Systems 1 – CS 6013

Computer Architecture and Operating Systems

Lecture 16: Virtual Memory, Part 2

MASTER OF SOFTWARE DEVELOPMENT (MSD) PROGRAM

J. DAVISON DE ST. GERMAIN

SPRING 2024

*(adapted from slides by Ryan Stutsman, Andrea Arpaci-Dusseau, and Sarah Diesburg, and MSD presentations)

Lecture 16 – Topics

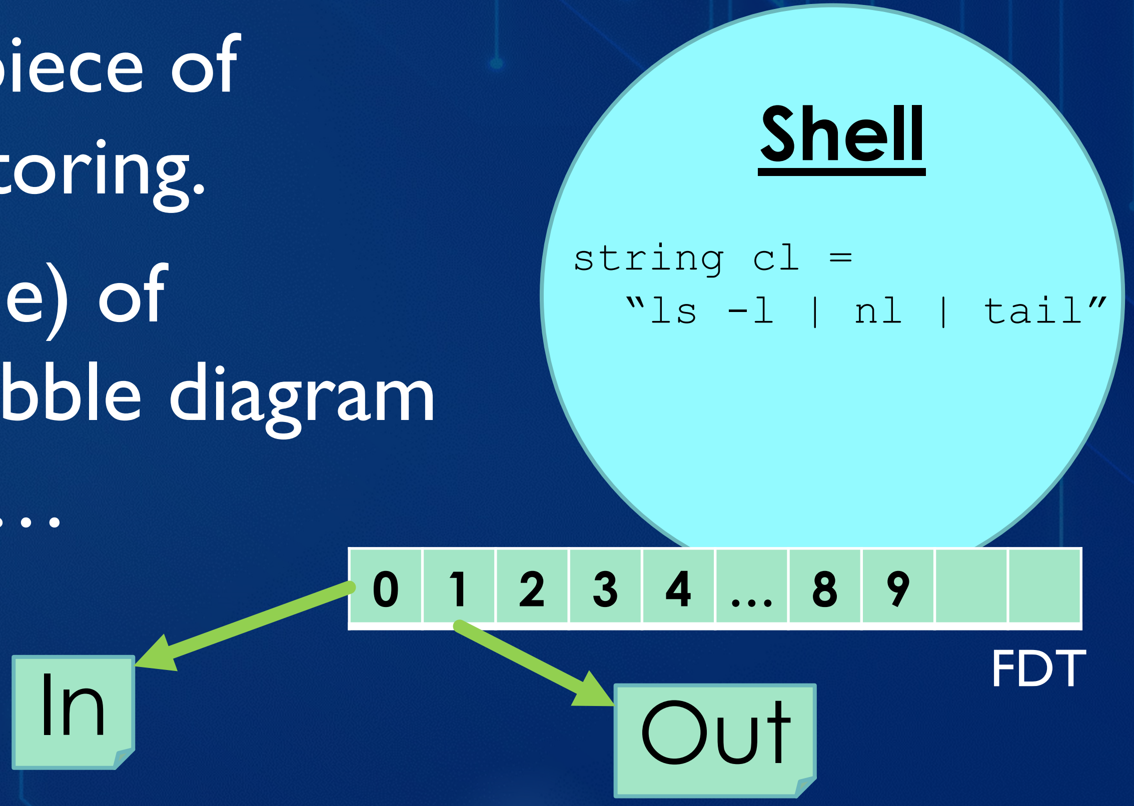
- Address Translation
  - Base / Bound
  - Segmentation

Announcements / Questions

- Unix Shell?
  - Check return codes on system calls
  - Everyone on track to have it done by Friday?

Unix Shell

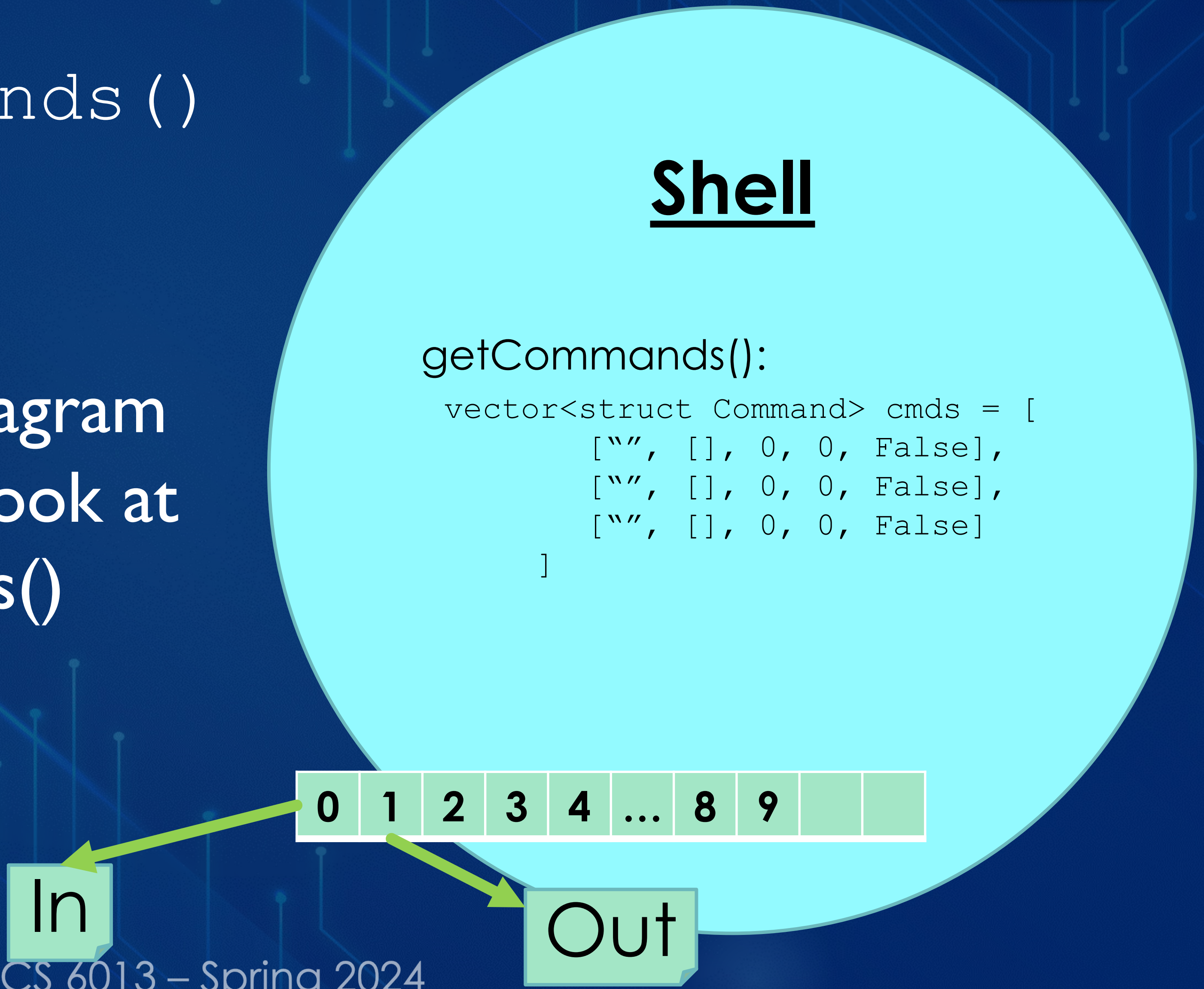
- The user has typed "ls -l | nl | tail"
  - Draw a process bubble picture that includes processes that exist, and every piece of information they are currently storing.
- Now, at the beginning (1st or 2nd line) of `getCommands()`, draw a new bubble diagram that shows all data that now exists…



Unix Shell

`ls -l | nl | tail`

- Beginning of `getCommands()`
  - Note, 0 == Std In FD
  - 1 == Std Out FD
- Now draw the bubble diagram detailing the way things look at the end of getCommands()



