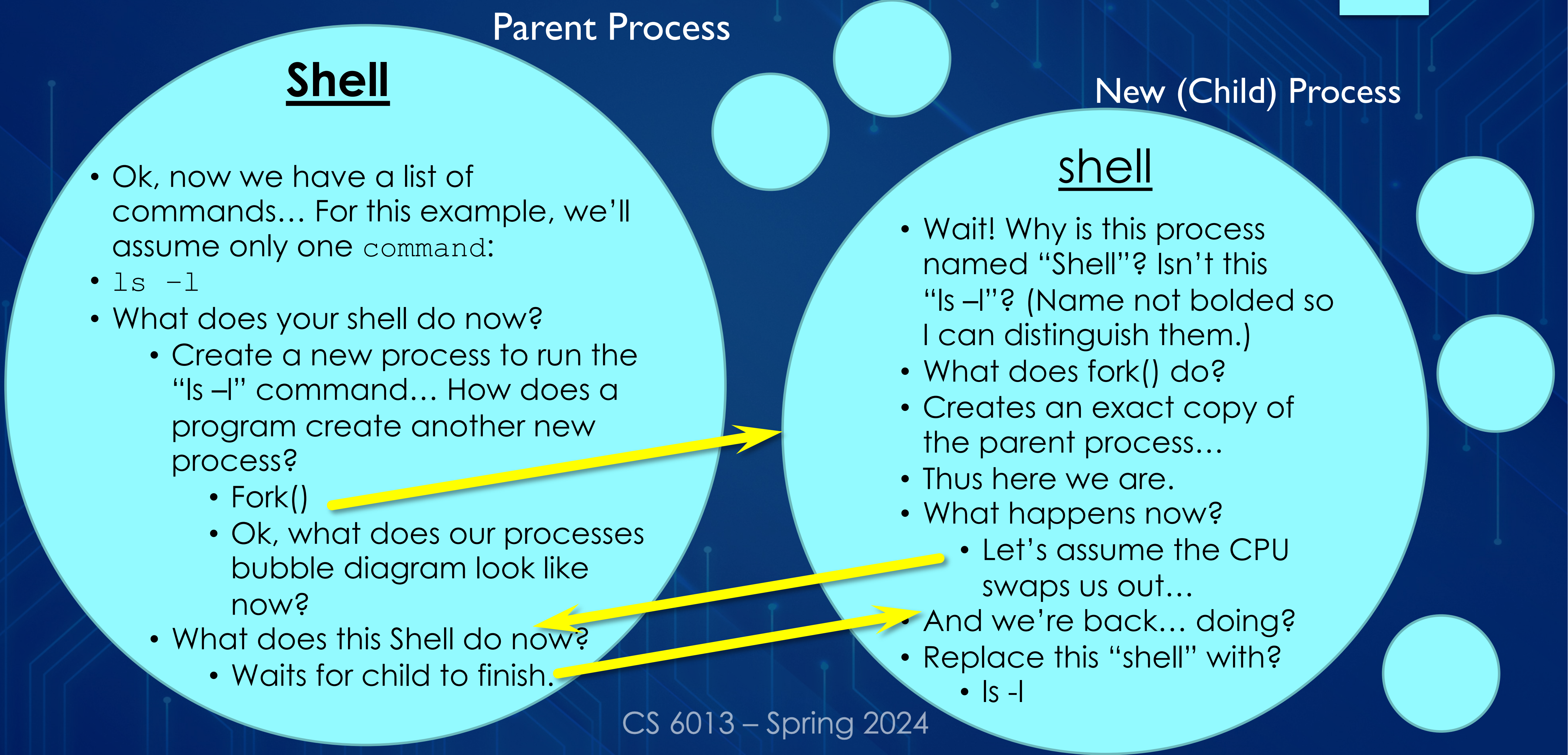
bash

ntpd

Network time Protocol Daemon

ps -def

Version 1 – Run a Single Command



Parent Process

Shell

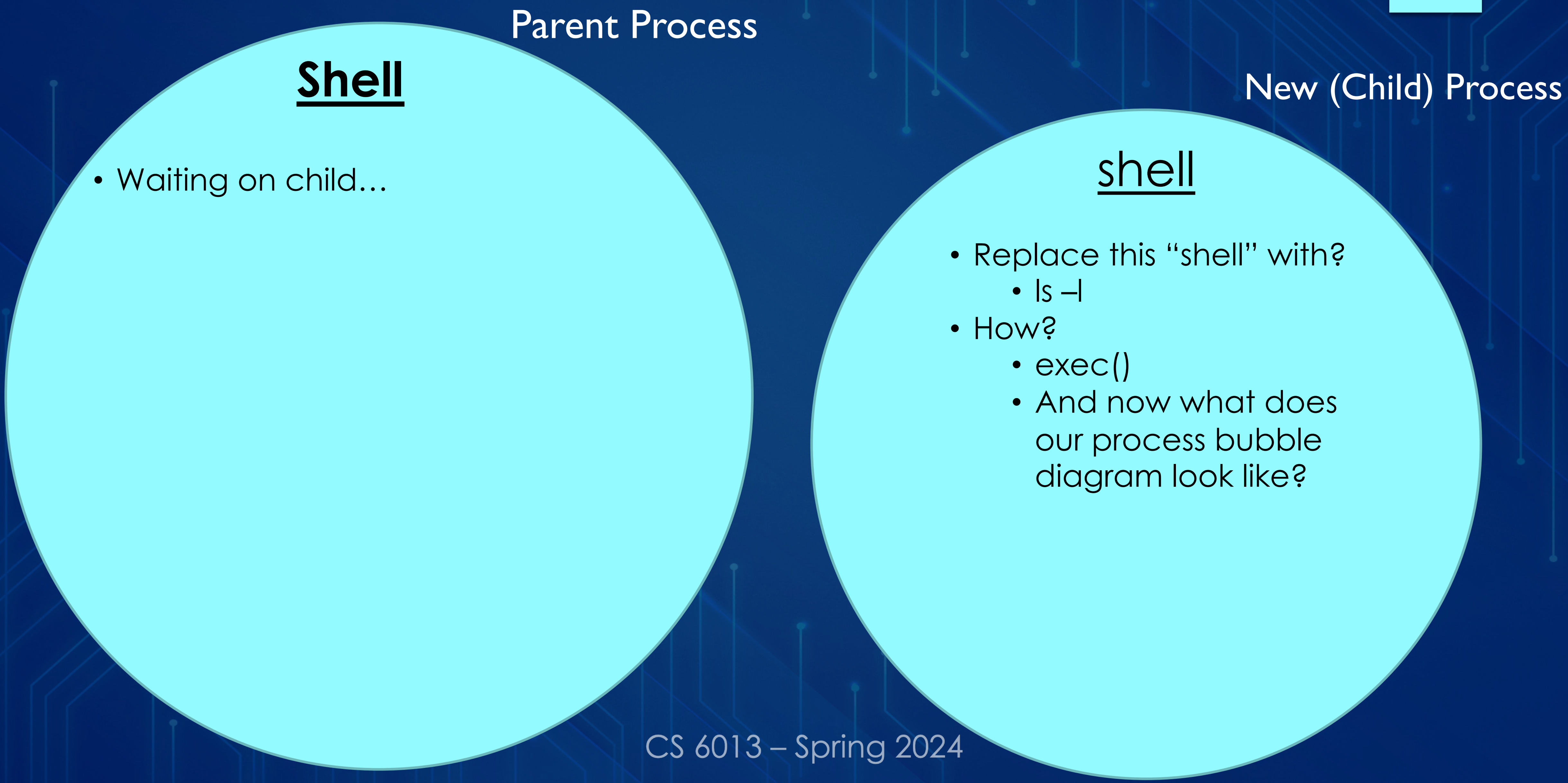
- Ok, now we have a list of commands… For this example, we’ll assume only one `command`:
- `ls -l`
- What does your shell do now?
  - Create a new process to run the “ls –l” command… How does a program create another new process?
    - Fork()
    - Ok, what does our processes bubble diagram look like now?
  - What does this Shell do now?
    - Waits for child to finish.

