Anita's Super Awesome Recitation slides

15/18-213: Introduction to Computer Systems
I/O and Virtual Memory, 28 Oct.2013

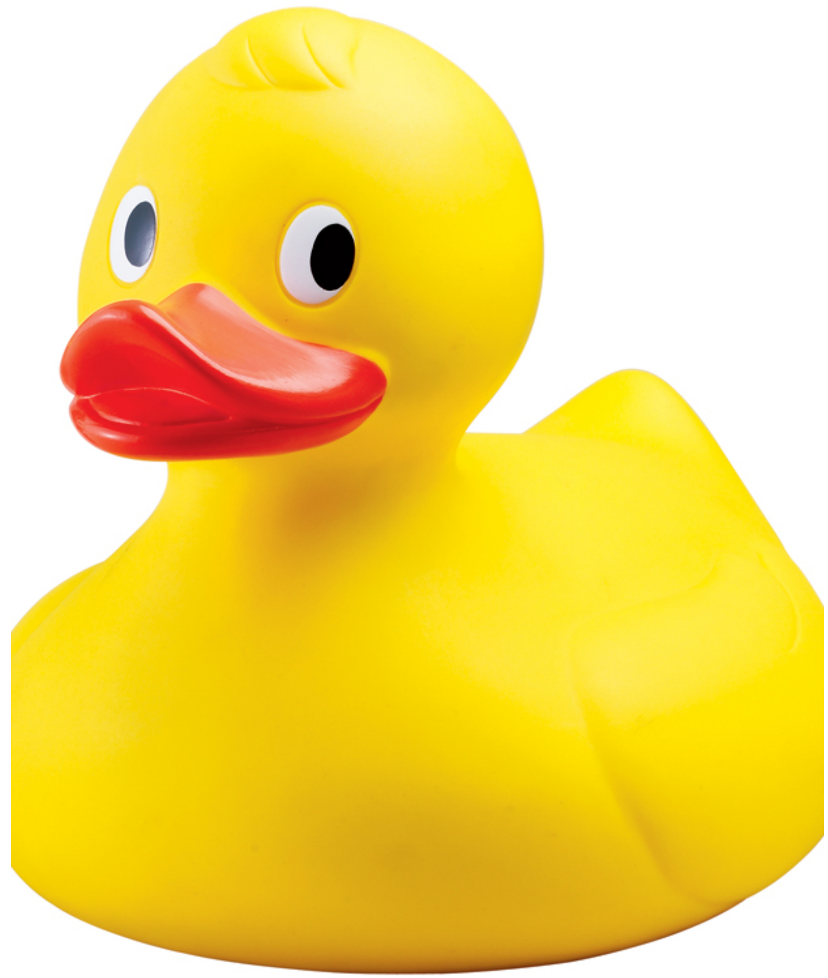
Ian Hartwig, Section E

Boring Stuff

- Shell Lab due THIS Thursday, 28 March 2013
  - We will have more TAs at office hours this week to speed up the queue
- Malloc Lab comes out this Thursday
  - My favorite lab!
  - Design and implement a memory allocator
- Pressing concerns?

Menu for Today

- Teensy Bit of Shell Lab
- I/O (with Pictures!)
- Virtual Memory
- Address Translation
- Extra: C Primer



Rubber Duck Debugging

"To use this process, a programmer explains code to an inanimate object, such as a rubber duck, with the expectation that upon reaching a piece of incorrect code and trying to explain it, the programmer will notice the error."

About sigsuspend()

- For those of you who still need help with it…
  - This site is pretty good
    - “Figure 10.22. Protecting a critcal region from a signal” is a really good example of how sigsuspend() works

I/O

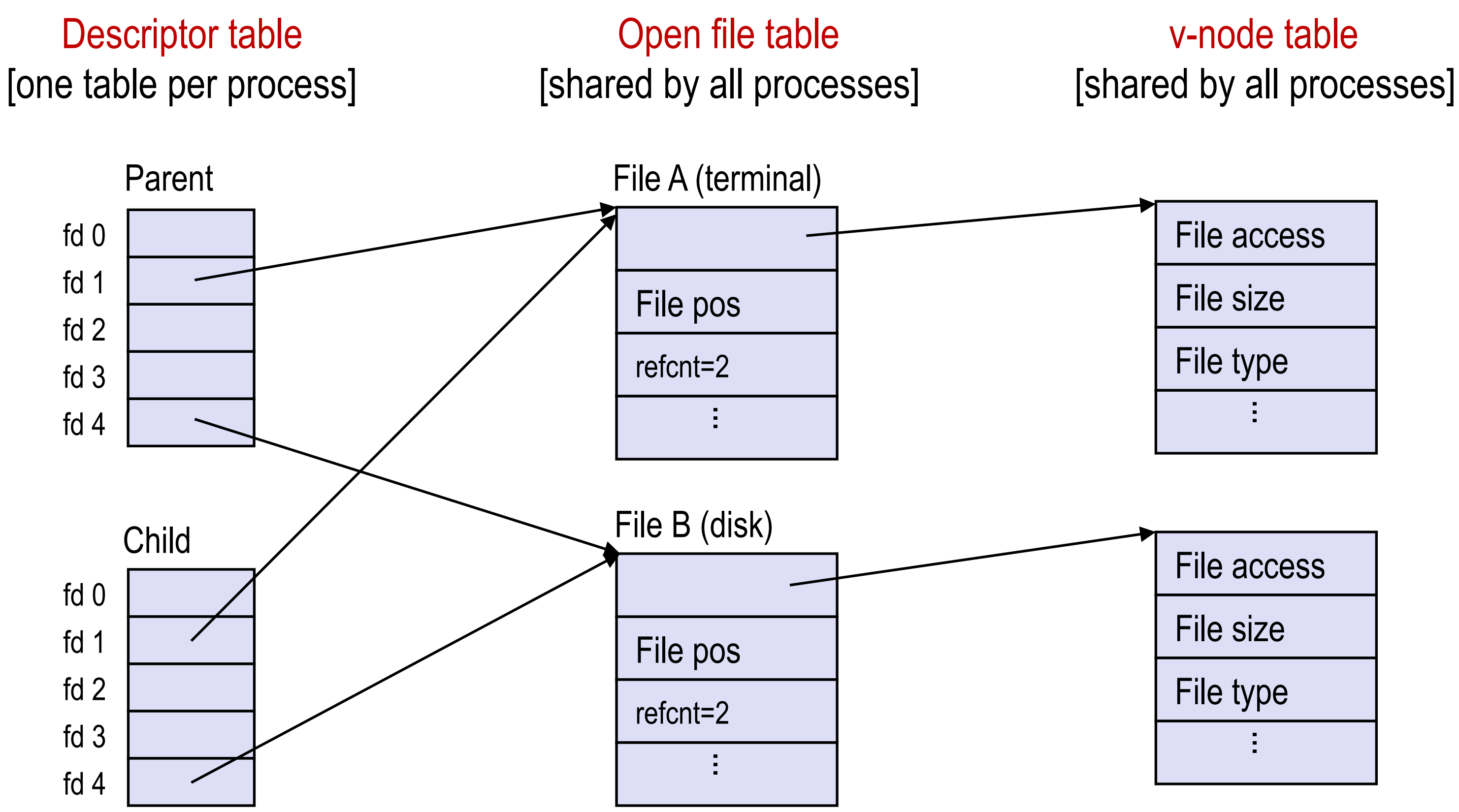
- Four basic operations
  - open()
  - close()
  - read()
  - write()
- What's a file descriptor?
  - Returned by open()
  - Some positive value, or -1 to denote error
  - `int fd = open("/path/to/file", O_RDONLY);`

File Descriptors

- Every process starts with these 3 by default
  - 0 – STDIN
  - 1 – STDOUT
  - 2 – STDERR
- Every process gets its own **file descriptor table**
- Forked processes share open file tables
- All processes share v-node tables
  - Contains the stat structure with info about a file

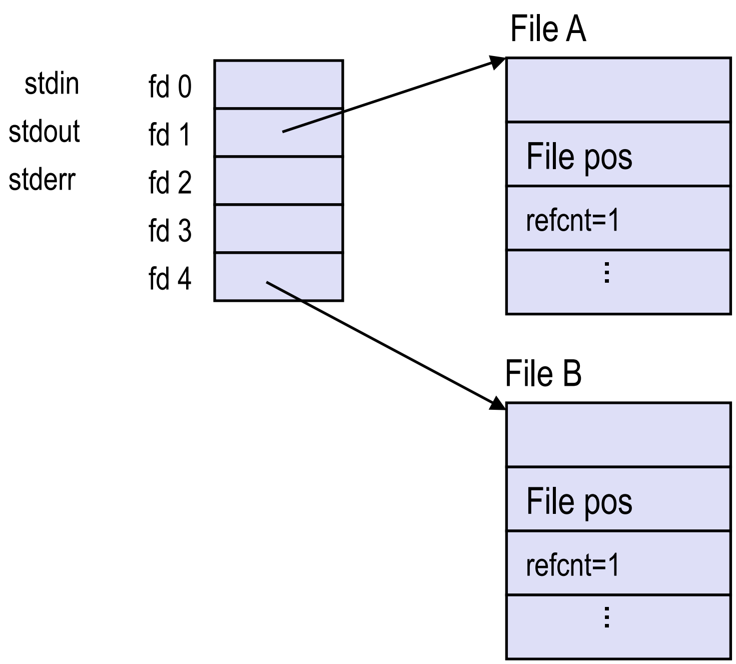
Parent and Child After fork()