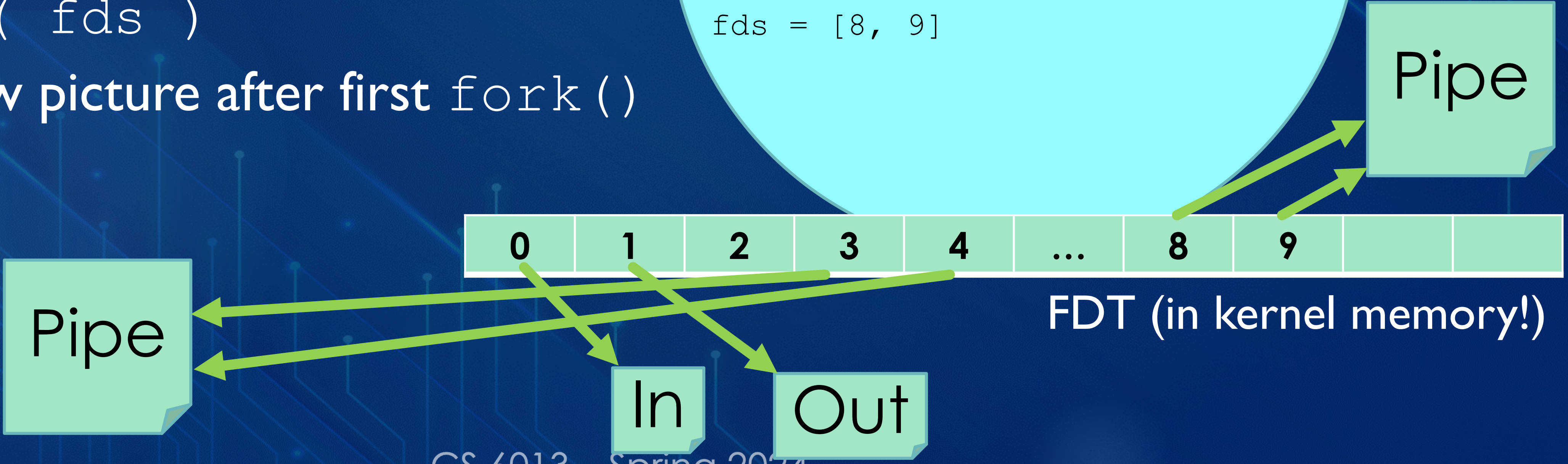
Unix Shell

`ls -l | nl | tail`

- End of `getCommands()`
  - Note, 0 == Std In FD
  - 1 == Std Out FD
- Why 3, 4, 8, 9? Where did these numbers come from? Mean?
  - `pipe( fds )`
- Now draw picture after first `fork()`

Shell

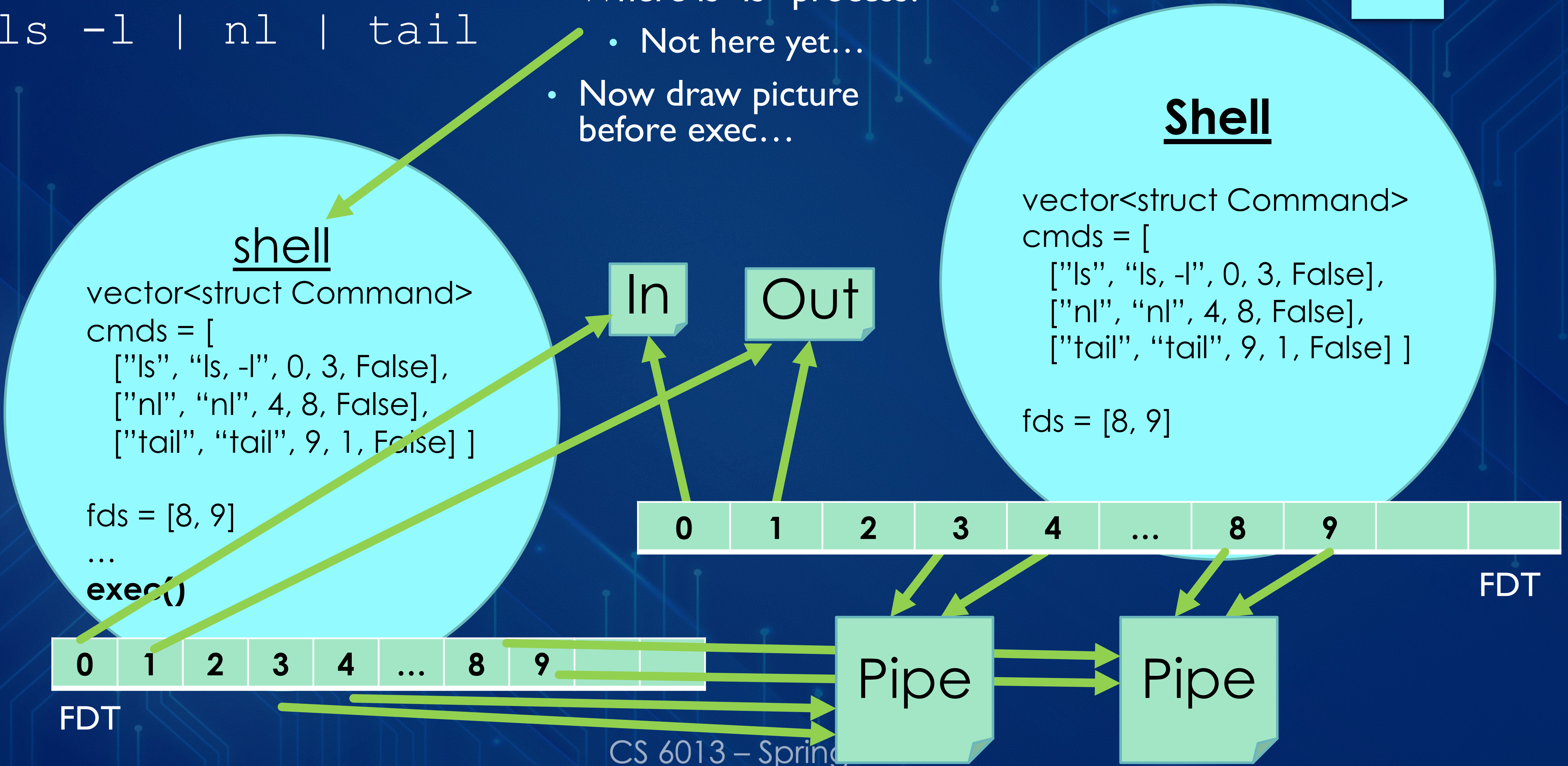
```
vector<struct Command> cmds = [
    ["ls", [ls, -l], 0, 3, False],
    ["nl", [nl], 4, 8, False],
    ["tail", [tail], 9, 1, False]
]

fds = [8, 9]
```



Unix Shell

`ls -l | nl | tail`

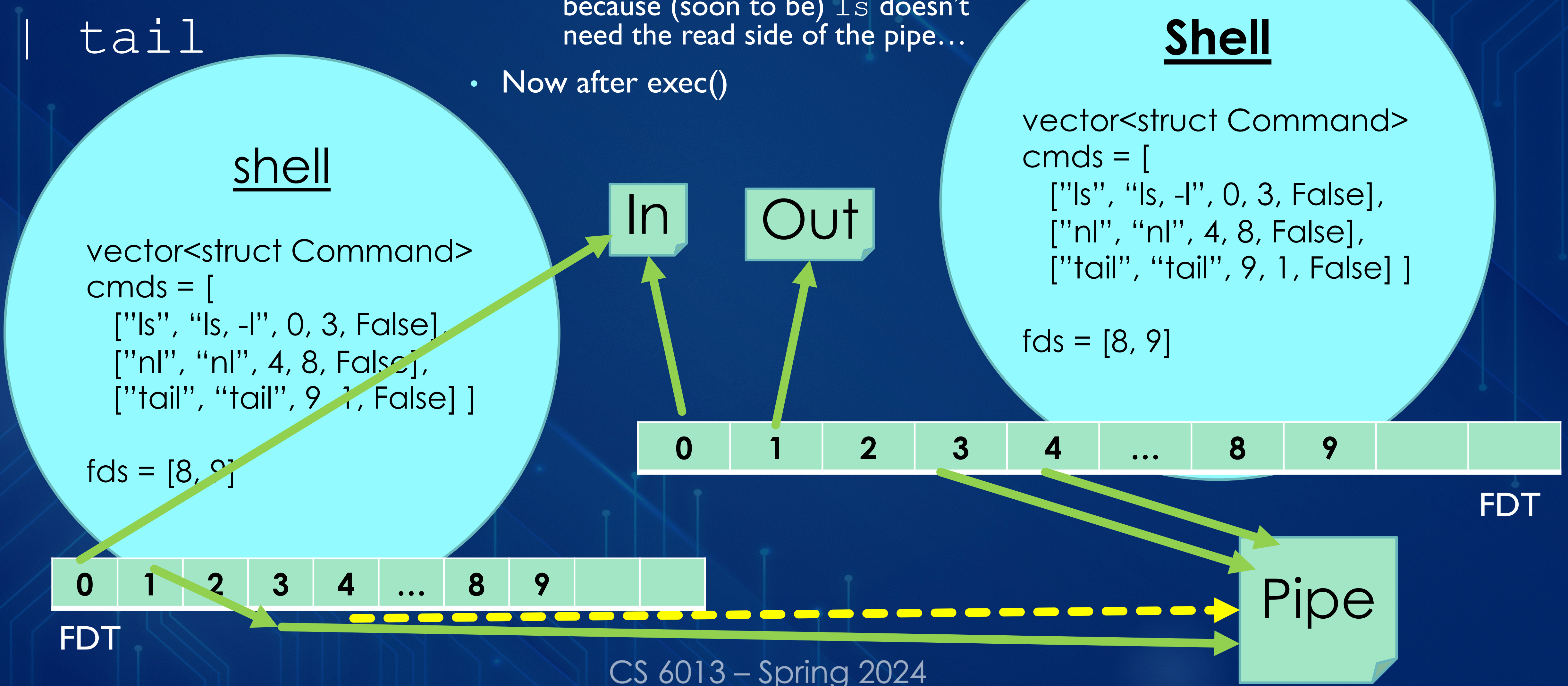
- After first fork():
- Where is “ls” process?
  - Not here yet…
- Now draw picture before exec…