New (Child) Process

shell

- Wait! Why is this process named “Shell”? Isn’t this “ls –l”? (Name not bolded so I can distinguish them.)
- What does fork() do?
- Creates an exact copy of the parent process…
- Thus here we are.
- What happens now?
  - Let’s assume the CPU swaps us out…
- And we’re back… doing?
- Replace this “shell” with?
  - ls -l

Version 1 – Run a Single Command



Parent Process

Shell

- Waiting on child…

New (Child) Process

shell

- Replace this "shell" with?
  - ls –l
- How?
  - exec()
  - And now what does our process bubble diagram look like?

Version 1 – Run a Single Command

Shell

- Waiting on child…

ls

- "Ah! I feel like a brand-new process!"
- Where'd the "-l" go?
  - Not part of the process name.
  - Stored in?
    - `main(` **`argv[]`** `)`
  - Next, what does `ls` do?
  - **`ls`** does its thing (lists files in the current directory)
- And then?
  - returns / exits
- And our bubble diagram?

Version 1 – Run a Single Command

Shell

- Waiting on child…
  - Child is gone (finished / exited)
- And we're back…
- Shell continues with?
  - Read command line.
  - `w | grep dav > out.txt`

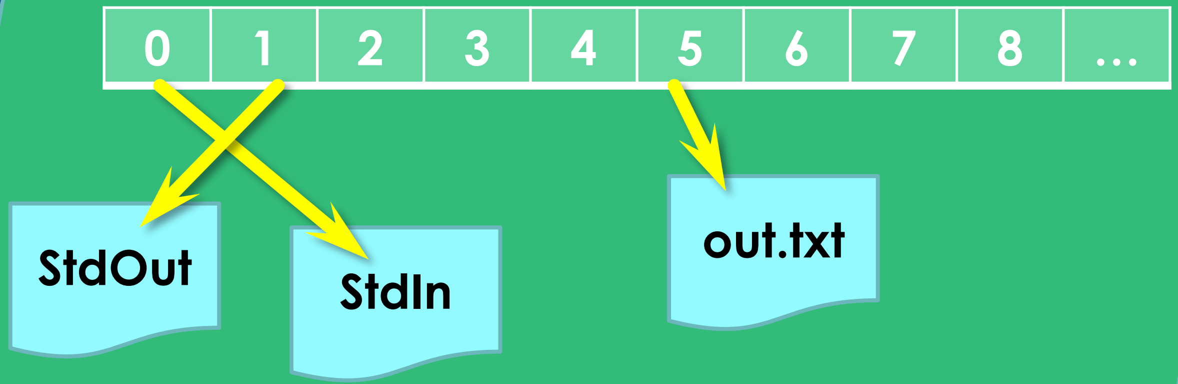
Version 2 – Single Command w/ output

Shell

- ls –l > out.txt
- What is "> "
  - Redirection of output to a file.
  - What does the `struct Command` look like after parsing the above command line?
    - `execName`?
      - ls
    - `argv`?
      - [ls, -l]   (Notice, no "> out.txt")
    - `background`?
      - False
    - `inputFd`?
      - 0 (standard in)
    - `outputFd`?
      - 5 (why 5?)

Kernel Memory

File descriptor table (for **Shell**)



- Note: Kernel memory diagram overlaps **Shell** process on purpose, why…
  - Because Shell has some of its data in the kernel!
- The FDT is indexed by a bunch of integers, but what does it actually contain*?
  - Each bucket contains a "pointer" to a file!!!
  - Which includes information (for the OS) on how to send/receive data to/from that thing (monitor, keyboard, actual file, pipe, etc).
- Back to our question, why 5?

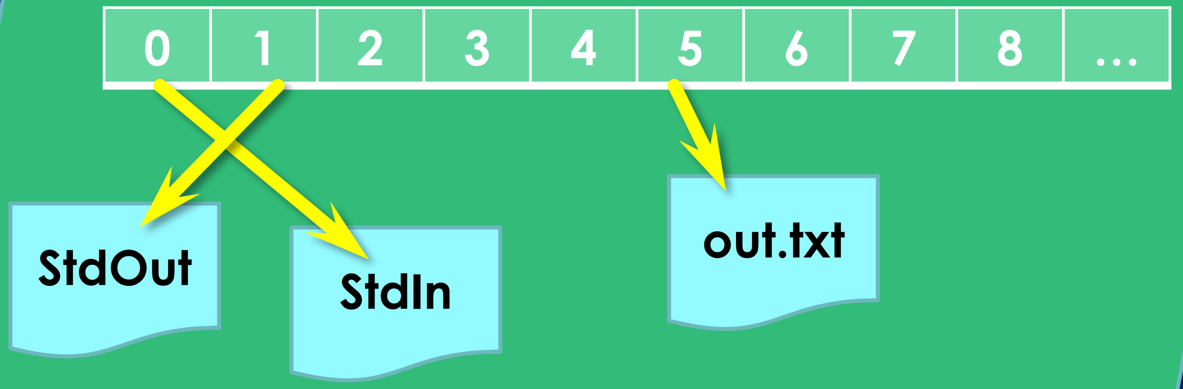
Version 2 – Single Command w/ output

Shell

- ls –l > out.txt
- So, what code does the shell run to create this file?
  - int fd = open(“out.txt”, …)
  - And then does what with this FD?
  - Remember the shell is currently in the process of parsing the command line…
    - What functions do this?
    - `tokenize()`
      - Breaks it into tokens but doesn’t know anything about them…
    - `getCommands()`
      - Yes, the open happens here…

Kernel Memory

File descriptor table (for **Shell**)