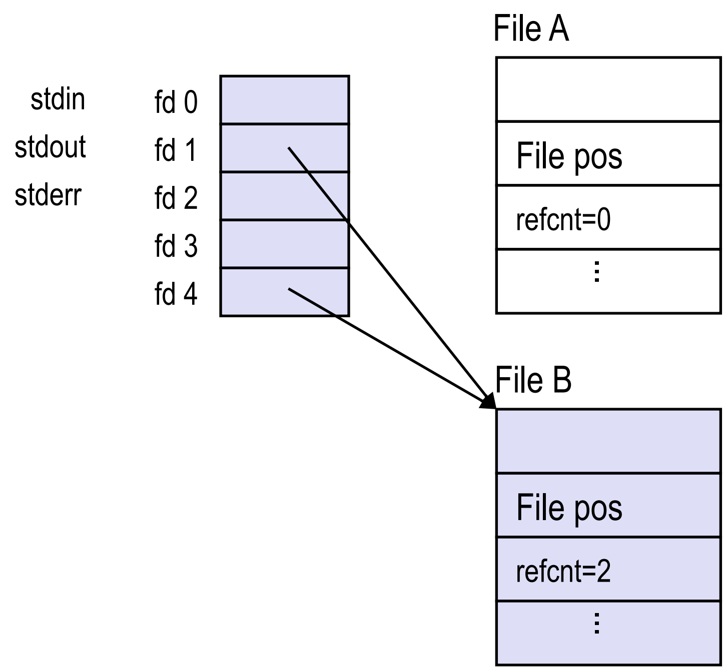
- Shamelessly stolen from lecture:



dup2() Super Relevant Example

- Use open() to open a file to redirect stdout
  - shelllab: Done before exec in the child process
- Call dup2(4,1)
  - Copies fd entries
  - Cause fd=1 to refer to disk file pointed at by fd=4





Magic Numbers are Gross

- If someone doesn't know what your code does, these could mean anything:
  - `0 – STDIN`
  - `1 – STDOUT`
  - `2 – STDERR`
- These are painfully obvious:
  - `STDIN_FILENO`
  - `STDOUT_FILENO`
  - `STDERR_FILENO`
- Defined for you in `<unistd.h>`

All the Lies

- Up to now, we've asked you to believe a couple of lies:
  - Each process has access to the entire system's memory.
  - The system has infinite memory.
  - Instructions have static addresses, even if you run the executable in more than one process at once.

All the Lies

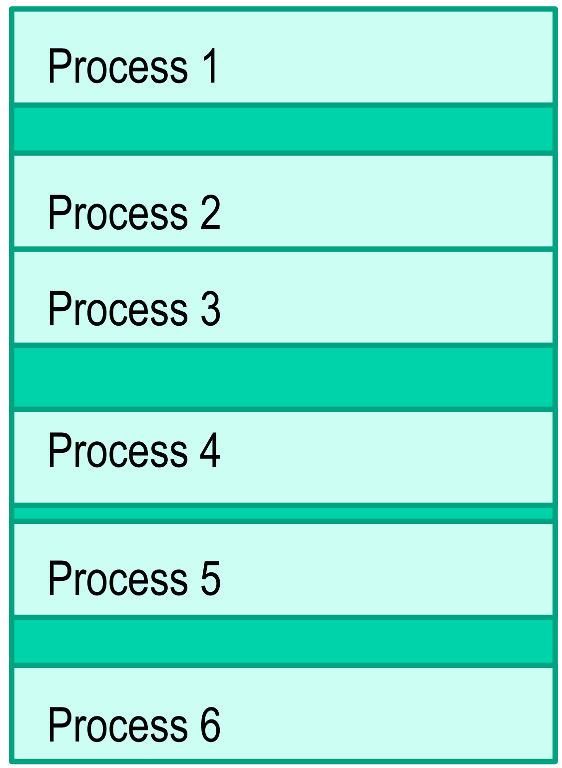
- Up to now, we've asked you to believe a couple of lies:
  - Each process has access to the entire system's memory.
  - The system has infinite memory.
  - Instructions have static addresses, even if you run the executable in more than one process at once.

- How do we make this possible?
  - ***Virtual Memory***

VM: Problems with Direct Mapping

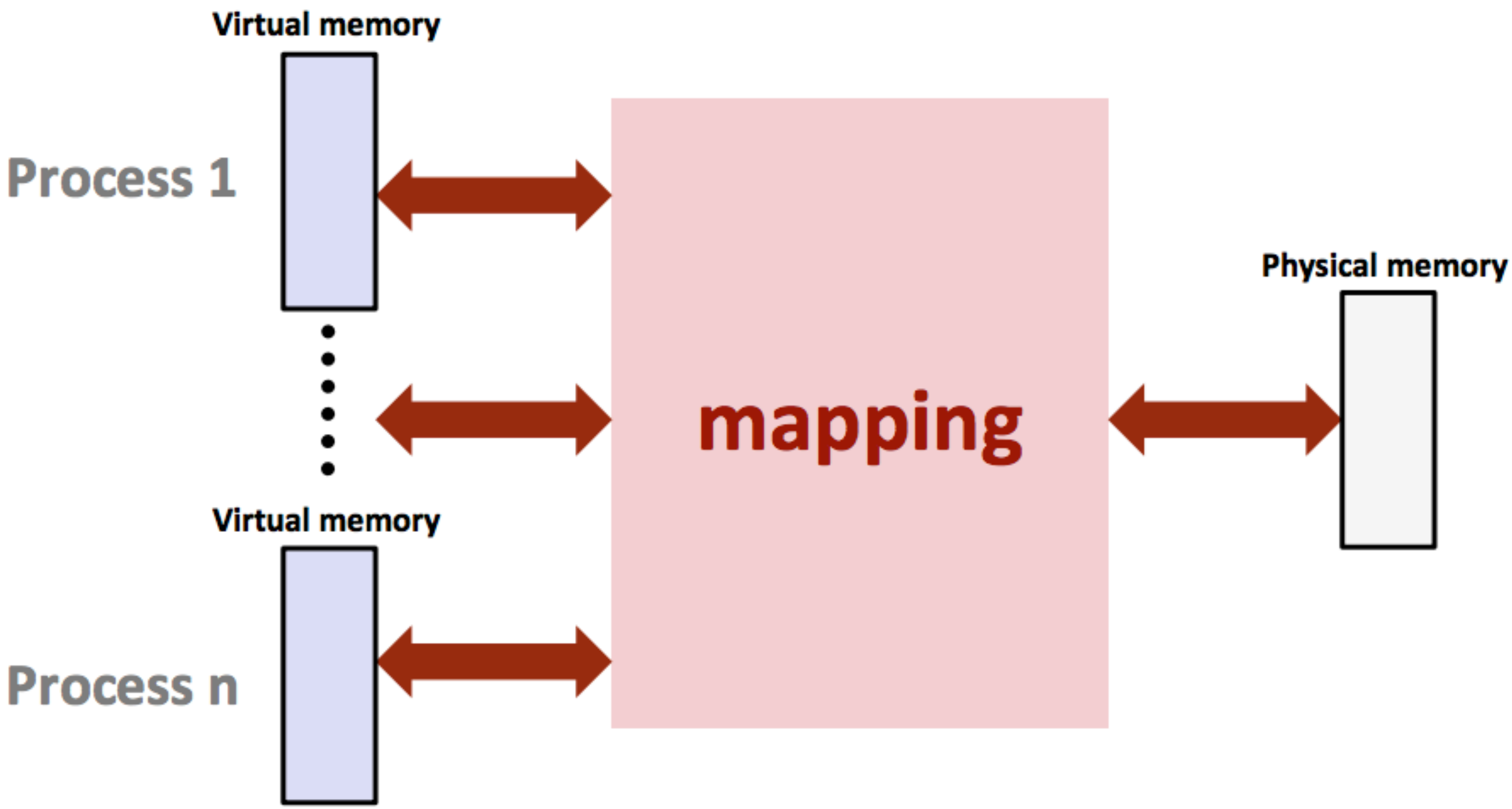
- Questions to ponder:
  - How can we grow processes safely?
  - What to do about fragmentation?
  - How can we make large contiguous chunks fit easier?