Unix Shell

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

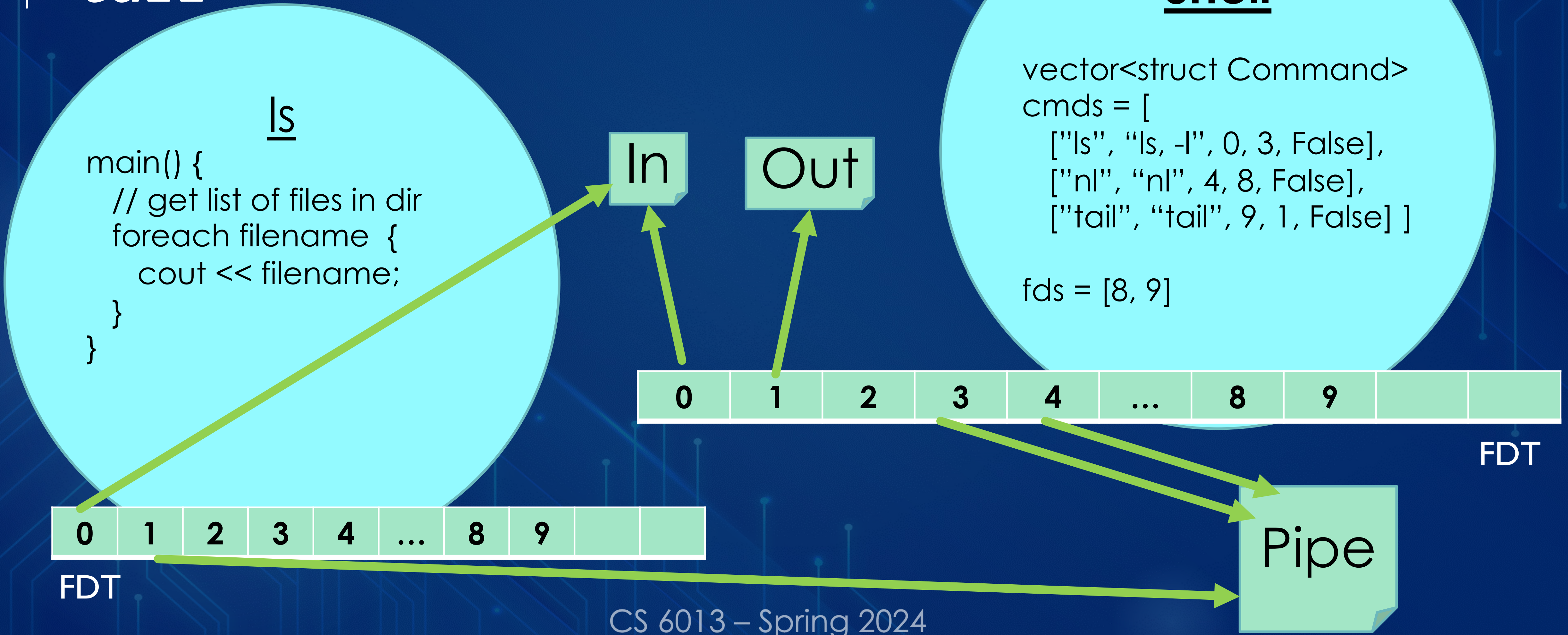
- Before exec()
  - Why / how is 1 pointing to 3?
    - 1 is "redirected" to 3 via dup2
  - FD #4 (in new <u>shell</u>) should be closed because (soon to be) `ls` doesn't need the read side of the pipe…
- Now after exec()



Unix Shell

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

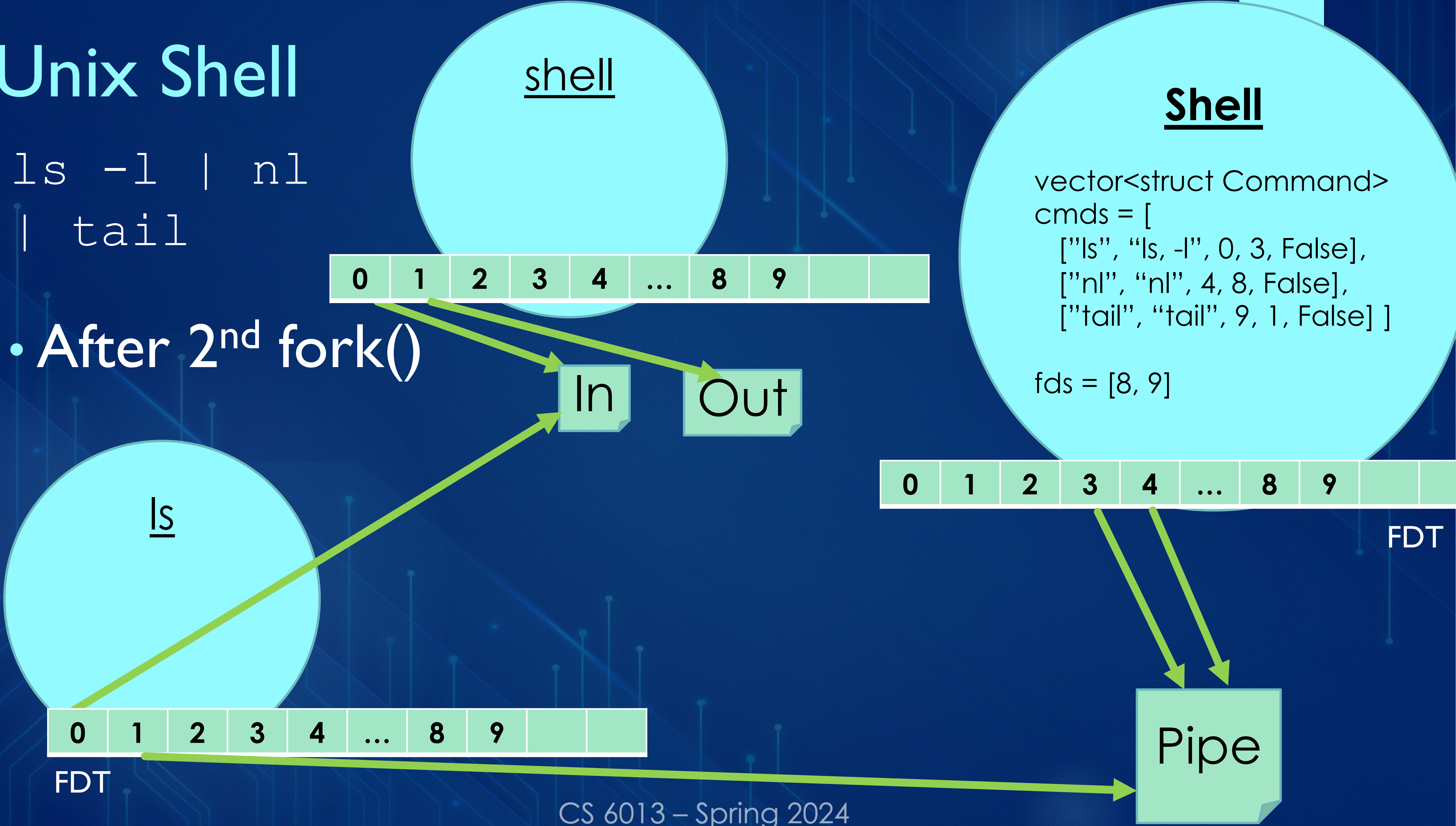
- After exec()
- Now what happens?



