Virtual Memory

15-213: Introduction to Computer Systems
Recitation 10: Nov. 2, 2015

Karthic Palaniappan

Agenda

- Shell Lab FAQs and I/O
- Malloc Lab Preview
- Virtual Memory Concepts
- Address Translation
  - Basic
  - TLB
  - Multilevel

Updates

- **Shell Lab is due Tuesday (tomorrow), 11:59 p.m.**
- **Malloc Lab is out Tuesday (tomorrow), 11:59 p.m.**
  - Due Thursday, Nov. 19
  - Start early!!
  - “The total time you spend designing and debugging can easily eclipse the time you spend coding.”

Shell Lab FAQ

- **"The traces behave differently from command-line input!"**
  - Some people are confused to find /bin/echo on their jobs list after running some trace files.
  - Some traces (e.g. trace05) print what they're running before they run them. They do this by using /bin/echo.
  - So if you see a mysterious /bin/echo show up on your jobs list, you shouldn't wonder *why it got on your jobs list*, you should wonder *why it never got deleted.*
  - Moral of the story: open the trace file and see what it does!

Shell Lab FAQ

- **Sigsuspend???**
  - You can only use waitpid() once, but there are probably two places you probably need to reap children (one for foreground jobs, one for background jobs).
  - Temptation: use waitpid() for background jobs; use `sleep()` or a tight loop (i.e., `while(1) {}`).
  - *Correct* solution: use sigsuspend to block your process until a signal arrives.
- **`int sigsuspend(const sigset_t *mask)`**
  - *Temporarily* replaces the process's signal mask with `mask`, which should be the signals you **don't** want to be interrupted by.
  - `sigsuspend` will return after an **un**blocked signal is received and its handler run. When it returns, it automatically reverts the process signal mask to its old value.

Shell Lab FAQ: sigsuspend example

```
int main() {
    sigset_t waitmask, newmask, oldmask;

    /* set waitmask with everything except SIGINT */
    sigfillset(&waitmask);
    sigdelset(&waitmask, SIGINT);

    /* set newmask with only SIGINT */
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGINT);

    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0) //oldmask now stores prev mask
           unix_error("SIG_BLOCK error");
    /* CRITICAL REGION OF CODE (SIGINT blocked) */
    /* pause, allowing ONLY SIGINT */
    if (sigsuspend(&waitmask) != -1)
           unix_error("sigsuspend error");

    /* RETURN FROM SIGSUSPEND (returns to signal state from before sigsuspend) */
    /* Reset signal mask which unblocks SIGINT */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
           unix_error("SIG_SETMASK error");
}
```

System Calls and Error Handling

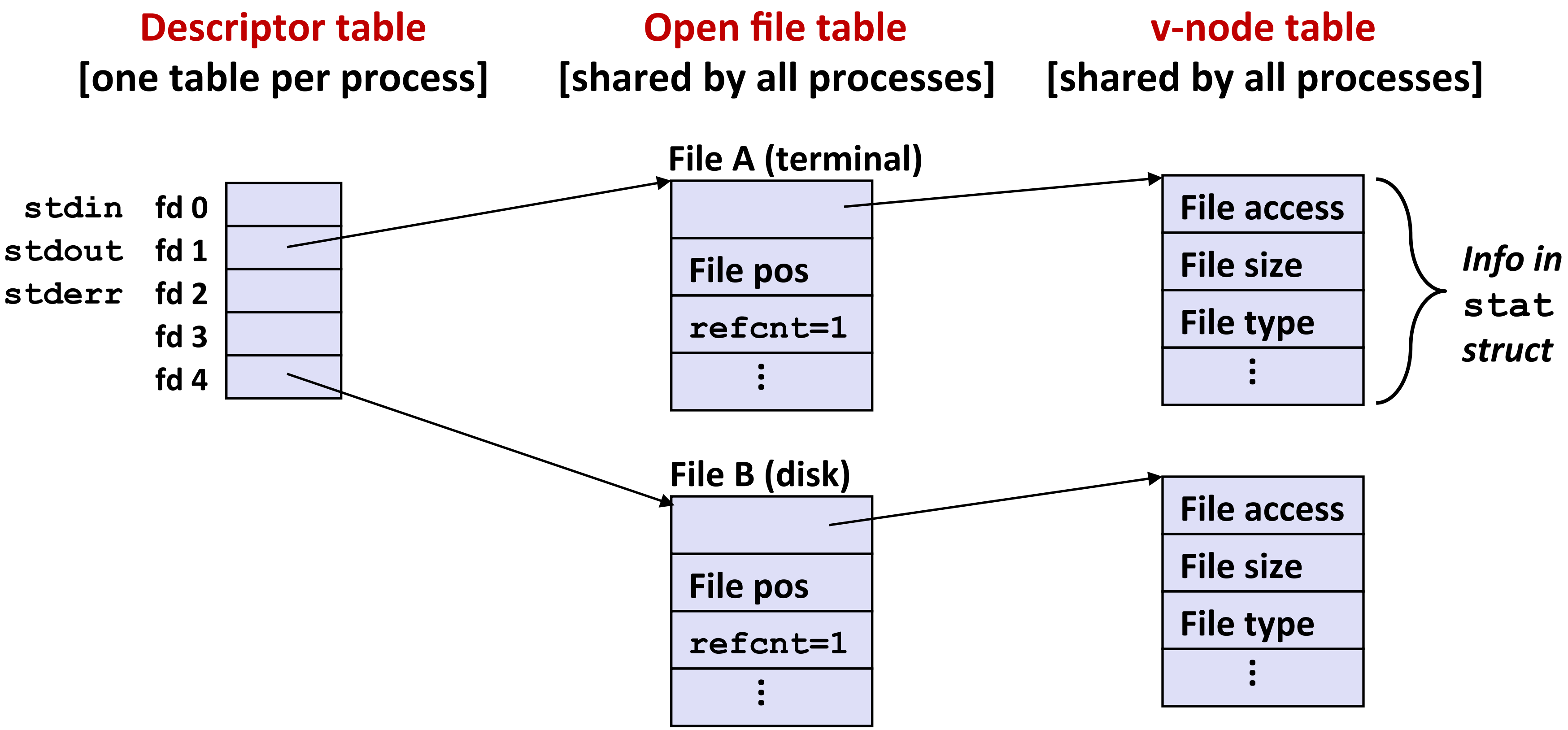
- **System Call Error Handling**
- **Always handle errors for every system call – #include <errno.h>**
  - Failed system calls almost always return -1
  - Global integer error number: errno
  - Getting error description: strerror(errno)
- **We deduct style points for not handling system call errors**
- **Do not lose style points here!**
- **Easy solution : Use wrappers from CSAPP website (Fork(),Execve(),Sigprocmask()…)**

I/O Basics

- **Four basic operations:**
  - open
  - close
  - read
  - write
- **What's a file descriptor?**
  - Returned by open.
  - int fd = open("/path/to/file", O_RDONLY);
  - fd is some positive value or -1 to denote error
- **Every process starts with 3 open file descriptors that can be accessed macros like STDOUT_FILENO**
  - 0 - STDIN
  - 1 - STDOUT
  - 2 - STDERR