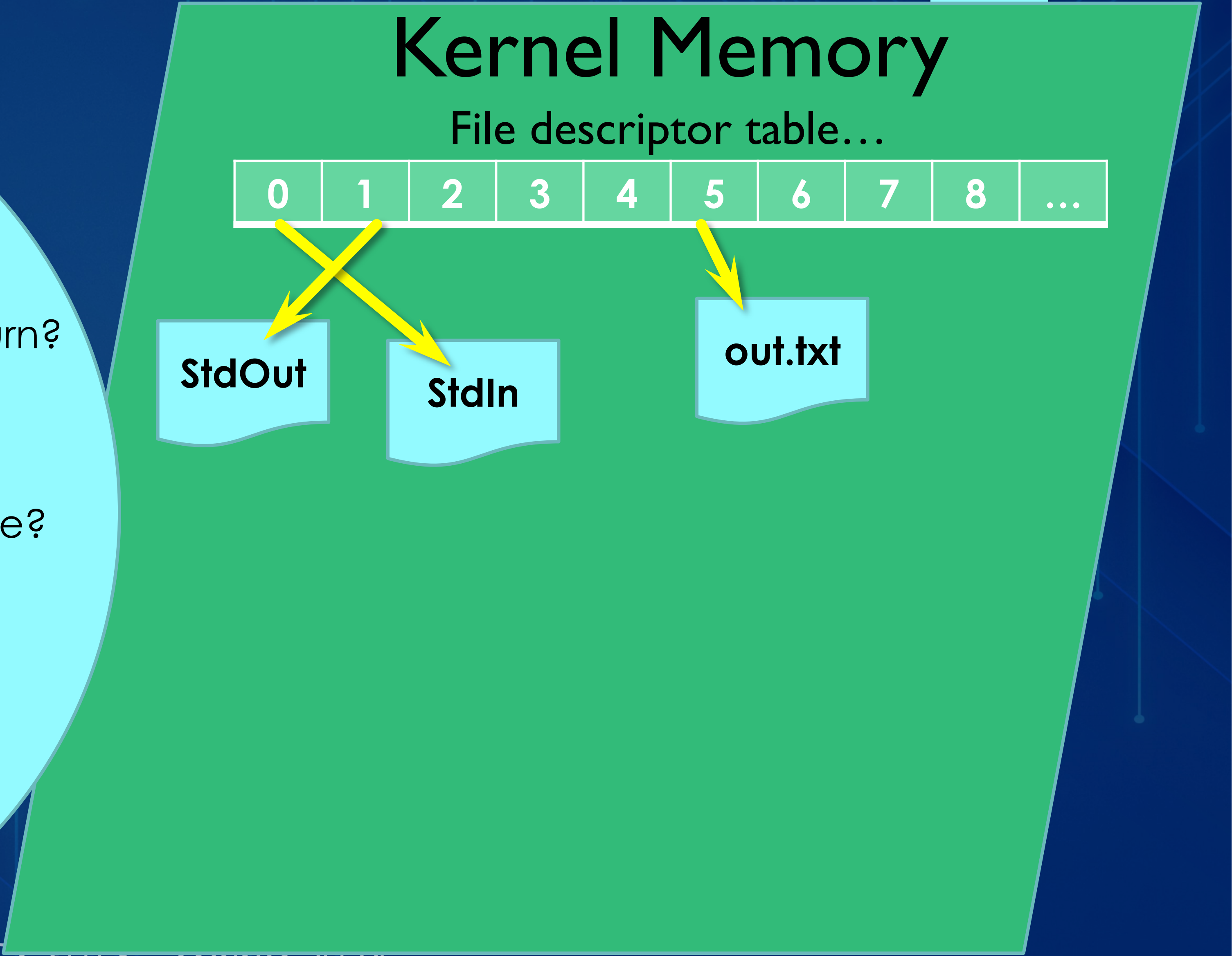
0	1	2	3	4	5	6	7	8	…

- Taking a step back… who creates (opens) out.txt?
  - The Shell. Why?
  - No easy / generic way to pass “> out.txt” to the child process.
  - Shell is responsible for setting up all the “initial” file descriptors for its children to use.

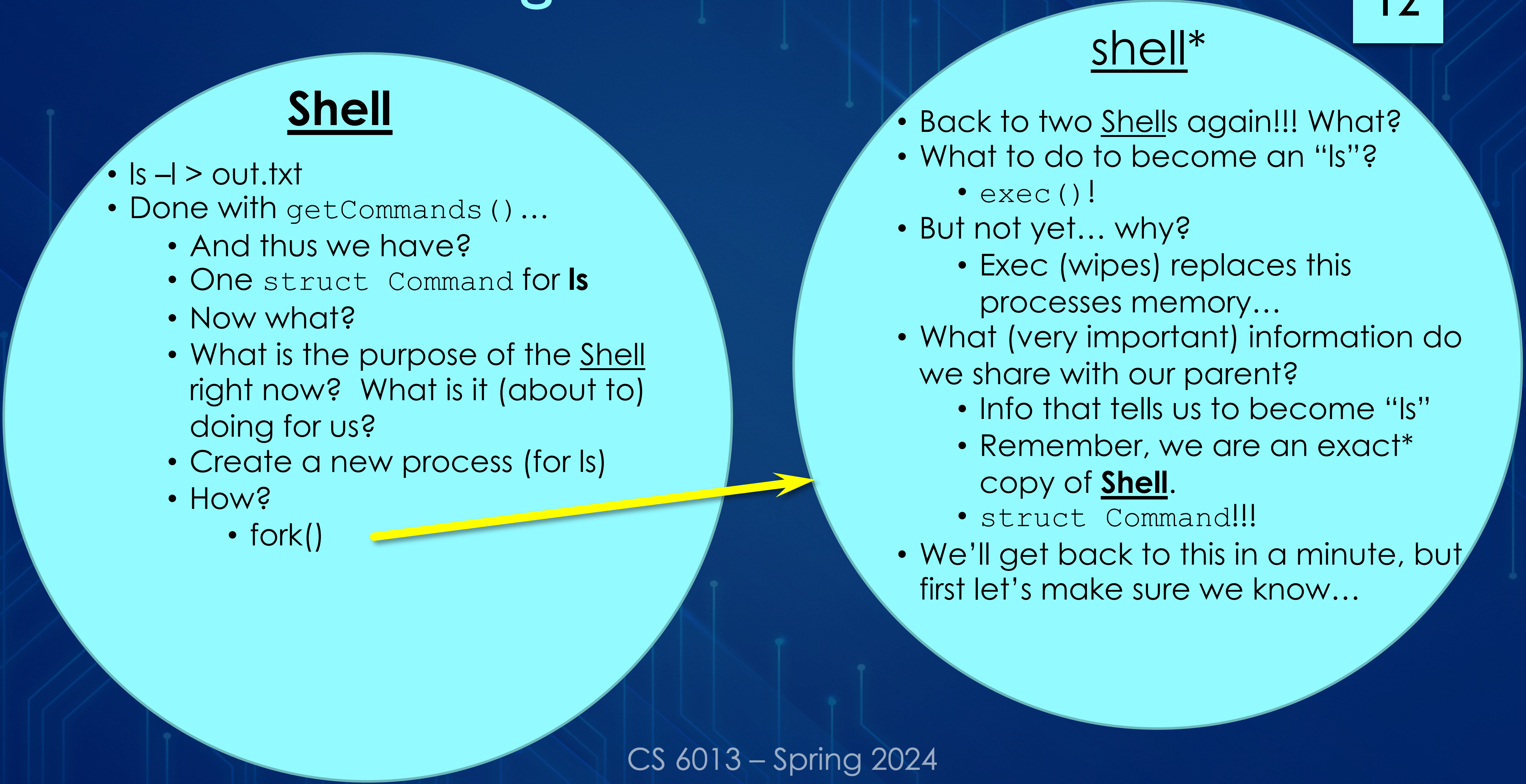
Version 2 – Single Command w/ output

Shell

- ls –l > out.txt
- `getCommands()`
  - `int fd = open( "out.txt"…`
  - What does getCommands return?
  - A vector of commands!
  - In this case?
    - A single `struct Command`.
  - At this point, that struct looks like?
    - `execName`: ls
    - `argv`: [ls, -l]
    - `background`: False
    - `inputFd`: 0 (standard in)
    - `outputFd`?
      - 5 (why 5?)
      - fd above has a value of 5!