Unix Shell

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

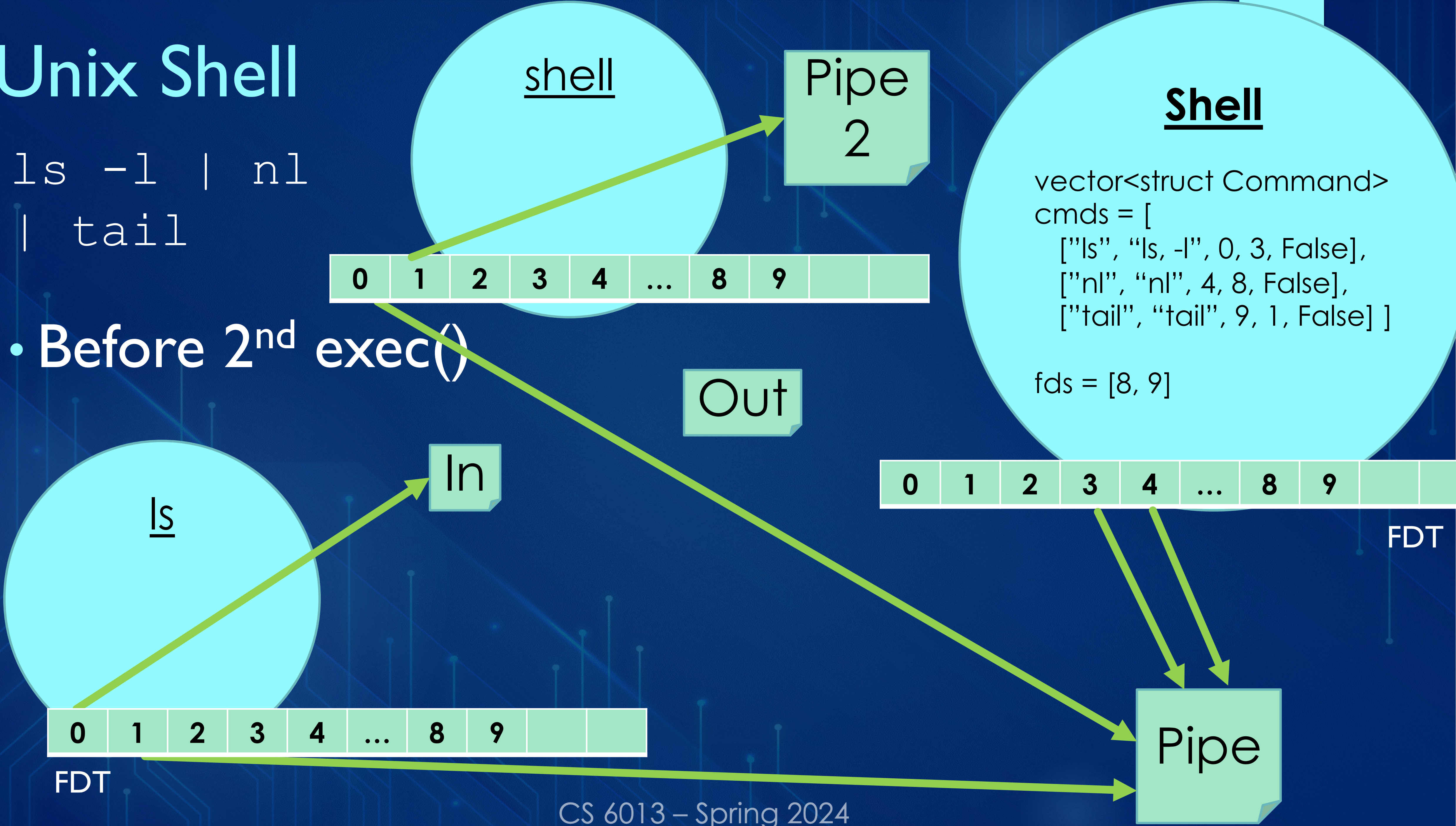
- After 2[nd] fork()



Unix Shell

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

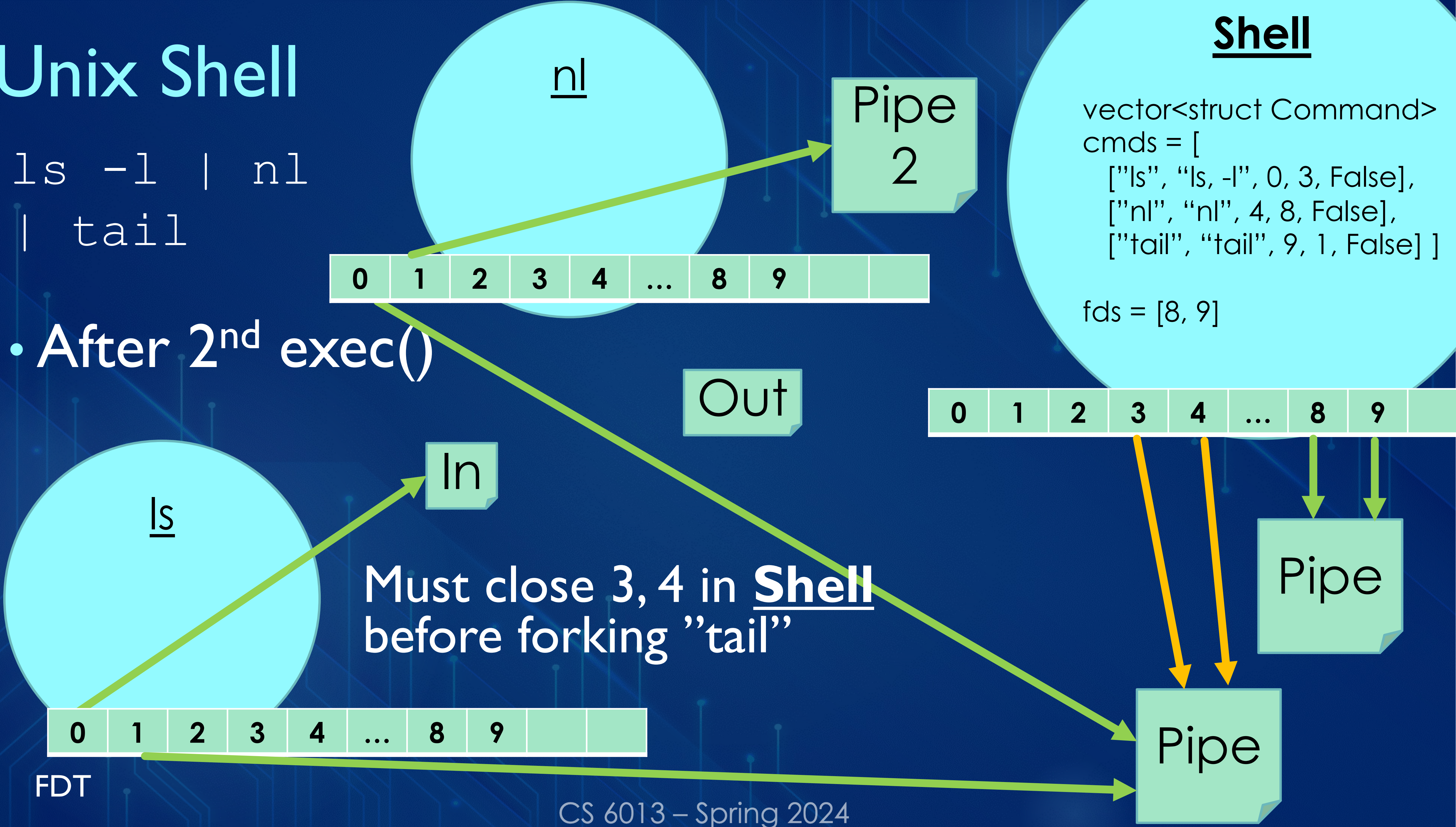
- Before 2nd exec()