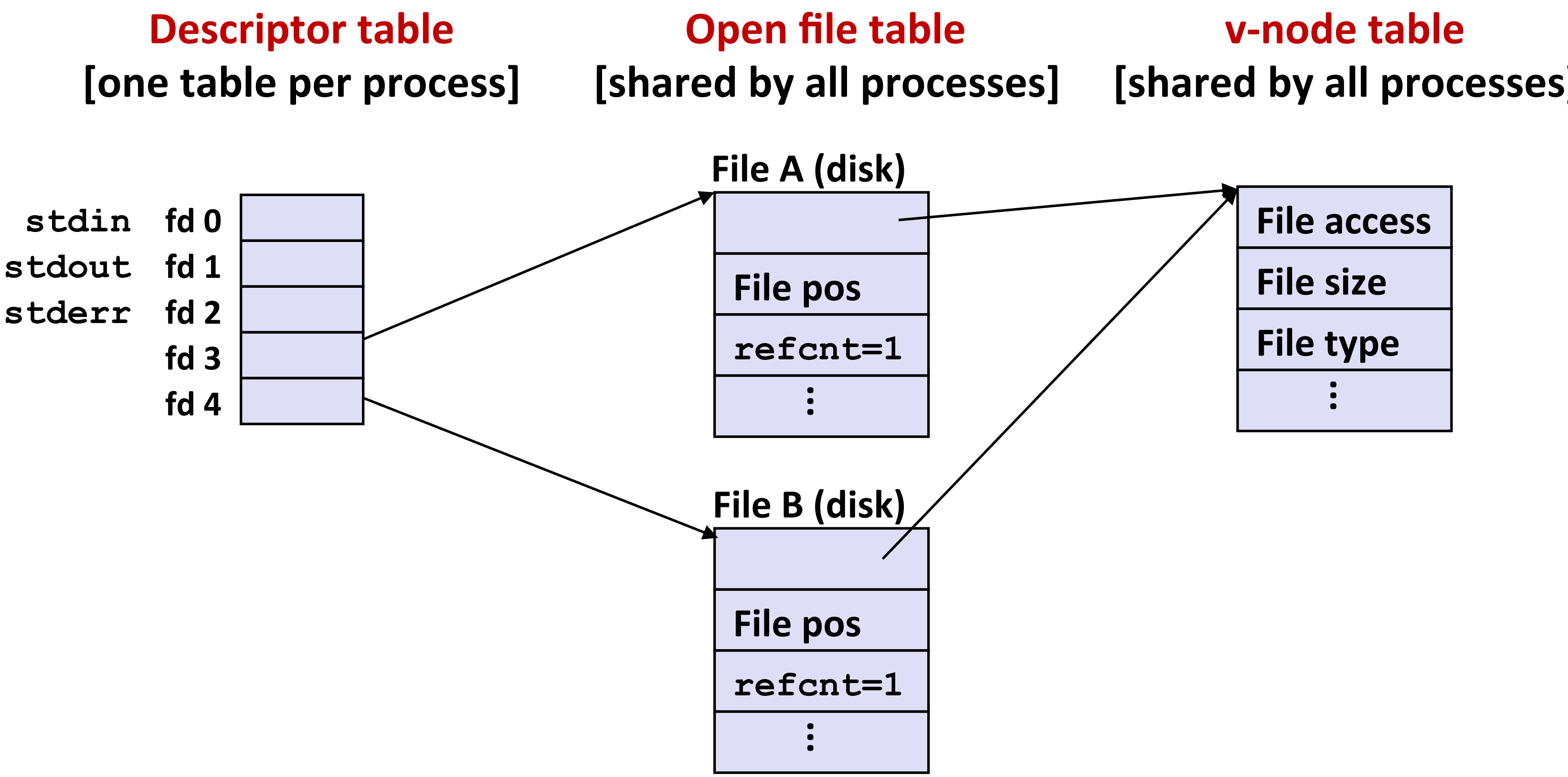
How the Unix Kernel Represents Open Files

- Two descriptors referencing two distinct open files. Descriptor 1 (stdout) points to terminal, and descriptor 4 points to open disk file



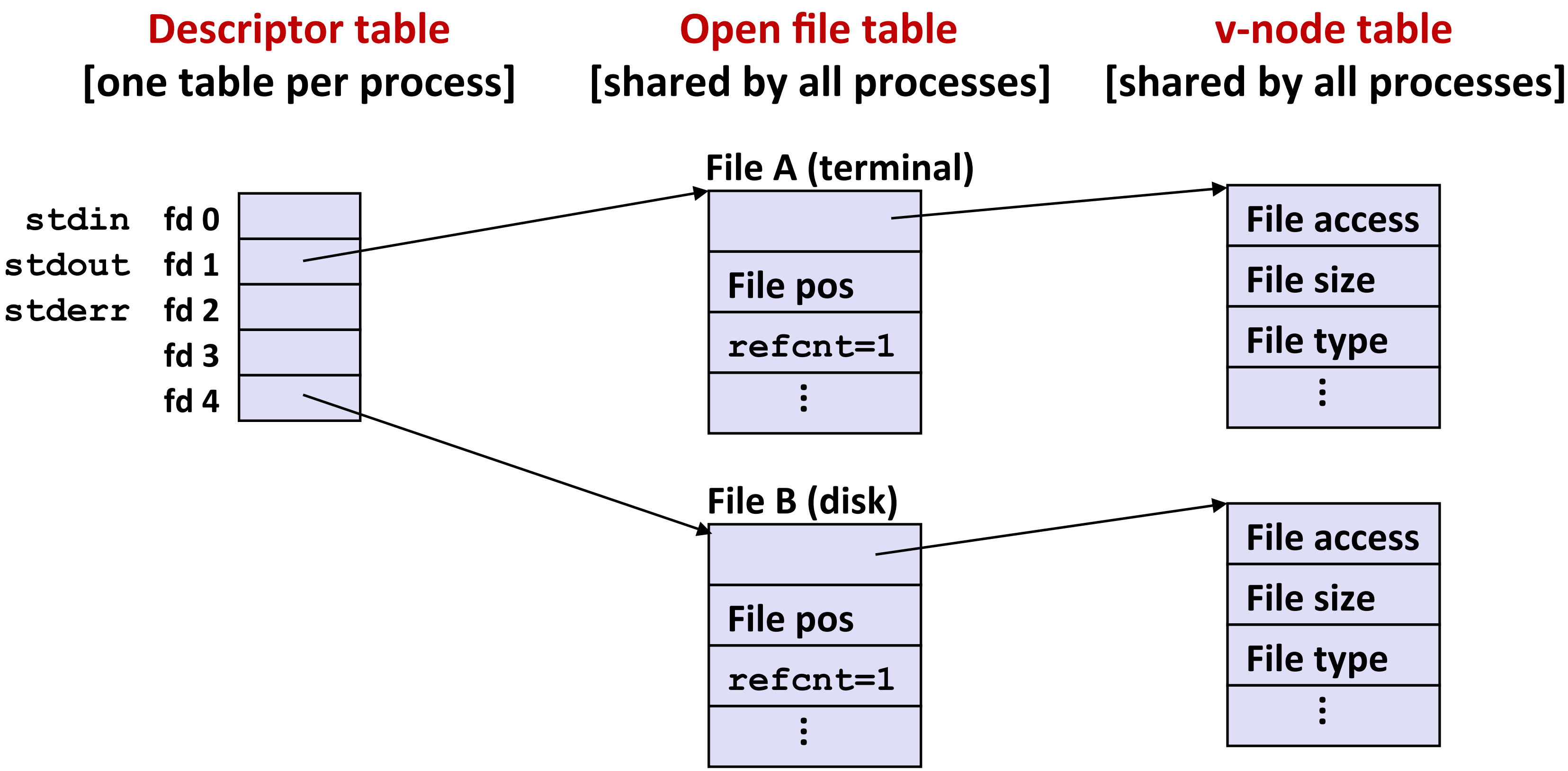
File Sharing

- **Two distinct descriptors sharing the same disk file through two distinct open file table entries**
  - E.g., Calling **`open`** twice with the same **`filename`** argument