Version 2 – Single Command w/ output



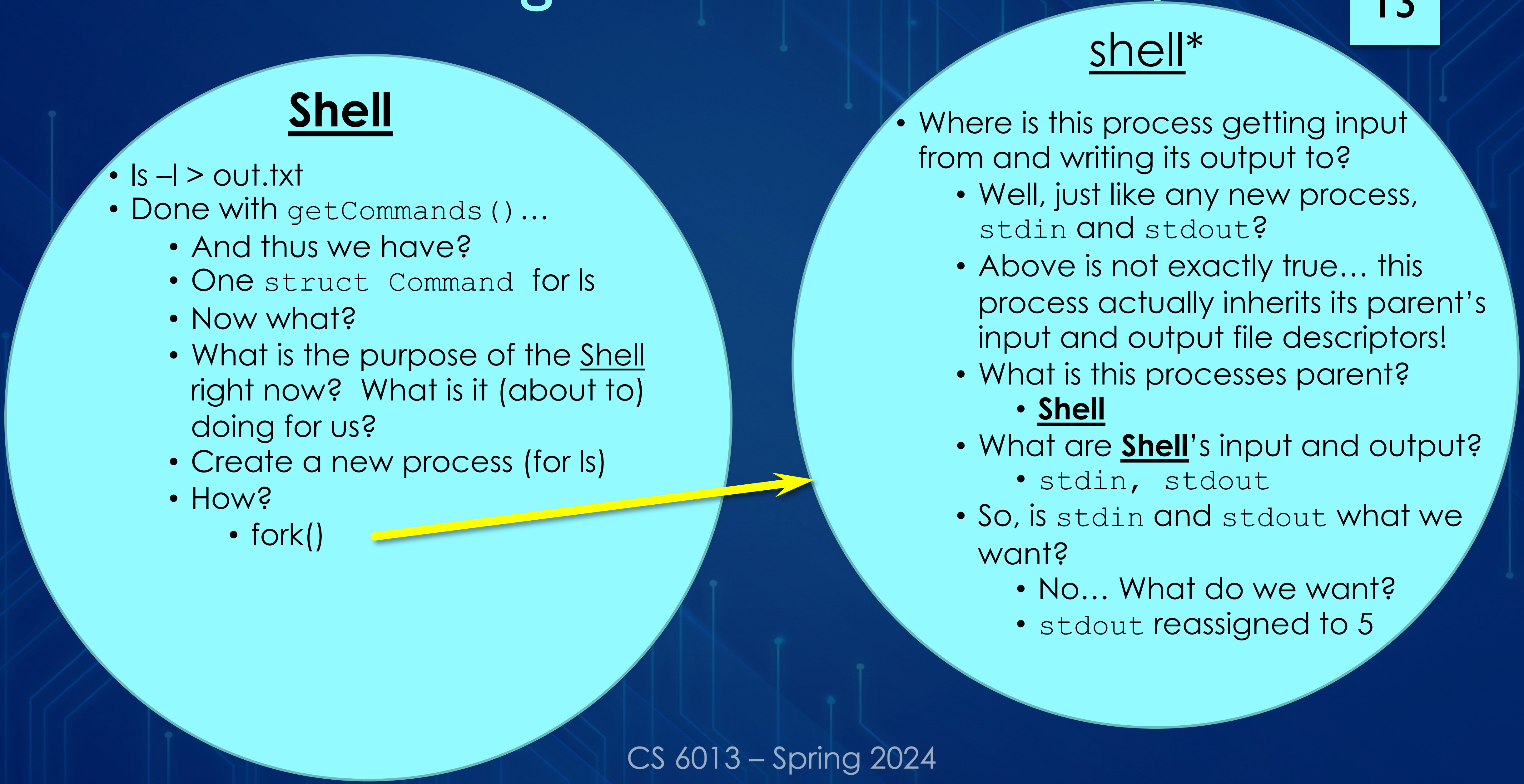
Shell

- ls –l > out.txt
- Done with `getCommands()`…
  - And thus we have?
  - One `struct Command` for **ls**
  - Now what?
  - What is the purpose of the Shell right now? What is it (about to) doing for us?
  - Create a new process (for ls)
  - How?
    - fork()

shell*

- Back to two Shells again!!! What?
- What to do to become an "ls"?
  - `exec()`!
- But not yet… why?
  - Exec (wipes) replaces this processes memory…
- What (very important) information do we share with our parent?
  - Info that tells us to become "ls"
  - Remember, we are an exact* copy of **Shell**.
  - `struct Command`!!!
- We'll get back to this in a minute, but first let's make sure we know…

Version 2 – Single Command w/ output



Shell

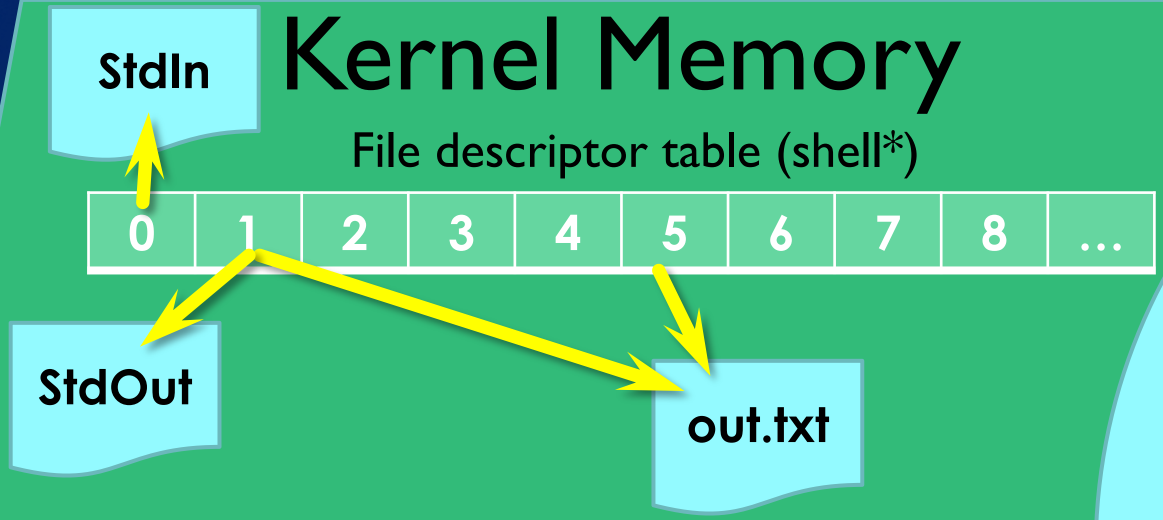
- ls –l > out.txt
- Done with `getCommands()`…
  - And thus we have?
  - One `struct Command` for ls
  - Now what?
  - What is the purpose of the Shell right now? What is it (about to) doing for us?
  - Create a new process (for ls)
  - How?
    - fork()

shell*

- Where is this process getting input from and writing its output to?
  - Well, just like any new process, `stdin` and `stdout`?
  - Above is not exactly true… this process actually inherits its parent's input and output file descriptors!
  - What is this processes parent?
    - **Shell**
  - What are **Shell**'s input and output?
    - `stdin, stdout`
  - So, is `stdin` and `stdout` what we want?
    - No… What do we want?
    - `stdout` reassigned to 5