Unix Shell

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

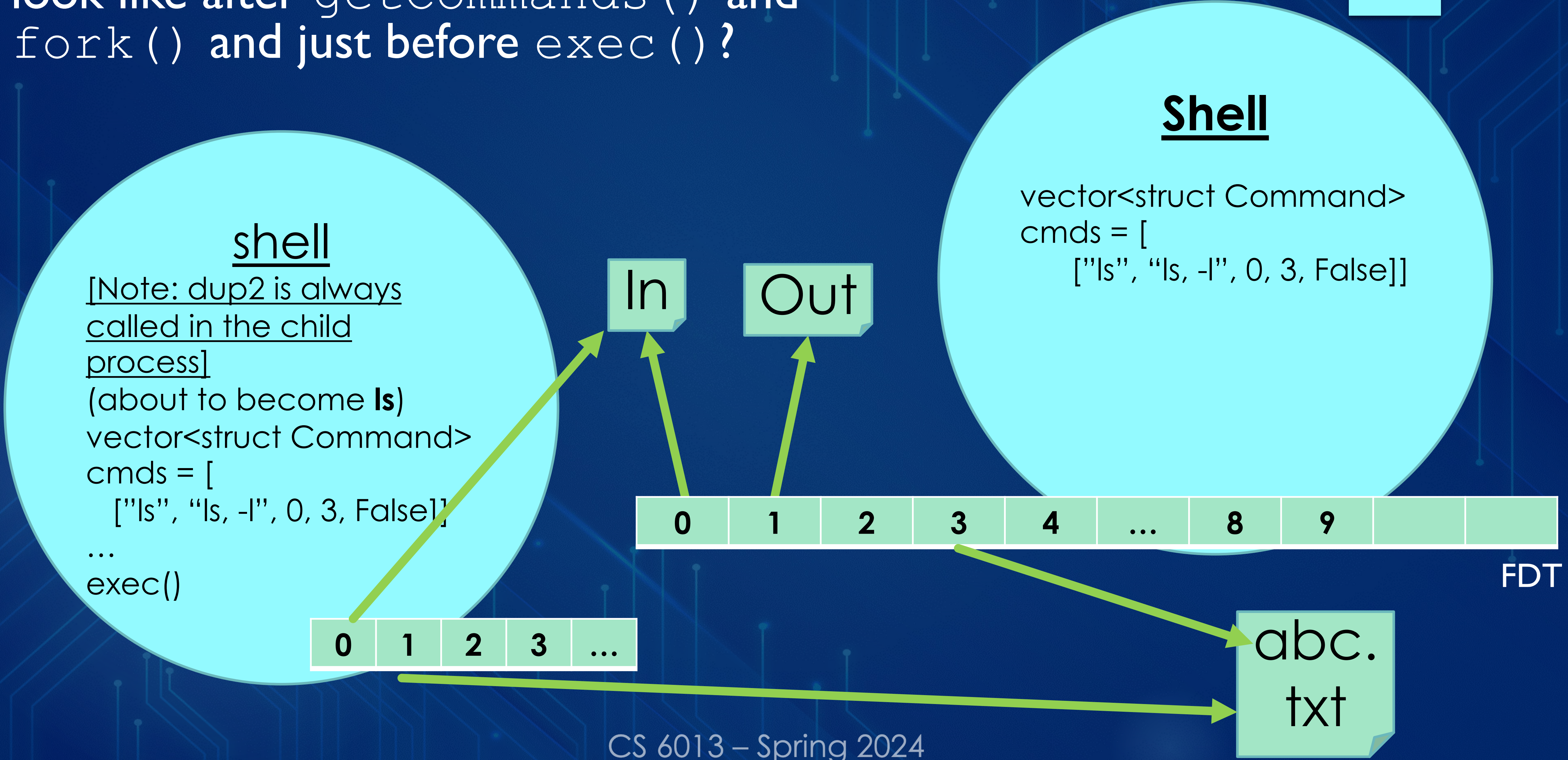
- After 2nd exec()



Must close 3, 4 in **Shell** before forking "tail"

Unix Shell

- What would "`ls -l > abc.txt`" look like after `getCommands()` and `fork()` and just before `exec()`?



Address Translation (v1)

- Static Relocation
  - At **load** (run) time, OS updates all addresses based on a base location.
  - Say +0x3000 (Max Process Memory Size)

Program [as written / compiled]

0x2fff — Stack

0x0abc — Heap

Virtual Addresses

0x0 — Code
int * intP = malloc()
[Library gives 0x0abc]

Process (**P4**) [when running]

0x5fff — Stack

0x3abc — Heap

Real* Addresses

0x3000 — Code
int * intP = malloc()
[Adjusted to **0x3abc**]

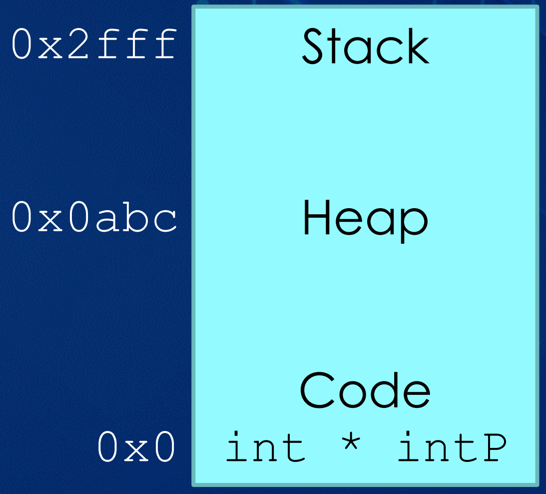
Physical Memory	
0x18000	**P2**
0x15000	**P3**
0x12000	
0x9000	
0x6000	**P1**
0x3000	**P4**
0x0	**OS**

- How is real memory arranged?
  - What's at 0x0?
    - OS
  - All other processes are slotted in.
- And P4?

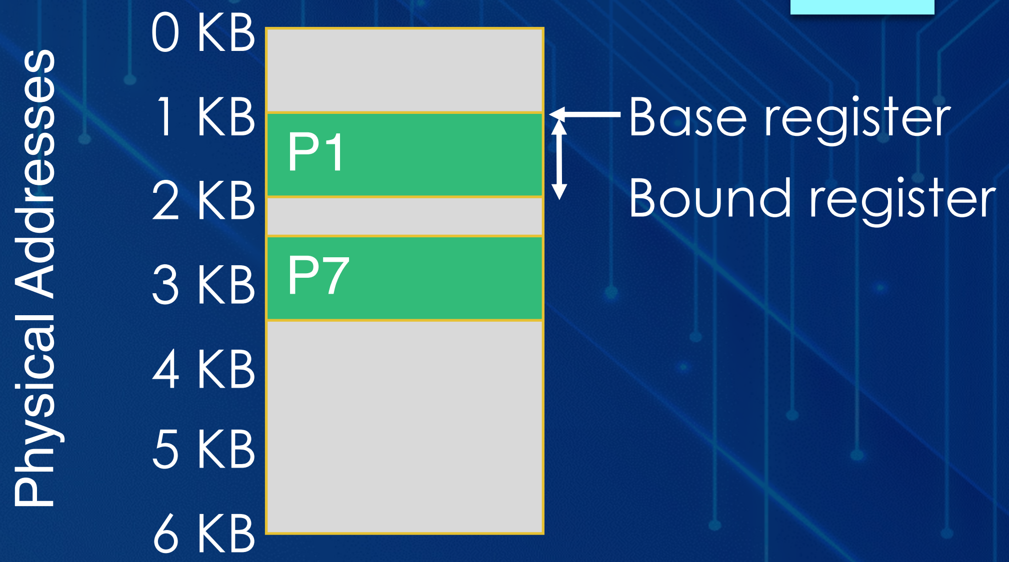
Base and Bound Approach (Translation v2)

- Fairly straightforward and easy to implement!
  - Similar to the Static Relocation – but done "on the fly"
- OS sets up address translation with privileged registers (cr3 and others) in CPU
  - Indicates where in physical memory the address space of the process starts (**base**)
  - And to what physical memory it extends (**bound**)
- All addresses are automatically translated by MMU
  - Hardware does the translation (ie: adds base (cr3) to every address)
- Of course, a user-mode process cannot change these registers
  - The OS changes the values of the base/bound register at context switch

Program's Virtual Address Space



Base & Bound Example

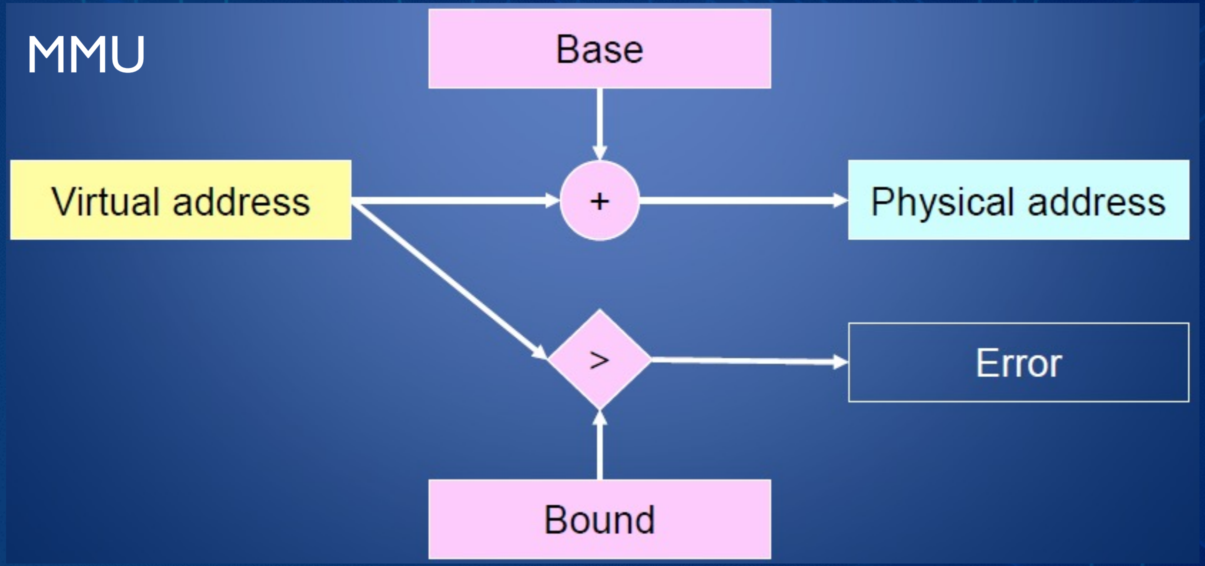


P1 is running

- Where are the base and bound values stored?
  - In registers (in the MMU – which is part of the CPU), which are only accessible by the OS (ie, when in kernel mode)