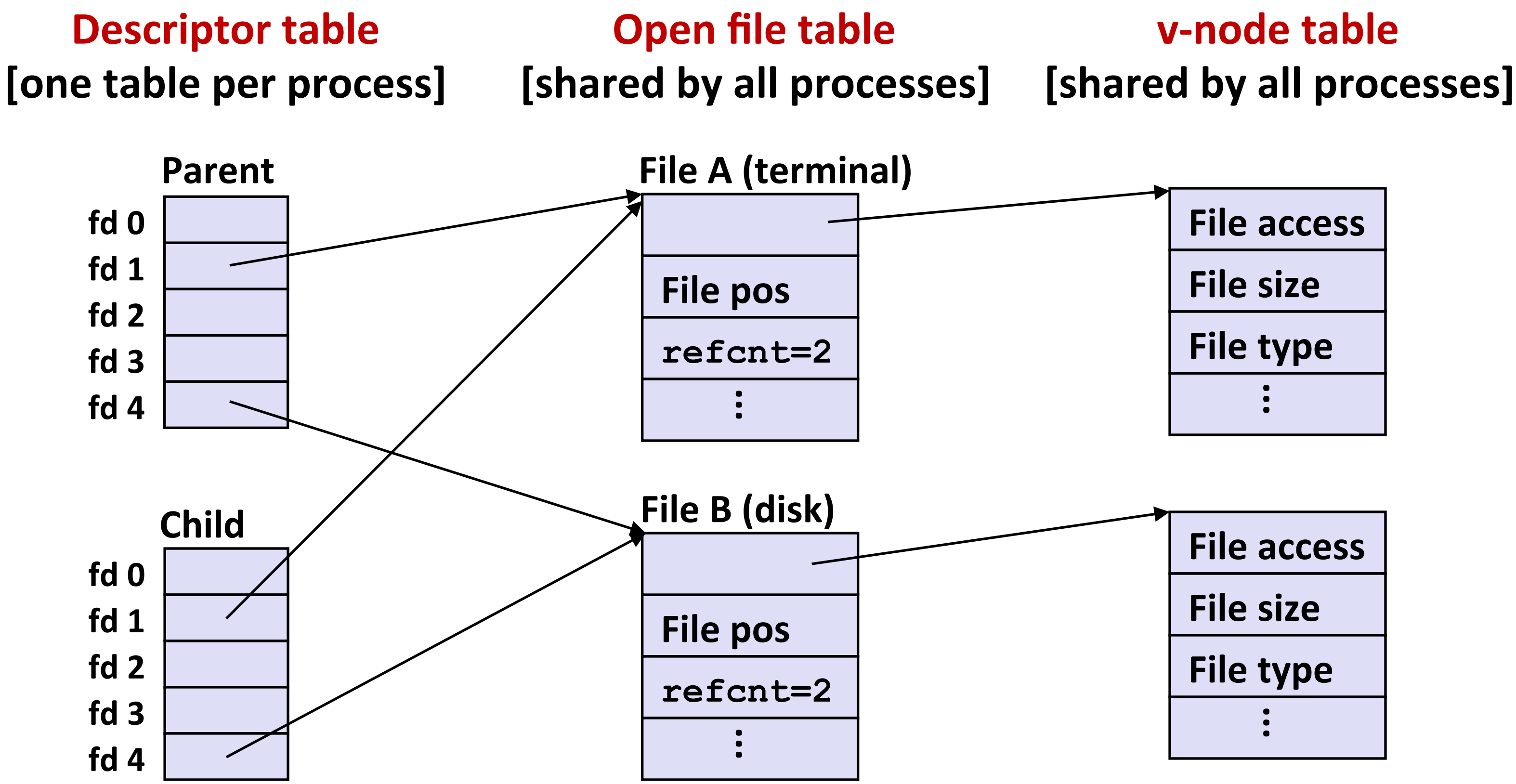
How Processes Share Files: `fork`

- A child process inherits its parent's open files
  - Note: situation unchanged by **`exec`** functions (use **`fcntl`** to change)
- *Before* **`fork`** call:



How Processes Share Files: `fork`

- **A child process inherits its parent's open files**
- ***After* `fork`:**
  - Child's table same as parent's, and +1 to each refcnt



I/O Redirection

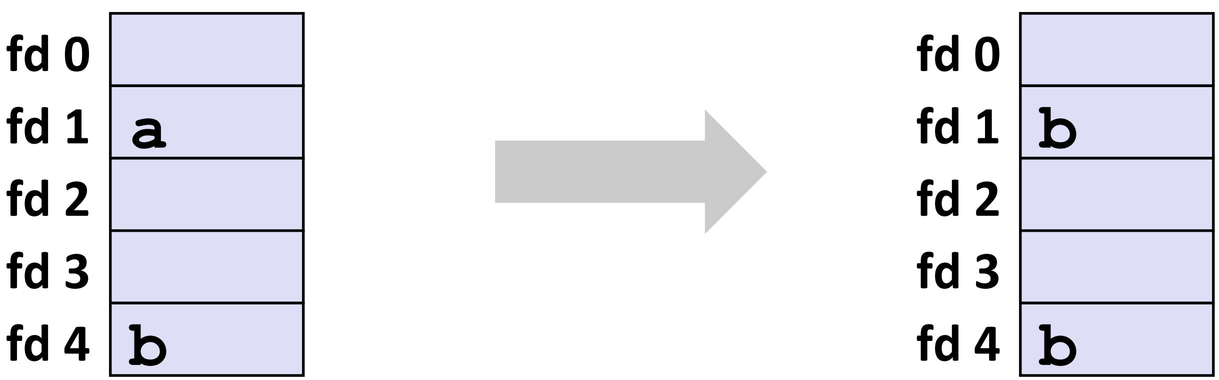
- **Question: How does a shell implement I/O redirection?**

  ```
  linux> ls > foo.txt
  ```

- **Answer: By calling the `dup2(oldfd, newfd)` function**
  - Copies (per-process) descriptor table entry **`oldfd`** to entry **`newfd`**