Direct Mapping Fragmentation



How do we Solve These Problems?

- We are scientists (and engineers)…

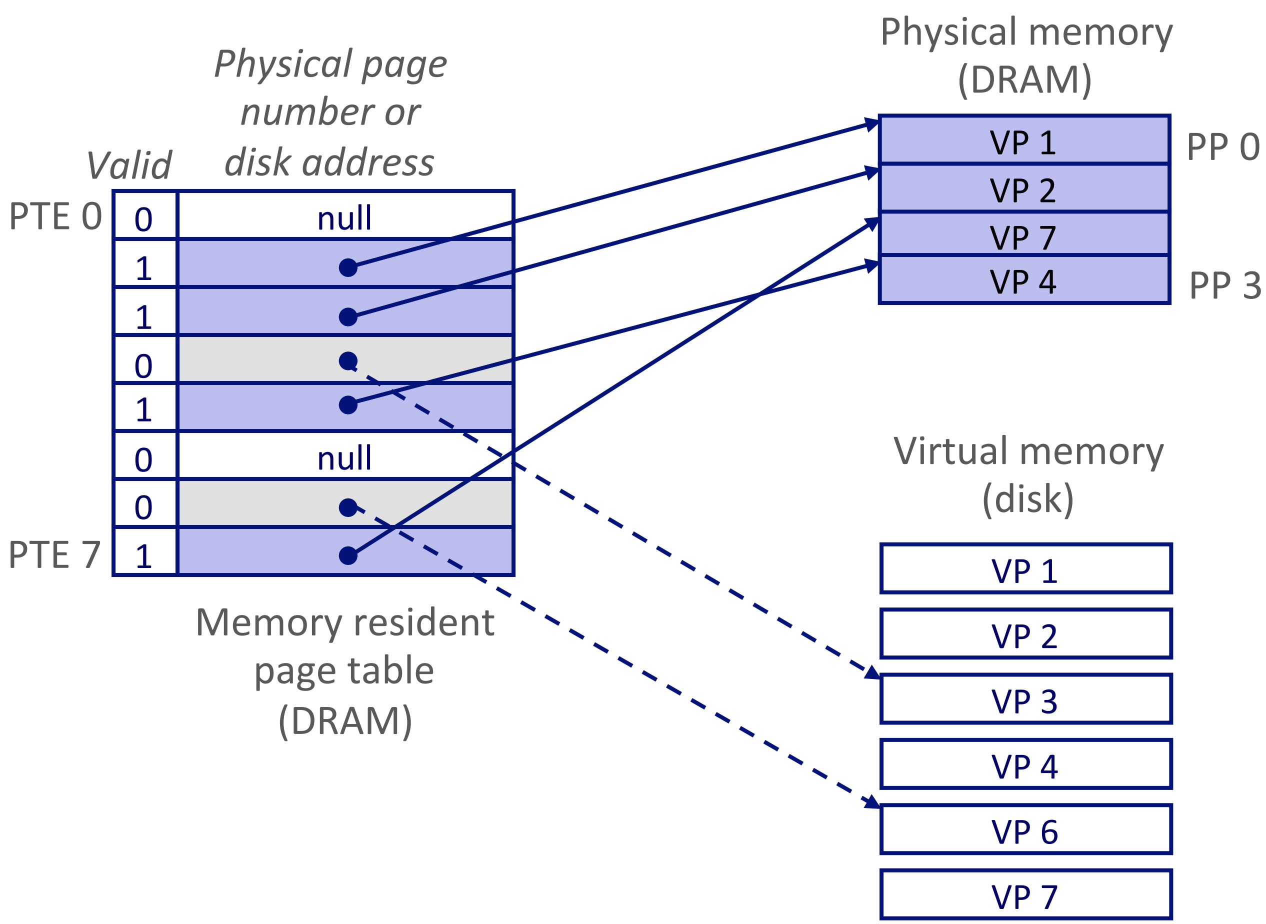


Virtual Memory

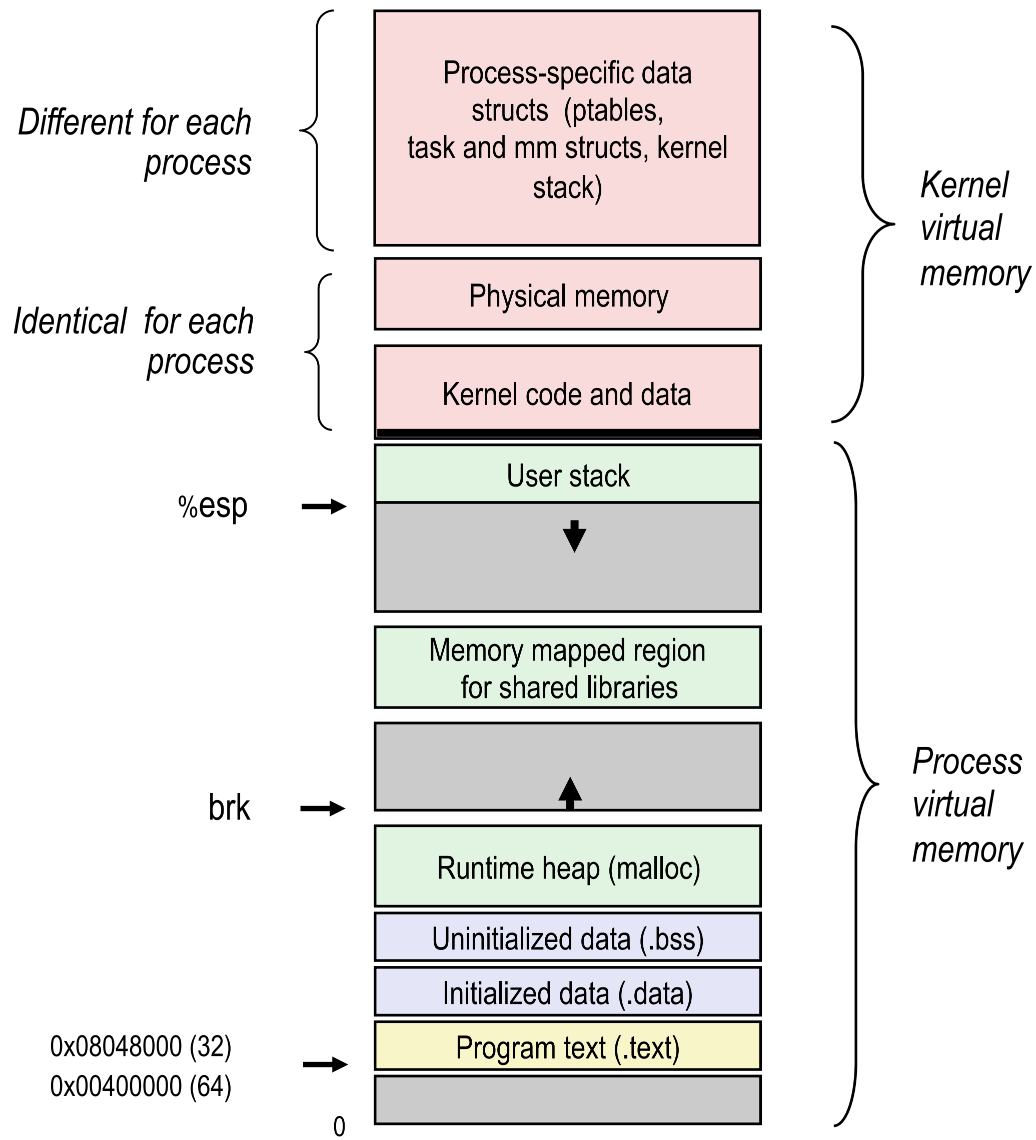
- ..Is the Best Thing Ever™
  - Demand paging
  - Memory Management
  - Protection
- Allows the illusion of infinite memory
  - Kernel manages page faults
- Each process gets its own virtual address space
  - Mapping is the heart of virtual memory