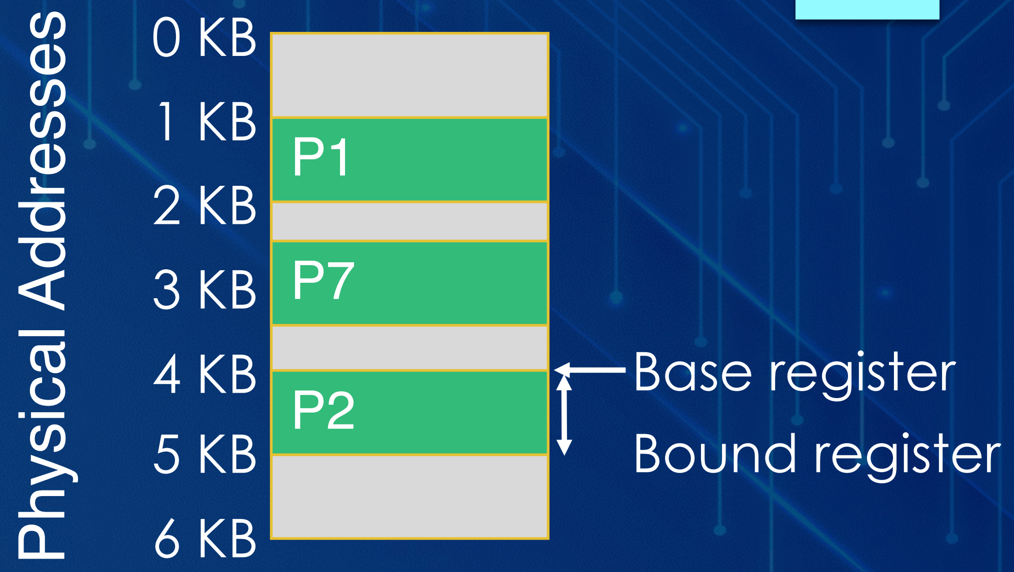
Base & Bound Implementation

- On <u>every</u> memory access of process
  - MMU compares logical address to bounds register
    - if logical address is greater (or *), then generate error
  - MMU adds base register to logical address to form physical address



* or less than base…

Base & Bound Example

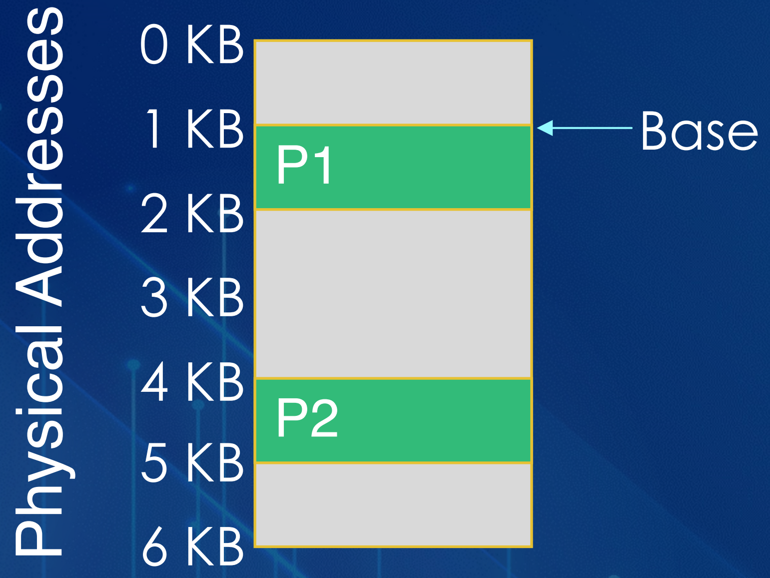


P2 starts running

- How (in general terms) does P2 to start running?
  - Context switch – which registers (in this situation) must be updated?
    - Base / Bound Registers (and of course, all the others)
- FYI, what does KB mean?
  - Kilobyte (1024 bytes)

Base & Bound Example

Can P1 access P2 memory?

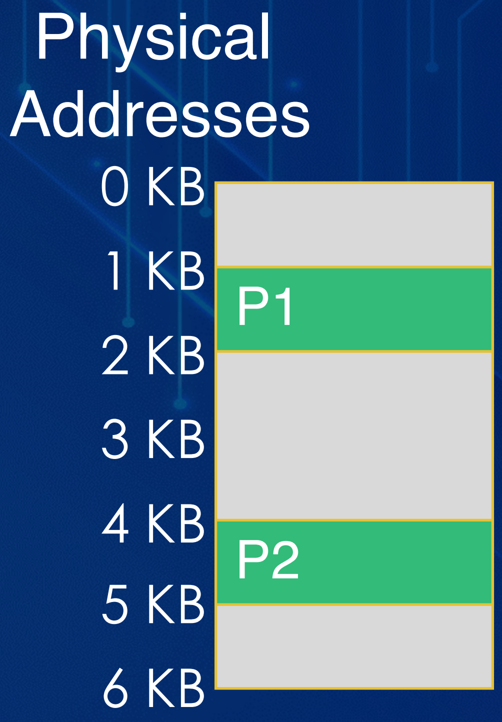


Virtual	Physical
`P1: load R1, [100]`	`load R1, [1124]`
`P2: load R1, [100]`	`load R1, [4196]`
`P2: load R1, [1000]`	`load R1, [5096]`
`P1: load R1, [1000]`	`load R1, [2024]`
`P1: load R1, [3072]`	Illegal bound error

- What happens to this virtual instruction?
  - Why did we add 1124?
  - This is the base register's value.
- What makes this happen?
  - MMU and base/bounds registers

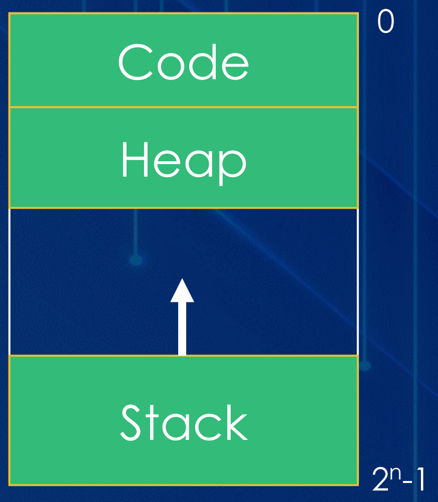
Base and Bounds Advantages

- Advantages
  - Protection (access/no access) across address spaces
  - Supports dynamic relocation* (see disadvantages)
  - Simple, inexpensive implementation
    - Only need a few registers and a little logic in MMU
  - Fast
    - Add and compare in parallel



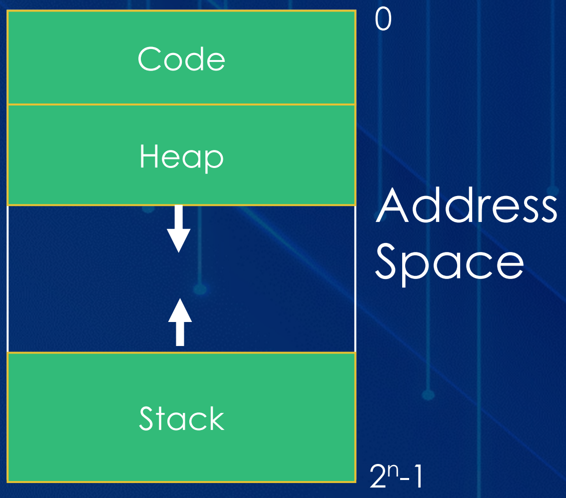
Base and Bounds Problems

- Disadvantages
  - Each process must be allocated contiguously in physical memory
    - Must allocate memory that may not be used by process (free space between stack and heap in Figure below)
      - Ie, "empty space" is mapped to physical memory – even though it isn't being used.
  - No partial sharing: cannot share limited parts of address space
- Not used because of its inflexibility… but is a good starting point for building a reasonable memory management system.