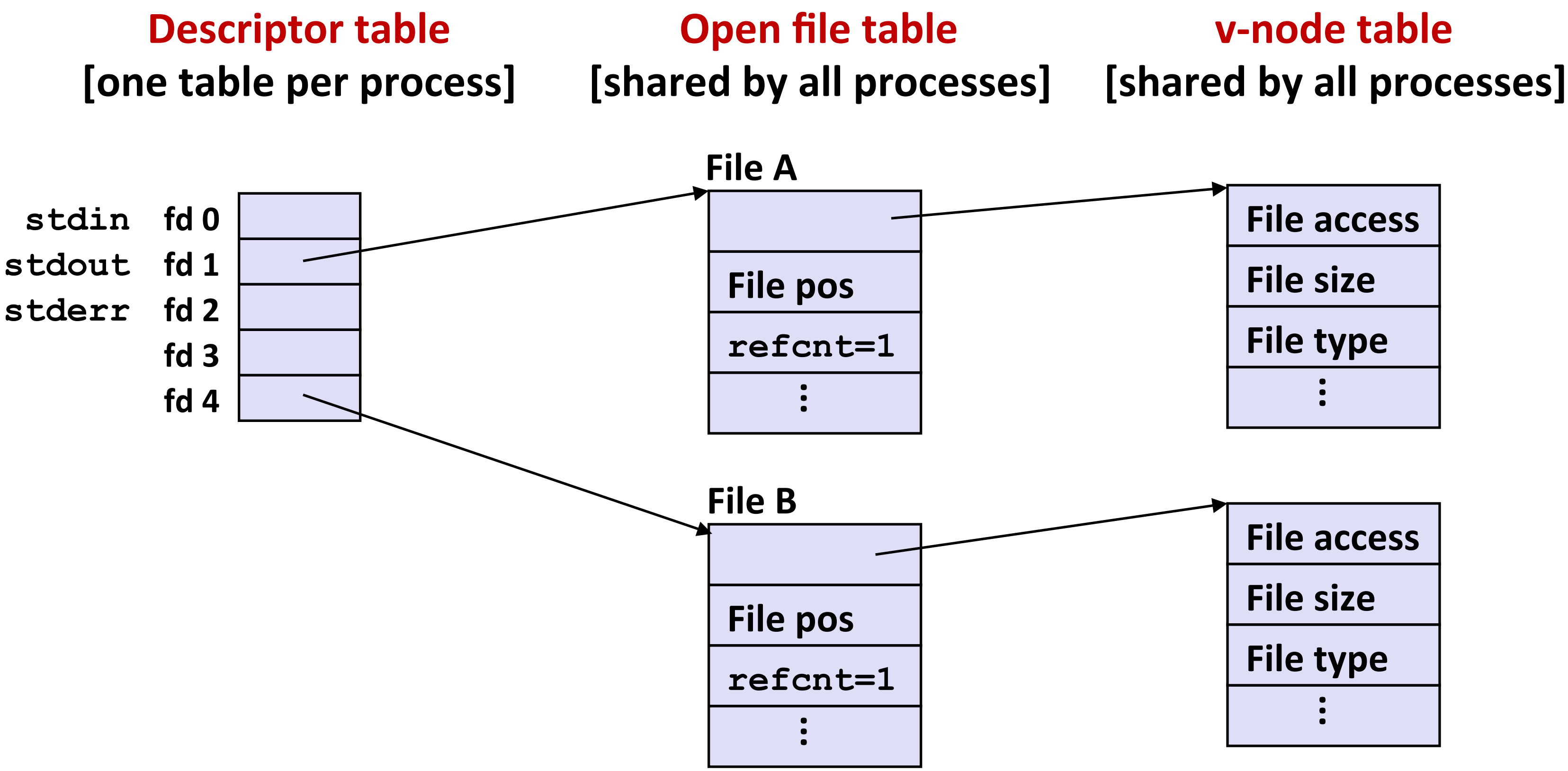
Descriptor table *before* `dup2(4,1)`

Descriptor table *after* `dup2(4,1)`



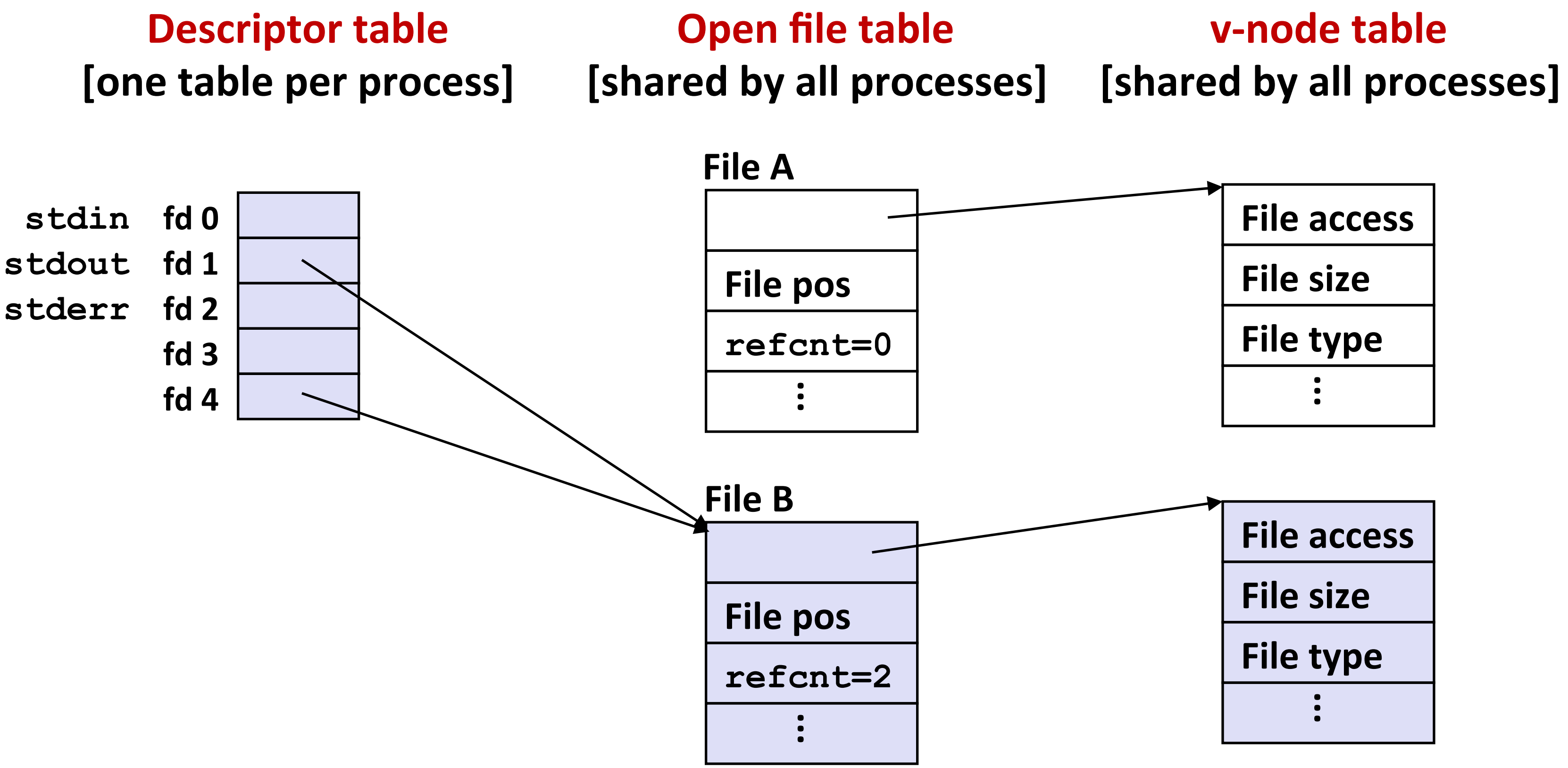
I/O Redirection Example

- **Step #1: open file to which stdout should be redirected**
  - Happens in child executing shell code, before **`exec`**



I/O Redirection Example (cont.)

- **Step #2: call `dup2(4,1)`**
  - cause fd=1 (stdout) to refer to disk file pointed at by fd=4



Malloc Lab Sneak Preview

- **You will write your own dynamic storage allocator – i.e., your own `malloc`, `free`, `realloc`, `calloc`.**
- **This week in class, you will learn about different ways to keep track of free and allocated blocks of memory.**
  - Implicit linked list of blocks.
  - Explicit linked list of *free* blocks.
  - Segregated lists of different *size* free blocks.
- **Other design decisions:**
  - How will you look for free blocks? (First fit, next fit, best fit…)
  - Should the linked lists be doubly linked?
  - When do you coalesce blocks?
- **This is exactly what you'll do in this lab, so pay lots of attention in class. ☺**

Malloc Lab Sneak Preview

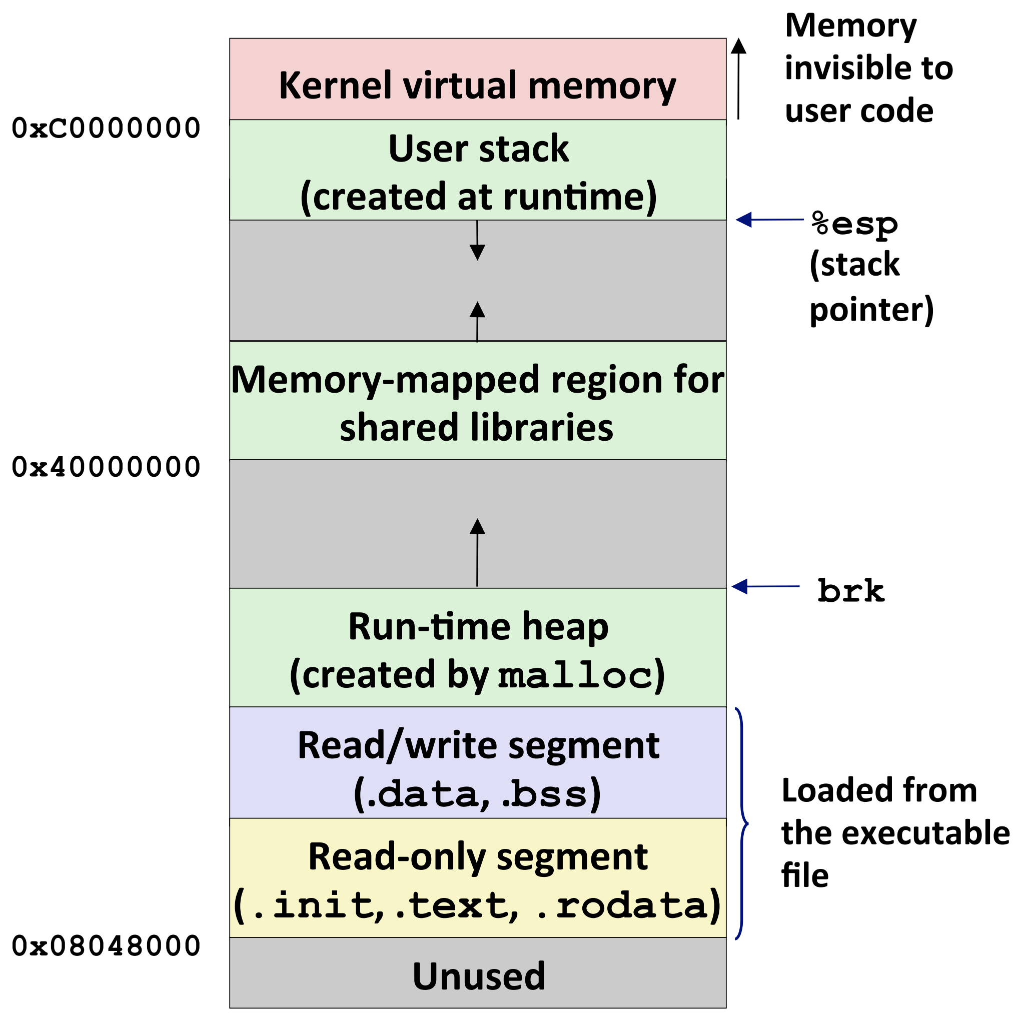
- **If you haven't been using version control so far, this is a good time to start.**
- **Workflow:**
  - Implement indirect linked lists. Make sure it works.
  - Implement explicit linked lists. Make sure it still works.
  - Implement segregated lists. Make sure it still works.
  - You WILL break things and need to revert.
- **Barebones guide to using git on the Shark Machines:**
  - `git init` starts a local repository.
  - `git add foo.c` adds foo.c to that repository.
  - `git commit -a –m 'Describe changes here'` updates your repository with the current state of all files you've added.