Enabling data structure: Page Table

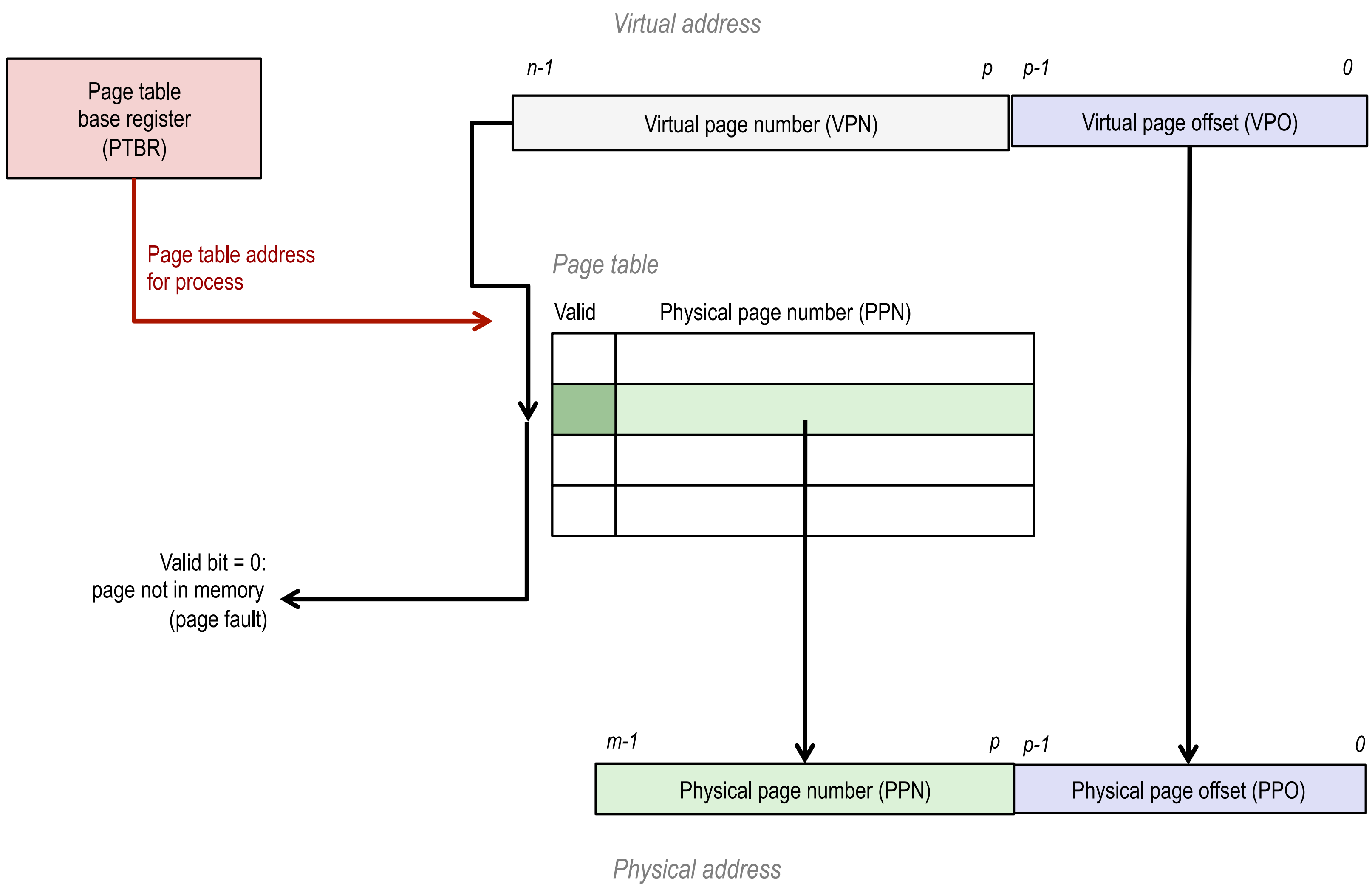
- A *page table* is an array of page table entries (PTEs) that maps virtual pages to physical pages
  - Per-process kernel data structure in DRAM