Version 2 – Single Command w/ output

Kernel Memory

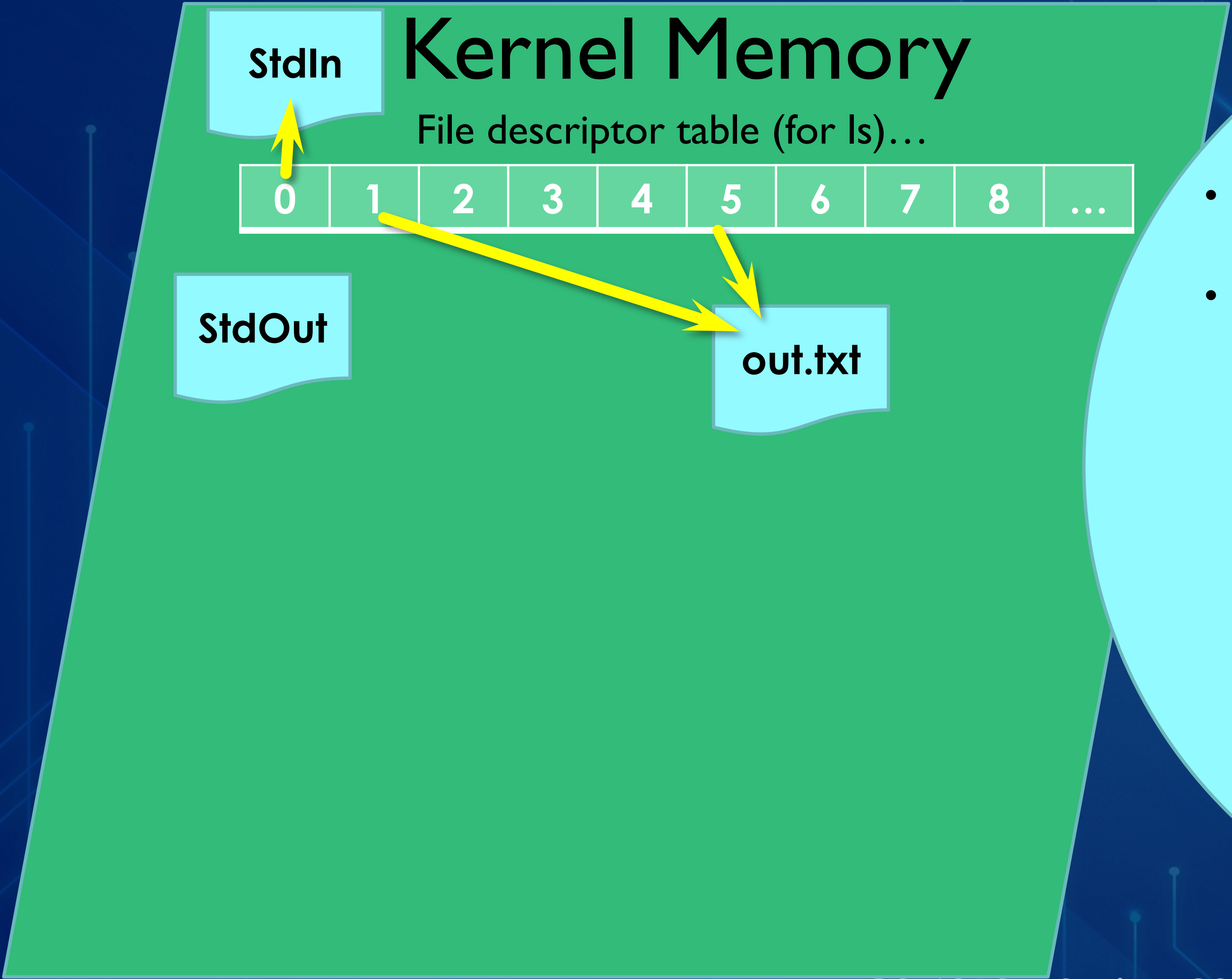


- Remember Kernel Memory and the FDT?
- **Shell** (seemingly a long time ago) opened "out.txt" and it was associated with FD 5. (before `fork()`)
- That still exists in Kernel memory and we have a copy of it.
- So dup2 updates this picture to look like…
- Note, there is a FDT for each process. The FDT we see here belongs to shell* (the copy), but began as an exact copy of which (whose) FDT?
  - **Shell** (original / main shell process)

shell*

- How to we change our `stdout` to be file descriptor 5?
- How do we "rename" a file descriptor?
- How do we duplicate a file descriptor and replace it with a different FD?
  - dup2()
- So now what are we (finally) ready to do?
  - Replace Shell with ls… how?
  - `exec()`
- But `exec()` will blow away all of our "memory" including our file descriptors…
  - Or does it? Where is the/our In/Out FD info stored?
    - In the Kernel FDT

Version 2 – Single Command w/ output



ls

- `exec()` is done, and this process is now `ls`
- While this process is busy doing its job…

ie: making system calls to ask the OS about the files in the current directory and then making more system calls to send that information to its “stdout”,

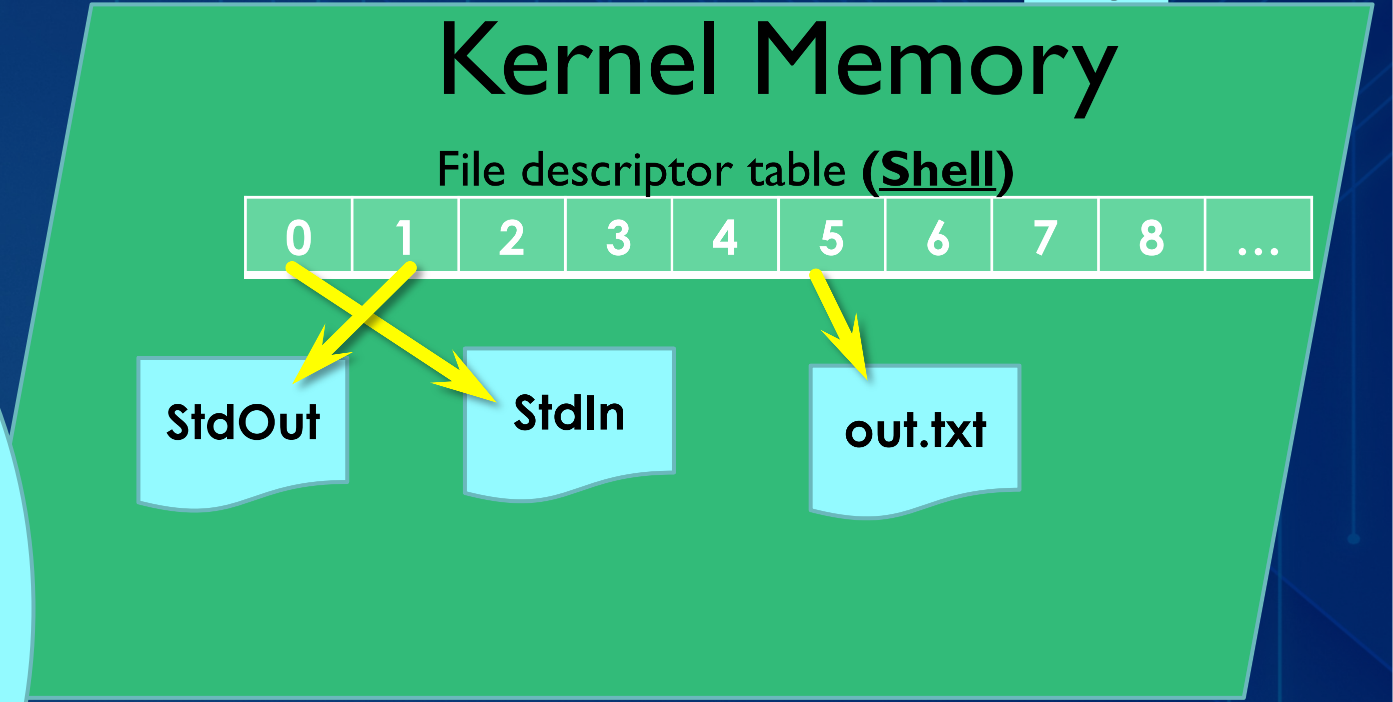
```
cout << fileInfoString << "\n";
```

…let’s jump back to **Shell**…

Back to Parent (Original/Real Shell)