Agenda

- Shell Lab FAQs
- Malloc Lab Sneak Preview
- **Virtual Memory Concepts**
- Address Translation
  - Basic
  - TLB
  - Multilevel

Virtual Memory Concepts

- We've been viewing memory as a linear array.
- But wait! If you're running 5 processes with stacks at `0xC0000000`, don't their addresses conflict?
- Nope! Each process has its own address space.
- How???



Virtual memory concepts

- We define a mapping from the **virtual** address used by the process to the actual **physical** address of the data in memory.

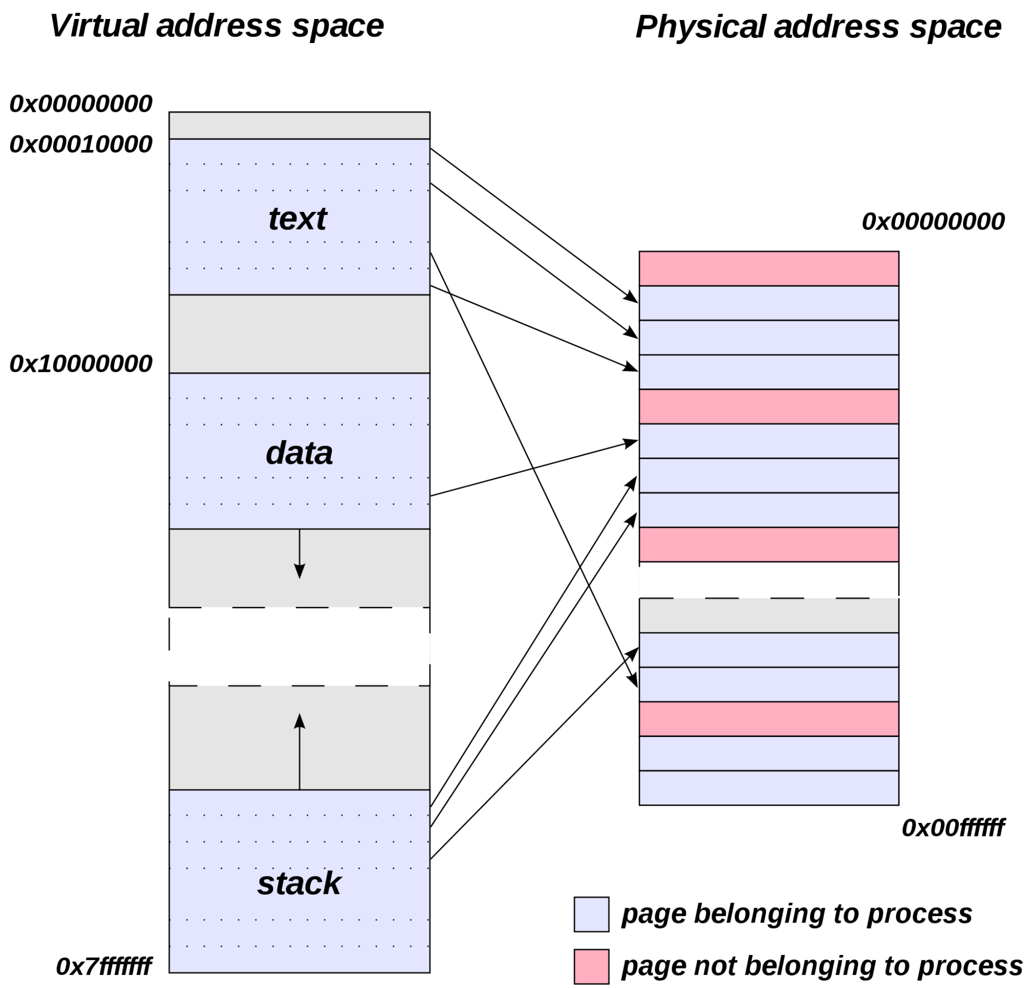
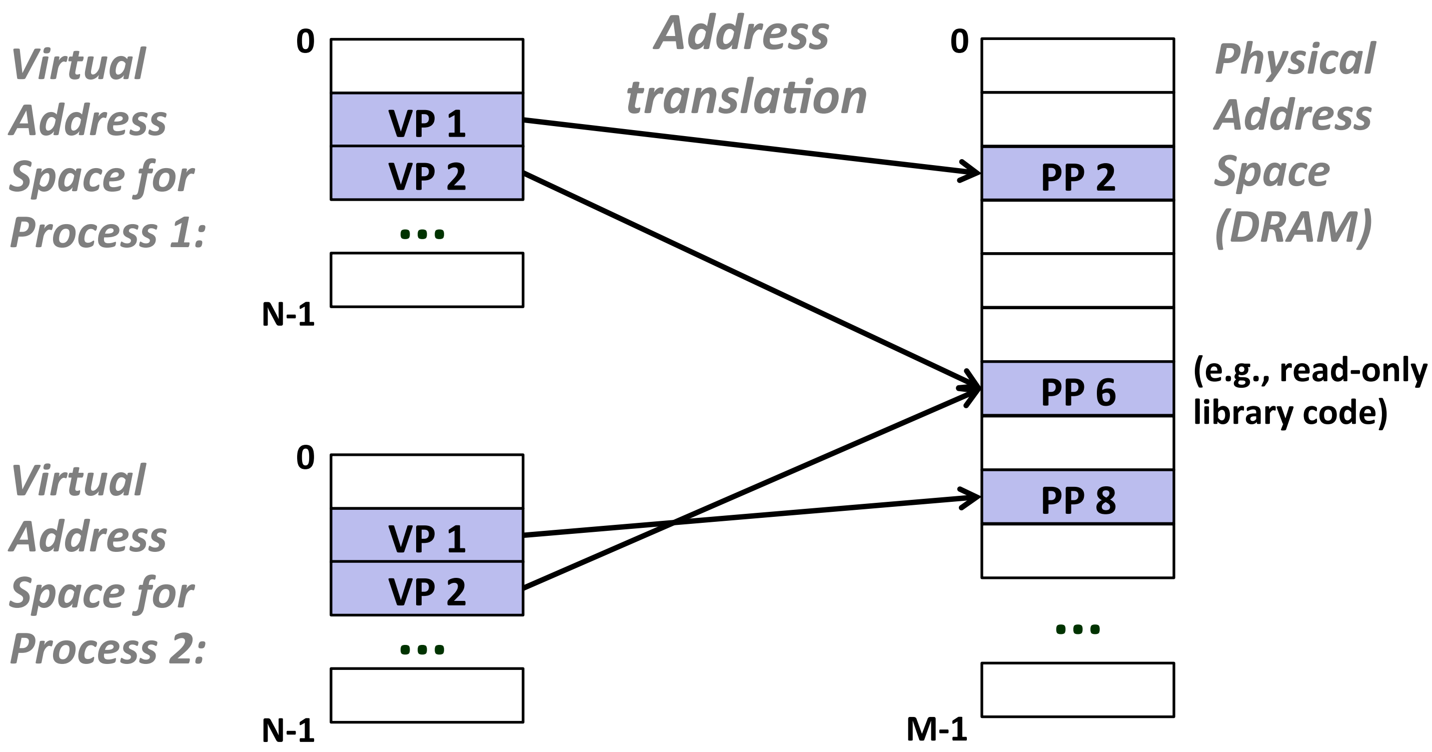