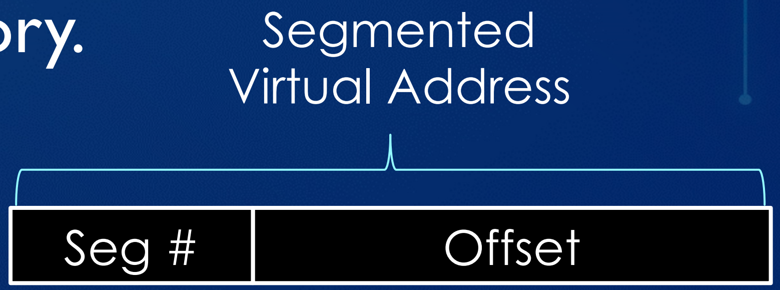
Segmentation

- Divide address space into logical **segments**
  - Each corresponds to logical entity in address space
  - Code, Stack, Heap
- Each segment can independently:
  - Be placed (separately) in physical memory
  - Grow and shrink
  - Be protected
    (separate read/write/execute protection bits)



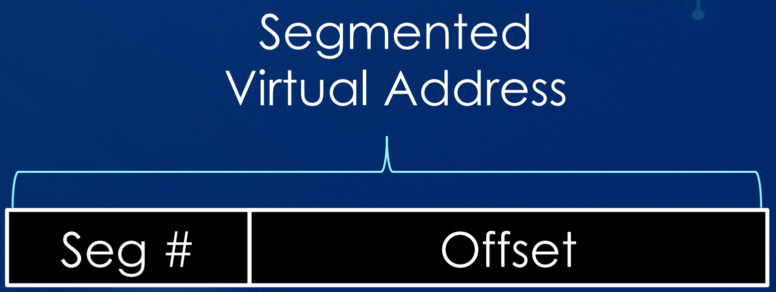
Segmented Addressing

- Process specifies segment and offset within segment
- How does process designate a particular segment?
  - Explicit: use part of virtual address
  - Top bits select segment (physical piece of real memory)
  - Low bits indicate offset within segment
- Each segment gets its own base and bound!
  - Base gives starting address for a segment.
  - Bound tells us the limit of that segment in memory.
- How many bits to use as the segment number?
  - Number of segments allowed.
  - Total size of virtual address space, etc

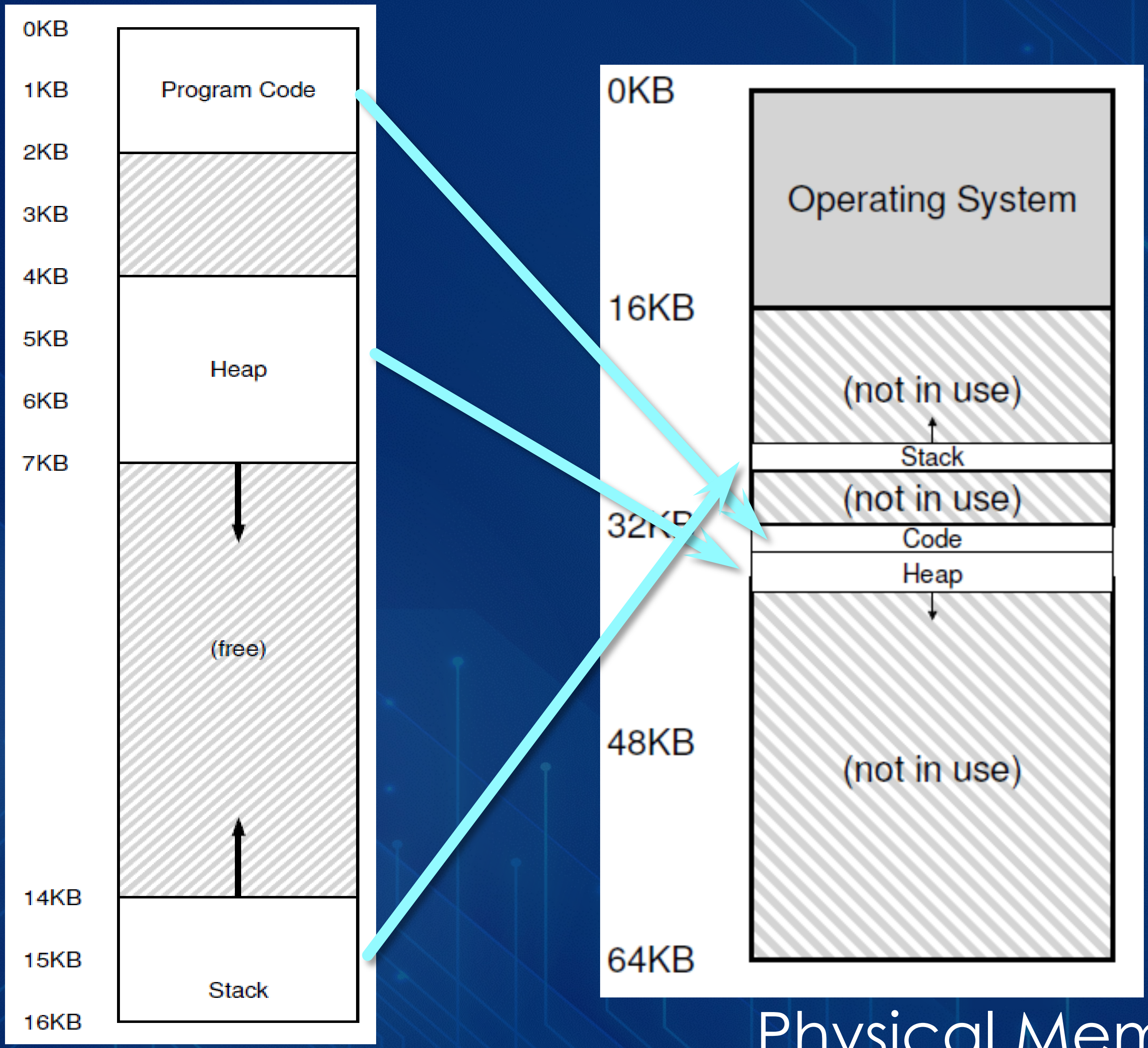


Segmented Address

- Assuming we are using 14-bit addresses…
- Amount of memory a program could use?
  - `16 K - 2^14`
- Now assume using Segmentation, and physical memory is divided into (only) 4 segments...
  - How many bits to represent the Seg #?
    - 2
  - How big can a segment of memory be?
    - `4 K — 2^12`



Examples



Virtual address space (Process 1).

Physical Memory

- Each segment on the left is mapped onto the real memory on the right.
- Notice OS is taking up some physical memory
- How much real memory does Process 1's Code segment use?
  - 2 KB of real memory – same as the virtual memory is uses

Example 1

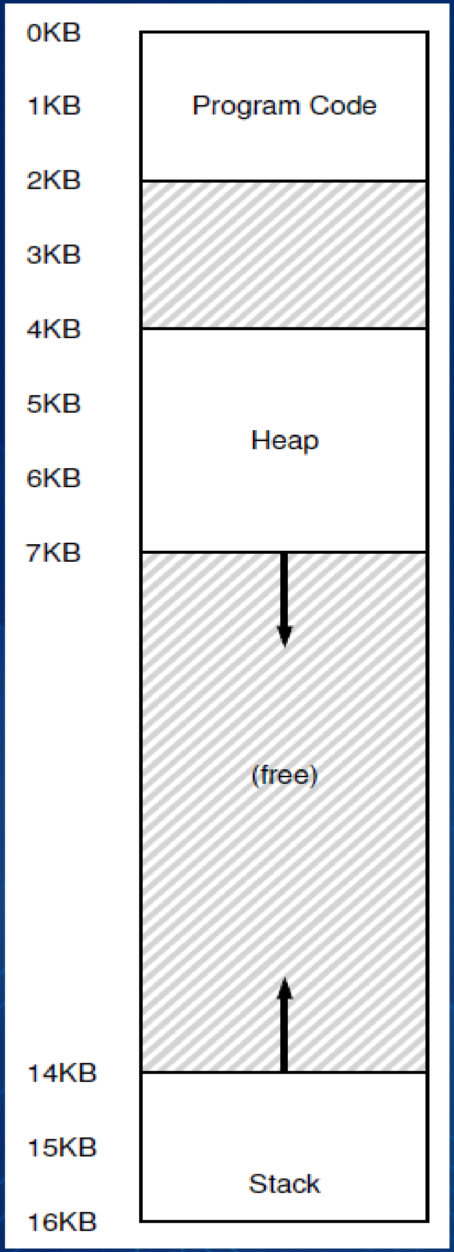
- Say code segment begins at 32KB in physical memory and is allotted 2KB.
- Consider the virtual address 100.
- What is the corresponding physical address?