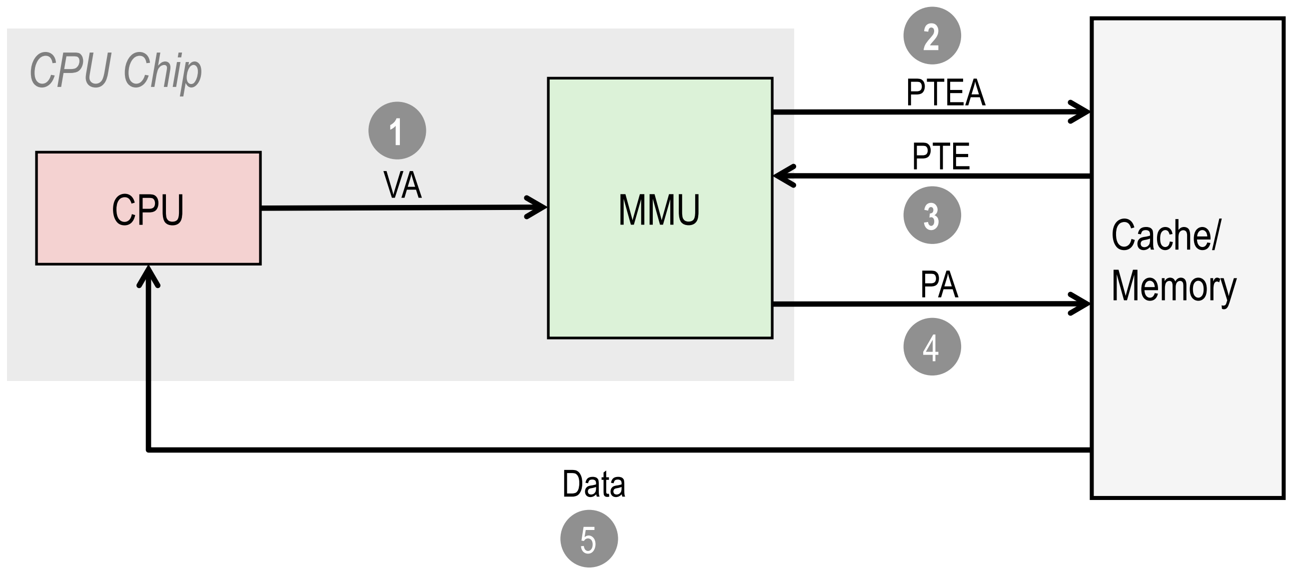
VM of a Linux Process



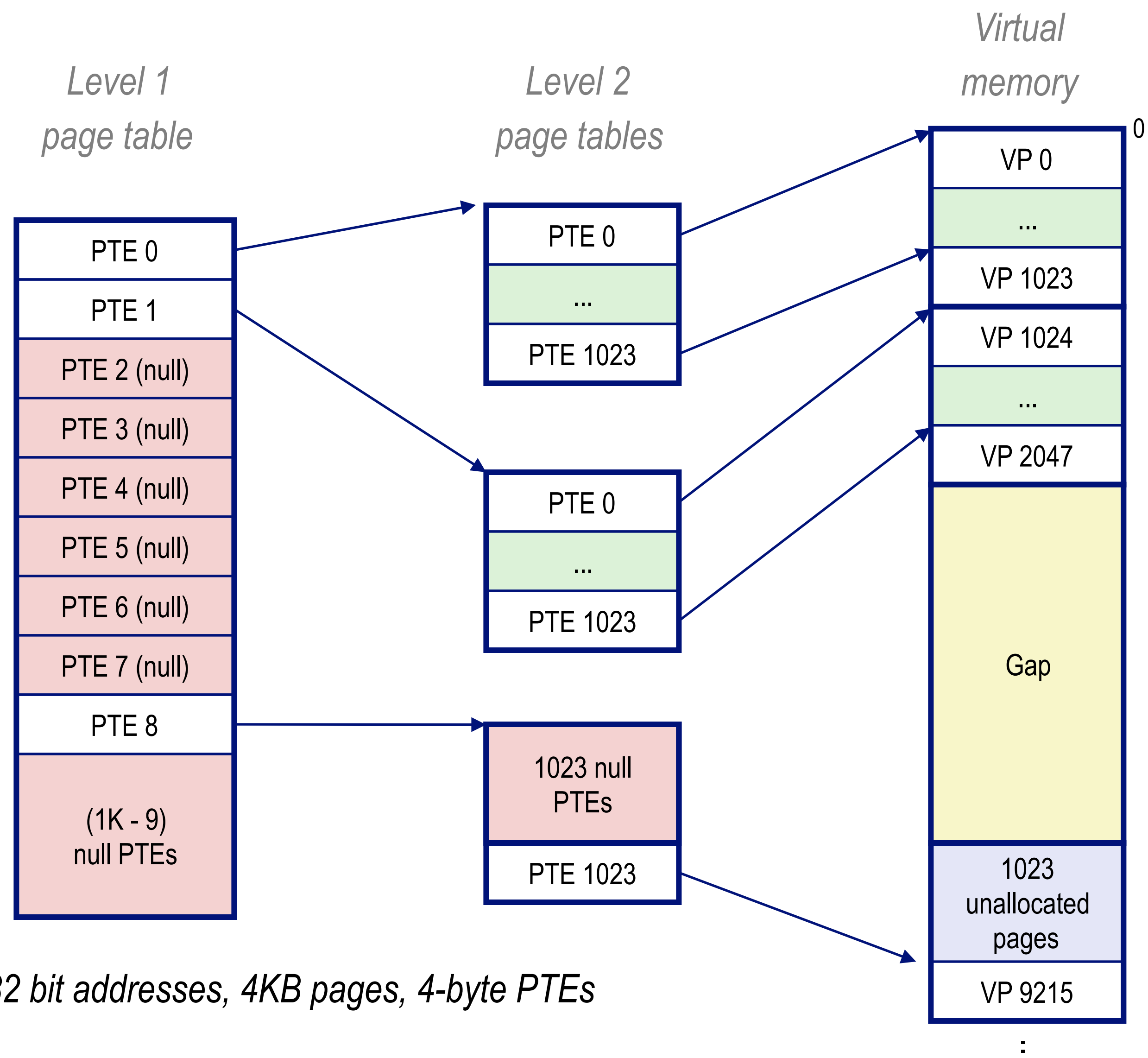
VM: Address Translations



Overview of a Hit

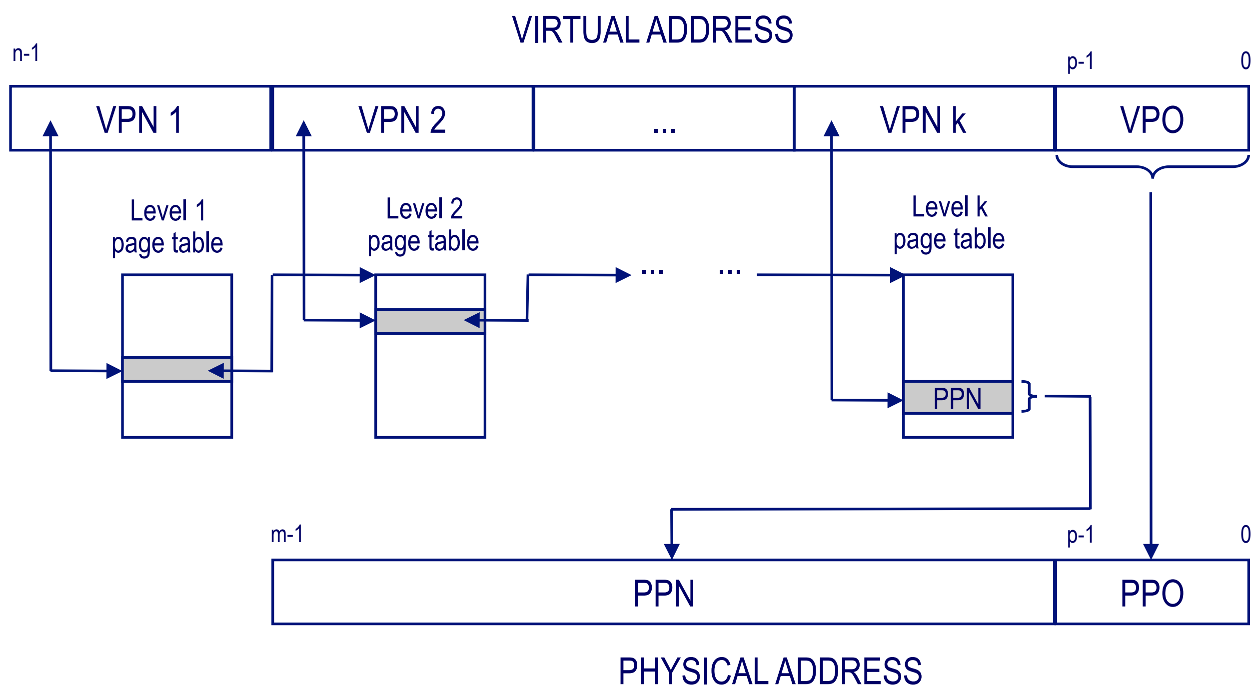


Two-Level Page Table



32 bit addresses, 4KB pages, 4-byte PTEs

Translating w/ a k-level Page Table



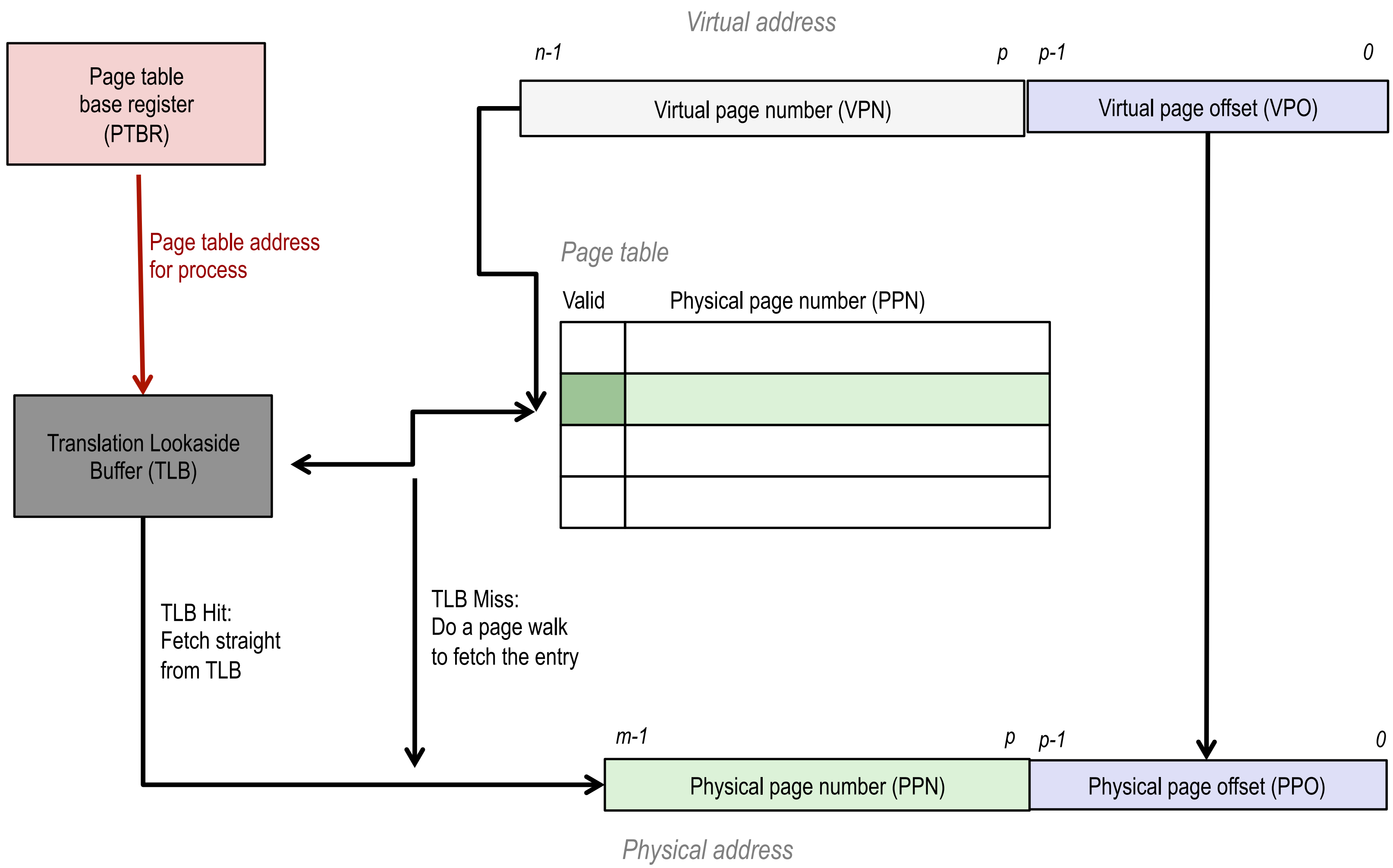
But Memory Accesses are Slow

- At least 2 memory accesses
  - Fetch page-table entry (PTE) from memory
  - Then fetch data from memory
- In x86, 3 memory accesses
  - Page directory, page table, physical memory
- In x86_64, 4 level page-mapping system
- What should we do?
  - Please don’t say insert a level of “indirection”