Image: http://en.wikipedia.org/wiki/File:Virtual_address_space_and_physical_address_space_relationship.svg

Virtual memory concepts

This explains why two different processes can use the same address. It also lets them share data *and* protects their data from illegal accesses. Hooray for virtual memory!



Virtual memory concepts

- **Page table**
  - Lets us look up the physical address corresponding to any virtual address. (Array of physical addresses, indexed by virtual address.)
- **TLB (Translation Lookaside Buffer)**
  - A special tiny cache just for page table entries.
  - Speeds up translation.
- **Multi-level page tables**
  - The address space is often sparse.
  - Use page directory to map large chunks of memory to a page table.
  - Mark large unmapped regions as non-present in page directory instead of storing page tables full of invalid entries.

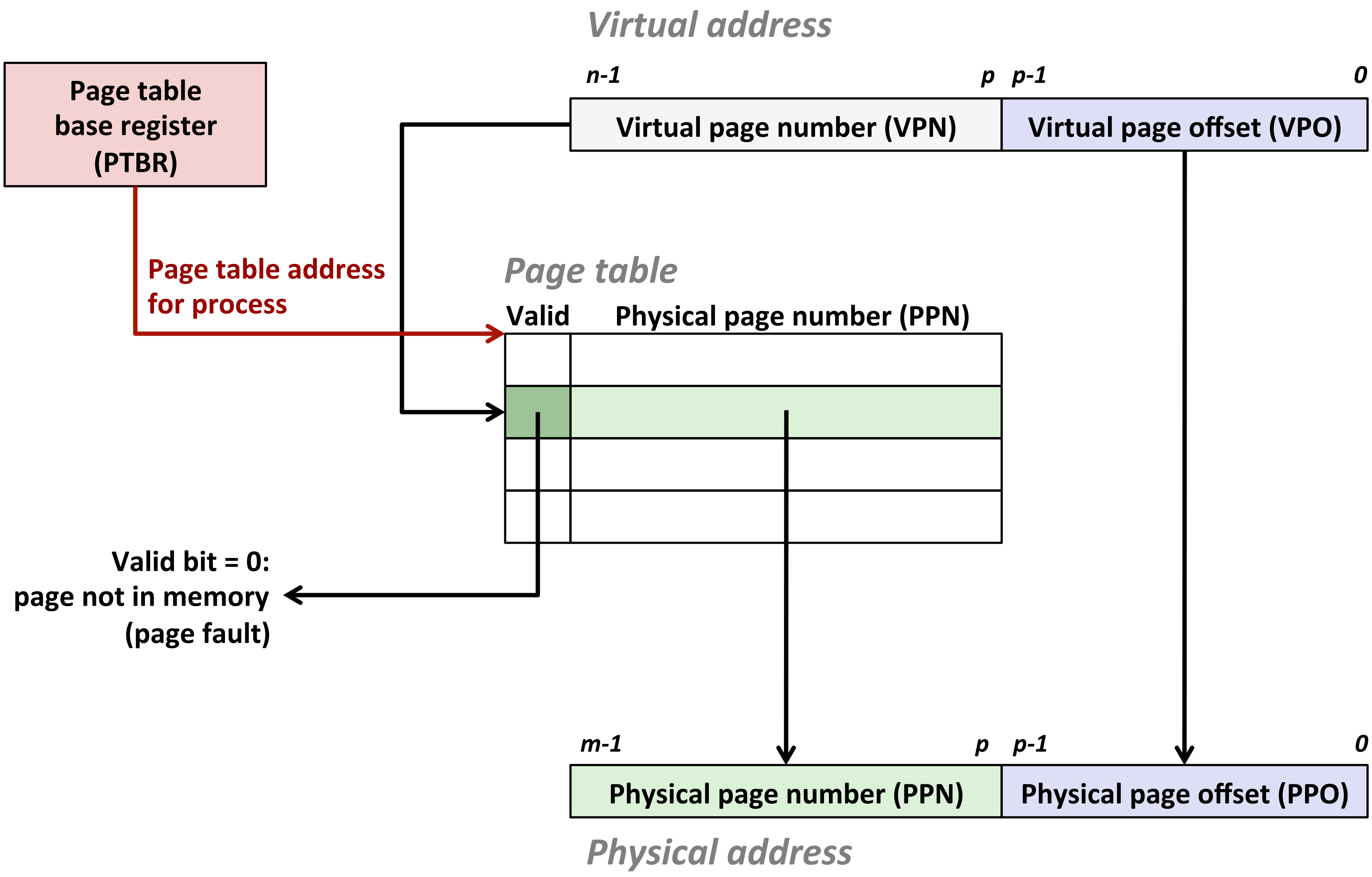
Agenda

- Shell Lab FAQs
- Malloc Lab Sneak Preview
- Virtual Memory Concepts
- **Address Translation**
  - Basic
  - TLB
  - Multilevel

VM Address Translation

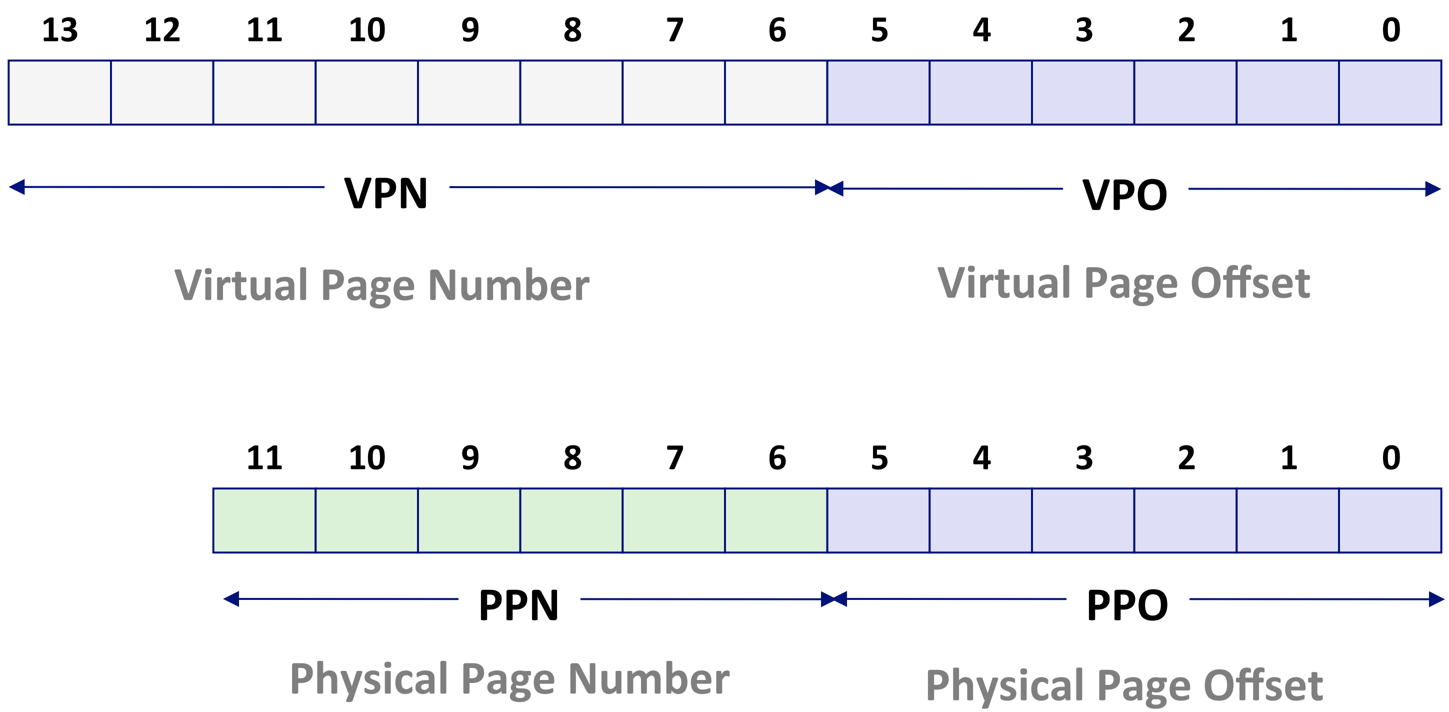
- **Virtual Address Space**
  - $V = \{0, 1, \ldots, N-1\}$
  - There are N possible virtual addresses.
  - Virtual addresses are n bits long; $2^n = N$.
- **Physical Address Space**
  - $P = \{0, 1, \ldots, M-1\}$
  - There are M possible physical addresses.
  - Virtual addresses are m bits long; $2^m = M$.
- **Memory is grouped into “pages.”**
  - Page size is P bytes.
  - The address offset is p bytes; $2^p = P$.
  - Since the virtual offset (VPO) and physical offset (PPO) are the same, the offset doesn’t need to be translated.