Shell

- The main **Shell**'s FDT still looks like this…
- What does it need to do?
- Does **Shell** do anything with out.txt?
  - So what should it do?
  - `close( fd )`
    // Remember fd == 5



Kernel Memory

File descriptor table (**Shell**)

0	1	2	3	4	5	6	7	8	…

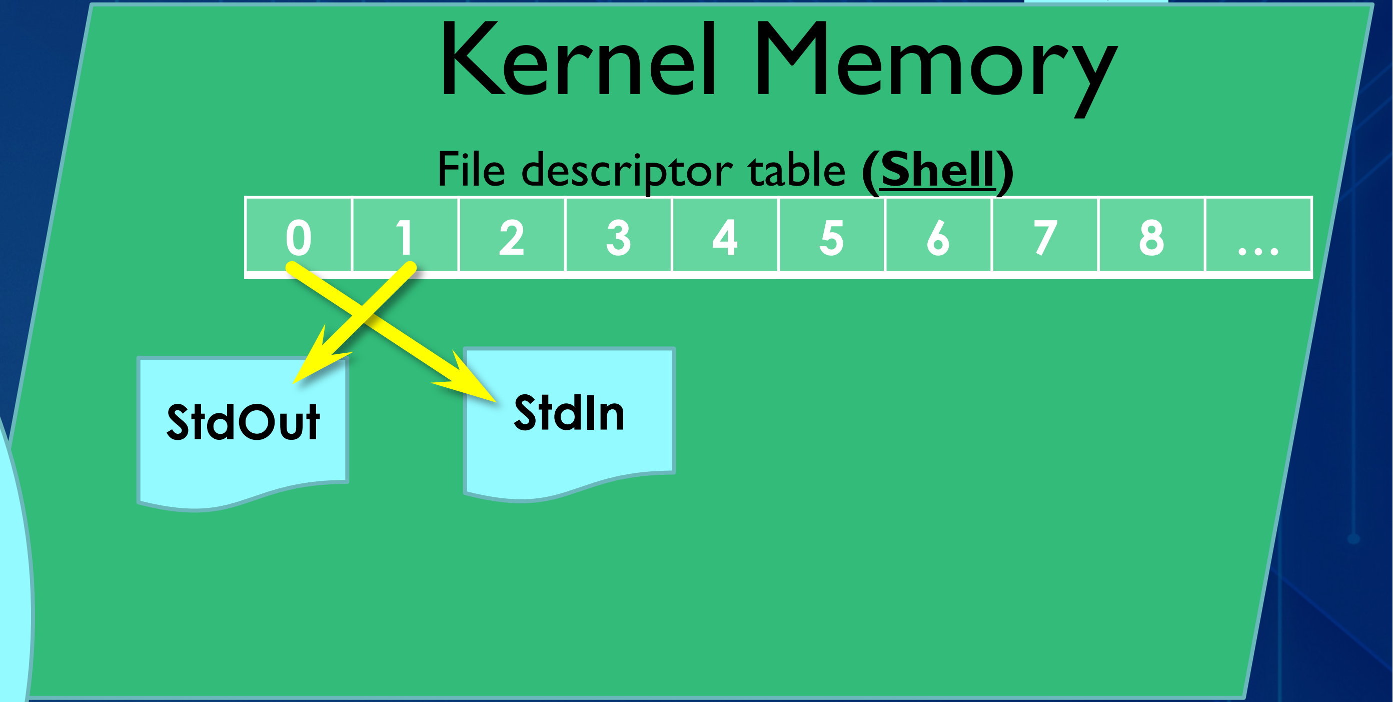
ls

Remember, I'm still out here. ☺

Version 3 – Multiple | Commands

Shell

- ls –l | nl | tail
- Step one?
  - tokenize()
- Step two?
  - getCommands()
- What happened when getCommands() parsed the first | (pipe) in the token list?
  - What does the | mean sitting there between **ls –l** and **nl**?
  - Send the output from **ls –l** to the input of **nl**!
- What will **ls** and **nl** become (shortly)?
  - Processes
  - How to send data between processes?
    - Pipes!