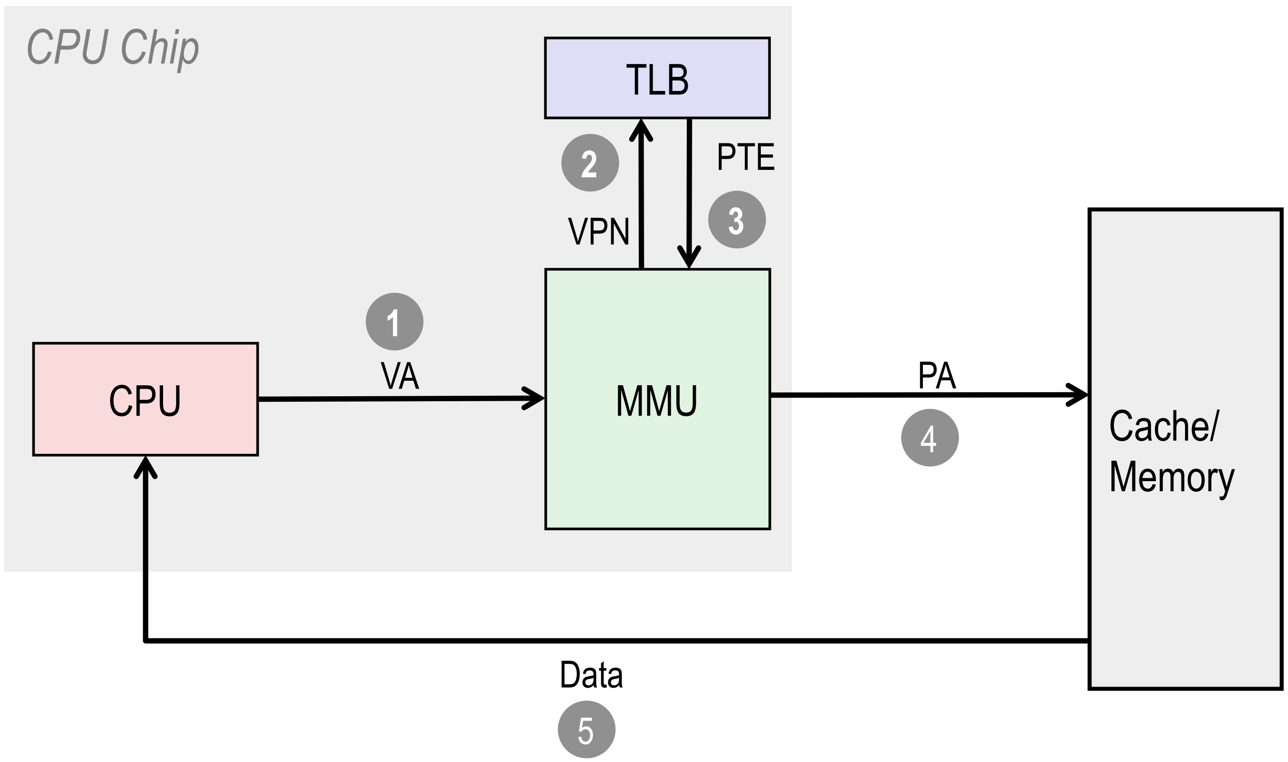
Translation Lookaside Buffer (TLB)

- Super fast hardware cache of PTEs
- Idea: Locality exists between memory accesses
  - Typically access nearby memory
  - Usually on the same page as current data
    - Arrays with loops
    - Program instructions