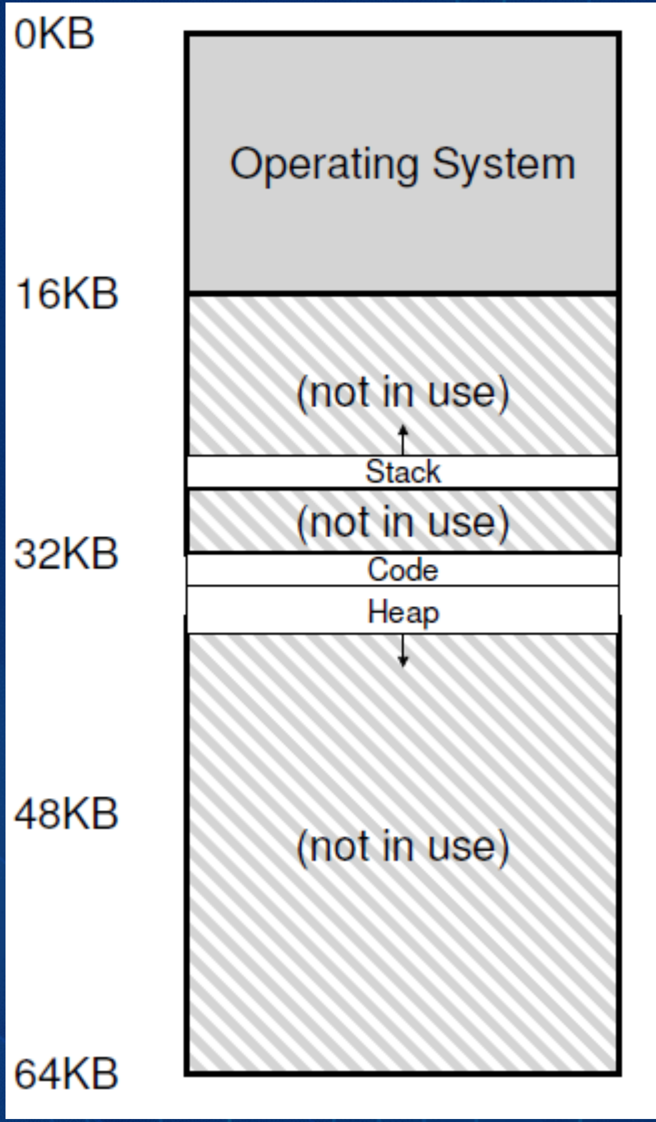
Example 1 - Answer

- Say code segment begins at 32KB in physical memory and is allotted 2KB.
- What is its physical address space?
  - 32KB to 34KB – 32768 to 34816
  - Note: 32 * 1 KB == 32 * 1024 == 32768
- What is physical address of virtual address 100?
  - Physical address = 32KB + 100 = 32868.
- Note: this is still within bounds of 32KB + 2KB.

Example 2

- Say heap segment starts at 34KB in physical memory, has size 3K.
- What is the **virtual** starting address of the heap segment?
  - Just looking at the figure on the left, we see the virtual starting address is 4096.
- What is the ending **physical** address of the heap?

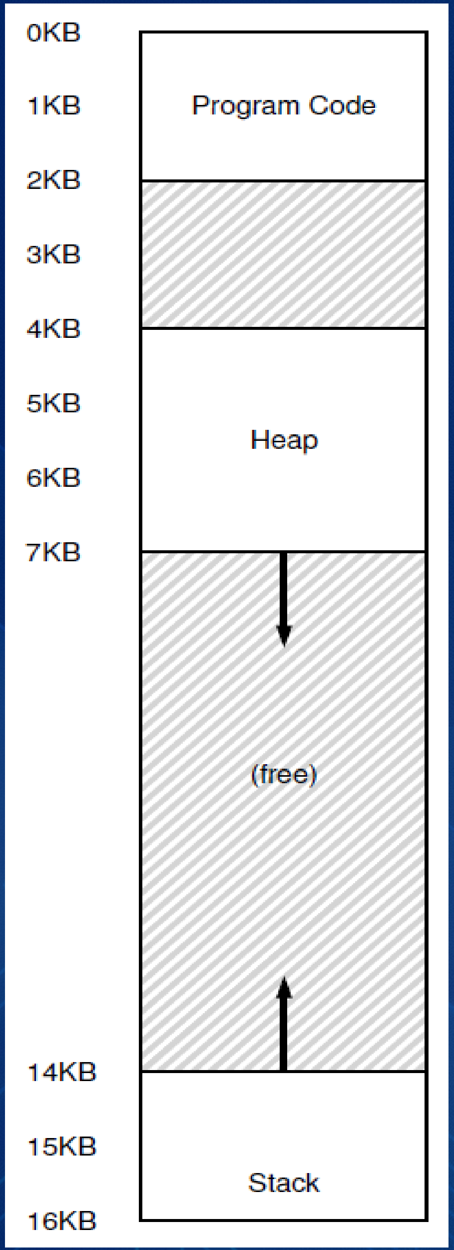




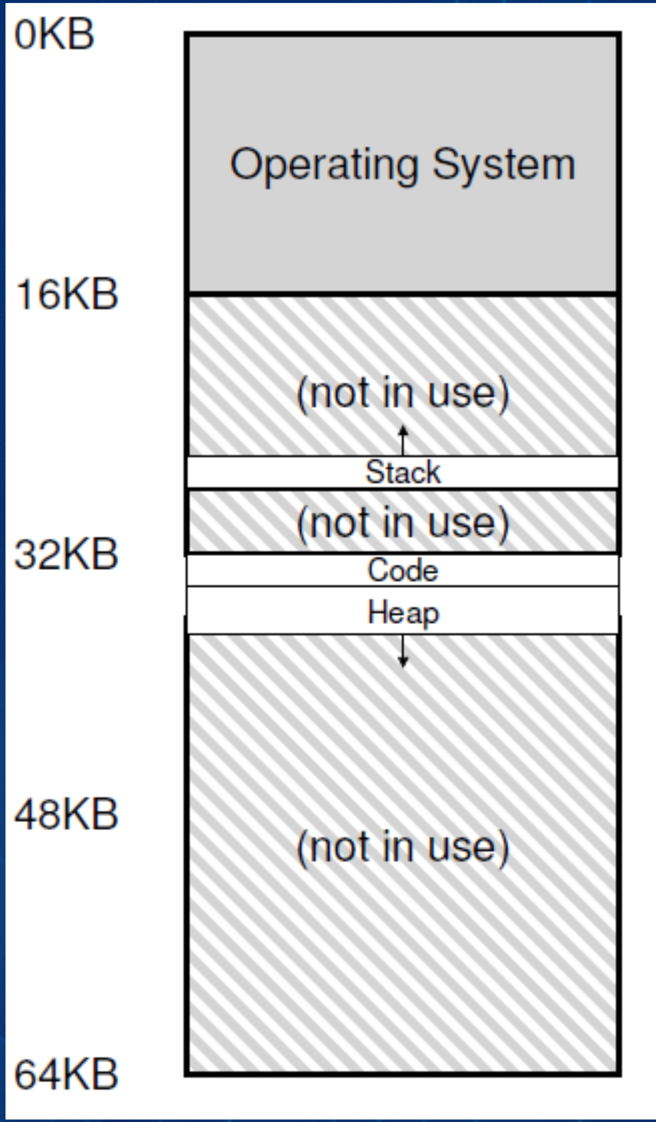
Virtual address space (Process 1).

Physical Memory

Example 3



Virtual address space (Process 1).



Physical Memory

- What is the physical address of virtual address 4200?
  - This one is a little trickier.
- First find the offset into the heap.
  - This is 4200 – (starting virtual address of heap).
  - That is, 4200 – 4096 = 104.
- Then add this offset to the base physical address of heap (34KB).
  - Result = 34KB + 104B = 34816 + 104 = 34920.

Segmentation Implementation

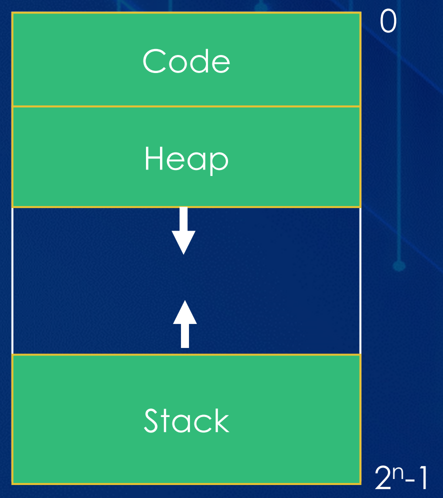
MMU has base/bound/permissions register per segment

- Example: 14-bit logical address, 4 segments
- How many bits for segment number?
- How many bits for offset?

Segment	Base	Bounds	R W
0	0x2000	0x6ff	1 0
1	0x0000	0x4ff	1 1
2	0x3000	0xfff	1 1
3	0x0000	0x000	0 0

Advantages of Segmentation

- Enables sparse allocation of address space
  - Stack and heap can grow independently
  - Heap: if no free memory, then sbrk()
- Protections for different segments
  - Read-only status for code
- Enables sharing of selected segments
- Supports dynamic relocation of each segment



Disadvantages of Segmentation

- External Fragmentation
  - Memory is full of little chunks of free space, but none of them are large enough for a single new segment.
  - Memory allocation requests may be denied.
- May still be really wasteful.
  - What if the heap is barely used, but it gets its own large segment?
  - Wasted space, contributes to fragmentation, and is not really fine-grained.

Conclusion

- HW+OS work together to virtualize memory
  - Give illusion of private address space to each process
- Add MMU registers for base+bounds so translation is fast
  - OS not involved with every address translation, only on context switch or errors
  - CPU has built-in hardware to do address mapping (translation)
- Dynamic relocation with segments is a good building block
  - Next lecture: Solve fragmentation with paging

~ Fin ~