Kernel Memory

File descriptor table **(Shell)**

0	1	2	3	4	5	6	7	8	…

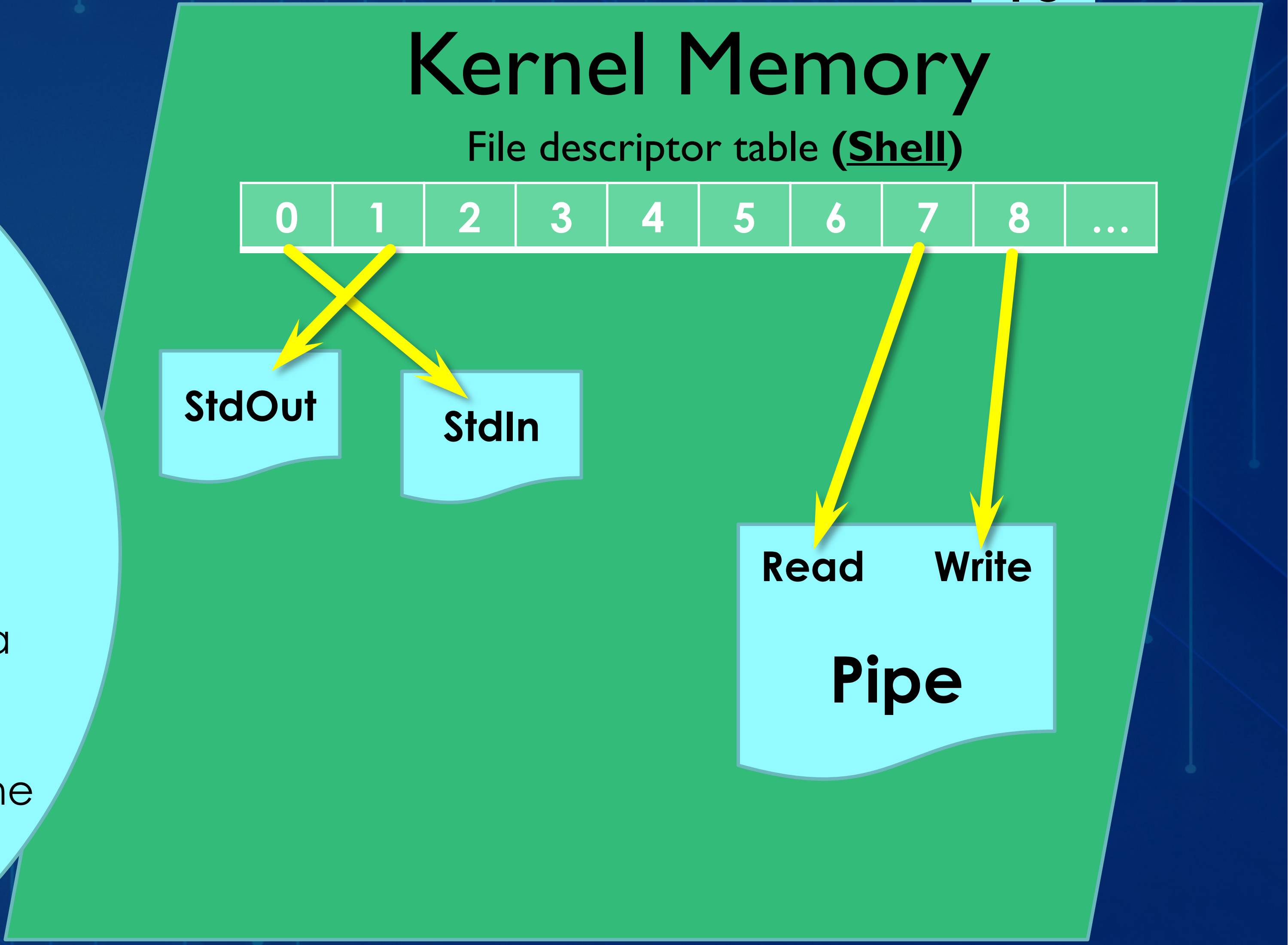
`ls`

Remember, I'm still out here. ☺

Version 3 – Multiple | Commands

Shell

- `ls -l | nl | tail`
- `getCommands()`
  - See a |, create a pipe:
  - `pipe( fds )`
- What are the values in `fds`?
  - Shrug, but let's assume 7 and 8.
  - fds[0] == 7 // read end of pipe
  - fds[1] == 8 // write end of pipe
- `pipe()` created a pipe object (data structure)… where is it?
- In the Kernel memory!
- Where are the 7 and 8 stored (not the pipe itself, just the FD numbers) with respect to **Shell**?
  - What function are we in?
    - In `getCommands()`



Version 3 – Multiple | Commands

Shell

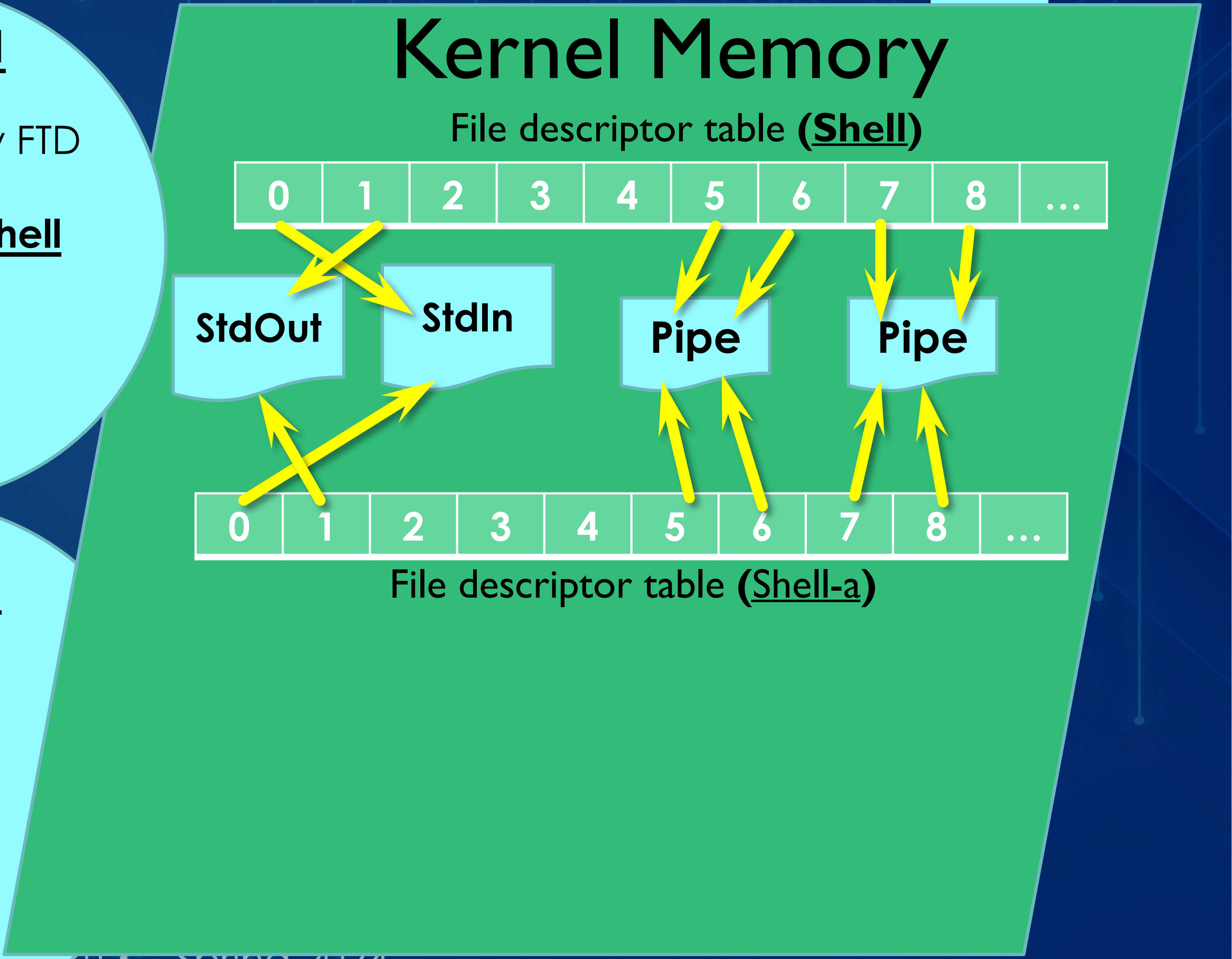
- ls –l | nl | tail
- fork -> **ls –l**
- fork -> **nl**
- fork -> **tail** (pipe (b) for **nl** to **tail** was also created.
  - **Bubbles now?**

Shell-a

- What does my FTD look like?
  - Copy of **Shell**

Shell-b

Shell-c



Kernel Memory

File descriptor table (Shell)