VM Address Translation



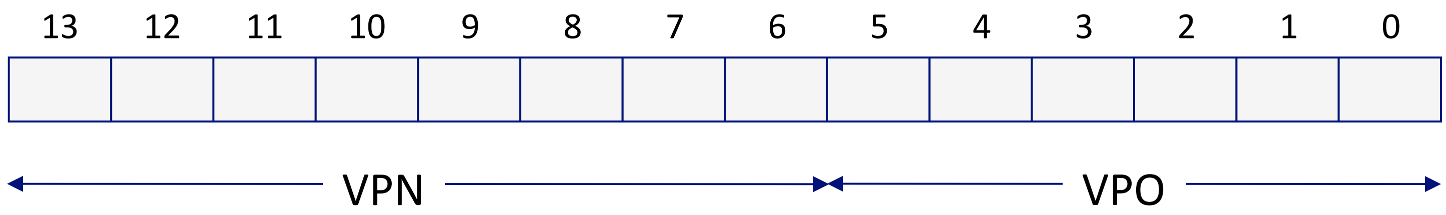
VM Address Translation

- **Addressing**
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bytes



Example 1: Address Translation

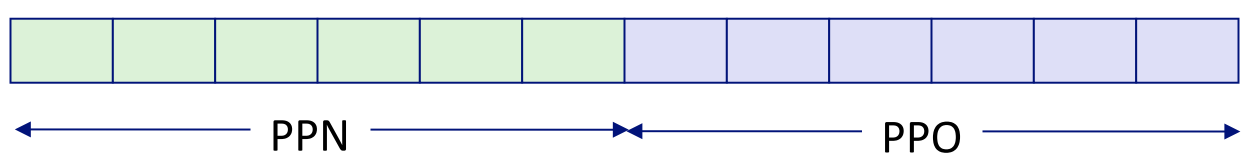
- Pages are 64 bytes. How many bits is the offset?
- Find `0x03D4`.



- VPN: _____
- PPN: ______
- Physical address: ___________

VPN	PPN	Valid
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

VPN	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1



Example 1: Address Translation

- Pages are 64 bytes. How many bits is the offset? $\log_2 64 = 6$
- Find `0x03D4`.

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0

VPN (bits 13–6) | VPO (bits 5–0)

- VPN: `0x0F`
- PPN: `0x0D`
- Physical address: `0x0354`

VPN	PPN	Valid
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

VPN	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

0	0	1	1	0	1	0	1	0	1	0	0

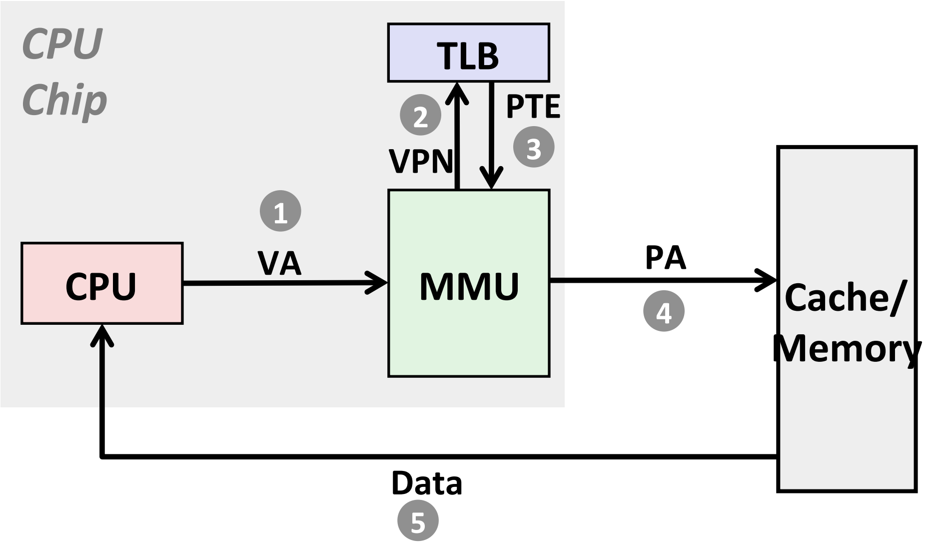
PPN (first 6 bits) | PPO (last 6 bits)

Agenda

- Shell Lab FAQs
- Malloc Lab Sneak Preview
- Virtual Memory Concepts
- **Address Translation**
  - Basic
  - TLB
  - Multilevel

VM Address Translation with TLB

- **That's nice and simple, but it doubles memory usage.**
  - One memory access to look in the page table.
  - One memory access of the actual memory we're looking for.
- **Solution:**
  - Cache the most frequently used page table entries in the TLB.
  - To look up a virtual address in the TLB, split up the VPN (not the whole virtual address!) into a TLB index and a TLB tag.



Example 2: Address Translation with TLB

1 MB of virtual memory

256 KB of physical memory

4 KB page size

TLB: 8 entries, 2-way set associative

- **How many bits are needed to represent the virtual address space?**
- **How many bits are needed to represent the physical address space?**
- **How many bits are needed to represent the offset?**

- **How many bits are needed to represent VPN?**
- **How many bits are in the TLB index?**
- **How many bits are in the TLB tag?**

Example 2: Address Translation with TLB

1 MB of virtual memory

256 KB of physical memory

4 KB page size

TLB: 8 entries, 2-way set associative

- **How many bits are needed to represent the virtual address space? 20. (1 MB = 2^{20} bytes.)**
- **How many bits are needed to represent the physical address space? 18. (256 KB = 2^{18} bytes.)**
- **How many bits are needed to represent the offset? 12. (4 KB = 2^{12} bytes.)**
- **How many bits are needed to represent VPN? 8. (20-12.)**
- **How many bits are in the TLB index? 2. (4 sets = 2^2 set bits.)**
- **How many bits are in the TLB tag? 6. (8-2.)**