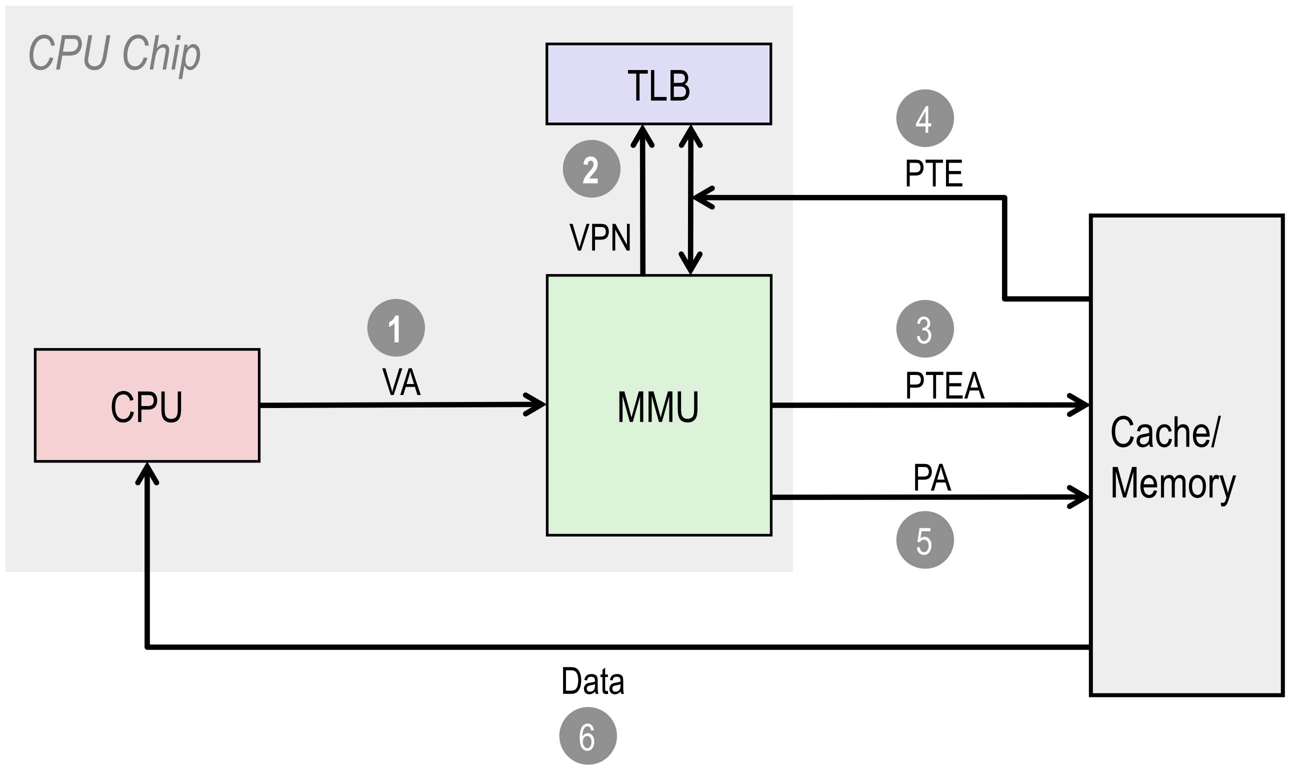
VM: Translations w/ TLB and Tables



Overview of a TLB Hit

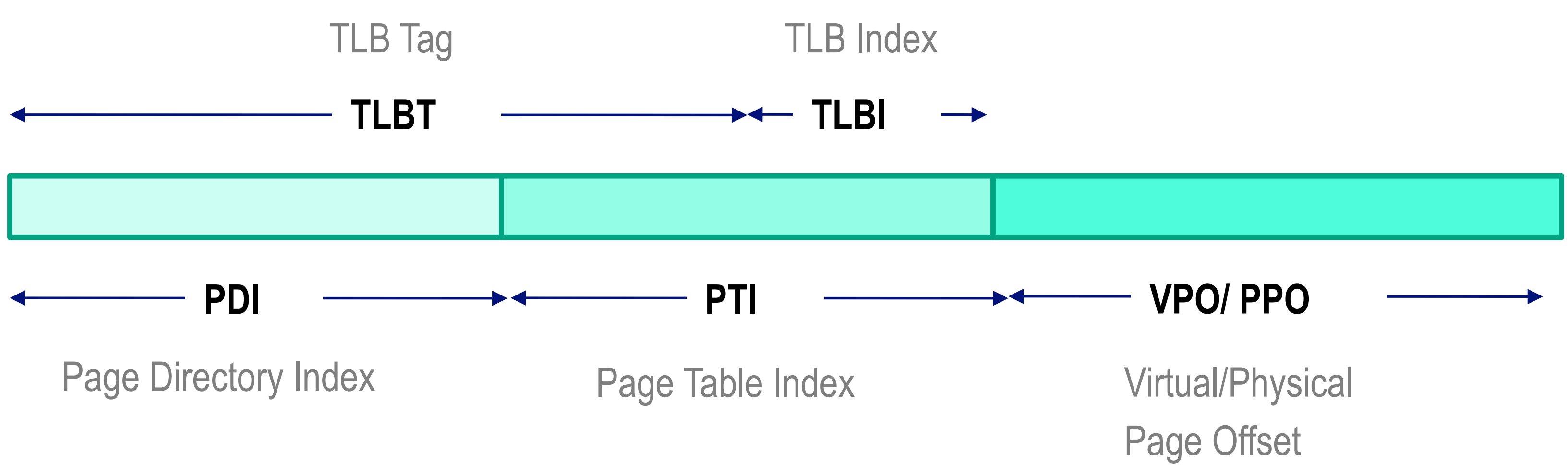


Overview of a TLB Miss

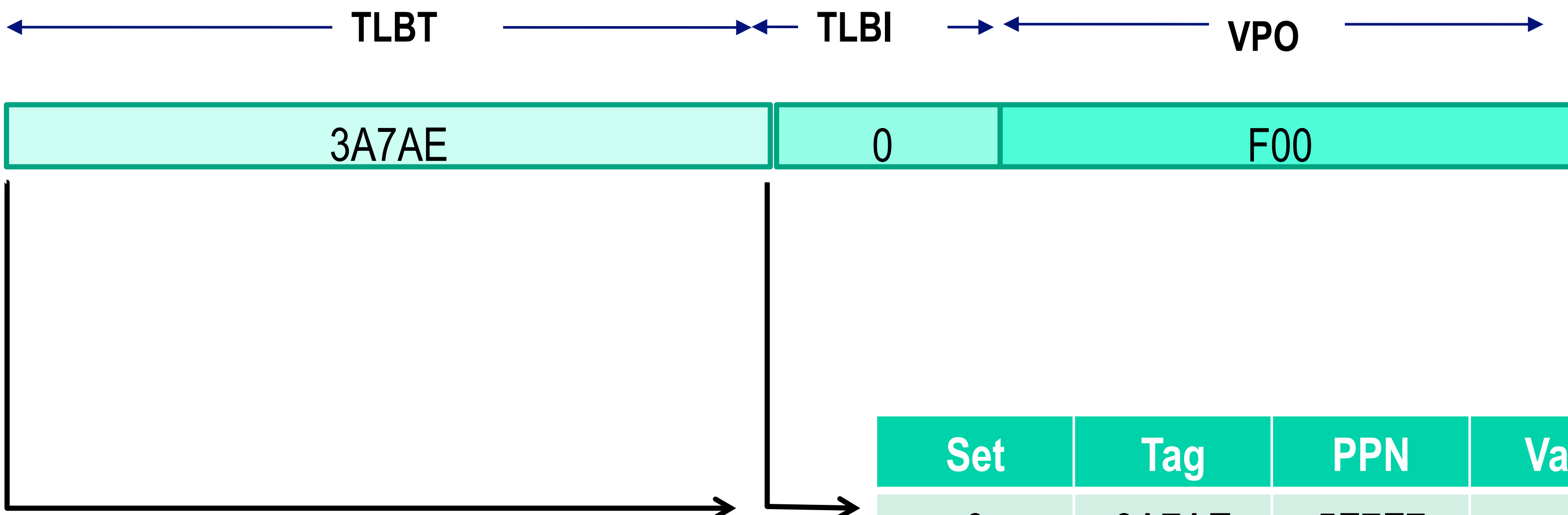


Tutorial: Virtual Address Translation

- Addressing
  - 32 bit virtual address
  - 32 bit physical address
  - Page size = 4 kb
- Paging
  - 10 bit page directory index
  - 10 bit page table index
  - 12 bit offset
- TLB
  - Direct Mapped
  - 4 entries



Tutorial: Address Translation Hit



Set	Tag	PPN	Valid
0	3A7AE	5F7F7	1
1			
2			
3			

- Always access TLB first

Tutorial: Address Translation Hit

TLBT	TLBI	VPO
3A7AE	0	F00

Set	Tag	PPN	Valid
0	3A7AE	5F7F7	1

VPO and PPO are always the same!

PPN	PPO
5F7F7	F00
Physical Page Number	Physical Page Offset

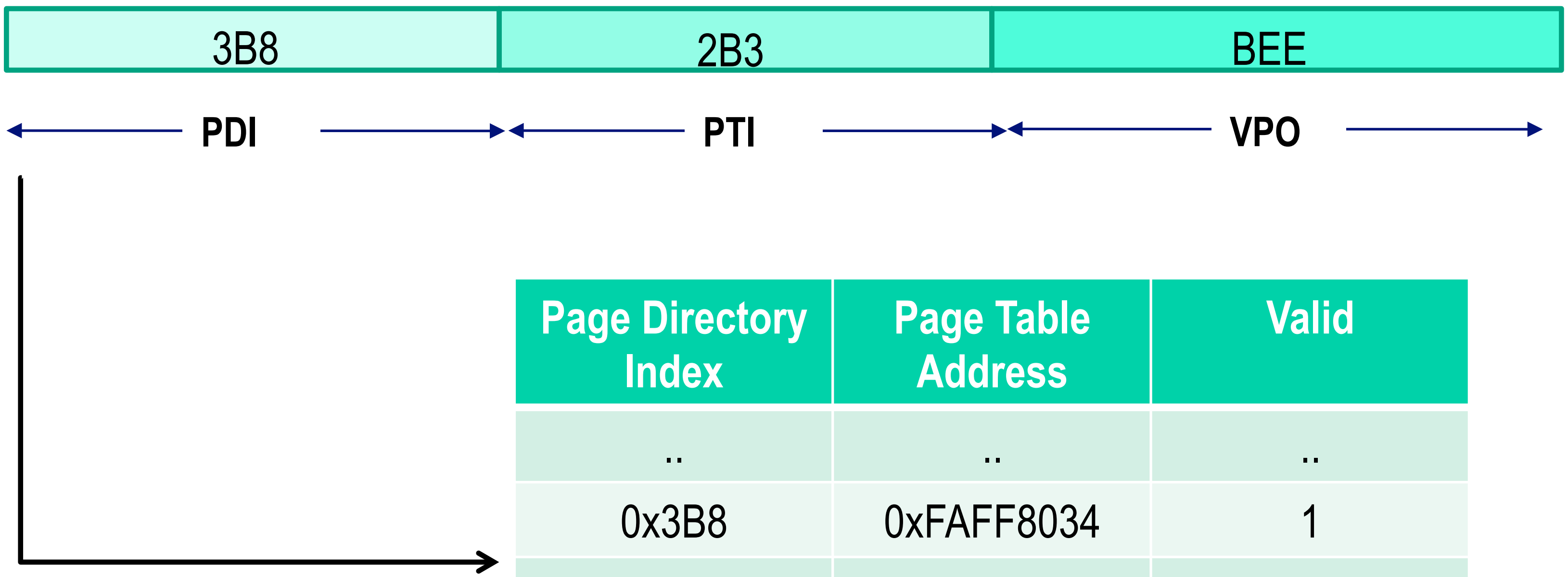
Tutorial: Address Translation Miss

TLBT	TLBI	VPO
3B8AC	3	BEE

Set	Tag	PPN	Valid
0			
1			
2			
3	3B8AC	DEAD	0

- TLB Miss! Do page walk

Tutorial: Address Translation Miss



Page Directory Index	Page Table Address	Valid
..	..	..
0x3B8	0xFAFF8034	1
0x3B9	..	..
...	..	..

Tutorial: Address Translation Miss

3B8	2B3	BEE
PDI	PTI	VPO

PDI	PTA	Valid
..	..	..
0x3B8	0xFAFF8034	1
0x3B9	..	..
...	..	..

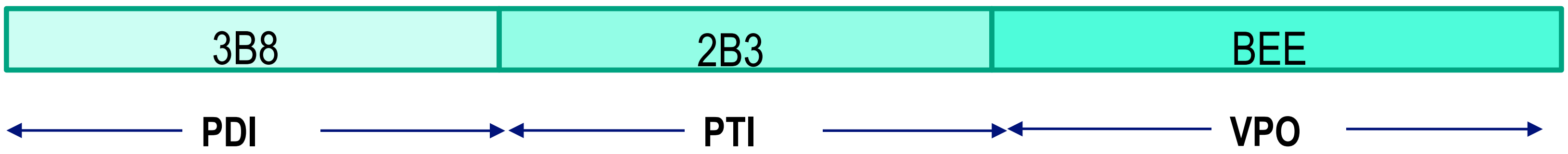
0xFAFF8034

PTI	PPN	Valid
..	..	..
...	...	...
...	..	..
...	..	..