0	1	2	3	4	5	6	7	8	…

StdOut

StdIn

Pipe

Pipe

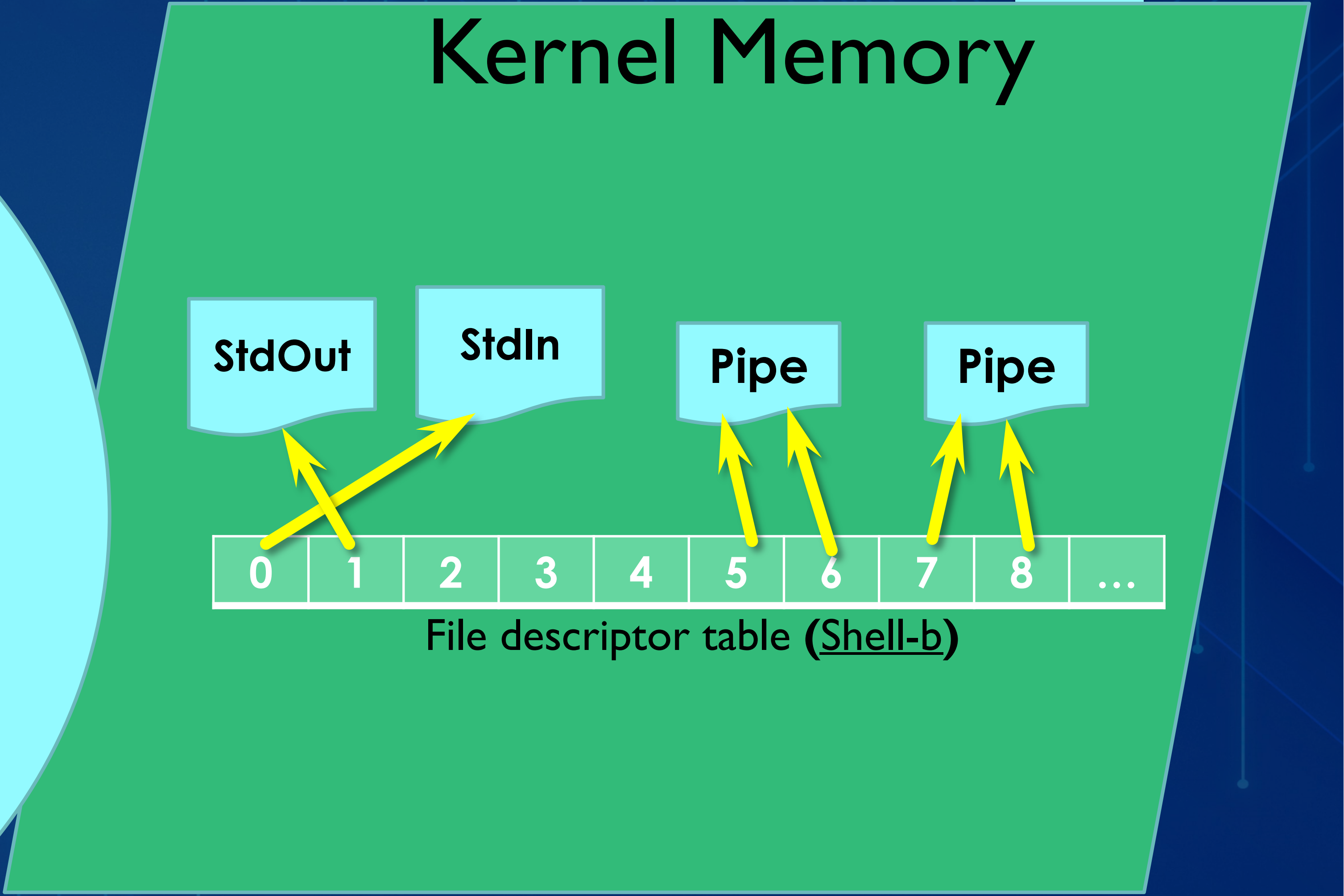
0	1	2	3	4	5	6	7	8	…

File descriptor table (Shell-a)

Version 3 – Multiple | Commands

Shell-b

- Soon I'll be **nl**
- But before I become **nl**, what do I need to do?
- What is my output supposed to be?
  - Sent to **tail**
- What is my input supposed to be?
  - Received from **ls -l**
- How do we do this?
  - How did we handle **ls > out.txt**?
- dup2, and dup2 again…

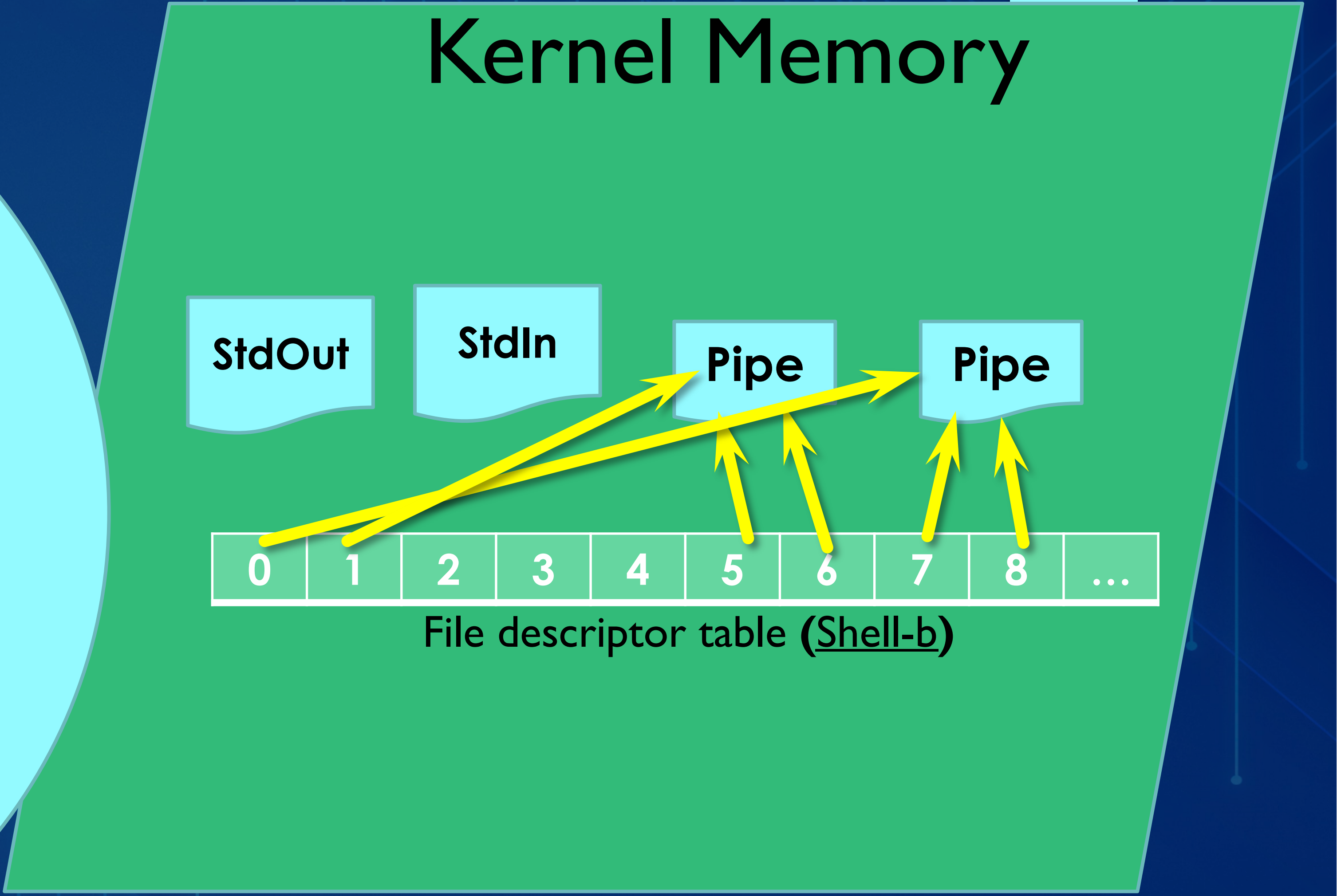


```
ls -l | nl | tail
```

Version 3 – Multiple | Commands

Shell-b

- Soon I'll be **nl**
- But before I become **nl**, what do I need to do?
- What is my output supposed to be?
  - Sent to **tail**
- What is my input supposed to be?
  - Received from **ls -l**
- How do we do this?
  - How did we handle **ls > out.txt**?
- dup2, and dup2 again…
  - Where are the FDs I'm replacing stored?
    - struct command
    - Specifically?
      - .inputFd
      - .outputFd



File descriptor table (Shell-b)

```
ls -l | nl | tail
```

Assignments

- Code Review?
  - Anyone want me to review their lab code?
- Unix Shell
  - More Questions?
- Named Pipes (FIFOs) – Coding / Using

~ *Fin* ~