Example 2a: Address Translation with TLB

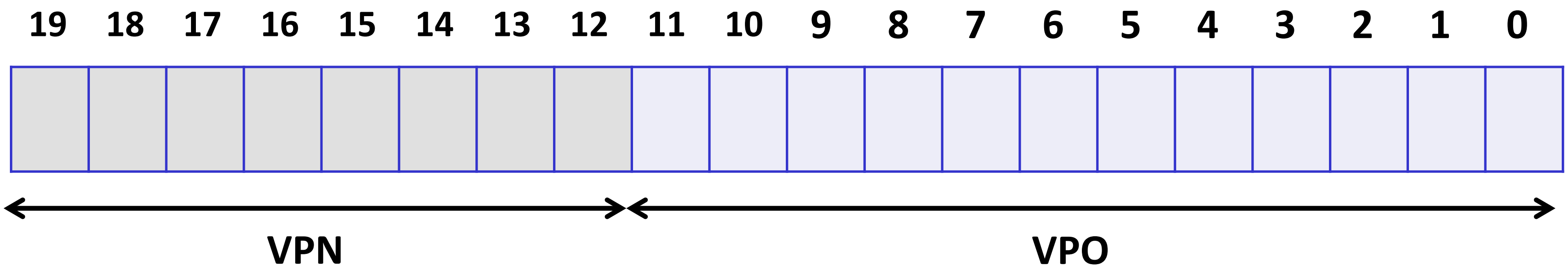
- **Translate `0x14213`, given the contents of TLB and the first 32 entries of the page table below. (All the numbers are in hexadecimal.)**

TLB			
Index	Tag	PPN	Valid
0	05	13	1
	3F	15	1
1	10	0F	1
	0F	1E	0
2	1F	01	1
	11	1F	0
3	03	2B	1
	1D	23	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	17	1	10	26	0
01	28	1	11	17	0
02	14	1	12	0E	1
03	0B	0	13	10	1
04	26	0	14	13	1
05	13	0	15	1B	1
06	0F	1	16	31	1
07	10	1	17	12	0
08	1C	0	18	23	1
09	25	1	19	04	0
0A	31	0	1A	0C	1
0B	16	1	1B	2B	0
0C	01	0	1C	1E	0
0D	15	0	1D	3E	1
0E	0C	0	1E	27	1
0F	2B	1	1F	15	1

Example 2a: Address Translation with TLB

`0x14213`



VPN:
TLBI:
TLBT:

TLB			
Index	Tag	PPN	Valid
0	05	13	1
	3F	15	1
1	10	0F	1
	0F	1E	0
2	1F	01	1
	11	1F	0
3	03	2B	1
	1D	23	0

TLB Hit!
PPN:
Offset:

Physical address:

Example 2a: Address Translation with TLB

0x14213

TLBT (bits 19–14) | TLBI (bits 13–12)

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	1

VPN (bits 19–12) | VPO (bits 11–0)

VPN: 0x14
TLBI: 0x00
TLBT: 0x05

TLB			
Index	Tag	PPN	Valid
0	05	13	1
	3F	15	1
1	10	0F	1
	0F	1E	0
2	1F	01	1
	11	1F	0
3	03	2B	1
	1D	23	0

TLB Hit!
PPN: 0x13
Offset: 0x213

Physical address:
0x13213

Example 2b: Address Translation with TLB

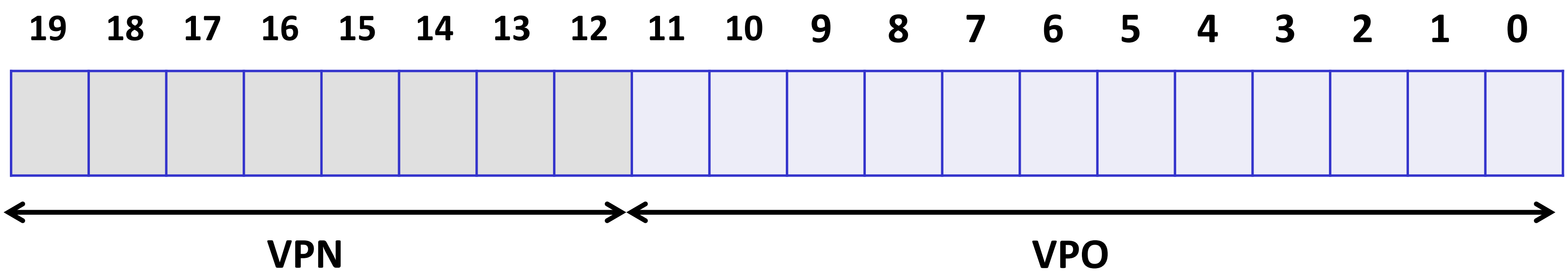
- Translate `0x1F213`, given the contents of TLB and the first 32 entries of the page table below.

TLB			
Index	Tag	PPN	Valid
0	05	13	1
	3F	15	1
1	10	0F	1
	0F	1E	0
2	1F	01	1
	11	1F	0
3	03	2B	1
	1D	23	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	17	1	10	26	0
01	28	1	11	17	0
02	14	1	12	0E	1
03	0B	0	13	10	1
04	26	0	14	13	1
05	13	0	15	1B	1
06	0F	1	16	31	1
07	10	1	17	12	0
08	1C	0	18	23	1
09	25	1	19	04	0
0A	31	0	1A	0C	1
0B	16	1	1B	2B	0
0C	01	0	1C	1E	0
0D	15	0	1D	3E	1
0E	0C	0	1E	27	1
0F	2B	1	1F	15	1

Example 2b: Address Translation with TLB

`0x1F213`



VPN:

TLBI:

TLBT:

TLB			
Index	Tag	PPN	Valid
0	05	13	1
	3F	15	1
1	10	0F	1
	0F	1E	0
2	1F	01	1
	11	1F	0
3	03	2B	1
	1D	23	0

Example 2b: Address Translation with TLB

`0x1F213`

TLBT (bits 19–14) | TLBI (bits 13–12)

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	0	1	1

VPN (bits 19–12) | VPO (bits 11–0)

VPN: `0x1F`
TLBI: `0x03`
TLBT: `0x07`

TLB			
Index	Tag	PPN	Valid
0	05	13	1
	3F	15	1
1	10	0F	1
	0F	1E	0
2	1F	01	1
	11	1F	0
3	03	2B	1
	1D	23	0

TLB Miss!

Step 2: look it up in the page table. ☹

Example 2b: Address Translation with TLB

0x1F213

TLBT | TLBI

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	0	1	1

VPN | VPO

VPN: 0x1F
TLBI: 0x03
TLBT: 0x07

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	17	1	10	26	0
01	28	1	11	17	0
02	14	1	12	0E	1
03	0B	0	13	10	1
04	26	0	14	13	1
05	13	0	15	1B	1
06	0F	1	16	31	1
07	10	1	17	12	0
08	1C	0	18	23	1
09	25	1	19	04	0
0A	31	0	1A	0C	1
0B	16	1	1B	2B	0
0C	01	0	1C	1E	0
0D	15	0	1D	3E	1
0E	0C	0	1E	27	1
0F	2B	1	1F	15	1

Page Table Hit
PPN:
Offset:

Physical address:

Example 2b: Address Translation with TLB

0x1F213

TLBT ← → TLBI

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	0	1	1

VPN ← → VPO

VPN: 0x1F
TLBI: 0x03
TLBT: 0x07

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	17	1	10	26	0
01	28	1	11	17	0
02	14	1	12	0E	1
03	0B	0	13	10	1
04	26	0	14	13	1
05	13	0	15	1B	1
06	0F	1	16	31	1
07	10	1	17	12	0
08	1C	0	18	23	1
09	25	1	19	04	0
0A	31	0	1A	0C	1
0B	16	1	1B	2B	0
0C	01	0	1C	1E	0
0D	15	0	1D	3E	1
0E	0C	0	1E	27	1
0F	2B	1	1F	15	1

Page Table Hit
PPN: 0x15
Offset: 0x213

Physical address:
0x15213

Agenda

- Shell Lab FAQs and I/O
- Malloc Lab Sneak Preview
- Virtual Memory Concepts
- **Address Translation**
  - Basic
  - TLB
  - Multilevel

Address Translation in Real Life

- Multi level page tables, with the first level often called a "page directory"
- Use first part of the VPN to get to the right directory and then the next part to get the PPN
- K-level page table divides VPN into k parts