0xFAFF9034

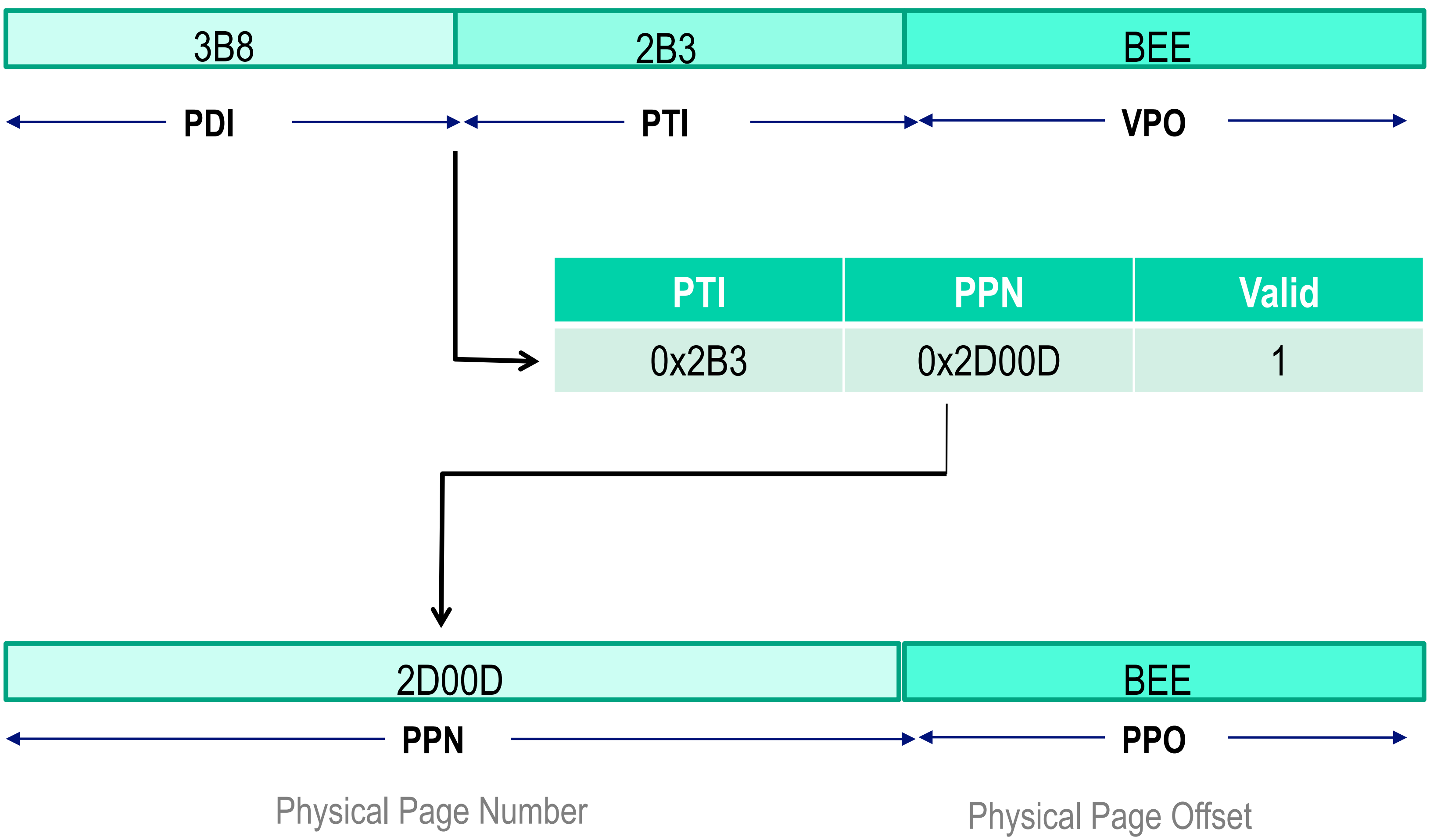
PTI	PPN	Valid
..	..	..
...	...	...
...	..	..
...	..	..

Tutorial: Address Translation Miss



Page Table Index	PPN	Valid
..	..	…
0x2B2	…	…
0x2B3	0x2D00D	1
…	…	…

Tutorial: Address Translation Miss



PTI	PPN	Valid
0x2B3	0x2D00D	1

Translation Macro Exercise

- 32 bit address: 10 bit VPN1, 10 bit VPN2, 12 bit VPO
- 4KB pages
- Define the following function like macros:
  - Page align

```
#define PAGE_ALIGN(v_addr) ________________________________
```

  - Gets VPN1/VPN2 as unsigned int from virtual address

```
#define VPN1(v_addr) ______________________________________
#define VPN2(v_addr) ______________________________________
```

  - Gets VPO as unsigned int from virtual address

```
#define VPO(v_addr) _______________________________________
```

  - Calculates the address of the page directory index

```
#define PDEA(pd_addr, v_addr) ______________________________
```

  - Calculate address of page table entry

```
#define PTEA(pd_addr, v_addr)______________________________
```

  - `Calculate physical address`

```
#define PA(pd_addr, v_addr) ________________________________
```

Translation Macro Solution

- 32 bit address: 10 bit VPN1, 10 bit VPN2, 12 bit VPO
- 4KB pages
- Define the following function like macros:
  - Page align

```
#define PAGE_ALIGN(v_addr) ((unsigned int) v_addr & ~0xfff)
```

  - Gets VPN1/VPN2 as unsigned int from virtual address

```
#define VPN1(v_addr) ((unsigned int) (((v_addr)>>22)&0x3ff))
#define VPN2(v_addr) ((unsigned int) (((v_addr)>>12)&0x3ff))
```

  - Gets VPO as unsigned int from virtual address

```
#define VPO(v_addr) ((unsigned int) ((v_addr)&0xfff))
```

  - Calculates the address of the page directory index

```
#define PDEA(pd_addr, v_addr) (((void **)pd_addr)+VPN1(v_addr))
```

  - Calculate address of page table entry

```
#define PTEA(pd_addr, v_addr)
    (((void **)PAGE_ALIGN(*PDEA(pd_addr,v_addr)))+VPN2(v_addr))
```

  - Calculate physical address

```
#define PA(pd_addr, v_addr)
    (((PAGE_ALIGN(*PTEA(pd_addr,v_addr)))) | VPO(v_addr))
```

Extra Stuff

- For next week, or for your enjoyment

All the C!

- "Saving you from malloc misery…"
- Basics
- Useful C Stuff
- Debugging
- Brian W. Kernighan and Dennis M. Ritchie,
  The C Programming Language, Second Edition,
  Prentice Hall, 1988

C and Pointer Basics

- Statically allocated arrays:
  - `int prices[100];`
  - Getting rid of magic numbers:
    - `int prices[NUMITEMS];`
- Dynamically allocated arrays:
  - `int *prices2 = (int *) malloc(sizeof(int) * var);`
- Which is valid:
  - `prices2 = prices;`
  - `prices = prices2;`
- The & operator:
  - `&prices[1]` is the same as `prices+1`
- Function Pointer:
  - `int (*fun)();`
  - Pointer to function returning int

Peeling the Onion (K&R p.101)

- `char **argv`
  - argv: pointer to a pointer to a char
- `int (*daytab)[13]`
  - daytab: pointer to array[13] of int
- `int *daytab[13]`
  - daytab: array[13] of pointer to int
- `char (*(*x())[])()`
  - x: function returning pointer to array[] of pointer to function returning char
- `char (*(*x[3])())[5]`
  - x: array[3] of pointer to function returning pointer to array[5] of char
- Takeaway
  - There is an algorithm to decode this (see K&R p. 101)
  - Always use parenthesis!!
  - Typedef

Why Typedefs?

- For convenience and readable code
- Example:
  - ```
    typedef struct
    {
     int x;
     int y;
    } point;
    ```
- Function Pointer example:
  - `typedef int(*pt2Func)(int, int);`
  - `pt2Func` is a pointer to a function that takes 2 int arguments and returns an int
    ```

Macros are Cool

- C Preprocessor looks at macros in the preprocessing step of compilation
- Use `#define` to avoid magic numbers:
  - `#define TRIALS 100`
- Function like macros – short and heavily used code snippets
  - `#define GET_BYTE_ONE(x) ((x) & 0xff)`
  - `#define GET_BYTE_TWO(x) ( ( (x) >> 8) & 0xff)`
- Also look at inline functions (example prototype):
  - `inline int fun(int a, int b)`
  - Requests compiler to insert assembly of max wherever a call to max is made
- Both useful for malloc lab

Debugging – Favorite Methods

- Using the `DEBUG` flag:
  - ```
    #define DEBUG
     . . .
    #ifdef DEBUG
     . . . // debugging print statements, etc.
    #endif
    ```
- Compiling (if you want to debug):
  - `gcc -DDEBUG foo.c -o foo`
- Using `assert`
  - `assert(posvar > 0);`
  - `man 3 assert`
- Compiling (if you want to turn off asserts):
  - `gcc -DNDEBUG foo.c -o foo`
    ```

Debugging – Favorite Methods

- Using printf, assert, etc only in debug mode:
-
```
#define DEBUG -or- //#define DEBUG
#ifdef DEBUG
  # define dbg_printf(...) printf(__VA_ARGS__)
  # define dbg_assert(...) assert(__VA_ARGS__)
  # define dbg(...)        __VA_ARGS__
#else
  # define dbg_printf(...)
  # define dbg_assert(...)
  # define dbg(...)
#endif
```

Little Things

- Usage messages
  - Putting this in is a good habit – allows you to add features while keeping the user up to date
  - `man -h`
- `fopen/fclose`
  - Always error check!
- malloc()
  - Error check
  - Free everything you allocate
- Global variables
  - Namespace pollution
  - If you must, make them private:
    - `static int foo;`

Questions and References Slide

- Rubber Duck 1
- Rubber Duck Debugging on Wiki
- Good sigsuspend() reference
- Indirection on Wiki
- Pictures stolen from lecture slides
- Stole from 15-410 Virtual Memory Slides
  - Lectures reside here
  - BTW, Prof. Eckhardt is super cool